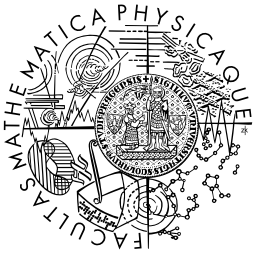


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Martin Mazáč

Hex Squared

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Matyáš Lorenc

Study programme: Computer Science - Artificial Intelligence (B0613A140006)

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

To everyone who ever said ”it’s just a game” — turns out, it’s a bit more complicated than that.

Title: Hex Squared

Author: Martin Mazáč

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Matyáš Lorenc, Department of Theoretical Computer Science and Mathematical Logic

Abstract: This thesis introduces Hex^2, a three-player variant of the board game Hex played on a hexagonal grid. Adding a third player opens up entirely new strategic possibilities and significantly increases the game's complexity. To explore effective ways of playing Hex Squared, several types of AI agents were designed and evaluated, ranging from simple heuristics and their adaptive variants based on evolutionary algorithms, to Monte Carlo Tree Search (MCTS) and its variant guided by a neural network. These agents were tested against each other in a series of matches. The results show that simple strategies are insufficient, and search-based methods, particularly MCTS enhanced with a neural network, perform best. This work presents the analysis of Hex^2 and offers insights into the potential of AI in multiplayer games, along with its web based implementation.

Keywords: Hex, Monte Carlo Tree Search, Self-Play, Reinforcement Learning, Neural Network

Název práce: Hex Squared

Autor: Martin Mazáč

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Matyáš Lorenc, Katedra teoretické informatiky a matematické logiky

Abstrakt: Tato práce představuje Hex^2, variantu deskové hry Hex pro tři hráče hranou na šestiúhelníkovém herním plánu. Přidání třetího hráče otevírá zcela nové strategické možnosti a výrazně zvyšuje celkovou složitost hry. Pro zkoumání efektivních způsobů hraní Hex^2 bylo vytvořeno a otestováno několik typů AI agentů: od jednoduchých heuristik či adaptivních založených na evoluci, přes Monte Carlo Tree Search (MCTS), až po MCTS řízené neuronovou sítí. Tito agenti byli navzájem porovnáváni v sérii zápasů. Výsledky ukazují, že jednoduché strategie nejsou dostatečně silné a nejlepší výkony podávají metody založené na prohledávání, zejména MCTS s podporou neuronové sítě. Práce přináší analýzu hry, její implementaci a nabízí shrnutí možností umělé inteligence ve hrách pro více hráčů.

Klíčová slova: Hex, Monte Carlo Tree Search, Self-Play, Zpětnovazební učení, Neuronová síť

Contents

Introduction

The game of Hex has long been a classic example of strategic depth emerging from simple rules and a widely used benchmark for artificial intelligence research. [1] Traditionally played by two players, Hex challenges AI agents to plan, adapt, and optimize strategies under deterministic and fully observable conditions. Yet, little attention has been given to extending this concept beyond two players, where the nature of strategy changes dramatically. Introducing a third player into Hex creates a fundamentally different environment - one that is more dynamic, less predictable, and substantially more complex.

This thesis explores Hex^2, a three-player variant of Hex played on a hexagonal board. Adding a third player not only increases the combinatorial complexity of the game but also introduces unique multiplayer dynamics. Players must constantly adjust their strategies, considering not only their own goals but also the evolving interactions between opponents. Unlike traditional Hex, where the focus lies on a direct competition, Hex^2 requires managing indirect influence and anticipating shifting alliances, making it a challenging domain for AI.

To approach this problem, we design and evaluate a variety of AI agents aimed at learning or discovering effective strategies for Hex^2. These range from simple heuristics to more sophisticated search-based methods such as Monte Carlo Tree Search [2], as well as agents that combine search with neural network guidance. [3] Through competitive experiments between these agents, we aim to assess which methods are capable of handling the game's complexity and whether classical AI techniques are sufficient, or need to be extended with learning-based approaches.

As part of this work, we also present a flexible and extensible software implementation of Hex^2 and the designed AI agents, creating a foundation for future research in multiplayer game AI. Since multiplayer environments remain underexplored in comparison to two-player settings, this work contributes to expansion of the understanding of AI strategies in games where multiple agents with conflicting interests interact simultaneously.

The following chapters introduce the classical Hex game and review relevant AI methods, formally define Hex^2 and its challenges, describe the implementation of the game and agents, present experimental results, and conclude with a discussion of key insights and possible future directions.

1 Background

We will start with an overview of the classic game of Hex, which will provide the foundation this thesis will extend and build upon.

After that we will go over possibilities of artificial intelligence in games.

1.1 Classic Hex

Hex is an abstract board game played on a rhombus shaped hexagonal grid as seen in Figure 2.1. Two players take turns placing stones of their color on the board, trying to connect the two opposite sides of the board with a continuous chain of stones of their color. First player able to do so wins.

This makes Hex a deterministic game, meaning the outcome is fully determined by the players' actions with no random elements involved. It is also a zero-sum game, where one player's gain is exactly balanced by the other player's loss. Furthermore, it is a game of perfect information, as all players have complete knowledge of all the previous moves and the current state of the board. Despite its simple rules and these nice properties, Hex represents a notoriously complex problem.

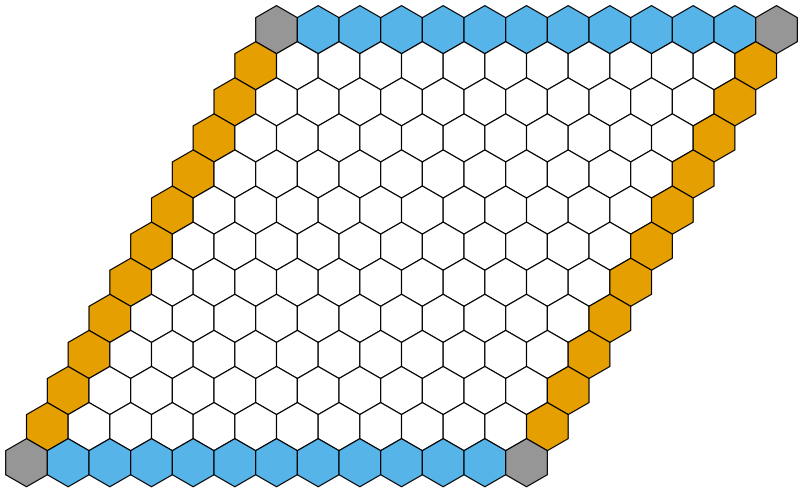


Figure 1.1 Standard size starting hex board.

1.1.1 Complexity

To better understand the complexity of solving Hex, we start by examining small board sizes for which the exact number of valid states is known. However, as the board size increases, computing this number precisely becomes practically infeasible. Therefore, it is necessary to employ methods that at least estimate the total number of possible board configurations.

A straightforward way to estimate this number, known as the *exponential estimate*, is based on the observation that each cell on the board can be in one of

three states: empty, occupied by the first player, or occupied by the second player. Thus, for an $n \times n$ board, there are approximately $\left\lceil \frac{3^{n^2}}{2} \right\rceil$ possible configurations.

For a tighter upper bound, John Tromp has shown [4] that the total number of unique board states U for an $n \times n$ board can be bounded using the following expression:

$$U = \frac{X + S}{2},$$

where

$$X = \sum_{i=0}^{n^2} \left[\binom{n^2}{\lceil \frac{i}{2} \rceil} \times \binom{n^2 - \lceil \frac{i}{2} \rceil}{\lfloor \frac{i}{2} \rfloor} \right], \tag{1.1}$$

and where

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{1.2}$$

Furthemore, if n is even, then S is given by

$$S = \sum_{i=0}^{n^2/4} \left[\binom{n^2/2}{i} \times \binom{n^2/2 - i}{i} \right], \tag{1.3}$$

else if n is odd, S is given by

$$S = 2 \times \sum_{i=0}^{\lfloor \frac{n^2}{4} \rfloor} \left[\binom{\lfloor \frac{n^2}{2} \rfloor}{i} \binom{\lfloor \frac{n^2}{2} \rfloor - i}{i} \right] + \sum_{i=0}^{\lfloor \frac{n^2}{4} \rfloor - 1} \left[\binom{\lfloor \frac{n^2}{2} \rfloor}{i} \binom{\lfloor \frac{n^2}{2} \rfloor - i}{i+1} \right] \tag{1.4}$$

Although this calculation may include some states that are invalid due to containing moves beyond the point where one player has already won, it gives us an upper bound that is tight enough to help us appreciate the approximate number of possible states. [1]

To establish a lower bound, we consider the number of valid board states with at most half of the cells filled, which were computed using brute force methods for boards up to size 8×8 by Hayward et al. [5].

Board Size	**Lower Bound**	**Valid states**	**Upper Bound**	**Exponential Estimate**
1x1	1	2	2	2
2x2	9	17	19	41
3x3	554	2,844	3,050	$9,842$
4x4	7.6×10^5	4,835,833	5,083,443	$21,523,361$
5x5	4.0×10^9	N/A	8.2×10^{10}	4.2×10^{11}
6x6	4.0×10^{14}	N/A	1.2×10^{16}	7.5×10^{16}
7x7	1.5×10^{20}	N/A	1.6×10^{22}	1.2×10^{23}
8x8	1.0×10^{27}	N/A	2.0×10^{29}	1.7×10^{30}
9x9	N/A	N/A	2.4×10^{37}	2.2×10^{38}
10x10	N/A	N/A	2.5×10^{46}	2.5×10^{47}
11x11	N/A	N/A	2.38×10^{56}	2.6×10^{57}
14x14	N/A	N/A	1.14×10^{92}	1.6×10^{93}

Table 1.1 Lower bound, valid states, upper bound, and exponential estimates for different board sizes.

As shown in Table 1.1, exact numbers of valid states become intractable for larger boards, and only bounds or estimates are available. These figures illustrate the exponential growth in complexity as board size increases.

The standard board size for Hex is 11×11, where the upper bound on the number of valid states is approximately 2.38×10^{56}. On a 14×14 board, this number grows dramatically to 1.14×10^{92}, underscoring the game's immense combinatorial complexity. This places Hex in a complexity class similar to Go [5].

Furthermore, in 1981, Stefan Reisch proved that Hex is PSPACE-complete, meaning it is at least as hard as the hardest problems in PSPACE [6].

1.1.2 Properties

Not long after the game was first introduced in 1942 by Piet Hein, several interesting properties were already known. One such characteristic is that the game may never end in a draw [1], therefore there is always exactly one of the two players able to connect his sides of the board.

Few years later in 1952 John Nash proved that the game is always won by the first player, given that both players play optimally. [7]

Such optimal policy has, however, been discovered only for smaller boards – an 8x8 board size of Hex was solved in 2009 using brute-force methods [4], and to this day it remains the largest solved board size – thus it is the larger boards that make for an interesting challenge.

1.2 Artificial intelligence in games

Artificial intelligence (AI) has revolutionized the way games are played and analyzed, providing tools and methods that allow computers to reach – and often surpass – human-level performance.

The following sections introduce key AI techniques used in games, including heuristics, evolutionary algorithms, reinforcement learning, neural networks, and search-based methods.

1.2.1 Heuristics

Imagine that we want to create an artificial player for a game. In every turn, this player needs to choose a move, ideally a good one. But figuring out which move is the best can be very difficult, especially in games with many possible moves and complicated strategies. One way to help the artificial player make decisions is to give it a simple rule that tells it which moves are better than others. Such a rule is called a heuristic.

A heuristic is a function that assigns a number to every possible move or position in the game. The higher the number, the better the move looks to the artificial player. The player then picks the move with the highest score.

For example, in a board game like Hex, a simple heuristic could say:

- Prefer moves that are close to the center of the board.
- Prefer moves that help connect your pieces together.

These are simple ideas that can guide the player to play reasonable moves, even without calculating everything that might happen in the future.

The advantage of heuristics is that they are fast and easy to use. Since the player does not try to think many moves ahead, it can quickly choose a move just by applying the heuristic.

However, heuristics also have limitations. Since they only look at the current position, and not at what might happen next, they can easily miss important things. For example, they might not see that a certain move leads to a win in two turns, or that an opponent can block them in the next move. Also, designing a good heuristic often requires human knowledge of the game, as we have to think about what makes a good move and write it down as a rule.

