Master Thesis

FrankenSearch: An Adaptive Approach to Search Algorithm Design

Leon Maria Haag

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Department of Advanced Computing Sciences
of the Maastricht University

Maastricht University
Faculty of Science and Engineering
Department of Advanced Computing Sciences

January 26, 2025

Abstract

Within the pursuit of Artificial General Intelligence (AGI), search algorithms remain a foundational approach to systematically solve complex decision-making tasks. This thesis introduces FrankenSearch, an adaptive framework for designing and executing search algorithms that aims to autonomously adapt to different problems. By decomposing established best-first search algorithms, including Monte-Carlo Tree Search (MCTS) and Proof-Number Search (PNS), into a set of smaller, parameterized building blocks, it becomes possible to restructure and recombine these components in new ways. The central innovation of this work lies in the development of the Search Algorithm Description Language (SADL), a Lisp-like language that provides a concise yet flexible mechanism for representing search algorithms in a structured form.

In its current scope, FrankenSearch focuses on two-player, deterministic, turn-taking, zero-sum board games and is specifically validated on the game of Lines of Action (LOA). Experiments confirm that SADL-based implementations of MCTS and PNS operate as intended, offering a functional proof of concept despite the computational overhead introduced by the external Ludii framework, which manages the game states in Java. In particular, the PNS implementation was applied to solve curated endgame positions in LOA, demonstrating correctness but also showing large memory usage when constructing detailed game trees.

FrankenSearch is built on a layered architecture: (1) domain-agnostic SADL specifications that can be parsed or interpreted in any language, (2) a Java-based engine to compile SADL code into executable components, and (3) an integration layer that applies these components to a specific domain—in this case, Ludii-based board games. This modular design enables each layer to be replaced or adapted independently, making it possible to use alternative parsers or custom game environments. Moreover, an embedded evolutionary algorithm leverages a Genetic Programming approach: by treating the Abstract Syntax Trees of SADL-based search algorithms as evolvable structures, new variants can be generated through mutation and recombination. Although none of the evolved variants have, so far, outperformed baseline MCTS under practical time constraints, these experiments illustrate the potential of a generative, component-driven methodology for designing novel search algorithms.

By systematically breaking down algorithms into interchangeable parts and describing them in SADL, FrankenSearch provides a flexible foundation for automated search algorithm generation. Building on this core framework, future research can explore more diverse domains, refine evolutionary strategies, and further optimize the data-handling layers to reduce overhead, ultimately broadening the framework's utility.

Preface

I would like to express my gratitude to my thesis supervisors, Prof. Dr. Mark Winands, Dr. Dennis Soemers, and Dr. Tony Garnock-Jones, for their invaluable guidance and support throughout this project. I am especially thankful to Prof. Winands for meeting with me regularly to discuss my progress and for providing consistent feedback that kept me on track. Special thanks to Dr. Garnock-Jones, who taught me so much about language design in such a short timeframe and generously allowed me to use his personal server for computational resources.

I also want to thank Chloé Crombach and Saurav Zacharias for always being available to discuss my ideas and for their encouragement throughout this journey. Your input and support made this process much easier and more enjoyable.

Contents

Chapter 1

Introduction

Adaptive search algorithms are a crucial research area in the pursuit of Artificial General Intelligence (AGI). This study lays the groundwork for a framework capable of autonomously adapting search algorithms to various problems by manipulating their descriptions in a custom-designed language.

Section 1.1 discusses the background and motivation for adaptive approaches, as well as their alignment with current efforts in artificial intelligence (AI) research. Section 1.2 presents the problem statement and specific research questions, while Section 1.3 provides an overview of the remaining chapters in this thesis.

1.1 Background and Motivation

In recent years, the field of artificial intelligence (AI) has garnered massive attention, both in academic research and in various industries and people's everyday lives (Dwivedi et al., 2021). Most of these instances are applications of so-called "narrow AI", specialized programs designed to solve a particular problem, manually fine-tuned to perform well in one task. To advance beyond these methods, there is increasing focus on Artificial General Intelligence (AGI), programs that could solve different problems and adapt to new, unseen situations (Gobble, 2019). The massive improvements within a short time period in the performance of large language models (LLMs) and the ability of tools like ChatGPT to assist in a wide variety of tasks have spurred interest and vision in AGI (Zhao et al., 2023). However, current research points towards diminishing returns in increased LLM training. The unpredictability of LLMs and their tendency to "hallucinate" are still far from solved, indicating that the gap from current LLM-based reasoning to true AGI is still significant (Pilditch, 2024; Xu et al., 2024).

As an alternative to the LLM-approach of "learning" through vast amounts of data, many complex tasks in a variety of contexts can be modeled through sequential decision-making. This approach frames problem solving as a search through a tree of possible decisions, with the goal of finding optimal solutions. Search is therefore a promising avenue in trying to achieve programs closer to AGI.

Games have long been recognized as an ideal testbed for developing and evaluating search algorithms due to their structured rules and clear scoring systems (Schaul et al., 2011). For those reasons, games are also used as a testbed for this research. This aspect is further discussed in Subsection 2.1.1.

Sironi and Winands (2021) identify current limitations of search algorithms and propose a framework to tackle these issues by being able to select and modify a search algorithm for any new problem. Algorithms like Monte-Carlo Tree Search (MCTS) or $\alpha\beta$-search have been used successfully in many different applications (Świechowski et al., 2023; Knuth & Moore, 1975). Other algorithms like Proof-Number Search (PNS) (Allis et al., 1994) are quite successful in specific endgame-situations, but are less generally applicable (Kishimoto et al., 2012).

