Testing the NEAT Algorithm on a PSPACE-Complete Problem

Angel Marchev, Jr.[1,2][0000−0002−5090−3123],
Dimitar Lyubchev[1][0009−0006−3970−6272], and
Nikolay Penchev[2][0009−0000−8955−2474]

[1] University of National and World Economy, Sofia 1700, 19 December 8 Street, Bulgaria
angel.marchev@unwe.bg
[2] Sofia University "St. Kliment Ohridski", Sofia 1113 Sofia, 125 Tsarigradsko Shose Blvd., bl.3, Bulgaria

Abstract. This paper investigates the efficacy of the Neuro-Evolution of Augmenting Topologies (NEAT) algorithm on PSPACE-complete problems, specifically utilizing the Sokoban puzzle. NEAT, which evolves both neural network topologies and weights, provides a promising approach for solving complex problems without predefined network architectures. We implemented NEAT using the neat-python library and tested it against several reinforcement learning (RL) algorithms, including Deep Q-Network (DQN) and Proximal Policy Optimization (PPO), within the OpenAI gym-sokoban environment. Our experiments involved extensive configuration variations to identify optimal settings for NEAT. Key findings indicate that NEAT solved the Sokoban problem, outperforming traditional RL variants. Our results highlight the importance of incremental structural growth and the protection of topological innovations. This study confirms NEAT's applicability to PSPACE-complete problems.

Keywords: Neuro-Evolution of Augmenting Topologies · PSPACE-Complete · Reinforcement learning.

1 Introduction

Artificial Neural Networks (ANNs) are pivotal in modern computational approaches, enabling advancements across numerous domains. However, the challenge of defining optimal network architectures often necessitates prior domain knowledge, creating barriers for zero-knowledge architecture searches. This study addresses the need for architecture-independent methods by focusing on the NEAT (Neuro-Evolution of Augmenting Topologies) algorithm, as proposed by Stanley and Miikkulainen [1]. Despite its potential, the literature lacks comprehensive investigations into NEAT's efficacy on complex, PSPACE-complete problems. This paper aims to fill that gap by testing NEAT on the Sokoban problem, a known PSPACE-complete task, to evaluate its performance without predefined network structures.

This research employs an extensive testing methodology to explore NEAT's capabilities, leveraging a sophisticated dataset and rigorous experimental design. By comparing NEAT against various Reinforcement Learning (RL) algorithms, we demonstrate the broader applicability of our findings. The use of a challenging problem like Sokoban, combined with multiple configurations and robust checks, ensures the relevance and robustness of our results. This approach not only answers the specific research question but also provides insights into the generalizability of NEAT for complex problem-solving.

Our findings highlight the critical role of fitness customization, species diversity, and the careful management of network complexity in optimizing NEAT's performance. Specifically, NEAT Configuration 1 successfully solved the Sokoban problem within 1000 iterations, showcasing the effectiveness of the proposed adjustments. In contrast, traditional RL variants like DQN, PPO, PPO optimized, and PPO CNN did not achieve success, underscoring potential limitations in their configurations for this specific task.

Through sensitivity analyses and robustness checks, we validated the reliability of our results. Variations in initial nodes, fitness functions, population sizes, mutation rates, and training environments were tested to ensure comprehensive evaluation. NEAT Configuration 1's success demonstrates the viability of our proposed modifications, while the success of Q-learning in solving the problem suggests its potential applicability. Other RL variants' failures indicate areas for further investigation and improvement.

This study builds upon foundational work in neuroevolution, particularly the seminal paper "Efficient Evolution of Neural Network Topologies" by Kenneth O. Stanley and Risto Miikkulainen. By extending NEAT's application to a PSPACE-complete problem, we contribute to the broader literature on ANN architecture optimization and problem-solving without prior domain knowledge. Our work intersects with studies on evolutionary algorithms, reinforcement learning, and complexity theory, providing a comprehensive perspective on NEAT's capabilities.

We show that NEAT can effectively solve PSPACE-complete problems like Sokoban without predefined network structures. Our study underscores the importance of tailored fitness functions in improving neuroevolutionary outcomes. We highlight how maintaining species diversity and managing network complexity can lead to more robust solutions. By comparing NEAT with various RL algorithms, we provide a thorough evaluation of their respective strengths and limitations.

This paper is organized in several sections:

- **Introduction**: Establishes the research question and its significance.
- **Testing Methodology**: Details the experimental design and evaluation criteria.
- **Simulation Problem Definition**: Describes the OpenAI gymnasium environments and the gym-sokoban game.
- **Baseline Solution: Reinforcement Learning**: Discusses the implementation and performance of RL algorithms.

- **Neuro-Evolution of Augmenting Topologies**: Explains the NEAT algorithm.
- **Implementation of NEAT**: Explains the NEAT software implementation, setup and configurations.
- **Key Findings and conclusions**: Presents the findings from the experiments, summarizes the main insights, and implications and suggests potential areas for further exploration and development.