Even though heuristics are simple, they are often very useful. In many cases, they are also used as part of more advanced AI methods.

1.2.2 Combining heuristics with evolutionary algorithms

There are cases where we might want to use multiple heuristics, e.g. we might want to choose one based on the stage of the game, but how do we learn when to use which?

One possible solution is to use evolutionary algorithms to learn a probability distribution over the available heuristics. Instead of combining them into a single evaluation function, the final strategy samples one heuristic at each decision point, with probabilities given by an evolved weight vector. This vector represents a categorical distribution over heuristics, where each weight corresponds to the likelihood of selecting a particular heuristic during play. Evolutionary algorithms can optimize this distribution based on performance in self-play or against benchmark opponents [8].

Evolutionary algorithms are a family of black-box metaheuristic optimization algorithms. At its core, every evolutionary algorithm starts with a population of random solutions, repeatedly applying genetic operators until a stopping criterion is met. This proces is illustrated in Figure 1.2.

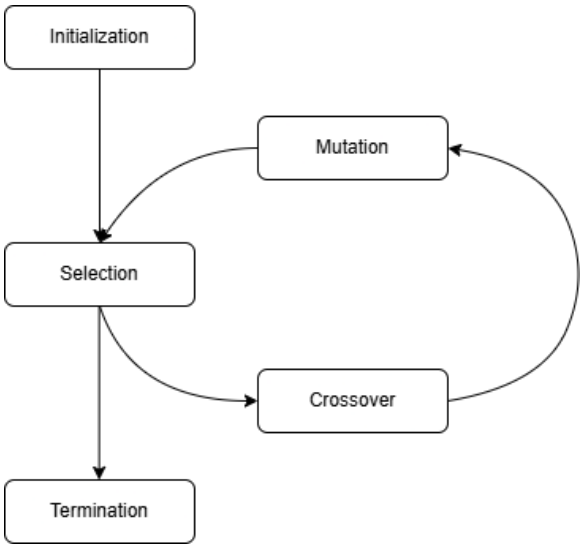


Figure 1.2 Flowchart of an evolutionary algorithm.

It is worth mentioning that the evolutionary algorithms rely on several hyperparameters, such as a population size, a mutation rate or a crossover rate. These need to be tuned for each problem separately, and finding the right values can be a challenging task. However, in general, we can say that increasing the mutation and crossover rates can lead to more exploration, while decreasing them encourages the algorithm to prefer exploitation. [8]

Selection

As the name suggests, this operator is used to select the best solutions from the population. However, in order to do that, we need to be capable of comparing two solutions, specifically, we need to be able to tell which one is better. This is done using a fitness function, which is a function that assigns a value to each solution based on how well it performs in the given task.

There are multiple ways to approach selection, but the most common ones are tournament and roulette wheel selections.

In tournament selection, a subset of the population is randomly chosen, and the best solution from this subset is selected. This process is repeated until the desired number of selected solutions is reached.

On the other hand, in roulette wheel selection, each individual is assigned a probability of being selected proportional to its fitness value. This allows us to steer the algorithm more towards exploration by adding a constant value to all the fitness values, lowering relative fitness differences. Contrary applying an exponential function to those same values broadens the differences between well and bad performing solutions, thus focusing on exploitation by selecting the currently best solutions.

The main advantage of the tournament selection is its ability to handle negative fitness values, which is not possible with roulette wheel selection, as negative probabilities would make no sense. Moreover, tournament selection is especially useful in games with multiple players, since it removes the need to define an explicit fitness function. Instead, individuals can simply compete against each other, and the winner of the match naturally emerges as the better solution

Crossover

This genetic operator is used to create a new individual by combining several existing ones. Implementation depends heavily on the problem at hand – for example, if our individual is a vector of real numbers representing a probability of using specific heuristic, we might use a simple uniform crossover, selecting each gene of the individual from one of the parents with equal probability.

Mutation

As evolutionary algorithm mimics the process of natural selection, we cannot forget about mutation, which adds randomness to the process by slightly modifying the individuals. It allows the algorithm to escape local optima, and explore the search space more thoroughly. In the simplest form it may be implemented as randomly changing a gene of a given individual with a given probability.

1.2.3 Neural Networks

Neural networks are computational models inspired by biological neural systems, particularly the human brain. Mathematically, a neural network can be viewed as a function $f_\theta : \mathbb{R}^d \to \mathbb{R}^k$ parameterized by a set of weights θ. These weights are the parameters of the function and are adjusted during the training process to minimize error. This function transforms an input x into an output through a series of transformations.

Artificial Neurons

A fundamental unit of a neural network is an *artificial neuron* which can be seen in Figure 1.3. The artificial neuron takes a set of input signals $x_1, x_2, \ldots, x_n$, each multiplied by a corresponding weight $w_1, w_2, \ldots, w_n$, sums them up, adds a bias b, and applies an activation function f. Formally, an output of the neuron is computed as follows:

$$\text{Output} = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{1.5}$$

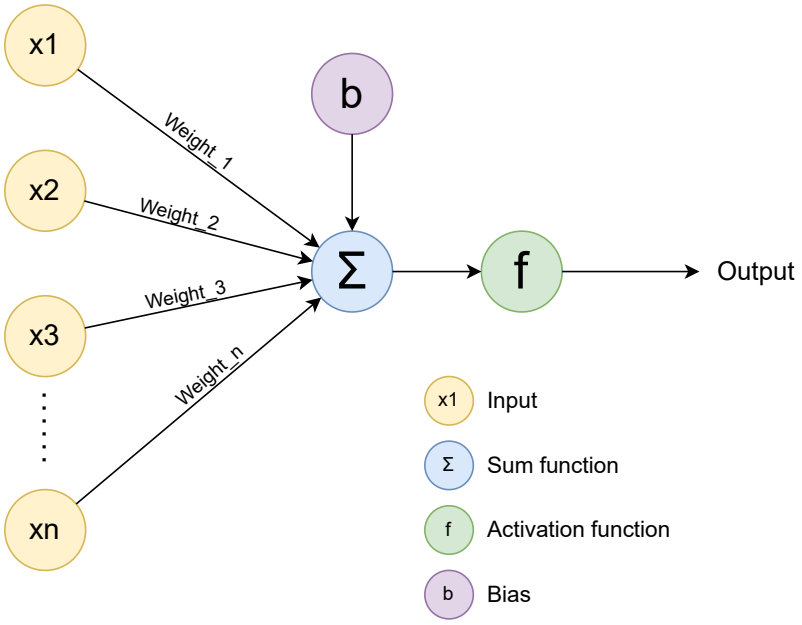


Figure 1.3 Artificial neuron.

Network Architecture

A typical neural network consists of multiple layers of artificial neurons, organized into an input layer, one or more hidden layers, and an output layer as seen in Figure 1.4. Each layer applies a transformation to the data, propagating information through the network. The entire process can be described as a composition of functions:

$$f_\theta(x) = (f_L \circ f_{L-1} \circ \cdots \circ f_1)(x), \tag{1.6}$$

where each f_i represents a layer in the network, and the composition of these layers defines the overall function implemented by the neural network.

Without a non-linear activation function, each neuron in a neural network would perform only a linear transformation. Consequently, stacking multiple layers of neurons would still result in a single linear transformation equivalent to a single-layer network, offering no additional representational power. Activation functions introduce non-linearity, enabling the network to approximate complex, non-linear functions and making deep architectures beneficial. [9]

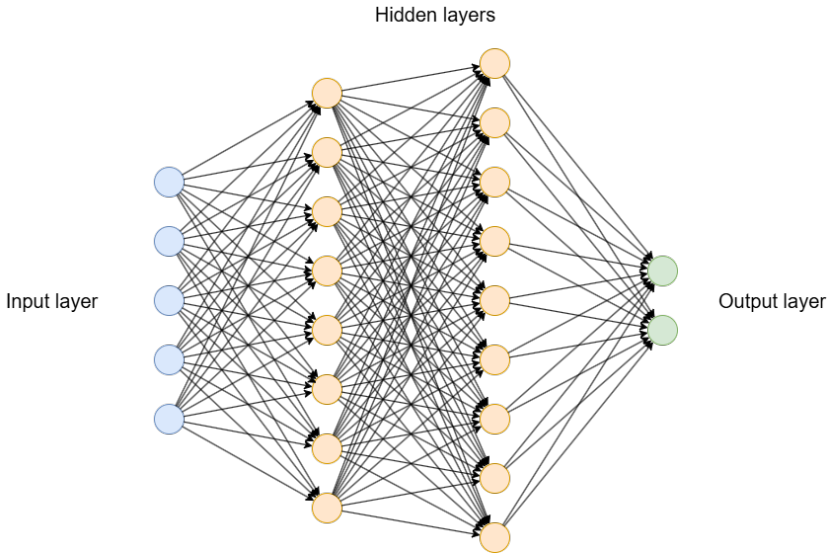


Figure 1.4 Fully connected neural network with two hidden layers.

Training Neural Networks

The parameters θ of a neural network can be learned, for example, through an optimization process known as a gradient descent. The goal is to minimize a loss function $L(f_\theta(x), y)$, which measures the difference between the network's predictions and the desired outputs. The weights are updated iteratively using:

$$w_{i+1} = w_i - \eta \nabla L(w_i), \tag{1.7}$$

where w_i represents the weight parameters at iteration i, η is the learning rate, and $\nabla L(w_i)$ is the gradient of the loss function. [9]

Network Topologies

A topology of a neural network refers to the specific arrangement and connection pattern of its neurons and layers. In other words, it describes how neurons are organized into layers and how these layers are interconnected. The choice of topology has a significant impact on the network's ability to solve particular types of problems.

At its simplest, a neural network can be seen as a series of layers stacked together, but different topologies introduce unique structural elements and connectivity patterns that make them suitable for specific tasks. Below, we introduce several selected important and widely used neural network topologies:

- **Fully Connected Networks (Dense Networks)**:
  Also known as multilayer perceptrons (MLPs), these networks consist of layers where each neuron is connected to every neuron in the preceding and succeeding layer. They are general-purpose models often used for structured data and problems where no spatial or temporal structure needs to be exploited.

- **Convolutional Neural Networks (CNNs)**:
  CNNs introduce convolutional layers that apply learnable filters to local regions of the input. This local connectivity and weight sharing make them particularly effective for image and spatial data, where local patterns such as edges or textures are crucial. CNNs are the backbone of most modern computer vision systems.

- **Residual Networks (ResNets)**:
  Residual networks extend traditional feedforward networks by incorporating skip connections, which bypass one or more layers. These connections allow gradients to flow more easily through deep networks, addressing some problems that may arise when training very deep architectures.

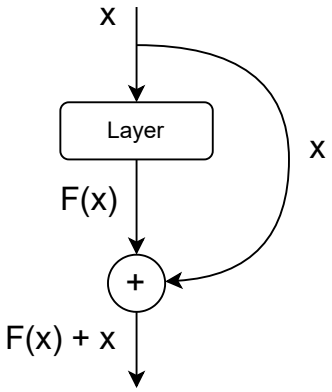


Figure 1.5 Residual Block in a neural network.

Each of these topologies represents a different approach to organizing and connecting neurons, and their design often reflects the nature of the problem they are intended to solve. In practice, these topologies are often combined to form more complex architectures that leverage the strengths of each.

Neural networks and games

Neural networks have become an essential tool in artificial intelligence for games. Their ability to approximate complex, non-linear functions makes them highly suitable for learning strategies, evaluating game states, and selecting actions in games with large search spaces and complicated dynamics.

They can be used in various roles depending on the game's needs. For example, instead of relying on handcrafted heuristic functions, a neural network can learn to evaluate the strength of a game state directly from data. This approach allows the AI to make more nuanced decisions, especially in situations where simple heuristics fall short. Neural networks can also represent policies, that is, mappings from game states to probabilities over possible actions. In this way, neural networks can serve as both evaluators of positions and decision-makers, providing a flexible alternative to static rule-based systems in complex game environments.