In all use cases, much human decision-making is involved in developing and applying search algorithms, starting from the choice of algorithm, usage of different variants, and even parameter tuning. The proposed framework aims to autonomously generate a search algorithm by selecting, combining, and adapting components of known search algorithms. Existing algorithms are abstracted into well-defined components that together form an algorithm. A structured language, independent of any specific programming language, should be able to represent any feasible search algorithm within the framework. A class grammar has been identified as a potentially useful method by Sironi and Winands (2021), as they identify and discuss three challenges:

- Developing a formal language to encode search algorithms.
- Developing methods that decide how to select good enough search algorithms for their application domain to address each given task without using any domain-specific information.
- Developing methods that automatically generate new search algorithms.

This thesis builds on the ideas of Sironi and Winands (2021) and aims to engage with all three of these challenges, focusing primarily on the first, as a structured way to represent search algorithms is a precondition for effectively tackling the other issues. Inspired by frameworks such as SATenstein (KhudaBukhsh et al., 2016), which parameterizes SAT solvers, and the Ludii framework (Piette et al., 2020) for describing games with simple building blocks, this thesis introduces FrankenSearch. FrankenSearch is a modular and adaptive framework for search algorithm design and execution, leveraging the newly proposed Search Algorithm Description Language (SADL) to encode both existing and novel search algorithms.

1.2 Problem Statement and Research Questions

While the vision is to work towards a framework capable of creating a wide variety of algorithms for a wide variety of problems, this thesis initially limits the scope, focusing on two-player, deterministic, turn-taking, zero-sum, perfect-information games as the main domain and limiting the algorithm space to best-first search algorithms. The previous motivation can be summarized into the following problem statement and four research questions, while keeping in mind the limited scope as a proof of concept.

Problem Statement: Developing a framework that decomposes search algorithms into small, parameterized components, enabling autonomous selection and adaptation.

The problem statement is addressed by the following three research questions:

1. **How can complex search algorithms such as Monte-Carlo Tree Search and Proof-Number Search be effectively decomposed into building blocks for the design of search algorithms?**

   To address the first question, the focus will be on identifying the fundamental components that constitute these algorithms and understanding how these components interact. Breaking down complex algorithms into reusable parts is a precondition for effectively describing, and ultimately recombining them.

2. **How can a structured language be developed to effectively represent search algorithms?**

   Designing a structured Search Algorithm Description Language (SADL) provides a standardized platform to define, manipulate, and apply both existing and novel search algorithms. The second question explores the principles of language design, including syntax and semantics, that can effectively encode search algorithms while maintaining a balance between simplicity and expressiveness.

3. **How can evolutionary algorithms be employed to generate, optimize, and evaluate new search algorithm variants effectively?**

   Evolutionary algorithms are powerful tools for exploring large search spaces. The third question investigates how these algorithms can be utilized within the framework to autonomously generate and optimize novel search algorithms based on existing SADL descriptions of algorithms. Within the domain of two-player game search that is used for this research, search algorithms are generally evaluated by their performance against other agents, as opposed to being able to be evaluated independently, as is common in evolutionary algorithms.

1.3 Thesis Overview

The remaining thesis is organized as follows: Chapter 2 provides a literature overview of search problems and presents a selection of existing search algorithms. Additionally, it examines existing parameterized algorithm- and game frameworks and provides background on language design and grammar, as well as evolutionary algorithms. Next, Chapter 3 introduces FrankenSearch, detailing the design of the Search Algorithm Description Language (SADL), including the decomposition of search algorithms into reusable components, starting with MCTS and PNS, and the compilation of the abstract language into executable code using Ludii games for testing. Afterwards, Chapter 4 presents experiments and results using selected games and algorithms. Finally, Chapter 5 offers conclusions and proposes future research directions.

Chapter 2

Foundations and Related Work

This chapter begins by providing background on search algorithms and the use of games as a testbed, while elaborating on the chosen scope for this thesis in Section 2.1. Next, Section 2.2 gives an overview of related work on modular frameworks, including those for General Game Playing (GGP), and provides a description of the Ludii Framework, which is used to apply algorithms to games in this thesis. Afterwards, Section 2.3 discusses some basic concepts of formal languages that serve as a backdrop for the design of the Search Algorithm Description Language (SADL). Finally, Section 2.4 introduces Evolutionary Algorithms (EAs), which are used to generate new search algorithm variants.

2.1 Search Algorithms for Games

Many games can be seen as search problems, where the search space consists of all possible combinations of moves, and the objective is to search through that space – usually with limited resources – in order to determine the best possible current move. In this thesis, the focus lies initially on two-player, zero-sum, perfect information games, a common dedicated research area (Schaeffer & van den Herik, 2002). Russell and Norvig (2020) describe these as *"deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite"*. *Perfect information* means that all agents have complete knowledge of the game state, including the history of all previous moves, and deterministic knowledge about any future possible move options (so there is no hidden information such as other player's cards in Poker and many other card games).