2 Testing Methodology

2.1 Using an API Game for Simulating the PSPACE-Complete Problem

To simulate the PSPACE-complete problem, we utilized the gym-sokoban environment from the OpenAI Gym third-party environments. Sokoban, a well-known PSPACE-complete puzzle game, provides a challenging testbed for evaluating the capabilities of neuroevolutionary algorithms like NEAT. The game's complexity and need for strategic planning make it an ideal candidate for this study.

2.2 Devise a Baseline Model

As a baseline model, we selected several popular Reinforcement Learning (RL) algorithms known for their applicability to similar types of problems. These included:

- **Deep Q-Network (DQN)**: A widely-used algorithm that combines Q-learning with deep neural networks.
- **Proximal Policy Optimization (PPO)**: A robust RL algorithm that balances exploration and exploitation.
- **Optimized PPO (PPO opt)**: An enhanced version of PPO with improved hyperparameter tuning.
- **PPO with Convolutional Neural Networks (PPO CNN)**: A variant of PPO that utilizes CNNs for better feature extraction.
- **Q-learning**: A traditional RL algorithm used as a comparative benchmark.

These baseline models were implemented and trained to solve the Sokoban problem, providing a standard against which to measure NEAT's performance.

2.3 Reproduce the Original Paper Using the Original Library

We reproduced the methodology outlined in Stanley and Miikkulainen's original NEAT paper using the neat-python library. This involved implementing the NEAT algorithm with its default settings and configurations as described in the seminal paper. Reproducing the original work was crucial for validating our modifications and ensuring the reliability of our comparative analysis.

2.4 Exhaust as Many Variation Options as Possible

To thoroughly explore NEAT's potential, we tested a wide range of configuration variations:

- **Initial Number of Nodes**: Starting with different numbers of nodes to assess the impact on learning efficiency.
- **Fitness Function Adaptation**: Customizing the fitness calculation to reward box-pushing and penalize inactivity, aiming to improve convergence rates.
- **Population Size and Generations**: Experimenting with different population sizes and numbers of generations to optimize exploration and solution finding.
- **Mutation Rates**: Introducing random mutations at each step to maintain genetic diversity and prevent premature convergence.
- **Training Environments**: Using multiple random levels to ensure the diversity of training environments and improve generalization.
- **Network Types**: Testing FeedForward=False (RNN) to evaluate the impact of recurrent connections.

These variations were systematically tested to identify the most effective configurations for solving the Sokoban problem.

2.5 Compare and Discuss Results

The final step involved comparing the performance of NEAT against the baseline RL models. Key metrics included the number of iterations required to achieve success, the success rate, and the time per iteration. Our analysis focused on:

- **Customizing Fitness Functions**: Evaluating how tailored fitness functions influenced learning outcomes.
- **Species Diversity**: Assessing the role of species diversity and its impact on maintaining unique network structures and decision-making processes.
- **Network Complexity**: Investigating the effects of different network complexities on problem-solving capabilities.

The results were discussed in detail, highlighting the strengths and limitations of each approach. Our findings underscore the importance of fitness customization, species diversity, and careful management of network complexity in optimizing NEAT's performance for complex tasks like Sokoban. The comparative analysis provided insights into the broader applicability of NEAT and RL algorithms for solving PSPACE-complete problems, contributing to the ongoing research in neuroevolution and reinforcement learning.

3 Simulation problem definition

3.1 OpenAI Gymnasium for Zero-Player Games Simulation

OpenAI Gymnasium is a toolkit for developing and comparing reinforcement learning (RL) algorithms. It provides a standardized environment and a diverse collection of tasks (environments) designed to benchmark RL algorithms. Gymnasium is an evolution of the original OpenAI Gym, offering enhancements and a more extensive set of tools to facilitate RL research and development.

OpenAI Gymnasium enables the simulation of zero-player games, where the agent (algorithm) interacts with the environment without human intervention. This setup is crucial for testing and training RL algorithms, as it allows the agent to learn optimal strategies through trial and error within a simulated environment. By providing a consistent interface and a suite of environments, Gymnasium allows researchers to focus on algorithm development and performance evaluation.

OpenAI Gymnasium environments are pre-defined scenarios in which RL agents can be trained and tested. These environments range from simple tasks, such as balancing a pole on a cart (CartPole), to complex simulations like robotic control (Mujoco). Each environment comes with a set of rules and objectives, providing a controlled setting for evaluating the effectiveness of RL algorithms.

In addition to the official environments provided by OpenAI Gymnasium, there are numerous third-party environments created by the community. These third-party environments extend the capabilities of Gymnasium by introducing new challenges and scenarios, allowing for broader testing and application of RL algorithms. Examples include environments for robotics, strategy games, and classic video games.