1.2.4 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm in which an agent learns to make decisions through interaction with an environment, guided by trial and error. By receiving feedback in the form of rewards or penalties based on its actions, the agent gradually learns to optimize its behavior over time. [10]

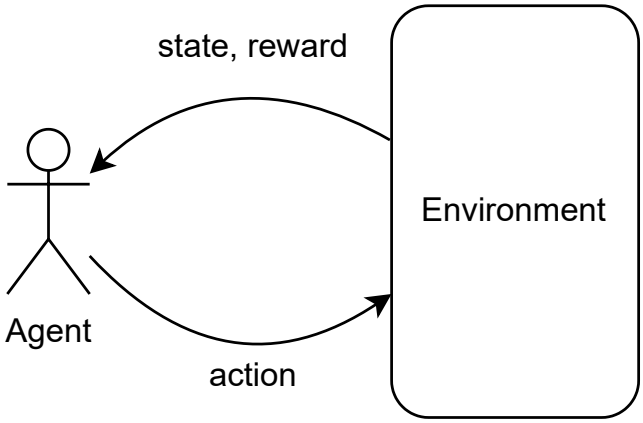


Figure 1.6 Reinforcement learning flowchart.

At the core of RL is the **Markov Decision Process** (MDP), formally defined by a tuple (S, A, P, R), where:

- S is a finite set of states of the environment.
- A is a finite set of actions.
- $P(s, a, s')$ is the transition function, defining the probability of transitioning to state s' given action a in state s.
- $R(s, a, s')$ is the reward function, specifying the immediate reward received when the agent transitions from s to s' after taking action a.

The transition function satisfies the Markov property, meaning the next state depends only on the current state and the action taken, not on the previous states or actions.

The agent's behavior is described by a policy $\pi : S \times A \to [0, 1]$, where $\pi(s, a)$ represents the probability of selecting action a in state s. The goal of reinforcement learning is to find an optimal policy π^* that maximizes the expected cumulative reward:

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \,\middle|\, \pi\right]$$

where $a_t = \pi(s_t)$ is the action taken by the agent at time t, and $\gamma \in (0, 1)$ is the discount factor ensuring convergence of the reward sum.

Reinforcement learning can also be applied in games; the agent learns to develop strategies that maximize its success within a set of rules and objectives. Games often provide a well-defined environment with clear states, actions, and rewards, making them ideal for applying RL methods. Through repeated interaction and feedback, the agent learns to optimize its decisions to achieve long-term goals, such as winning against an opponent or reaching a favorable outcome. This learning process allows the agent to discover complex and effective strategies without those being explicitly programmed.

Neural networks can also be trained using RL via interaction with an environment. Instead of minimizing a predefined loss function as in supervised learning, RL optimizes the reward signal that guides learning.

There are two primary roles a neural network can take in reinforcement learning. A policy network maps states directly to actions, thereby controlling the agent's behavior. Alternatively, a network can serve as a value function, estimating the expected return of a given state (or a state-action pair) to help guide decision-making.

One commonly used concept in reinforcement learning is the **Q-function**, $Q(s, a)$, which represents the expected cumulative reward obtained by taking action a in state s, and then following a particular policy. Learning an accurate approximation of the Q-function allows the agent to evaluate the long-term value of actions and choose those that maximize its expected reward.

The training process typically involves gradient-based optimization using algorithms such as:

- **Policy Gradient Methods** - directly optimize a policy by adjusting network parameters in the direction of higher expected rewards.
- **Q-learning with Deep Neural Networks** – approximates the Q-function to estimate state-action values and improve decision-making.
- **Actor-Critic Methods** - combine two components: the *actor*, which represents the policy and selects actions, and the *critic*, which estimates the value function to guide and improve the actor's learning.

Alternatively, **Neuroevolution** can be used to train neural networks by evolving their parameters and structure through evolutionary algorithms.

In all cases, the training follows a trial-and-error approach, using exploration strategies to balance between discovering new actions and exploiting known rewards.

1.2.5 Search-based algorithms

Search-based algorithms play a crucial role in decision-making by systematically exploring potential future game states to determine the optimal move in a given position, and can be broadly classified into two categories: uninformed and informed search algorithms.

At the core of these algorithms is the concept of a *game tree*, which is a tree-structured representation of all possible sequences of moves that can be played from a given starting position. Each node in the game tree represents a game state, while the edges correspond to the available actions or moves leading from one state to another. By traversing the game tree, search algorithms aim to identify the best sequence of moves according to a defined objective, such as maximizing the chance of winning or minimizing the opponent's options.

Uninformed search algorithms operate without any additional heuristics or domain-specific knowledge, requiring exhaustive exploration of the game tree. As a result, they can be computationally expensive.

In contrast, informed search algorithms leverage problem specific heuristic evaluations to guide the search process, often significantly improving efficiency and decision quality. [11]

Minimax

Let's start with the Minimax algorithm, which will also serve as a basis for other algorithms that build on top of it.

Minimax operates under the assumption that two players take turns making moves, with opposing goals: one player, called the *maximizing player*, aims to maximize the utility value (e.g., their chance of winning), while the other, the *minimizing player*, seeks to minimize this value, effectively working against the maximizing player. Both players are assumed to act rationally, always selecting the best available move according to their objective.

The algorithm systematically explores the game tree by evaluating all possible future game states. It traverses the tree down to terminal nodes, assigns utility values based on the game's outcome, and then propagates these values back up the tree. At each decision point, the maximizing player selects the move that yields the highest value, while the minimizing player picks the move that results in the lowest value, effectively modeling an optimal adversary. [12]

The time complexity of Minimax is $O(b^d)$, where b is the average branching factor (i.e., the number of legal moves per position), and d is the depth of the search tree. [13]

An interesting property of Minimax is that, in the case of two-player zero-sum games, the set of strategies it produces forms a Nash equilibrium [14], meaning that neither player can improve their outcome by unilaterally changing their strategy if the other player stays with theirs.

Based on the values propagated up the game tree, the algorithm ultimately selects the action corresponding to the child node with the best value for the player whose turn it is to move. Thus, Minimax not only determines the optimal value of the current game state but also prescribes the optimal action to take in that position.

An issue with Minimax is, however, that we have to search the whole game tree, which can often be computationally expensive and in many cases practically impossible. This can be at least partially dealt with by using Alpha-Beta Pruning.

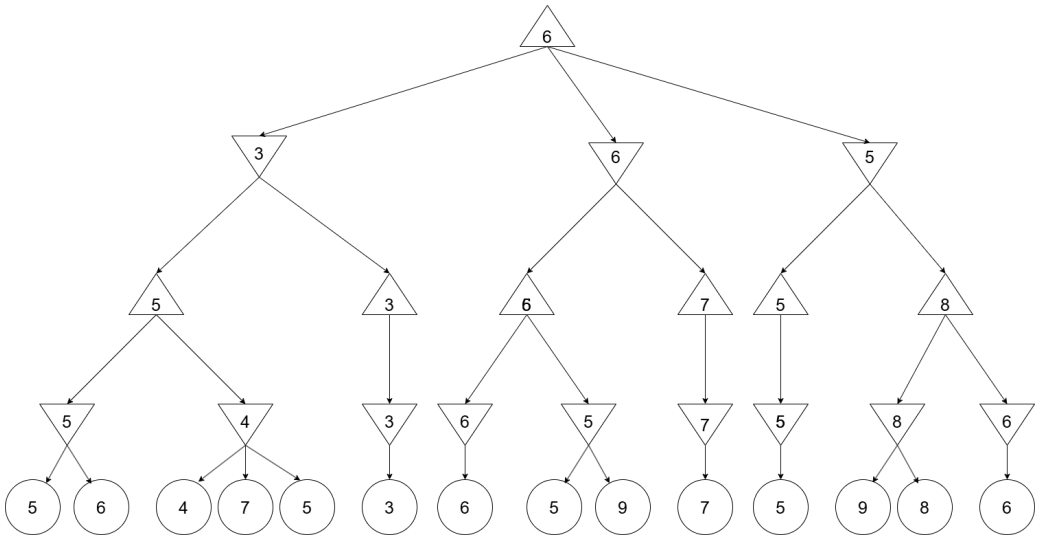


Figure 1.7 Two player game tree with values propagated using minimax.

Alpha-Beta Pruning

Alpha-Beta Pruning is an enhancement of Minimax that significantly reduces the number of nodes evaluated by eliminating branches that cannot influence the final decision. The algorithm introduces two parameters in each node:

- α: the best score the maximizing player can guarantee so far, which he would obtain by choosing an alternative action sometime earlier in the game
- β: the best score the minimizing player can guarantee so far, which he would obtain by choosing an alternative action sometime earlier in the game

As the tree is traversed, these bounds are continuously updated. The maximizing player attempts to increase α by finding higher-valued moves, whereas the minimizing player attempts to decrease β by finding lower-valued moves.

Pruning occurs when the algorithm detects that a subtree cannot possibly affect the final decision. This happens in either of the following cases:

- If the minimizing player finds a move with a value less than or equal to α, it means the maximizing player already has a better alternative available somewhere on the path to the root – i.e., earlier in the game – and would never allow the game to reach this state. Hence, the remaining branches under this node can be pruned, meaning this node with its subtree do not need to be further evaluated.
- Similarly, if the maximizing player encounters a move with a value greater than or equal to β, it means the minimizing player would prevent the game from reaching this point, and the branch can be discarded.

Thus, Alpha-Beta Pruning effectively eliminates parts of the game tree that cannot affect the final choice because, under rational play, neither player would select a move that worsens their own position.

This optimization can reduce the effective search complexity from $O(b^d)$ for minimax to $O(b^{d/2})$, allowing deeper searches within the same computational budget. [13]

However despite this improvement, many games are still too complex to be solved using Alpha-Beta Pruning, as the branching factor and the depth of the game tree are simply too big.

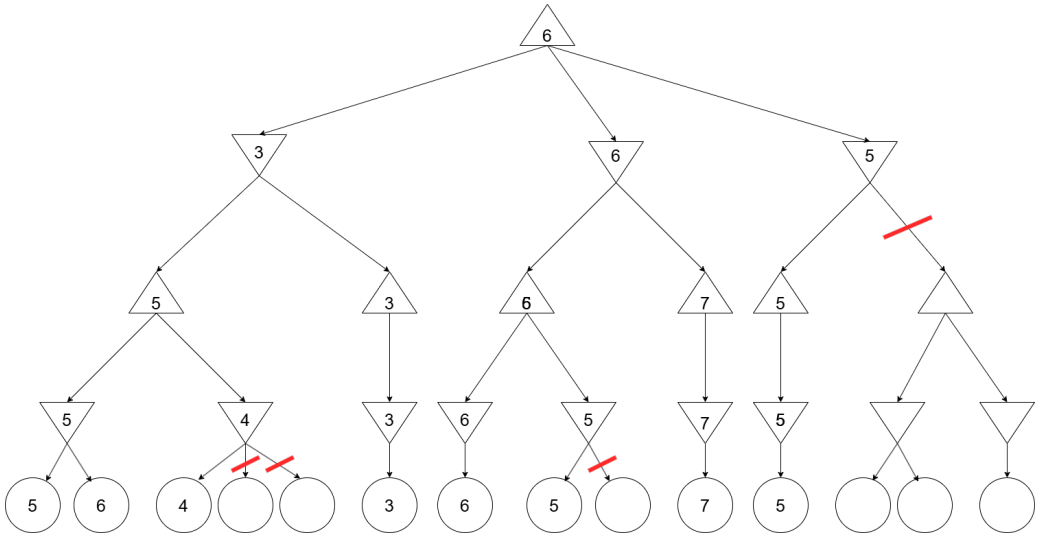


Figure 1.8 Alpha-Beta Pruning gets us to the same result faster.

This issue can be mitigated by limiting the search to a certain depth instead and using a heuristic evaluation function to estimate the value of the leaf nodes. Those values are then propagated back up the tree as before.

Such approach unfortunately introduces new problem, which is so-called horizon effect. [15] The calculations fall victim to this effect when a significant change occurs just over the depth cutoff. Quiescence search is one of the methods used to address this issue essentially by extending the search until a stable position is reached.

Heuristic alpha-beta tree search explores part of the game that is wide but shallow. In case of huge branching factors or insufficiently good evaluation function we might want to consider using Monte Carlo Tree Search instead, which searches part of the game that is narrow but deep.