The combination of these attributes is an important condition for many search algorithms to work. It implies that the "best" move for a given player in a given situation will be the most damaging action for the opponent, and that with perfect information, one can try to predict the move that the opponent might make in the future, assuming they are a rational player, i.e. trying to maximize their utility (or gain) in the game. Chess, Checkers, Go, and Tic-Tac-Toe are common examples of two-player, zero-sum, perfect information games.

These ideas allow for tree search to be effectively employed to tackle the problem. Starting from the current board position (or abstract game state) as the root node, each node in the first ply represents the game state after each possible move. Nodes in the second ply represent the game state after any subsequent move (usually by the opponent player), respectively. One of the earliest search algorithms using these ideas is *Minimax* (von Neumann & Morgenstern, 1944), which searches through the tree by always assuming that the current player chooses the available move that will maximize their gain, while the opposing player will subsequently choose the move that will minimize the first player's gain (because the opponent will maximize their own gain, which, based on zero-sum, directly implies minimizing the other player's gain). The actual "gain" can be determined by calculating the entire search tree until terminal positions are reached and then backpropagating the result. While this works, for example, in Tic-Tac-Toe and other simple games, it is not feasible for the large trees of most games because of time and memory constraints. Alternatively, various heuristics might be used to evaluate a non-terminal game state.

Minimax generally follows a Depth-First Search (DFS) approach, and while it alone is not efficient enough to be widely used beyond simple demonstrations, it laid the foundation for the family of $\alpha\beta$-algorithms to emerge. These algorithms use pruning of non-optimal branches to significantly reduce the search space and focus resources on promising moves. A number of enhancements (e.g., transposition tables, move ordering) have been proposed over the years, making $\alpha\beta$ one of the most widely used algorithms in game search (Marsland, 1986; Saffidine et al., 2012).

While the pruning of $\alpha\beta$ and enhancements like move ordering help focus on promising moves, the core actual tree-search operation is generally Depth-First Search. In contrast, many other algorithms follow a *best-first-search* approach, where at every iteration of the tree search, the currently most promising (or "best") node is further investigated. Since this thesis investigates the possible generation of new search algorithms by combining elements of existing ones, an initially narrow scope of algorithms that follow a somewhat similar structure is useful. Henceforth, the focus lies on best-first-search algorithms. Two specific instances – Monte-Carlo Tree Search and Proof-Number Search – are explained in detail (Subsections 2.1.2 and 2.1.3, respectively) and are used as a baseline for the remainder of this thesis.

2.1.1 Games as Testbed for Search

Before diving deeper into specific search algorithms, the choice and high utility of games as a testbed for search shall be emphasized. As detailed in Chapter 1, despite the apparent focus on games, the broader goal is to advance search algorithms for use-cases in a wide variety of industries, and get closer to Artificial General Intelligence (AGI). Games have long been identified as a useful environment to develop and test (search-) algorithms, with applications beyond games (Schaul et al., 2011; Yannakakis & Togelius, 2018). Almost all games by nature have clear and fixed rules that define all possible actions in any given state. Similarly, games usually have a clear and deterministic scoring method, declaring a win, draw, or loss, possibly supplemented by a numerical score that indicates the margin of victory (or defeat). Single-player games often only have a score as a benchmark of performance, without concept of winning or losing (e.g., achieving a high score in Space Invaders after "Game Over"). These attributes make it rather convenient to test and evaluate different algorithms on such games, compared to other real-world use-cases of search algorithm, where the actual performance might be much harder to quantify, e.g., resource allocation and scheduling of surgeries in the context of hospital operations (Cardoen et al., 2010; Guerriero & Guido, 2011). Additionally, the vast number and diversity of games reflect a wide variety of types of problems to be solved, and General Game Playing frameworks (see Section 2.2) allow to easily test algorithms on a wide variety of problems with just one interface.

In line with the focus on games, domain-specific vocabulary like "player", and "move" are used throughout this thesis, but unless stated otherwise, the ideas always apply more generally beyond games, read "agent" and "action", respectively, for the previous examples.

2.1.2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a best-first search algorithm widely used in artificial intelligence, especially in game playing. The algorithm builds a search tree incrementally by using random simulations to estimate the value of different actions, making it highly effective in vast search spaces where traditional methods often fall short. The concept of combining random playouts with tree search can be traced back to Abramson (1987). The method was formalized by Coulom (2007), who introduced the term “Monte-Carlo Tree Search”. A considerable advancement in MCTS was the development of the *Upper Confidence Bound applied to Trees* (UCT; see details below) by Kocsis and Szepesvári (2006), which provides a systematic way to balance exploration and exploitation during the search process.

MCTS garnered wider attention after its successful application in computer Go. Google DeepMind combined MCTS with neural networks to create AlphaGo. AlphaGo’s victory over professional Go players, particularly its win against Lee Sedol in 2016, highlighted the potential of MCTS in solving complex decision-making problems and marked a milestone in AI development (Silver et al., 2016).

The MCTS Algorithm

MCTS consists of the following four main steps (also see Figure 2.1), which are repeated iteratively, usually until a computational budget (such as time) is exhausted (Chaslot, Winands, van den Herik, et al., 2008):

- **Selection:** Starting from the root node, the algorithm selects successive child nodes until it reaches a leaf node. The selection process is guided by a policy designed to balance exploration of new nodes and exploitation of nodes with high estimated rewards. The UCT formula (Kocsis & Szepesvári, 2006) is often used for that guidance, and is explained below.
- **Expansion:** If the selected leaf node does not represent a terminal state, one or more child nodes are generated and added to the tree. This step allows the algorithm to explore new states that have not yet been evaluated.
- **Simulation:** A play-out or rollout is performed from the newly added node until a terminal state is reached. The outcome of this simulation provides an estimate of the node’s value. This simulation is traditionally conducted using random actions, but can also incorporate domain-specific heuristics to improve accuracy.
- **Backpropagation:** The results of the simulation are propagated back through the nodes that were selected during the current iteration. This process updates the statistics of each node, such as the visit count and the average reward, which are used to guide future selections.