3.2 The Complexity of Sokoban

Gym-Sokoban is a third-party environment in OpenAI Gymnasium, simulating the classic Sokoban puzzle game. Sokoban is a PSPACE-complete problem where the player (agent) pushes boxes to designated target locations within a grid. The complexity of the game arises from the need for strategic planning and optimal movement sequences to avoid irreversibly blocking boxes. Gym-Sokoban provides a challenging testbed for RL algorithms, requiring them to develop sophisticated strategies to solve the puzzles efficiently.

- **Objective:** Push all boxes to target locations.
- **Environment:** A grid-based puzzle with walls, boxes, and targets.
- **Challenges:** Avoiding deadlocks and planning optimal moves.

The Gym-Sokoban environment is particularly useful for testing the performance of RL algorithms on complex, strategic tasks that require a high degree of foresight and planning.

Sokoban has been extensively analyzed through the lens of computational complexity theory. Initially, it was demonstrated that the computational problem

of solving Sokoban puzzles is NP-hard [4],[5]. Subsequent research established that it is also PSPACE-complete [6],[7].

Computers face significant challenges in solving non-trivial Sokoban puzzles due to the high branching factor, which entails numerous legal moves at each step, and the considerable search depth required, involving many moves to find a solution [8]. Even relatively small puzzles can necessitate extended solutions [9].

Sokoban serves as an excellent testbed for the development and assessment of planning algorithms [10],[11]. The first recorded automated solver, Rolling Stone, was created at the University of Alberta. This solver's core methodologies have influenced many subsequent solvers, employing a traditional search algorithm augmented with domain-specific insights [12]. The Festival solver, utilizing the FESS algorithm, was the pioneer in solving all 90 puzzles of the commonly used XSokoban test suite [13]. Nonetheless, even the most advanced automated solvers struggle with many of the more complex puzzles that human solvers can resolve given sufficient time and effort [14].

In computational complexity theory, a decision problem is deemed PSPACE-complete if it can be solved using a memory amount that scales polynomially with the input length (polynomial space) and if every other problem solvable in polynomial space can be polynomially reduced to it. PSPACE-complete problems are considered the most challenging problems within PSPACE, the class of decision problems solvable within polynomial space, because solving any one of these problems would facilitate solving any other problem in PSPACE with relative ease.

Examples of PSPACE-complete problems include determining properties of regular expressions and context-sensitive grammars, verifying the truth of quantified Boolean formulas, identifying step-by-step transitions between solutions of combinatorial optimization problems, and solving various puzzles and games.

4 Baseline Solution - Reinforcement Learning

4.1 Principal Concepts of Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by taking actions in an environment to maximize cumulative rewards. Key components of RL include the agent, environment, states, actions, rewards, and policy.

- **Agent**: The learner or decision maker.
- **Environment**: Everything the agent interacts with.
- **State s_t at time t**: A representation of the current situation of the environment.
- **Action a_t at time t**: A set of all possible moves the agent can make.
- **Reward R_t at time t**: Feedback from the environment to evaluate the action taken.

– **Policy π_θ parameterized by θ**: The strategy that the agent employs to determine actions based on the current state.
– **Policy update process $\pi_\theta(s_t) \rightarrow \pi_\theta(s_{t+1})$ from state s_t to s_{t+1}**: The implemented change in the policy reflecting the previous states and rewards.

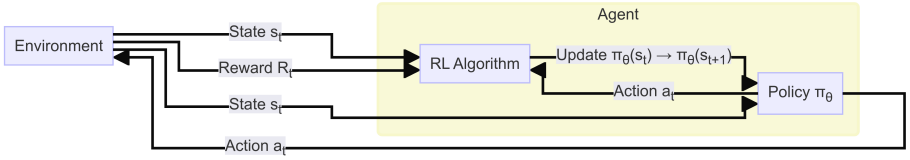


Fig. 1: Flowchart of the principal process of Reinforcement learning

Figure 1 represents a reinforcement learning (RL) process involving an environment and an agent. The environment E provides a reward R_t to the RL algorithm D. The environment E also provides the current state s_t to both the RL algorithm D and the policy π_θ in the agent. The policy π_θ determines an action a_t based on the current state s_t. This action a_t is executed in the environment E, leading to the next state s_{t+1}. The action a_t is also used by the RL algorithm D to update the policy. The RL algorithm D updates the policy π_θ from $\pi_\theta(s_t)$ to $\pi_\theta(s_{t+1})$. This update involves adjusting the parameters θ to improve future actions based on the reward R_t. The updated policy π_θ is used in subsequent interactions with the environment. This loop continues iteratively, refining the policy π_θ to maximize cumulative rewards. The RL algorithm can be any standard algorithm such as Q-learning, Deep Q-Network (DQN), or Proximal Policy Optimization (PPO).