Monte Carlo Tree Search

Unlike in the previous approaches, we will now be using the average utility over a number of simulations of complete games. Monte Carlo Tree Search (MCTS) constructs a search tree incrementally through a series of simulations, iterating over four main phases:

1. **Selection**
   Starting from the root node, the algorithm recursively selects child nodes according to a selection strategy that balances exploration and exploitation,

such as the Upper Confidence Bound (UCB1). This process continues until it reaches a node that is not fully expanded or a terminal node. A node is said to be *expanded* when all possible actions (child nodes) from the corresponding game state have already been added to the search tree. The UCB1 formula used for the node selection is given by:

$$UCB = Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \tag{1.8}$$

where $Q(s,a)$ is the empirical mean utility of taking action a in state s, $N(s)$ is the total visit count of state s, and $N(s,a)$ is the number of times action a was taken in state s. The constant C controls the balance between exploration and exploitation.

In simple terms exploitation refers to selecting the action we currently think would be the best, while exploration is all about selecting actions that may not look that promising as of now, but could potentially turn out to be, in fact, better in the long run.

But how exactly does this formula behave during a run of the algorithm? The resulting value is a sum of two terms, first term can be looked at as the exploitation term, and the second one as the exploration term. It is quite easy to see that due to the nature of logarithm, the exploration term will fade off over time, as the number of visits to the node increases, therefore as the algorithm progresses, it will start to exploit more and more.

2. **Expansion**
   If the selected node is not terminal and not fully expanded, one of its unvisited child nodes is added to the search tree. This new node represents a successor state resulting from an action that has not yet been explored. By gradually expanding new nodes, the algorithm incrementally grows the search tree, allowing it to explore new parts of the game space. The specific unvisited child is typically selected arbitrarily or uniformly at random from the remaining unexplored actions, as the goal at this stage is to ensure that all actions are eventually considered.

3. **Simulation**
   A rollout is performed from the newly added node until a terminal state is reached, providing an estimated outcome for that state. The simplest variant of the algorithm uses random rollouts, where game is completed with all players choosing their actions at random, but often domain-specific heuristics are utilized as they can significantly improve the performance of the algorithm.

4. **Backpropagation**
   The result of the simulation is propagated back through the selected nodes, updating their statistics, including visit counts and rewards. The values of the nodes are updated to reflect the outcomes of the simulated games, ensuring that promising actions are reinforced over time.

MCTS continues iterating these four phases until a computational budget is exhausted, such as a fixed number of iterations or time limit. The best move is typically chosen based on the highest visit count or highest average utility. [2]

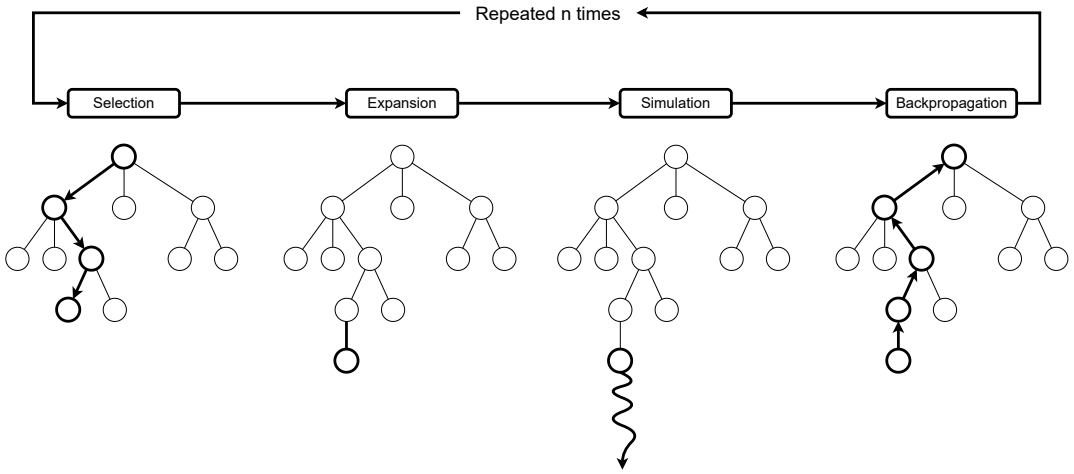


Figure 1.9 Monte Carlo Tree Search flowchart.

While Monte Carlo Tree Search is a powerful tool for decision-making in complex games, its performance can be significantly enhanced when combined with a neural network that provides value and policy estimates. This approach, used in *AlphaGo Zero* [3], allows the search to be guided more efficiently and is especially useful in games with vast branching factors, such as our three-player Hex variant.

Neural Network Guidance. The key idea is to train a neural network that, given a game state, predicts two crucial pieces of information:

- **Policy** - a probability distribution over possible moves, indicating which actions are likely to be strong.
- **Value** – an estimate of the expected outcome of the game from that state, typically represented as a number between -1 and 1, where 1 indicates a guaranteed win for the current player, -1 indicates a certain loss, and values closer to 0 reflect uncertain or evenly matched positions.

These predictions are then used within MCTS in two main ways:

1. The policy network's output biases the selection phase, preferring moves that are likely to be good.
2. The value estimate can replace costly rollouts during the simulation phase, allowing the search to focus on promising lines without having to simulate games to completion.

Training of neural network for MCTS

The first part is generating the training data. This is achieved using self-play, where many games are played between MCTS agents with simulation phase

replaced by the neural network. After each run of the MCTS where one move is selected, the board state and the calculated policy moves are stored. After the game is finished, the winner is determined, and a value of -1, 0, or 1 is assigned to each previously stored datapoint to indicate the game outcome from the perspective of the player who made the move. If the player associated with a given datapoint won the game, it receives a value of 1; if they lost, it receives -1; and in the case of a draw, a value of 0 is used.

The network is then taught on this data using supervised learning, with the policy head being trained to match the MCTS probabilities and the value head being trained to match the game outcome.

After, the MCTS agent with the new network fights itself with previous version of network and if the new one wins at least 55% of the time, it becomes the new best network, which is used to generate new data. Otherwise, more data is generated using the previous network and the process is repeated.

2 Hex Squared

Hex^2 is an extension of the classic game of Hex, introducing a third player and replacing the traditional rhombus-shaped hexagonal grid with a fully hexagonal one, whence the name Hex squared. While the objective of the game remains unchanged, these modifications bring significant strategic implications.

Players take turns in the following order: RGB - red, green, and blue and unlike classic Hex, Hex^2 can result in a draw, and the presence of two opponents introduces new tactical considerations and strategic possibilities. However, this also increases the game's complexity, as the branching factor at the start is three times higher than in classic Hex. This stems from the fact that the board can be conceptualized as three interlocking rhombi from the original Hex.

It is also worth noting that, as in classic Hex, larger boards tend to reduce volatility [1]. In the context of Hex^2, volatility refers to the degree to which early moves or tactical mistakes can dramatically influence the outcome of the game. On smaller boards, limited space forces players into direct confrontations early on, making each move potentially decisive. In contrast, larger boards offer more room for strategic maneuvering and recovery, allowing players to develop their positions more flexibly and reducing the impact of any single move. Thus, as the board size increases, the game becomes less dependent on sharp early tactics and more focused on long-term strategy.

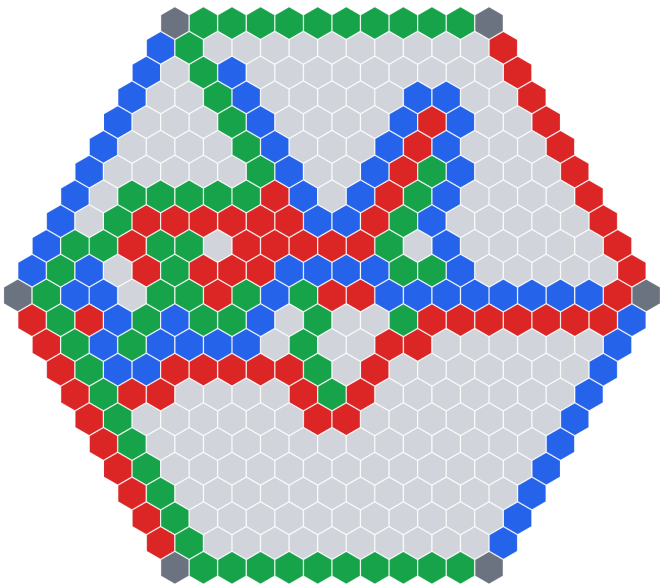


Figure 2.1 Hex^2 board, green player has won

2.1 Winning Strategy

Unlike in classic Hex, where the first player has a guaranteed win if they play optimally [7], we hypothetize that no such winning strategy exists in Hex^2. This can be viewed intuitively: Suppose the red player has just moved and is now one step away from connecting their two sides. Typically, there will be only one, or at most two possible moves to complete this connection. However, there exists a more complex scenario, which we will address later. In the typical situations, the remaining players always have the opportunity to block these final moves, thereby preventing the connection.

Importantly, the blocking move cannot result in a win for the blocking player. If the blocking player were truly just one move away from winning themselves, their imminent threat would have been apparent during their previous turn, prompting the other players to block them instead of allowing them to proceed unchallenged.

This mutual vigilance and ability to block ensure that no player can guarantee a win.

Additionally, it is worth noting that in Hex^2, achieving a draw is generally considered a better outcome than suffering an outright loss. When faced with an inevitable defeat, players aim for a draw as a last-resort strategy, effectively denying victory to the other participants.

Let us now analyze the two previously mentioned typical scenarios in detail.

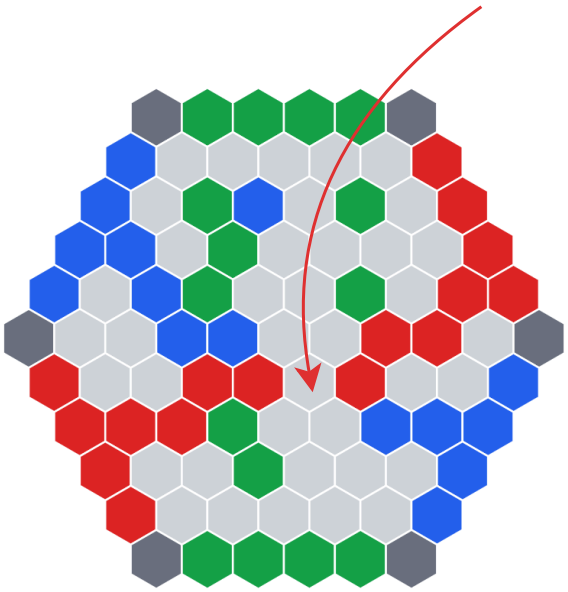


Figure 2.2 The red player has just made a move. Only one move is required to block the red player's victory.

The situation illustrated in Figure 2.2 demonstrates a critical moment where

the red player is one step away from winning. This implies that a bottleneck exists: there is only a single point through which the other two players can still connect their edges. Recognizing this, the green player, who is playing right after the red player, will immediately occupy this crucial hexagon to maintain their ability to connect their edges in the future. By doing so, the green player ensures they remain a viable competitor rather than leaving an opening for the red player to secure victory.

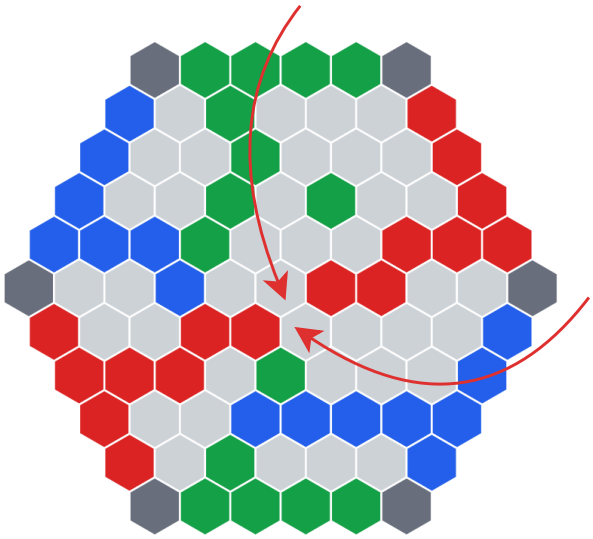


Figure 2.3 The red player has just made a move. Two moves are required to block the red player's victory.