The four steps of MCTS – Selection, Expansion, Simulation, and Backpropagation – work together to incrementally build a search tree that balances exploration and exploitation (Chaslot, Winands, van den Herik, et al., 2008). By repeating these steps, MCTS is able to make increasingly informed decisions, which is why it is particularly effective in complex decision-making environments, such as strategic games. One aspect that sets it apart from other search algorithms like $\alpha\beta$-search is that there is no need for extra heuristics to evaluate a given non-terminal board position. By collecting statistics from random playouts, MCTS implicitly evaluates any node over time (Browne et al., 2012).

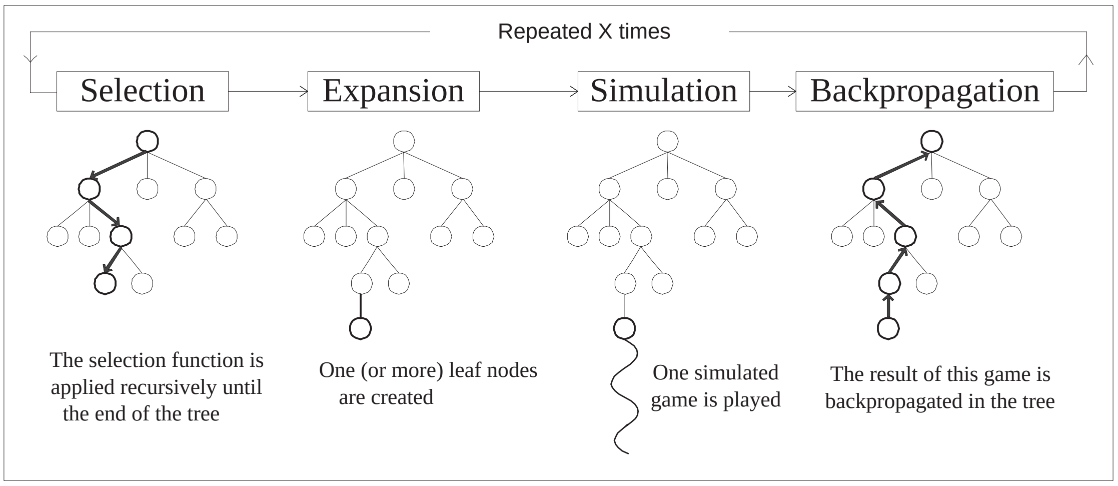

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until the end of the tree
One (or more) leaf nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 2.1: Diagram of Monte-Carlo Tree Search (taken with permission from Chaslot, Winands, van den Herik, et al. (2008))

Upper Confidence Bound for Trees Formula

The Upper Confidence Bound for Trees (UCT), proposed by Kocsis and Szepesvári (2006), is a formula used during the *Selection* phase of MCTS to balance exploration (trying out less visited nodes) and exploitation (focusing on nodes with high average reward). The UCT value for a node i is given by:

$$\mathrm{UCT}_i = \overline{X_i} + C\sqrt{\frac{\ln N_p}{N_i}} \tag{2.1}$$

where:

- $\overline{X_i}$ is the average reward of node i,
- C is a positive constant that controls the balance between exploration and exploitation,
- N_p is the number of times the parent node has been visited,
- N_i is the number of times node i has been visited.

$\overline{X_i}$ is the average reward of the node, which encourages selecting nodes that have yielded good results in previous simulations, and therefore acts as the *exploitation* term. $\sqrt{\frac{\ln N_p}{N_i}}$ is the *exploration* term, which encourages visiting nodes that have been visited fewer times. The logarithmic factor $\ln N_p$ ensures that as the number of simulations increases, the exploration term decreases, reducing the likelihood of repeatedly selecting less promising nodes. Finally, the constant C acts as a balancing term, setting the priority between exploration and exploitation.

Modern Variants and Applications

Since its inception, MCTS has undergone numerous adaptations and improvements to enhance its performance across various domains. The integration with neural networks, as in the previously mentioned AlphaGo, has significantly boosted the algorithm's efficiency and applicability, handling various complex games (Silver et al., 2018), but also guiding research like drug discovery (Wang et al., 2020). Parallelization techniques, such as root and tree parallelization, have been developed, using multi-core processors to accelerate the algorithm's execution (Chaslot, Winands, & van den Herik, 2008). In domain-specific applications, MCTS has been tailored to incorporate prior knowledge, particularly in games like Go, where patterns and heuristics guide the search process more effectively (Gelly & Silver, 2007). Additionally, methods like Rapid Action Value Estimation (RAVE) (Gelly & Silver, 2011) have been introduced to enhance MCTS by updating node values based on the statistical value of moves across different game states, especially when move order is less critical. Wider applications of MCTS include combinatorial- and scheduling problems, as well as Procedural Content Generation (PCG) (Browne et al., 2012).