4.2 RL Algorithms Used

Q-Learning is a value-based off-policy RL algorithm that aims to learn the value of the optimal policy independently of the agent's actions. It updates the Q-values (quality of actions) iteratively using the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_a Q(s',a') - Q(s,a)] \tag{1}$$

where α is the learning rate, γ is the discount factor, r is the reward, and s', a' are the next state and action.

Deep Q-Network (DQN) extends Q-Learning by using deep neural networks to approximate the Q-values. This allows it to handle high-dimensional state spaces. DQN employs experience replay and target networks to stabilize training.

Proximal Policy Optimization (PPO) is a policy gradient method that aims to balance exploration and exploitation. It optimizes a surrogate objective function while ensuring that the new policy is not too far from the old policy to maintain stable learning:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{2}$$

where $r_t(\theta)$ is the probability ratio and $\hat{A}_t$ is the advantage estimate.

PPO with Convolutional Neural Networks (PPO CNN) is a variant of PPO that incorporates CNNs to extract features from high-dimensional input spaces, such as images. This is particularly useful for tasks involving visual data, where CNNs can effectively capture spatial hierarchies in the input.

5 NeuroEvolution of Augmenting Topologies

5.1 Principal Concepts of NEAT

Neuroevolution (NE), the artificial evolution of neural networks using genetic algorithms [1], has been highly effective in reinforcement learning tasks, particularly those with hidden state information. This paper introduces NEAT (NeuroEvolution of Augmenting Topologies), which evolves both neural network topologies and weights. Neuroevolution (NE) has shown promise in reinforcement learning tasks by evolving artificial neural networks (ANNs). However, evolving both topology and weights can enhance performance, although it might complicate the search. NEAT addresses this by:

1. Using historical markings to line up genes for meaningful crossover,
2. Speciating to protect topological innovations,
3. Growing structures incrementally from minimal initial structures.

NEAT enhances neuroevolution by both optimizing and complexifying solutions incrementally. Each component (historical markings, speciation, minimal initial structure) is critical for efficient evolution, making NEAT a powerful approach for complex reinforcement learning tasks.

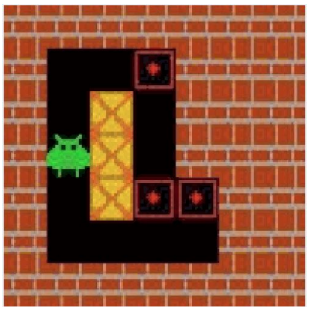
Genetic Encoding: Each genome in NEAT consists of a list of connection genes and node genes. Connection genes refer to two nodes, specify weights, enable bits, and innovation numbers, which help align genes during crossover. Mutations in NEAT can change connection weights or structures by adding connections or nodes. Figure 2 shows an example of genetic encoding.

Historical Markings: To perform crossover between diverse genomes, NEAT uses historical markings. Each new gene created by mutation is assigned a unique innovation number, which helps identify and match genes from different genomes. This avoids the problem of competing conventions and simplifies the crossover process.

$$\delta = c_1 \frac{E}{N} + c_2 \frac{D}{N} + c_3 \bar{W} \tag{3}$$

Where:

- E is the number of excess genes,
- D is the number of disjoint genes,
- $\bar{W}$ is the average weight difference of matching genes,
- N is the number of genes in the larger genome,
- c_1, c_2, c_3 are coefficients.



(a)

```
[[0 0 0 0 0 0 0]
 [0 1 1 2 0 0 0]
 [0 1 4 1 0 0 0]
 [0 5 4 1 0 0 0]
 [0 1 4 2 2 0 0]
 [0 1 1 1 1 0 0]
 [0 0 0 0 0 0 0]]
```

(b)

```
NEAT input layer:  [0 0 0 0 0 0 0 0 1 1 1 0 0 0 5 4 2 0 0 0 0 2 4 1 0 0 0 0 4 1 1 1 0 0 0 2
 1 1 1 0 0 0 0 0 0 0 0 0]
```

(c)

Fig. 2: Example of Genetic encoding - Game level (a), Level Encoding (b), Input Layer Encoding (c)

Protecting Innovation through Speciation: NEAT speciation protects structural innovations by allowing them to optimize without direct competition. Genomes are divided into species based on compatibility distance, ensuring diversity. Fitness sharing adjusts individual fitness by the number of species members, promoting niche preservation (see figure 3).

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(d(i,j))} \tag{4}$$

Where:

- f'_i is the adjusted fitness,
- f_i is the original fitness,
- $sh(d(i,j))$ is the sharing function,

– n is the number of individuals.

Minimizing Dimensionality: NEAT starts with a minimal structure (no hidden nodes) and grows only necessary structures through mutations. This reduces the search space dimensions, improving efficiency and avoiding unnecessary complexity.