In contrast, the second scenario illustrated in Figure 2.3 requires coordination between the remaining players to block red's victory. Since two blocking moves are necessary, the green player can claim one of the crucial tiles, leaving the responsibility of blocking the other tile to the blue player. Assuming rational play, the blue player will always block. This is because securing a draw is considered a better outcome than allowing red to win outright. In such cases, blocking serves as a last-resort strategy to deny victory to an opponent rather than conceding defeat.

These scenarios illustrate how Hex^2 often requires players to cooperate tactically against a common threat, even when victory is no longer a realistic possibility for them. Rational players will prioritize drawing over outright losing, as it allows them to deny an opponent's win and potentially prolong the game to their advantage. Consequently, blocking and coordination become essential elements of strategy, even if only to secure a draw.

Having examined the two most common scenarios where a player's imminent victory can be blocked by one or two moves, we now turn our attention to a third, more intricate situation.

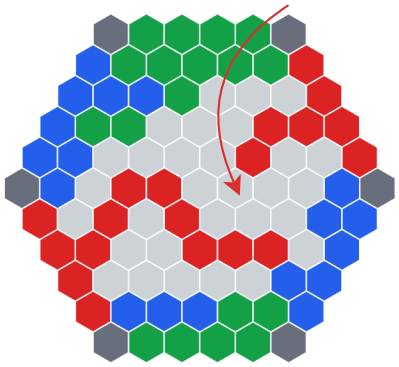


Figure 2.4 The red player has just made a move. Potentially three moves could be required to block the red player's victory after their next move.

If the red player plays their next move to the critical place indicated by the red arrow in Figure 2.4, they will be one step away from winning, there will be three separate points through which red can complete their connection. Even with coordinated effort, the other two players cannot block all three simultaneously. As shown in the figure, at most three distinct points can be simultaneously threatened as winning moves. This upper bound arises from the nature of the Hex^2 board and turn-based gameplay. Since a player can only place one stone per turn, they can only extend their connection by a limited amount and geometrically, there can be no more than three distinct positions through which a player can complete their path in the next round.

Attempting to threaten a fourth such point would imply that the player already had multiple winning paths available before their previous move that could be secured in the next round – effectively meaning they were already on the verge of winning. But in that case, the opponents (assuming rational play) would have recognized and blocked those threats in the previous round. Thus, any situation in which more than three simultaneous winning threats appear must have evolved from a position that was already winning and already should have triggered a response.

To identify such critical spots, any player can examine all shortest paths connecting the edges possibly using the already occupied positions (which are not counted towards the length of the path) for both the opponents. If the shortest path have length 2, the player then checks whether there are three such shortest paths that share a single common hexagon. If such a hexagon exists, it is the critical point that must be secured. While multiple critical threats can occur throughout the game, at most one such threat can be active in any given round, since a single move cannot establish multiple three-way threats.

When players recognize this critical spot, they will prioritize occupying it for two main reasons: first, it immediately blocks the opponent's winning attempt; second, given that the game has reached a state where this spot is critical, it indicates that the opponents's edges are already close to being connected – occupying the spot thus offers the best opportunity to break through the bottleneck and still keep fighting for a win.

2.2 Complexity

Having explored the complexity of classic two-player Hex, a natural extension is to examine how the introduction of a third player affects the size of the game's state space in Hex^2. With three players taking alternating turns on a hexagonal board, the branching factor increases dramatically—especially in the early stages—leading to a significant escalation in overall complexity.

To estimate the size of the state space, we begin with the formula for the number of hexagons in a regular hexagonal tiling of edge length a, which we will use throughout this section:

$$h(a) = 3a^2 - 3a + 1$$

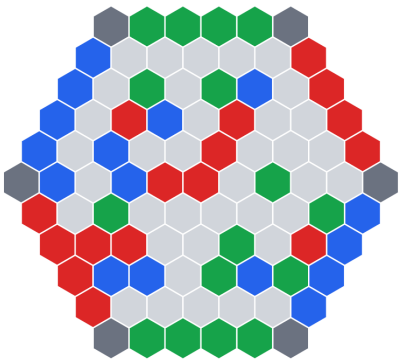


Figure 2.5 A Hex^2 board of edge length 6.

Furthermore, for a given Hex^2 board, which is a regular hexagon, let n refer to the length of a full edge of this hexagon, which consists of all the preoccupied positions along the side plus the two neutral positions in both adjacent corners, one at each end, where the edge connects to adjacent sides. For illustration, the game board presented in figure 2.5 is a board of edge length 6.

For a Hex^2 board with edge length n, the number of initially unoccupied interior cells is equal to $h(n-1)$, which we will, for simplicity, denote as h for a given board.

2.2.1 Upper Bound

Given that each hexagon on the board that is unoccupied at the beginning of the game can be in one of four possible states – occupied by one of the three players or unoccupied – the upper bound on the number of possible board configurations can be calculated as:

$$U = 4^h$$

This calculation provides a rough estimate of the state space size, which grows exponentially with board size. For instance, on a board of edge length $n = 8$, we have:

$$h = h(7) = 3(7)^2 - 3(7) + 1 = 127$$

Thus, the upper bound is:

$$4^{127} \approx 2.8 \times 10^{76}$$

For a board of edge length $n = 11$, we similarly compute:

$$h = h(10) = 3(10)^2 - 3(10) + 1 = 271$$

and the corresponding upper bound becomes:

$$4^{271} \approx 1.4 \times 10^{163}$$

2.2.2 Lower bound

In classic Hex, a lower bound for complexity was calculated by brute-force enumeration of partially filled board states. For Hex^2, we adopt a different approach: estimating the number of minimal-length winning paths for a single player and then extrapolating the number of compatible board configurations involving moves from all players.

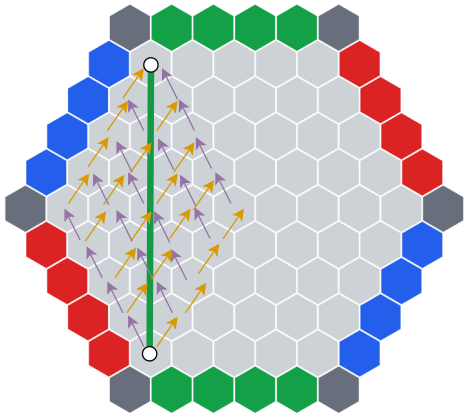


Figure 2.6 A minimal-length connection between two opposite endpoints. The arrows represent all the possible minimal connections between two fixed endpoints.

For a game board of edge length n, we define the shortest possible winning path for a player as the minimal connection between their two opposing edges of

the board. Then, the length p of the shortest path connecting two opposite edges is given by:

$$p = 2n - 3$$

Consider a minimal-length winning path that starts and ends at two fixed, strictly opposing points adjacent to the player's edges, as is shown in Figure 2.6. The path must have a total length of p, meaning it consists of $p - 1$ actual steps, i.e., transitions between consecutive points of the path.

To create such a path, the player must alternate between two directions on the hexagonal grid — rightward and leftward — in a balanced way. In the case of a symmetric shortest path, exactly half of the steps (i.e., $\frac{p-1}{2}$) go in one direction, and the remaining half go in the other.

Since we need to choose which of the $p-1$ total steps are taken in the rightward direction (with the leftward steps then automatically determined), the number of distinct such paths is given by the binomial coefficient:

$$\binom{p-1}{\frac{p-1}{2}}$$

We define the boundary line as the sequence of hexagons that spans one side of the board as highlighted in Figure 2.7. Let e be the number of hexagons along a boundary line, which is given by:

$$e = n - 1$$

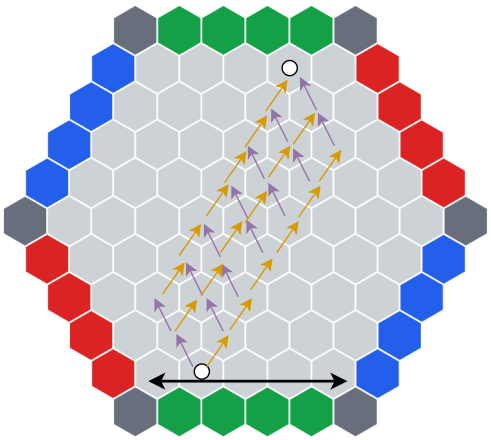


Figure 2.7 Hexagonal grid with a highlighted boundary line, showcasing different shortest paths between two specific endpoints.

Now suppose we fix the starting point of a path to the leftmost point of the bottom boundary line, but allow the endpoint to vary along the boundary line of the opposing edge. Each step that shifts the endpoint further along that edge effectively replaces a leftward move with a rightward one. In the context of

shortest paths on a hexagonal grid, this change increases the number of rightward moves by one for each unit shift.

Because each shortest path consists of a specific number of left and right moves, changing that balance alters how many such paths exist. For a fixed start and a shifted endpoint (by $i-1$ positions), the number of corresponding shortest paths is given by:

$$\binom{p-1}{\frac{p-1}{2}+i-1}$$

Summing this over all possible endpoint positions, we obtain the total number of minimal-length connections between a fixed start point and all endpoints on the opposing boundary:

$$\sum_{i=1}^{e}\binom{p-1}{\frac{p-1}{2}+i-1}$$

To capture all the shortest paths between all the possible starting and ending hexagons, we sum over all combinations. This is analogous to the previous case with a fixed starting point, where we summed over all possible endpoints; here, we additionally sum over all starting positions along the corresponding boundary line.

$$P=\sum_{k=1}^{e}\sum_{i=1}^{e}\binom{p-1}{\frac{p-1}{2}+i-k}$$

Next, we estimate the number of valid configurations where a player achieves a minimal-length winning path while the other two players fill the remaining board without interference. Let the board contain h initially unclaimed hexagons. Suppose the winning player wins using the shortest possible path. Then this player uses p of them. We assume both the other players place exactly $p-1$ non-interfering stones without overlapping any previously claimed tiles.

The number of ways to distribute these moves without interference is:

- $\binom{h-p}{p-1}$: Ways for the second player to place their stones.
- $\binom{h-2p+1}{p-1}$: Ways for the third player to place their stones.

The total number of valid configurations where one player wins using a minimal-length path while the other two players occupy parts of the remaining board without interference is then estimated from below as:

$$\mathrm{L}=3\cdot\sum_{k=1}^{e}\sum_{i=1}^{e}\binom{p-1}{\frac{p-1}{2}+i-k}\cdot\binom{h-p}{p-1}\cdot\binom{h-2p+1}{p-1}$$

To put this into perspective, on a board of edge length $n=8$, the estimated lower bound for Hex^2 is around 1.2×10^{32}, while for classic Hex, the estimate is 1.0×10^{27}. For a board of edge length $n=11$, the estimated lower bound for Hex^2 grows to approximately 2.9×10^{46}, illustrating the exponential increase in complexity driven by both board size and the introduction of a third player.

3 Implemented AI

This section describes the AI agents implemented for Hex[2].

3.1 Heuristic-Based Agents

The simplest AI agents in the game rely on heuristic evaluation to make decisions.

3.1.1 Random Agent

The most basic AI agent selects a move at random from the list of all available moves. Although it does not exhibit rational behavior, it serves as a useful baseline for performance comparison.

3.1.2 Center-Control Agent

This agent prioritizes moves that claim unoccupied hexagons near the center of the board, under the assumption that central control provides a strategic advantage as is the case in classic Hex.

3.1.3 Edge-Control Agent

In contrast to the center-control agent, this AI favors moves that occupy hexagons as far from the center as possible.

3.1.4 Path-Finding Agent

Using search, finds the shortest path connecting the edges of the board. From this path, it selects free cell closest to the halfway point of the path. Towards the end of the game it may not be possible for such path to exist, so the agent will fall back to random move in such case.

3.2 Adaptive Heuristics with Evolution

The previously discussed heuristic agents operate independently with fixed strategies, but different heuristics may be more effective at different points in the game. Instead of adhering to a single heuristic throughout, an AI agent could dynamically adjust its strategy as the game progresses.