2.1.3 Proof-Number Search

Proof-Number Search (PNS) is a best-first search algorithm specifically designed for solving problems that can be modeled as AND-OR trees. Originally introduced by Allis et al. (1994), PNS has been widely used in two-player perfect-information games (Kishimoto et al., 2012). However, the algorithm is domain-independent and can be applied to any problem where the objective can be framed as a proof or disproof task.

The core idea of PNS is to determine the game-theoretical value of a position or state by systematically proving or disproving whether a particular *proof goal* can be achieved. The proof goal is problem-specific and defines the success criteria for the search. For instance, in games, the proof goal might be *win* or *at least draw*, depending on the context.

Algorithm Overview

PNS models the problem space as an AND-OR tree, where nodes represent states or positions and are categorized as follows:

- **OR-nodes** represent decision points where the maximizing player is to move. To achieve the proof goal at an OR-node, at least one child must satisfy the proof goal.
- **AND-nodes** represent decision points where the minimizing player is to move. To achieve the proof goal at an AND-node, all children must satisfy the proof goal.

Each node in the tree is associated with two values:

- **Proof Number (PN):** The minimum number of leaf nodes that must be proved to confirm that the current node satisfies the proof goal.
- **Disproof Number (DN):** The minimum number of leaf nodes that must be disproved to confirm that the current node does not satisfy the proof goal.

The PNS algorithm selects the node that is most likely to help achieve or refute the proof goal by focusing on the node with the most favorable proof and disproof numbers.

Proof and Disproof Numbers

Assigning and updating proof and disproof numbers is essential to the PNS algorithm:

- Leaf Nodes: A terminal node that satisfies the proof goal has a proof number of 0 and a disproof number of ∞, while a node that does not satisfy the proof goal has a proof number of ∞ and a disproof number of 0. An unknown leaf node has both proof and disproof numbers set to 1.

- Internal Nodes:
  - The proof number of an OR-node is the minimum of its children's proof numbers (since proving one child is sufficient to achieve the proof goal for the OR-node).
  - The proof number of an AND-node is the sum of its children's proof numbers (since all children must be proved to achieve the proof goal for the AND-node).
  - The disproof number behaves analogously, but with the roles of AND and OR nodes reversed.

Steps of Proof-Number Search

PNS operates through the following iterative steps until the game tree is either fully solved or the computational resources are exhausted:

- **Selection:** Start at the root and follow the most promising path by selecting the child with the lowest proof number at OR-nodes and the child with the lowest disproof number at AND-nodes. This process continues until a leaf node is reached.
- **Expansion:** The selected leaf node is expanded by generating its children, which are then evaluated. The proof and disproof numbers for these children are set based on their evaluation.
- **Backpropagation:** The proof and disproof numbers of the expanded node's ancestors are updated based on the newly assigned values of the expanded node and its siblings. This step ensures that the most promising path remains the focus of the search.

Variants and Enhancements

Over the years, several variants and enhancements of the original PNS algorithm have been proposed to improve its efficiency and applicability, including:

- **df-PN (Depth-First Proof-Number Search):** This variant transforms the best-first search nature of PNS into a depth-first search, which is more memory efficient. It is particularly useful in deep game trees where memory resources are limited (Nagai, 1999).
- **PN^2:** This method employs a second-level PNS to manage memory usage more effectively, by solving subproblems locally before integrating them into the larger search tree (Breuker, 1998).
- **PDS-PN (Proof and Disproof Number Search - Proof Number):** A hybrid approach that combines depth-first and best-first strategies, first using a depth-first search to explore deep variations before applying PNS for more precise evaluation (Winands et al., 2003).
- **Parallel PNS:** Techniques have been developed to parallelize the PNS algorithm, allowing it to take advantage of modern multi-core processors. This parallelization significantly speeds up the search process by distributing the workload across multiple threads (Kaneko, 2010; Saito et al., 2009).

Proof-Number Search has been applied successfully in various domains, particularly in solving complex endgames in board games like Connect Four, Go, and Chess (Kishimoto et al., 2012). PNS has also found use in AI beyond games, for example in model checking (Saffidine, 2012) in the field of logic, as well as retrosynthesis, the synthesis planning of new target molecules in Chemistry (Franz et al., 2022).

2.1.4 Combining MCTS and PNS

As detailed in the previous subsections, both Monte-Carlo Tree Search (MCTS) and Proof-Number Search (PNS) have been used effectively in a variety of tasks, particularly in game AI. MCTS can successfully handle large and complex search spaces through its stochastic simulations. PNS, on the other hand, with its focus on binary goal-search tasks, can excel in endgame scenarios, but struggles when applied to the root of a large search tree, as it can quickly exhaust computational resources. This difference in the strengths of MCTS and PNS, along with the similarity in their core approach of iterative best-first search, makes it enticing to combine these algorithms. The goal is to leverage the exploration capabilities of MCTS while incorporating the deterministic proof and disproof mechanisms of PNS to improve overall decision-making, particularly in endgames or smaller subtrees.

The *MC-PNS* algorithm, proposed by Saito et al. (2006), applies Monte-Carlo evaluations to enhance Proof-Number Search in Go, successfully integrating the flexibility of Monte-Carlo simulations with the deterministic strengths of PNS, showing improvements in memory efficiency and search time, in the Go Life-and-Death problem.