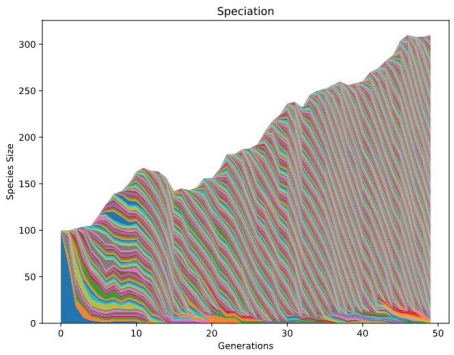


Fig. 3: Increasing Speciation

5.2 Flowchart of NEAT Algorithm

Figure 4 represents the complete flowchart of the NEAT algorithm:

– **Start with Minimal Network Structure**: Initialize each network in the population with the minimal structure, typically consisting only of input and output nodes without any hidden nodes.
– **Initialize Population**: Create an initial population of these minimal networks.
– **Evaluate Fitness**: Assess the performance of each network in the population using a fitness function specific to the task at hand.
– **Termination Check**: Determine if the termination criteria are met (e.g., a network achieves a desired fitness level or a maximum number of generations is reached). If yes, proceed to the best network found.
– **Speciation**: Divide the population into species based on network topological similarities to protect innovation.
– **Selection**: Select the best-performing networks within each species for reproduction based on their fitness.
– **Reproduction**: Generate offspring through reproduction. This involves either mutation or crossover.
– **Mutation Check**: Decide whether a mutation will occur. If yes, proceed to add a connection or node.

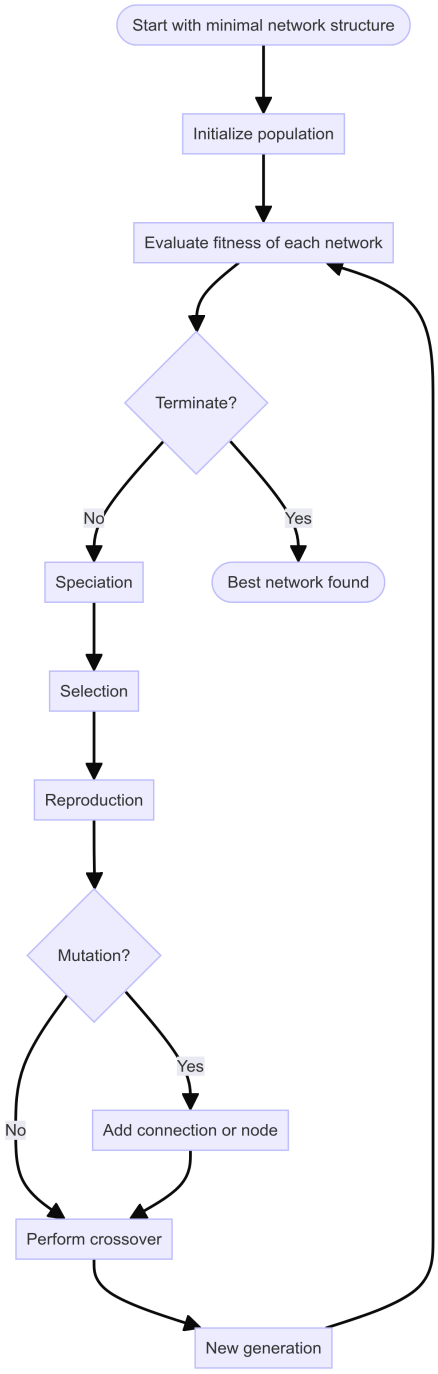


Fig. 4: Flowchart of the principal process of NEAT

- **Add Connection or Node**: Introduce structural mutations by adding a new connection between nodes or inserting a new node into an existing connection.
- **Perform Crossover**: If no mutation occurs, combine genomes from two parent networks to produce offspring.
- **New Generation**: Form a new generation of networks from the offspring.
- **Repeat Evaluation**: Evaluate the fitness of the new generation and repeat the process from the fitness evaluation step until the termination condition is met.
- **Best Network Found**: Once the termination condition is satisfied, the best network is identified as the result.

6 Implementation of NEAT

For the software implementation the Python NEAT library is used [3]. The main method for modifying the behaviour of the algorithm is through a configuration file for better reproducibility and easier automation.

6.1 Defining the Experiment Using a Configuration

Defining an experiment using the NEAT configuration file involves several key steps, each corresponding to a section of the file. The NEAT algorithm utilizes a variety of parameters to guide the evolution of neural networks. These parameters can be grouped into several categories:

Algorithm Parameters: These parameters define the overall behavior of the NEAT algorithm, such as the fitness criterion used for selection, the threshold at which evolution stops, the population size, and whether the population resets on extinction.

Genome Parameters: These settings govern the default properties of the genomes, including activation functions, aggregation functions, biases, compatibility, connections, nodes, network structure, response, and weights. For activation and aggregation functions, the default function, mutation rate, and available options are specified. Biases parameters relate to initialization, mutation, and replacement. Compatibility parameters include coefficients for disjoint genes and weight differences. Connection and node parameters define probabilities for adding or deleting connections and nodes. Network structure parameters include initial connection settings, feed-forward configuration, and the number of input, hidden, and output nodes. Response parameters specify initialization, mutation, and replacement for node responses. Weight parameters cover initialization, mutation, and replacement for connection weights.