To enable this adaptability, we introduce an evolutionary approach that allows the AI to select the most suitable heuristic based on the current board state. This method leverages evolutionary techniques to refine decision-making over time.

Two distinct types of individuals were evolved, both achieving the same performance. The first type used a simple vector of probabilities, from which a strategy was sampled. The second type allowed for learning different probabilities at various stages of the game. Here, each individual was represented by a simple neural network that processed the board state and produced a probability distribution

over the available heuristics. Selection was performed by grouping individuals into sets of three, having them compete against each other over multiple rounds. The individual with the highest number of wins advanced to the next generation. Mutation was applied by adding random noise from a normal distribution to the network's weights. No crossover was used in the final implementation.

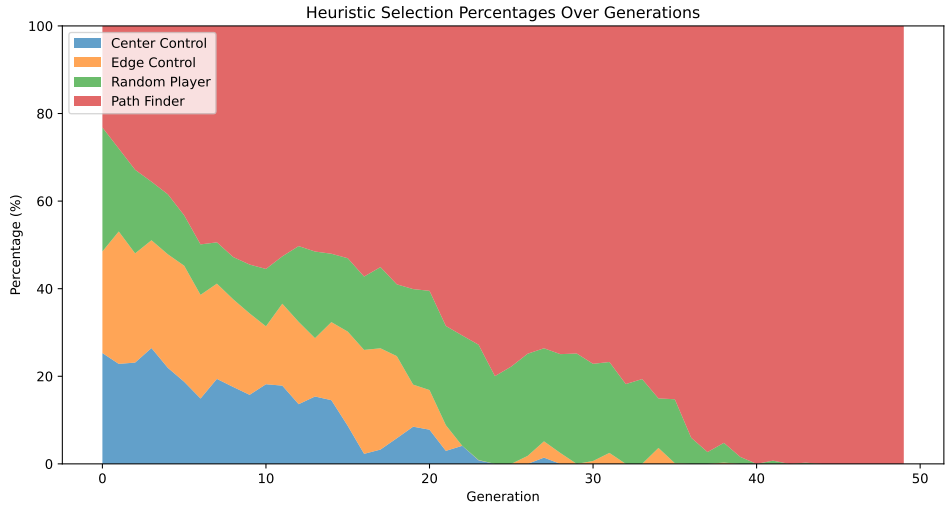


Figure 3.1 Heuristic selection statistics over time for the evolutionary approach using a neural network to choose heuristics based on board state, indicating the percentage of times each heuristic was chosen.

While it's not hard to predict that the random agent strategy would be outperformed at most stages of the game, evolution has demonstrated that all heuristics are consistently dominated by the path-finding agent. Regardless of the game stage, it provides the most effective choice of cells.

Therefore, it makes no sense to employ any mix of heuristics, as the pure path-finding agent is always the stronger choice.

3.3 Monte Carlo Tree Search

The majority of AI agents in Hex^2 revolves around MCTS and its variations, as it is widely regarded as the state-of-the-art algorithm for decision-making in games with large branching factors.

Several variations of MCTS have been implemented in Hex^2 to enhance its performance. These modifications focus on improving the simulation phase, which is crucial for determining state values and guiding the search more efficiently.

3.3.1 MCTS with a random rollout

Uses a random rollout strategy to simulate the game from the current state.

3.3.2 MCTS with a heuristic rollout

Employs a heuristic evaluation function from the path-finding agent to guide the simulation phase, providing a more informed estimate of the state value.

3.3.3 MCTS with a neural network

As introduced in Section 1.2.5, the neural network provides both policy and value estimates for each state. The policy guides move selection during the search, prioritizing more promising moves during the expansion phase. The value estimate replaces the simulation phase – rather than running random playouts to the end of the game, the network directly predicts the expected outcome from the current state. These predictions are then used to update the tree through backpropagation. More details on the neural network architecture can be found in Section 3.5.

3.3.4 MCTS with a neural network v2

Since the neural network and the classic MCTS excel at different stages of the game, this agent uses a combination of both. In the early game, where the branching factor is high and brute-force search is ineffective, the neural network guides MCTS by providing policy and value estimates. As the board fills up and the game tree becomes smaller and more tractable, the agent switches to classic MCTS with random rollouts, which performs better in the late game.

It is important to note that this agent uses the exact same neural network as its first version, sharing the identical architecture and weights. The only difference lies in the later stages of the game, where the neural network's value prediction is no longer used, and the simulation phase falls back to random rollouts.

3.4 Stand-alone neural network

Since the network used in combination with MCTS gives us value estimate as well as probability a of winning over all possible moves, it can be used without the MCTS as well, simply by selecting the move which is most likely to yield a win according to the network.

3.5 Neural Network Architecture

Our network is inspired by the architecture of AlphaGo Zero [3], with the key difference that it operates only on the current game state, without incorporating historical positions.

Training was conducted over 4–6 self-play cycles. In each cycle, MCTS agents guided by the current network played against each other. A new network was accepted if it won at least 55% of games against the previous version. Each iteration included several thousand games.

3.5.1 Architecture Details

The input to the network consists of three channels representing the board state – one for each player. The network processes the input through the following structure:

- **Initial block:** A single convolutional layer with 256 filters of size 3×3, followed by batch normalization and ReLU activation.

- **Residual tower:** Five residual blocks, each with:
  - Two convolutional layers with 256 filters.
  - Batch normalization and ReLU after each convolution.
- **Policy head:**
  - A 1×1 convolutional layer with 2 filters, batch normalization, and ReLU.
  - A fully connected layer to a vector of size b^2, where b is the board width.
  - Softmax activation to obtain a valid probability distribution over all moves.
- **Value head:**
  - A 1×1 convolutional layer with 1 filter, batch normalization, and ReLU.
  - A fully connected layer with 256 units and ReLU.
  - A final linear layer with a tanh activation, outputting a value in $[-1, 1]$.

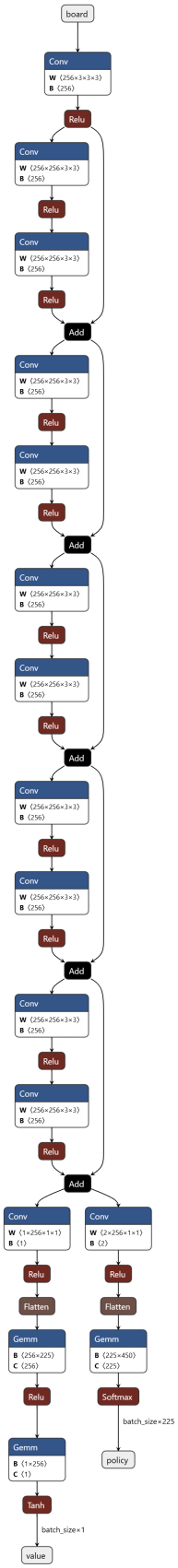


Figure 3.2 Neural network architecture inspired by AlphaGo Zero.

4 Experiments

This chapter presents the results of a series of experiments conducted to evaluate the performance of various AI agents in Hex^2. The goal was to compare different approaches to decision-making in this novel three-player game, ranging from basic heuristics to more advanced search-based methods.

The experiments are structured in a progressive fashion: we begin by assessing the effectiveness of simple heuristic-based agents. These are followed by matchups between heuristics and a standalone neural network, providing insight into the learning capabilities of the trained models. Next, we examine different Monte Carlo Tree Search (MCTS) variants–both standalone and combined with heuristics or neural networks. Finally, we conclude with a comprehensive comparison of the strongest agents from each category.

All matches were played on a hexagonal board with edge length 8 which corresponds to the medium-sized variant of Hex^2 and each match configuration was evaluated over multiple games to reduce randomness and improve result reliability. Due to the three-player nature of Hex^2, draws were common and are explicitly included in the summaries.

4.1 Heuristics

In this subsection, we evaluate the performance of the implemented heuristic-based agents. Each matchup involves two competing heuristics, while the third player is always the Random agent. This setup allows us to focus on direct comparisons between the two heuristics under test, as the Random player introduces minimal interference

PathF	**EdgeC**	**CenterC**	**Random**	**Draws**
98	-	-	0	2
44	14	-	-	42
98	-	0	-	2
-	4	-	2	94
-	-	0	1	99
-	61	0	-	39

Table 4.1 Heuristic tournament summary. Each row represents a match configuration, showing the number of wins for each player and the number of games that ended in a draw. The third random player is implicit and not included in the table.

Table 4.1 shows the matchups between different heuristic agents. The PathFinderHeuristic clearly dominates in all its pairings, often winning nearly all games. EdgeControlHeuristic performs moderately well, while CenterControlHeuristic appears ineffective, registering no wins. The high number of draws in several configurations highlights the balance or indecisiveness between weaker strategies.

Player	Total Wins
PathFinderHeuristic	240
EdgeControlHeuristic	79
CenterControlHeuristic	0
RandomPlayer	3
Total Draws	273

Table 4.2 Overall heuristic tournament summary.

Summarizing the heuristic-only tournament, PathFinderHeuristic is the standout performer with 240 wins. EdgeControlHeuristic follows distantly with 79, while CenterControlHeuristic is completely ineffective. The RandomPlayer manages only 3 wins, confirming its inadequacy. The large number of draws (273) illustrates the complexity and possibility of stalemates in Hex[2].

4.2 Strongest heuristic vs. NN

To evaluate the performance of the trained neural network, we ran a series of matches against the strongest heuristic agent, PathFinderHeuristic.

Player	Wins
Neural network	0
PathFinderHeuristic	83
RandomPlayer	0
Draws	17

Table 4.3 NeuralNetworkPlayer, PathFinderHeuristic and RandomPlayer.

First, we evaluate agents in a 1v1 setting with the third player always being the RandomPlayer. As we can see in Table 4.3, the neural network fails to secure a single win in this matchup, while the PathFinderHeuristic dominates the competition. The RandomPlayer, unsurprisingly, also has no wins. The result suggests that the neural network at this stage is undertrained or unable to generalize well against a strong heuristic.

Player	Wins
Neural network	0
2x PathFinderHeuristic	0
Draws	100

Table 4.4 NeuralNetworkPlayer vs. 2x PathFinderHeuristic.

Especially when facing two PathFinderHeuristics, the neural network is unable to find any winning strategy, as expected given its prior results. The result is a full draw, suggesting extreme caution or inaction by all players, possibly indicating a deadlock due to lack of initiative from the network.

Player	Wins
2x Neural network	0
PathFinderHeuristic	100
Draws	0

Table 4.5 2x NeuralNetworkPlayer vs. PathFinderHeuristic.

To further probe the robustness of the PathFinderHeuristic, we also evaluated its performance when playing against two neural networks simultaneously.

This match shows the PathFinderHeuristic wins every single game against two neural networks, highlighting once again its strong performance and the weakness of the current neural model.

Player	Total Wins
NeuralNetworkPlayer	0
PathFinderHeuristic	183
Total Draws	117

Table 4.6 Tournament summary, NN vs path-finder heuristic.

Across all matches against the PathFinderHeuristic, the neural network fails to win a single game. This result reinforces the conclusion that the trained model is not yet competitive with the best heuristic. The high number of draws (117) suggests the neural network is unable to execute effective threats.

4.3 MCTS random rollout vs MCTS with heuristic rollout

To evaluate the impact of different rollout strategies in Monte Carlo Tree Search, we conducted a series of matches between the two MCTS agents – one using random rollouts and the other leveraging the PathFinderHeuristic during the simulation phase.

Player	Wins
MCTS with random rollout	2
MCTS with PathFinderHeuristic rollout	5
Total Draws	93

Table 4.7 Tournament summary. MCTS with random rollout vs MCTS with PathFinderHeuristic rollout.

This comparison of two MCTS variants demonstrates a slight edge for the agent using the PathFinderHeuristic in its rollouts. However, the overwhelming majority of games end in draws, indicating that both MCTS variants are fairly conservative or evenly matched.

4.4 MCTS NN vs MCTS NN v2

To compare two variants of Monte Carlo Tree Search guided by a neural network, we evaluated the original MCTS NN agent against a modified version (v2) that switches to random rollouts in the late game. This approach aims to combine the strengths of neural guidance in the early game with the efficiency of classic rollouts when the search space becomes smaller.