Doe et al. (2022) proposed a new hybrid algorithm called *PN-MCTS* (later refined by Kowalski et al. (2025)), which integrates proof and disproof numbers into the MCTS tree structure to improve decision making. Specifically, proof and disproof numbers are used for:

- **Final Move Selection**: Prioritizing moves that lead to proven outcomes, ensuring the algorithm chooses moves that guarantee wins when possible.
- **Subtree Solving**: Identifying subtrees that have been proven or disproven allows the algorithm to skip unnecessary simulations of those subtrees, saving computational resources.
- **Enhanced UCT Selection**: Modifying the standard UCT approach by incorporating proof and disproof numbers into the selection process, allowing for more informed decision-making when evaluating node quality.

This approach has shown significant improvements, with experiments demonstrating up to a 96.2% win rate compared to standard MCTS in Lines of Action (Kowalski et al., 2025).

Given the proven effectiveness of combining MCTS and PNS in various games, hybrids of these two present a promising initial scope and proof of concept for a general and adaptive framework for search algorithms, which is the focus of this thesis. By decomposing and recombining key components of these algorithms, the framework aims to generate adaptive search algorithms that can autonomously adjust to different problems.

2.2 Parameterized and Modular Frameworks

Frameworks that are designed to be easily adaptable, either by simple manual parameterization, or even autonomous adaptation are useful in many research areas, allowing easy application in a wide variety of problems. This section provides an overview of two kinds of modular frameworks that are relevant for this thesis: Subsection 2.2.1 presents some adaptive algorithms like SATenstein (KhudaBukhsh et al., 2016) that inspired this thesis, while Subsection 2.2.2 gives an introduction into General Game Playing (GGP) frameworks, and specifically Ludii (Piette et al., 2020).

2.2.1 Adaptive Algorithms

Within the context of adaptive algorithm frameworks a broad distinction can be made between portfolio-based solution that primarily try to select the best algorithm for a given problem from a portfolio of existing algorithms, and modular/generative solutions that try to create novel algorithms by adapting and recombining individual parts of the algorithm.

Xu et al. (2008) described SATzilla, a portfolio-based framework to solve SAT (Satisfiability) problems. SATzilla uses machine-learning methods to select an existing SAT solver from its portfolio based on features of the given SAT problem instance. KhudaBukhsh et al. (2016) extended that idea to build SATenstein, a highly customizable and modular framework that can automatically generate SAT solvers by combining various algorithmic components of existing solvers. The SATenstein approach is a main inspiration for this thesis, which tries to do to search algorithms what SATenstein does to SAT solvers. In recognition to that, the title of this thesis, FrankenSearch, pays homage to SATenstein's reference to "frankensteining" solutions from existing parts.

In the field of game AI, various portfolio-based approaches have been investigated, enabling agents to handle a wide variety of games without manual tuning for each game. A notable example is the Hierarchical Portfolio Search (HPS) (Churchill & Buro, 2021), used in the strategy game Prismata. HPS was designed to manage the complex decision-making processes found in strategy games, where the state and action spaces are vast. By utilizing a portfolio of search algorithms, HPS reduces the search space by selecting a small subset of actions, which are then explored in-depth by algorithms like MCTS or $\alpha\beta$-search.

In General Game Playing (GGP; see Subsection 2.2.2 below), self-adaptive algorithms have been employed to dynamically adjust playing strategies during gameplay. Świechowski and Mańdziuk (2014) introduced a framework in which GGP agents autonomously select the best strategy from a portfolio based on the game's characteristics. This adaptive approach is particularly important in GGP, where agents must handle a variety of games without prior knowledge of the game's rules, making it essential for the agent to adjust its strategy in real time.

Bontrager et al. (2021) applied similar ideas to the General Video Game AI framework (GVGAI; see Subsection 2.2.2 below), analyzing the performance of different algorithms across varied game types. Game features are eventually matched to AI agents, setting a foundation for portfolio-selection in Video Game AI, improving general game-playing capabilities.

Beyond portfolio approaches, Cazenave (2024) presents a method for discovering exploration terms in MCTS algorithms. Monte-Carlo Search is used to generate and evaluate mathematical expressions for exploration terms in MCTS, showing that the discovered terms outperform standard ones. This thesis builds on such ideas in proposing a framework for generative algorithm adaptations.

2.2.2 General Game Playing

General Game Playing (GGP) (Pitrat, 1968) is an area within artificial intelligence (AI) focused on developing systems that can play a variety of games without any pre-programmed knowledge of their rules. Unlike specialized game players such as IBM's Deep Blue (Campbell et al., 2002), which was designed specifically for chess, GGP agents must be versatile, interpreting the rules of each new game autonomously and applying generalized strategies rather than game-specific tactics (Kowalski, 2016).

The Stanford GGP framework, central to this research area, introduced a formal language known as the Game Description Language (GDL) (Genesereth et al., 2005). GDL encodes games using a logical structure that specifies game states, legal moves, and win conditions, allowing agents to interpret and interact with these descriptions. This design supports the broader aim of GGP to develop systems that perform well across diverse and dynamic environments. The AAAI's[1] open competitions in GGP have further driven innovation in this field (Genesereth et al., 2005).

Another approach is the General Video Game AI (GVGAI) competition framework with the Video Game Description Langauge (VGDL) to represent games (Perez-Liebana et al., 2019). GVGAI extends the GGP principles to video games, where unpredictability and real-time requirements demand even greater flexibility from GGP systems. More than board games, video games often introduce elements of partial information and dynamic environments, requiring GGP agents to respond swiftly and adaptively to unforeseen changes.