Speciation Parameters: These parameters determine how genomes are grouped into species to maintain diversity within the population. This includes the compatibility threshold for considering genomes as part of the same species.

Stagnation Parameters: These parameters address the conditions under which species are considered stagnant and the criteria for maintaining or eliminating such species. This includes the method for calculating species fitness, the

maximum number of generations a species can stagnate, and the number of elite species preserved.

Reproduction Parameters: These parameters govern the reproductive process, including the number of elite individuals preserved each generation and the proportion of individuals that survive to the next generation.

6.2 Experimental Design

Each set of experiments is designed to investigate how different configurations of NEAT parameters impact the evolution of neural networks. The parameters vary in aspects such as connection management, node handling, mutation rates, and species management, all of which play critical roles in shaping the networks' ability to learn and solve tasks. By systematically varying these parameters, the study aims to uncover the most effective configurations for evolving robust and efficient neural networks. See table 1 for precise parameter values for each experiment.

Table 1: Configurations for NEAT experiments

parameter	Experimental set 1			Experimental set 2		
	config-1	config-2	config-3	config-4	config-5	config-6
fitness_threshold	10	10	10	7	7	7
pop_size	1000	500	200	500	500	500
activation_default	sigmoid	sigmoid	relu	sigmoid	sigmoid	sigmoid
activation_mutate_rate	0.05	0.1	0.1	0.03	0.03	0.03
aggregation_mutate_rate	0	0.1	0.1	0	0	0
bias_mutate_power	0.5	1	1	0.88	0.88	0.88
bias_mutate_rate	0.7	0.7	0.7	0.7	0.3	0.7
conn_add_prob	0.5	0.7	0.7	0.7	0.3	0.7
conn_delete_prob	0.5	0.2	0.2	0.4	0.15	0.4
enabled_mutate_rate	0.01	0.05	0.05	0.01	0.01	0.01
feed_forward	1	1	1	1	0	1
node_add_prob	0.3	0.5	0.5	0.5	0.3	0.6
node_delete_prob	0.15	0.2	0.2	0.3	0.15	0.3
num_hidden	1	2	0	0	0	0
response_mutate_power	0	1	1	0.88	0.88	0.88
response_mutate_rate	0	0.7	0.7	0.7	0.7	0.7
response_replace_rate	0	0.1	0.1	0.1	0.3	0.3
weight_mutate_power	0.5	1	1	0.88	0.88	0.88
weight_mutate_rate	0.8	0.8	0.8	0.8	0.4	0.8
weight_replace_rate	0.1	0.1	0.1	0.1	0.1	0.3
compatibility_threshold	3	2	2	2	2.5	2
max_stagnation	20	10	10	10	15	10
species_elitism	2	1	2	2	2	2
elitism	2	1	5	2	2	2

Experimental Set 1 In the first set, the experiments share common parameters that set a baseline for comparison. These parameters include the fitness threshold, mutation rates for biases and weights, and the structure of the neural network. These parameters are essential as they determine the basic setup of the neural networks, including how they evolve over generations and adapt to the given task.

Configuration 1 - This configuration uniquely emphasizes higher probabilities for connection deletion and lower mutation rates for enabling connections. It features a minimal hidden layer architecture, with specific settings for node addition and deletion rates, which are crucial for exploring the network's complexity. It also focuses on limited mutation power and rates for response parameters, with a higher compatibility threshold to encourage diversity.

Configuration 2 - Configuration 2 aligns closely with Configuration 3 but includes specific adjustments in the number of hidden nodes and elitism values. These settings are intended to balance the evolutionary pressure by controlling how many top-performing individuals are preserved each generation. The higher node addition probability and lower node deletion rate aim to incrementally increase the network's complexity.

Configuration 3 - Similar to Configuration 2, this configuration modifies the number of hidden nodes and adjusts the elitism values to preserve more top performers. It retains higher mutation power for responses and a moderate compatibility threshold to maintain species diversity, ensuring a robust exploration of possible solutions.

Experimental Set 2 The second set of experiments introduces different common parameters to create a new baseline for the experiments. This includes a lower fitness threshold, a fixed population size, and specific activation and aggregation mutation rates. These parameters ensure that the neural networks are initialized with consistent properties and evolve under controlled mutation rates, focusing on how activation and aggregation functions impact performance.

Configuration 5 - Configuration 5 distinguishes itself with a focus on lower connection deletion probabilities and a non-feed-forward architecture, which allows recurrent connections. This setup aims to explore how recurrent connections can influence learning and adaptation. It also features lower mutation rates for weights and specific settings for node and response mutations to test their impact on the network's adaptability and stability.