Player	**Wins**
MCTS NN	0
MCTS NN v2	24
Total Draws	76

Table 4.8 Tournament summary. MCTS NN vs MCTS NN v2.

The later version of MCTS with neural network guidance (v2) convincingly outperforms the earlier version, winning 24 games while the older version scores none. This demonstrates a tangible improvement between network versions and suggests v2 has better evaluation or search guidance capabilities.

4.5 Comparison of Best Agents: Heuristic, MCTS with Heuristic Rollout, and MCTS with NN v2

To conclude the evaluation, a final tournament was held between the strongest agents from each category: the top heuristic-based agent PathFinderHeuristic, the MCTS agent with heuristic rollouts, and the most advanced agent using MCTS with a neural network (v2). This matchup offers a direct comparison of the three main approaches explored — pure heuristics, search-based methods, and learning-based agents. Results from the round-robin tournament are summarized in Table 4.9 and highlight the consistency and relative strength of each strategy under identical conditions.

Player	**Wins**
PathFinderHeuristic	0
MCTS with PathFinderHeuristic rollout	4
MCTS with Neural network v2	17
Draws	79

Table 4.9 Tournament summary. PathFinderHeuristic, MCTS with PathFinder-Heuristic rollout, MCTS with Neural network v2.

This final tournament shows MCTS with neural network v2 as the clear winner. It outperforms both the pure heuristic and the heuristic-based MCTS. These results suggest that combining MCTS with an improved neural network leads to the strongest decision-making capabilities in Hex^2 so far.

5 Discussion

The evaluation of AI agents in Hex^2 reveals a distinct contrast between simple, rule-based strategies and more sophisticated, adaptive approaches. While heuristic agents provided a useful starting point, their limitations became clear as the complexity of the game increased. On the other hand, search-based methods—particularly those enhanced with neural networks—demonstrated a stronger ability to adapt to the unpredictable nature of three-player dynamics. This section discusses the performance of each type of agent and identifies the factors that contributed to their success or failure.

5.1 Heuristics

The tournament between heuristics has shown that the PathFinderHeuristic which is the most complicated and the most dynamic heuristic is also the strongest one, with only the EdgeControlHeuristic being to draw majority of the games against it, but also win a smaller portion of them.

We may also notice that the EdgeControlHeuristic consistently outperformed the CenterControlHeuristic, winning every single matchup. This could indicate that in the three-player variant of Hex, occupying the center is not as advantageous as it is in the classic two-player version where central control is considered strong, especially on smaller boards that have been fully solved. [1]

5.2 Neural network

Since NN on its own has lost the vast majority of games against the PathFinderHeuristic and was able to draw only a small portion of the remaining games, it is safe to say on its own NN is not a very strong player. This is mainly due to the network not being able to learn when it is capable of finishing up the game and "go for the kill" and a substantially more training would be needed to achieve that.

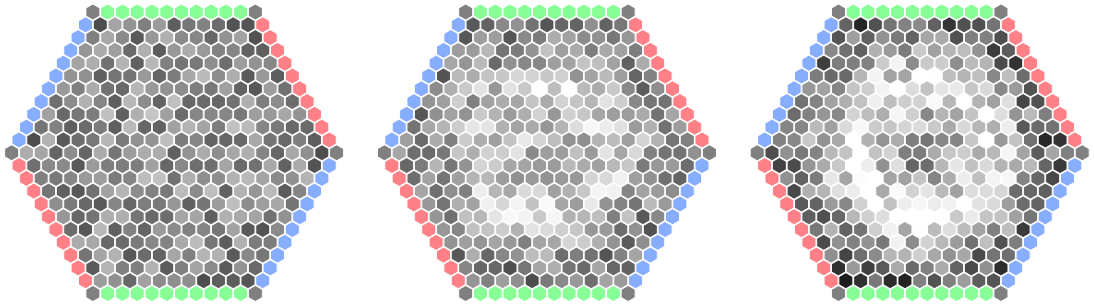
Despite all that, the neural network has revealed many interesting things about the game itself and gave us a general idea of how to approach it.

After applying slight color correction to the output values—such as increasing the contrast to improve readability—two main patterns have emerged.

The first is that while the network initially favors the center of the board for its opening move, it gradually learns to avoid the very center. This is likely because the center is the most contested area, and the network has learned it is better to avoid it early in the game.

The second pattern is the so-called bridge move, a known strategy from classic Hex. It involves placing stones one space apart, making it easier to connect them later while forcing opponents to use two moves to block the path.

These evolving preferences can be clearly seen in Figure 5.1, which illustrates how the neural network's first-move policy changes over time with training—from random dispersion to more structured, strategic positioning.



Before training. After 1200 games. After 4000 games.

Figure 5.1 Neural network's first move policy over time.

5.3 MCTS

The MCTS has shown to be a very strong player, being able to beat all the heuristics with ease even with random rollouts. While the gameplay of basic MCTS tends to be very weak early on, as the game progresses and the game tree shrinks and the search space becomes more manageable, the MCTS starts to play almost optimally.

An improvement in the form of complex rollout, using the same algorithm as PathFinderHeuristic, has shown to be an improvement over the basic MCTS, being able to beat the basic MCTS in a head-to-head match. This however comes at a cost of greatly increased time complexity, as the rollout is orders of magnitude slower. So if we ignore the difference in time it takes for their respective turns, with the same number of iterations, the MCTS with complex rollout is the winner here.

5.4 MCTS with neural network

While NN is not great at all on its own, it turned out to be a great guidance for MCTS at the early stages of the game, when the game tree is simply too large to be searched. That is where the network shines, often being able to recognize potential threats and block them before they truly unfold and giving the MCTS a considerable boost compared to random rollouts.

6 Hex Squared implementation

This chapter describes the software implementation of the Hex^2 game. It explains the client-server architecture, technologies used for both frontend and backend, and the design choices that enable live gameplay, spectating, and AI integration. Particular emphasis is placed on maintainability, extensibility, and ease of deployment across platforms.

The full implementation of Hex^2, including the game logic, AI agents, and web-based interface, is available on GitHub:

`https://github.com/justMazi/Hex-Squared`

6.1 Software architecture

The game is implemented as a web application with a client-server architecture. This allows for any amount of spectators to watch the game in real time. The game state is stored on the server and is updated and retrieved by the clients through REST API calls. The server is responsible for validating the moves and ensuring that the game state is consistent. The clients are responsible for rendering the game state and sending the moves via HTTP requests to the server.

Each of the parts contains a dockerfile, that allows for easy deployment of the application in a containerized environment, so it is platform independent and can be run on any machine with docker engine.

In the root folder there is also a docker-compose file, which allows for easy deployment of the whole application with a single command.

6.1.1 Backend

The backend of Hex^2 is implemented as a .NET Core application, acting as the authoritative game engine and API provider. It exposes a set of REST API endpoints that allow the frontend to retrieve and modify the game state. The backend is responsible for enforcing game rules, validating moves, and allows for synchronization between players and spectators.

The architecture follows a clean architecture pattern, separating the solution into distinct layers.

Inspired by a DDD (Domain Driven Design) approach, at it's core there is a domain layer, containing business classes without any sort of external dependencies. In this case there will be domain-specific entities like Game, GameState, Player etc. Every other layer has either direct or transitive dependency to it.

Application layer holds so-called services. Those usually contain the main business logic that defines how the entities interact with each other and how the data is moved around. In this case, there is for example GameService which wraps the underlying repository and holds additional logic. Additionally, there are interfaces for the services as well as interfaces for the repositories.

It is the implementation of the repository contracts that is stored in the infrastructure layer. Here we actually define how the data is stored and retrieved, what sort of persistence is used and so on.

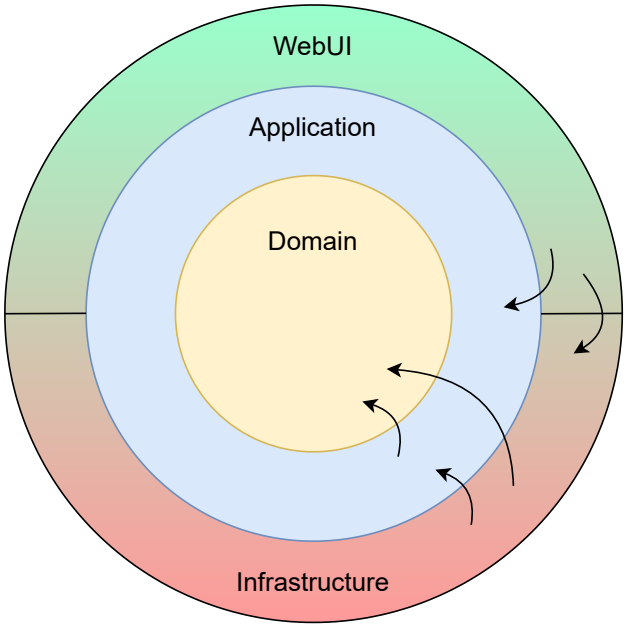


Figure 6.1 Onion diagram of the clean architecture pattern, arrows represent compile time project dependencies.

Finally, the presentation/WebUI layer exposes the API to clients, mapping endpoints to the underlying services and handling incoming requests and outgoing responses.

So why exactly should one go through all this hassle? It becomes rather apparent once your codebase starts to grow in size, or you want to change some parts of the application. For example, if you decide to switch from a simple in memory storage to an SQL database, you would only need to change the implementation of the repository in the infrastructure layer. Such change could even make switch to asynchronous IO operations and the rest of the application would run just as before. All it takes is only to fulfill the contract. All that is thanks to the separation of concerns and proper usage of interfaces.

Another advantage is way easier testing, since all the repositories and services rely on dependency injection and their respective instances are retrieved from the DI container at runtime, you can mock the dependencies and test the rest in isolation.

The game itself is implemented as a finite state machine. This allowed for an easier and clearer implementation of the game rules and state transitions.

Another AI agent can simply be added by defining new inheritor of abstract class **AiPlayer** and implementing the **CalculateBestMoveAsync** method. Because the backend exposes an API endpoint that using reflection returns all the inheritors of **AiPlayer** and frontend consumes this endpoint, the new AI agent will be automatically available on the frontend.

6.1.2 Frontend

The frontend of Hex[2] is built using the Svelte framework, a modern JavaScript framework that compiles components into highly efficient vanilla JavaScript, that is then executed in the browser. Svelte was chosen for its simple minimal syntax, easy reactivity and performance.

The frontend is responsible for rendering and refreshing of the game state as well as utilizing browser cookies for distinguishing players. This ensures the player can continue the game even after refreshing the page or closing the browser, and at the same time, that the other players cannot impersonate them.

6.1.3 State representation in MCTS

Since MCTS requires rapid state evaluations and rollouts, the game state representation must be as lightweight as possible. It is not feasible to reuse the implementation of the game state from the backend state machine, as such representations tend to be designed for correctness and maintainability rather than computational efficiency. Instead, a minimal and optimized state representation is used to facilitate high-speed simulations and tree expansions.

In C# it is implemented as a 2D array of bytes. Since we flatten the hexagon, two of the corners remain unused. The huge performance gain is however worth it as this is merely a small drawback. An example of such representation can be seen in Table 6.1.

-1	-1	-1	-1	-1	-1	-1	-1	2	2	2	2	2	2	-1
-1	-1	-1	-1	-1	-1	3	0	0	0	0	0	0	0	1
-1	-1	-1	-1	-1	3	0	0	0	0	0	0	0	0	1
-1	-1	-1	-1	3	0	0	0	0	0	0	0	0	0	1
-1	-1	-1	3	0	0	0	0	0	0	0	0	0	0	1
-1	-1	3	0	0	0	0	0	0	0	0	0	0	0	1
-1	3	0	0	0	0	0	0	0	0	0	0	0	0	1
-1	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
1	0	0	0	0	0	0	0	0	0	0	0	0	3	-1
1	0	0	0	0	0	0	0	0	0	0	0	3	-1	-1
1	0	0	0	0	0	0	0	0	0	0	3	-1	-1	-1
1	0	0	0	0	0	0	0	0	0	3	-1	-1	-1	-1
1	0	0	0	0	0	0	0	0	3	-1	-1	-1	-1	-1
1	0	0	0	0	0	0	0	3	-1	-1	-1	-1	-1	-1
-1	2	2	2	2	2	2	-1	-1	-1	-1	-1	-1	-1	-1

Table 6.1 Efficient 2D game state representation. Each positive number represents a tile occupied by a player. Negative one represents an unplayable tile, and zero an empty tile. (Obviously, -1 overflows to 255, but the functionality remains just the same.)