Ludi and Ludii

Ludi (Browne, 2011a) is a system designed to model, analyze, and automatically generate abstract games. One of its core ideas is the ability to generate new games by recombining simple, modular components known as *ludemes*. This approach allows Ludi to not only analyze existing games but also evolve novel games through a process of game design automation. The game *Yavalath* (Browne, 2011b) is one of the best-known examples of this capability, showcasing Ludi's potential to create games with strategic depth from modular components.

Building on the success of Ludi, the Ludii system (Piette et al., 2020) expands the ludemic approach into a comprehensive framework for general game playing (GGP). While Ludi focused on generating new games, Ludii is designed to model, play, and analyze a wide range of traditional strategy games. By using high-level ludemes, Ludii provides a concise and modular way to represent diverse game rules and mechanics, making it more flexible and computationally efficient than previous systems like the Game Description Language (GDL).

A key advantage of Ludii is the simplicity of its game descriptions. For instance, a game like Tic-Tac-Toe can be defined with just a few lines of code, as shown in Figure 2.2. In contrast, GDL requires far more verbose logic to describe even simple games. This ludemic model makes Ludii a powerful tool not only for game modeling and AI research but also for adaptive algorithm testing.

```
(game "Tic-Tac-Toe"
    (players 2)
    (equipment {
        (board (square 3))
        (piece "Disc" P1)
        (piece "Cross" P2)
    })
    (rules
        (play (move Add (to (sites Empty))))
        (end ("Line3Win"))
    )
)
```

Figure 2.2: Ludii description of Tic-Tac-Toe.

Ludii's concise, Lisp-like structure—where games are described in nested symbolic expressions—aligns with its modular approach. This structure allows for flexibility and recursive

[1]Association for the Advancement of Artificial Intelligence

representations, similar to how Lisp handles symbolic computation. For more details on the language design principles and how they influence the Search Algorithm Description Language (SADL), refer to Section 2.3.

In this thesis, Ludii serves as a testbed for evaluating new search algorithms generated through SADL. Its ludemic approach, which allows modular and flexible game descriptions, is a key inspiration for SADL's design. SADL aims to decompose and recombine search algorithms in a similarly modular fashion, enabling the creation of adaptive algorithms that can be tested across a range of games.

2.3 Language Design

The design of formal languages, whether for programming, algorithm description, or other use-cases, follows principles that ensure clarity, extensibility, and efficiency. This section introduces relevant language design principles, focusing on programming paradigms, grammar, modularity, and the influence of languages like Lisp and Prolog on symbolic computation.

2.3.1 Programming Paradigms

Programming paradigms are fundamental styles of programming that guide the structure and elements of computer programs. Two primary paradigms are *imperative programming* and *declarative programming*, each offering different approaches to problem solving (Sebesta, 2012).

Imperative Programming

Imperative programming focuses on *how* to perform computations. It describes algorithms as sequences of explicit commands that change a program's state through assignments, loops, and control structures. Languages such as C, Java, and Python are predominantly imperative. This paradigm is characterized by:

- **State Mutations**: Variables represent memory locations that can be modified.
- **Control Flow**: The order of execution is defined by the programmer through constructs like conditionals and loops.
- **Procedural Abstraction**: Code is organized into procedures or functions that encapsulate specific tasks.

Imperative Languages therefore provide fine-grained control over program execution, suitable for performance-critical applications.

Declarative Programming

Declarative programming focuses on *what* the desired outcome is, rather than detailing the steps to achieve it. It abstracts away the specific control flow, allowing the programmer to specify the logic of computation without describing its control flow. Declarative programming encompasses several sub-paradigms, including functional programming and logic programming, which are further outlined below.

In all cases, declarative languages allow for more abstract and concise expressions of logic and computation, often leading to shorter and clearer code.

Functional Programming Functional programming treats computation as the evaluation of mathematical functions and avoids changing state or mutable data (Hudak, 1989). Notable characteristics are:

- **Immutability**: Data structures are immutable; once created, they cannot be altered.
- **First-Class Functions**: Functions are treated as first-class citizens and can be passed as arguments or returned from other functions.
- **Recursion**: Iterative processes are often expressed recursively.

Languages like Haskell and ML are functional languages, with theoretical foundations in lambda calculus, introduced by Church (1941).

Logic Programming Logic programming is based on formal logic, where programs consist of logical statements, and computation is performed through inference. Common properties include:

- **Facts and Rules**: Programs are written as a set of facts and inference rules.
- **Goal Resolution**: Computation involves proving queries or goals using the facts and rules.
- **Unification and Backtracking**: The language engine attempts to satisfy goals through pattern matching and backtracking over possible solutions.

Prolog (Programming in Logic) is a prominent logic programming language that exemplifies these concepts (Clocksin & Mellish, 2003).

2.3.2 Formal Grammar

A formal grammar defines the structure of valid expressions within a language. It uses a set of production rules to generate strings of symbols, representing both code and data. In computer science, formal grammars such as *Context-Free Grammars* (CFGs) are widely used to define programming languages (Chomsky, 1957). Knuth (1964) introduced Backus-Naur Form (BNF), a notation for expressing CFGs, which has become standard in language specifications. Three goals of a well-designed grammar are:

- **Expressiveness**: Ability to represent complex ideas concisely.
- **Simplicity**: Avoiding unnecessary complexity to facilitate understanding and maintenance.
- **Parsability**: Enabling efficient parsing by compilers or interpreters.