Configuration 4 - This configuration shares several parameters with Configuration 6 but stands out with its unique settings for node addition and response replacement rates. It emphasizes higher mutation rates for weights and a standard feed-forward structure, aiming to balance network complexity and mutation stability.

Configuration 6 - Configuration 6 modifies the node addition and response replacement rates to further explore their effects on network evolution. It retains the higher mutation rates for weights and a feed-forward structure, similar to

Configuration 4, ensuring a focused comparison on how these parameters influence the evolutionary dynamics.

7 Key Findings and Conclusions

7.1 Solvability of PSPACE-complete problem using NEAT

The NEAT algorithm was tested on the PSPACE-complete problem of solving the Sokoban puzzle. NEAT was able to successfully solve the Sokoban problem within 1000 iterations (20 generations), outperforming traditional reinforcement learning (RL) algorithms such as DQN, PPO, and their variants, underscoring its potential in solving complex PSPACE-complete problems without predefined network architectures. Table 2 presents the empirical results from all experiments, including the baseline models.

Table 2: Empirical results from all experiments

algo	variant	iters	success	time/iter
RL	DQN	5000	No	45
RL	PPO	5000	No	26
RL	PPO opt	5000	No	29
RL	PPO CNN	5000	No	27
RL	Q-learn	1000	Yes	23
NEAT	config 1	1000	Yes	11
NEAT	config 2	1000	No	14
NEAT	config 3	1000	No	13
NEAT	config 4	1000	No	15
NEAT	config 5	1000	No	16
NEAT	config 6	1000	No	17

Different configurations of NEAT were extensively tested to identify important factors of solvability. Configuration 1's success highlights the importance of higher probabilities for connection deletion, lower mutation rates for enabling connections, and maintaining minimal hidden layer architecture for effective problem-solving.

Traditional RL algorithms like DQN, PPO, PPO optimized, and PPO CNN did not achieve success in solving the Sokoban problem within 5000 iterations. Q-learning, a simpler RL variant, successfully solved the problem, suggesting its potential applicability alongside NEAT.

Fitness customization played a crucial role in improving NEAT's learning process, enabling the algorithm to reward strategic moves and penalize inactivity. Species diversity, ensured through NEAT's speciation mechanism, was vital in maintaining unique network structures and promoting innovative solutions. Managing network complexity by starting with minimal structures and allowing

incremental growth through mutations proved effective in navigating the solution space efficiently.

7.2 The Configuration Setups

Through various tests with different configuration setups, we found:

- **Customizing Fitness**: Customizing the fitness function improved the learning process.
- **Feedforward vs. Recurrent**: No significant difference was observed between feedforward and recurrent networks (feedforward=False) for Sokoban during the first 50 generations with a population of 500.
- **Species Influence**: The number of species greatly influences behavior:
  - Higher number of hidden nodes (and layers) proportionally increases the number of species.
  - More hidden nodes lead to greater diversity and a higher chance of improving fitness scores and finding optimal solutions.

7.3 The Software Implementation

The neat-python library's mechanisms provide critical insights into the functioning of NEAT:

- **Speciation**:
  - The population is divided into species based on genetic similarity.
  - Speciation protects innovation by preventing newly mutated genomes from competing directly with more mature ones.
  - Each species is assigned a fitness score, typically the average fitness of its members.
  - Speciation allows search to proceed in multiple spaces simultaneously.
  - Without speciation, structural innovations do not survive, and the population quickly converges on initially well-performing topologies.
- **Fitness Sharing**: Reduces the fitness of similar individuals within a species to encourage diversity and prevent any single individual from dominating.
- **Selection**:
  - Parents are selected based on their fitness, with fitter individuals having a higher selection probability.
  - **Stochastic Universal Sampling (SUS)**: Ensures a more even distribution of offspring among individuals according to their fitness.
  - Neat-python uses a replacement strategy where a portion of the least fit individuals are replaced by new offspring. If a species stagnates after a certain number of generations, it can be replaced.
- **Crossover**:
  - Neat-python uses a specialized crossover mechanism that aligns genes from both parents to preserve as much functionality as possible.
- **Mutation**:

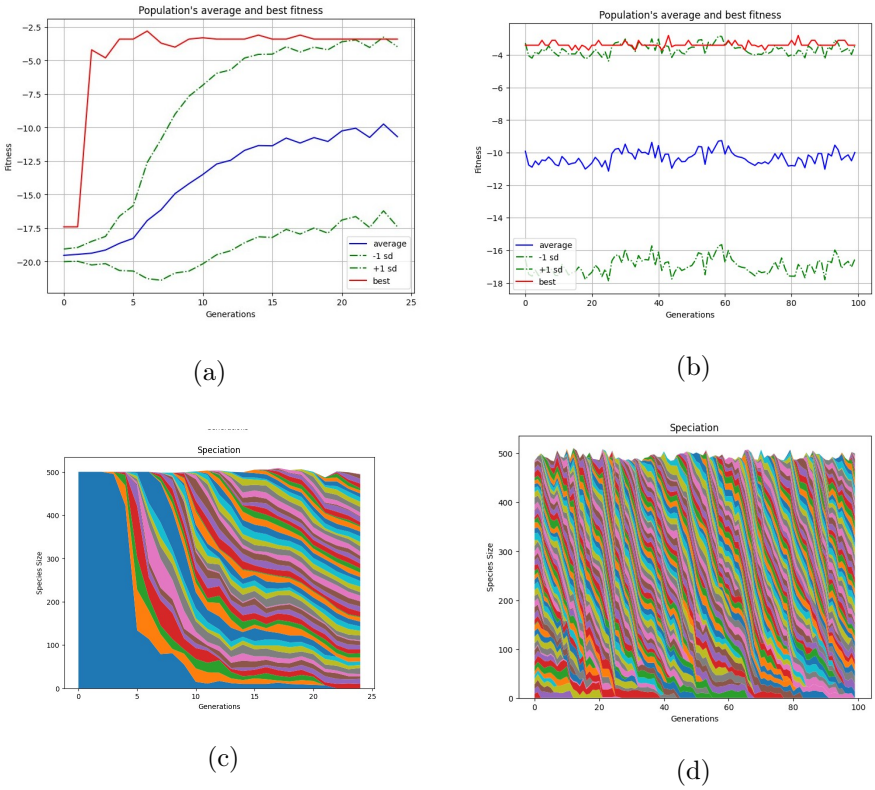


Fig. 5: Training results (a,b) vs. respective speciation (c,d)

- Direct influence on parameters such as weights, biases, number of nodes, and connections.
- Cannot directly influence actions taken at each step (this was additionally implemented by us).

– Technical limitations: Scaling the solution is limited to CPU optimizations due to the nature of the neat-python and gym-sokoban libraries.

7.4 Future Research

Future goals for this research include exploring custom neuro-evolutionary architecture search, which involves developing different genome structures and initialization methods. Additionally, improving the crossover mechanisms and initial hyperparameter optimization are key areas of focus. These enhancements aim to reduce the necessity of experimenting with different configuration files by optimizing these parameters from the outset. By advancing these aspects, the efficiency and effectiveness of neuro-evolutionary algorithms like NEAT can be significantly improved, facilitating more robust and adaptive solutions to complex problems.

Acknoledgements

This work was financially supported by the UNWE Research Programme.

References

1. Stanley, K.O., Miikkulainen, R.: Efficient evolution of neural network topologies. In: Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600), vol. 2, pp. 1757–1762. IEEE, Honolulu (2002). https://doi.org/10.1109/CEC.2002.1004508
2. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347 (2017). https://doi.org/10.48550/arXiv.1707.06347
3. McIntyre, A.: NEAT-Python 0.92 documentation (2019). CodeReclaimers, LLC. Accessed July 1, 2024. https://neat-python.readthedocs.io/
4. Fryers, M., Greene, M.: Sokoban. Eureka 54, 25–32 (1995)
5. Dor, D., Zwick, U.: SOKOBAN and other motion planning problems. Computational Geometry 13(4), 215–228 (1999). https://doi.org/10.1016/S0925-7721(99)00017-6
6. Culberson, J.C.: Sokoban is PSPACE-complete. Technical Report TR 97-02, Dept. of Computing Science, University of Alberta (1997)
7. Hearn, R.A.: Games, Puzzles, and Computation. PhD thesis, Massachusetts Institute of Technology, pp. 98–100 (2006)
8. Junghanns, A., Schaeffer, J.: Sokoban: Improving the Search with Relevance Cuts. Theoretical Computer Science 252(1-2), 5–19 (2001). https://doi.org/10.1016/S0304-3975(00)00080-3
9. Holland, D., Shoham, Y.: Theoretical analysis on Picokosmos 17. Archived from the original on 2016-06-07
10. Junghanns, A., Schaeffer, J.: Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. Theoretical Computer Science 252(1-2), 4–19 (1998)
11. Virkkala, T.: Solving Sokoban. MSc thesis, University of Helsinki, p. 1 (2011)
12. Junghanns, A., Schaeffer, J.: Sokoban: Enhancing general single-agent search methods using domain knowledge. Artificial Intelligence 129(1-2), 219–251 (2001). https://doi.org/10.1016/S0004-3702(01)00109-6
13. Shoham, Y., Schaeffer, J.: The FESS Algorithm: A Feature Based Approach to Single-Agent Search. In: 2020 IEEE Conference on Games (CoG), pp. 1–8. IEEE, Osaka (2020). https://doi.org/10.1109/CoG47356.2020.9231929
14. Damgaard, B. (2024, June 4). Open Test Suite - Numbers. Sokoban Solver Statistics. Retrieved from https://sokoban-solver-statistics.sourceforge.io/statistics/OpenTestSuite/Numbers.html