6.2 User Guide

In order to get the application up and running, both the backend and the frontend need to be running and listening on their respective ports. The simplest

way to achieve this is to use the provided `docker-compose.yml` file located in the root folder of the project.

Prerequisites The host system must have the Docker Engine installed in order to run the application using Docker Compose. Docker is not installed by default on most operating systems and requires manual installation. However, Docker Desktop, which includes Docker Compose, is available for **Windows**, **macOS**, and **Linux**. Installation guides for each platform are available on the official Docker website[1].

- **Windows**: Docker Desktop requires Windows 10/11 with WSL2 backend enabled.
- **macOS**: Docker Desktop supports both Intel and Apple Silicon chips.
- **Linux**: Docker Engine and Compose can be installed separately via package managers depending on the distribution.

Once Docker is installed and running, the application can be started simply by executing the following command in the root folder:

```
docker compose up
```

Running the Application After the images are built and the containers are started, the frontend will be accessible at `http://localhost:3100`, and the backend will run at `http://localhost:8100`.

You can now navigate to the frontend address in your browser. On the introductory screen, you can either generate a new link or copy an existing one using the copy button. This link can be shared with other players, allowing them to join the session.

The game offers 3 board sizes to choose from. If you plan to play against AI, you have the option to select from several AI agents, each featuring a distinct strategy and level of difficulty. This choice determines which AI agents populate the game when you use the "Fill with AI" button on the subsequent screen, which appears after entering the game.

[1] `https://docs.docker.com/get-started/get-docker/`

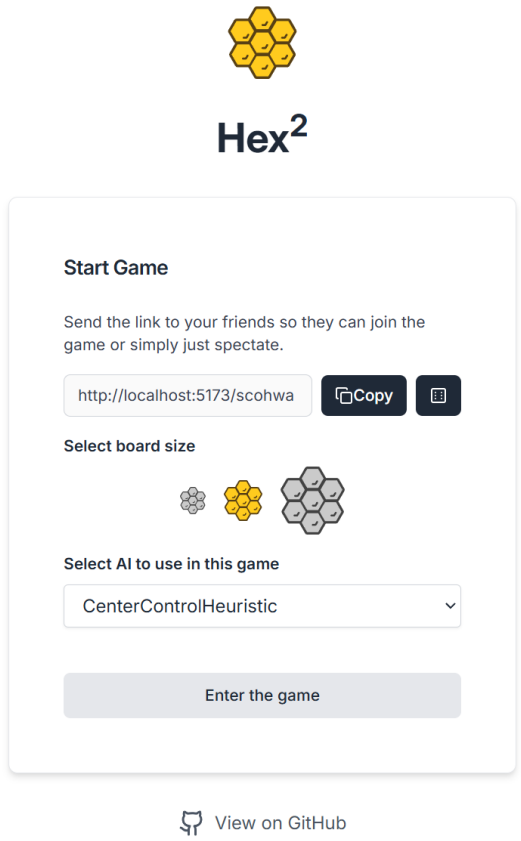


Figure 6.2 Hex^2 intro screen

After entering the game, several buttons will be available for color selection. Each of the three colors has a corresponding button, allowing players to choose according to their preference. Should a player wish to switch their selected color before the game begins, they can do so by selecting another available color, which automatically frees up their previous choice for others.

The game can also start without human players, allowing AI to take all three slots while human participants only spectate. The game officially begins once all three colors are taken.

At the end of the game, players are presented with the result and have the option to restart while keeping their previously chosen colors.

Throughout the rounds, on the left side of the screen a scoreboard tracks and displays the number of wins for each player, providing an ongoing record of performance. Draws are not counted toward any player, as they do not contribute to anyone's score.

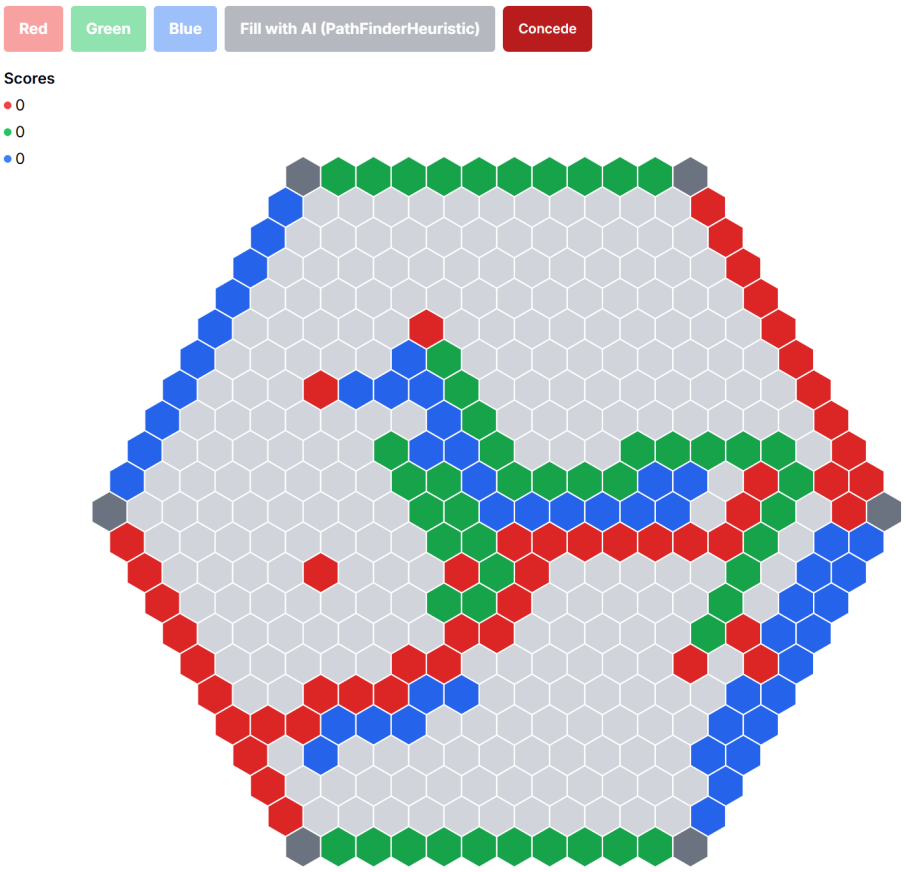


Figure 6.3 In progress game of Hex[2]

At any point during the game, players can concede by clicking the respective button. Their turn will be skipped, and the game will continue without them.

If there is no activity registered in a game for 24 hours, that game is cleaned up by the server.

Conclusion

In this thesis, we introduced and implemented Hex^2, a novel three-player variant of the classical Hex game. By extending Hex to include a third player and redesigning the board into a fully hexagonal shape, we created a game that brings entirely new dynamics, strategies, and challenges. Our work explored this game from both theoretical and practical perspectives, covering its formal properties, unique game mechanics, and AI-based approaches for playing it effectively.

As Hex^2 represents a completely new game, an important part of our work was to explore how AI can approach this multiplayer setting. To that end, we developed and tested a wide variety of AI agents - ranging from simple heuristic-based agents and adaptive heuristics via evolutionary algorithms, to Monte Carlo Tree Search and neural-network-guided MCTS. The results of these experiments reveal several key insights.

Firstly, simple heuristics were insufficient to compete at a high level, with the PathFinderHeuristic outperforming all other basic strategies and often either winning or at least drawing. Other heuristics, such as CenterControl and Random, failed to pose a meaningful challenge, underscoring both the complexity of the game and the limitations of naive strategies. This also reflects a fundamental shift in strategic priorities – while the classic game revolves around controlling the center, in Hex^2, the most critical battles often take place around it.

Secondly, pure neural networks, although promising, failed to achieve competitive strength. While neural networks managed to capture some patterns — such as avoiding contested center positions and using skip moves — they struggled to finish games efficiently, often failing to exploit winning opportunities. This suggests that although deep learning holds potential for multiplayer games, substantially more training data would be needed to reach competitive performance. Ultimately, this highlights the enduring value of search-based methods, which remain essential for strong gameplay by enabling effective planning and tactical foresight.

Finally, MCTS proved to be the most successful AI approach, reliably defeating all heuristic-based agents. Especially when combined with neural network guidance, MCTS achieved the strongest performance, taking advantage of both search-based and learned knowledge. Notably, neural network-enhanced MCTS excelled in the early game where the search space is vast and brute-force search is ineffective, while switching to pure MCTS in the late game allowed for more precise tactical play. This combination highlights the complementary nature of search and learning-based methods in complex environments.

Beyond AI performance, this thesis also contributes a fully working software implementation of Hex^2, including a modular AI architecture and web-based interface. The solution is designed to be easily extensible for future AI development and experimentation, and fully reproducible thanks to the backend experiments project, which automates systematical benchmarking of agents.

Bibliography

1. Browne, Cameron. *Hex Strategy: Making the Right Connections*. 2000. isbn 978-1-56881-117-8.
2. Browne, Cameron B.; Powley, Edward; Whitehouse, Daniel; Lucas, Simon M.; Cowling, Peter I.; Rohlfshagen, Philipp; Tavener, Stephen; Perez, Diego; Samothrakis, Spyridon; Colton, Simon. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*. 2012, vol. 4, no. 1. Available from doi: 10.1109/TCIAIG.2012.2186810.
3. Silver, David; Schrittwieser, Julian; Simonyan, Karen; Antonoglou, Ioannis; Huang, Aja; Guez, Arthur; Hubert, Thomas; Baker, Lucas; Lai, Matthew; Bolton, Adrian, et al. Mastering the game of Go without human knowledge. *Nature*. 2017. Available from doi: 10.1038/nature24270.
4. Henderson, Philip; Arneson, Broderick; Hayward, Ryan B. Solving 8×8 Hex. In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*. IJCAI, 2009, pp. 505–510.
5. Hayward, Ryan B.; Wang, Philip J. Solving 8 x 8 Hex. 2006. Available also from: https://webdocs.cs.ualberta.ca/~hayward/papers/solve8.pdf.
6. Reisch, Stefan. Hex ist PSPACE-vollständig. *Acta Informatica*. 1981.
7. Ryan B. Hayward, Bjarne Toft. *Hex The Full Story*. 2019. isbn 978-0-36714-422-7.
8. Eiben, A.E.; Smith, James E. *Introduction to evolutionary computing*. Springer Berlin Heidelberg, 2015. isbn 978-3-66244-873-1.
9. Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. *Deep Learning*. MIT Press, 2016. isbn 978-0262035613.
10. Sutton, Richard S.; Barto, Andrew G. *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press, 2018. isbn 978-0262039246.
11. Pearl, Judea. *Heuristics: Intelligent search strategies for computer problem-solving*. Addison-Wesley, 1984.
12. Russell, Stuart J.; Norvig, Peter. *Artificial Intelligence: A modern approach*. Pearson, 2021. isbn 978-0-13461-099-3.
13. Knuth, Donald E.; Moore, Ronald W. An analysis of alpha-beta pruning. 1975. Available from doi: 10.1016/0004-3702(75)90019-3.
14. Martin J. Osborne, Ariel Rubinstein. *A Course in Game Theory*. 1994. isbn 978-0-26215-041-5.
15. *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973.

List of Figures

List of Tables

List of Abbreviations

- AI – Artificial Intelligence
- API – Application Programming Interface
- CNN – Convolutional Neural Network
- DI – Dependency Injection
- DDD – Domain-Driven Design
- FSM – Finite State Machine
- MCTS – Monte Carlo Tree Search
- MDP – Markov Decision Process
- NN – Neural Network
- NLP – Natural Language Processing
- RNN – Recurrent Neural Network
- RL – Reinforcement Learning
- ReLU – Rectified Linear Unit
- REST – Representational State Transfer
- SQL – Structured Query Language
- UCB1 – Upper Confidence Bound 1