Modularity is another key principle of many languages, allowing complex systems to be broken down into smaller, reusable components. Object-oriented languages like C++ and Java exemplify this by structuring code around classes and objects (Stroustrup, 1997).

2.3.3 Symbolic Languages and Lisp

Symbolic programming languages focus on the manipulation of symbols and symbolic expressions. *Lisp* (LISt Processing), introduced by McCarthy (1959), is one of the earliest high-level programming languages and has been influential in the field of artificial intelligence (AI) due to its capabilities in symbolic reasoning.

Lisp is considered a multi-paradigm language, supporting both imperative and functional programming styles. Key features include:

- **S-expressions and Uniform Syntax**: The fundamental data structure in Lisp is the *S-expression* (symbolic expression; details below), representing both code and data as nested lists (Graham, 1993). This uniform representation enables Lisp programs to manipulate their own code structure directly and flexibly, as code is represented using the same list structure as data.
- **Dynamic Typing**: Types are associated with values at runtime rather than variables at compile time.

These characteristics make Lisp particularly suited for recursive algorithms, tree-based computations, and rapid prototyping.

S-Expressions

As defined by McCarthy (1959), an *S-expression* (symbolic expression) is either

1. an atom of the form `X`, or
2. an expression of the form `(X . Y)`, where `X` and `Y` are S-expressions.

Modern S-expression notations allow for expressions of more than two underlying S-expressions, for example `(X Z Y)` (as an abbreviated version of `(X . (Y . (Z . ())))`). Therefore, it can simply be said that an S-expression is either an atom, or a parenthesized list of S-expressions.

Prefix Notation and Tree Representation

Lisp uses *prefix notation* in its S-expressions, also known as Polish notation, where the operator precedes its operands. For example: `(+ 3 4)`
This notation aligns naturally with tree representations of expressions:

- The `+` symbol is the root node.
- The numbers `3` and `4` are child nodes.

This structure is therefore useful for representation in abstract syntax trees (ASTs), which are central in compilers and interpreters. It also simplifies the implementation of recursive algorithms, as each sub-expression can be treated as a tree itself.

Lisp's use of S-expressions as its surface syntax unlocks powerful metaprogramming techniques, such as macros, which enable programmers to create new syntactic constructs in ways not easily possible in other languages (Abelson et al., 1996). By representing code directly as lists, Lisp allows programs to inspect and modify their own structure at runtime, making it uniquely suited for tasks that benefit from introspection and manipulation of program structure.

2.3.4 Applications in AI and Search

Languages like Lisp and Prolog have been instrumental in the development of AI due to their capabilities in symbolic computation and reasoning. Lisp's flexibility and metaprogramming capabilities facilitate the manipulation of symbolic information and the implementation of complex data structures. As early as the 1960s, Lisp was used in the development of expert systems, automated theorem proving, and heuristic search algorithms (Nilsson, 1998).

Prolog's logic programming paradigm enables powerful inference mechanisms through its declarative nature, making it suitable for knowledge representation, reasoning, and solving problems with logical constraints (Norvig, 1992).

2.4 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a class of optimization techniques inspired by the principles of natural evolution, such as selection, mutation, and recombination (Eiben & Smith, 2015). They are commonly applied in domains that involve complex optimization problems, where traditional deterministic methods struggle due to high-dimensionality, non-linearity, or lack of analytical gradients. EAs leverage populations of candidate solutions that iteratively over generations to gradually improve the quality of solutions, making them particularly useful for adaptive and generative frameworks.

Subsection 2.4.1 briefly describes the core principles of EAs, while Subsection 2.4.2 distinguishes some types of EAs and how they are used for this thesis (see Subsection 4.1.2 for the FrankenSearch experiments using EAs).

2.4.1 Principles of Evolutionary Algorithms

EAs operate by mimicking biological evolution. The process generally involves four main stages:

1. **Initialization**: A population of candidate solutions, often represented as chromosomes (strings of genes), is randomly initialized. These candidate solutions can represent anything from simple binary strings to complex data structures.

2. **Selection**: Individuals from the population are selected to participate in creating offspring. Selection methods, such as roulette-wheel selection, tournament selection, or rank-based selection, favor individuals with higher fitness scores. Fitness here represents the quality of a solution concerning the objective function.

3. **Genetic Operators**: New individuals are generated from selected parents using genetic operators, primarily *crossover* and *mutation*:

    (a) **Crossover** involves combining genes from two parent solutions to produce offspring, introducing diversity while preserving the structure of the parents.

    (b) **Mutation** applies random changes to individual genes, which helps to explore the search space and avoid premature convergence.

4. **Replacement**: After creating a new generation of solutions, some or all of the previous population is replaced. The specifics of different replacement strategies influence the algorithm's exploration-exploitation balance.

This iterative process continues until a termination criterion, such as a maximum number of generations or a target fitness level, is met.

2.4.2 Types and Applications of Evolutionary Algorithms

Among the various forms of EAs, Genetic Algorithms (GAs) and Genetic Programming (GP) are of particular interest. Genetic Algorithms (GAs) (Holland, 1992) are the most widely used variant of EAs. They represent candidate solutions as strings of genes, often binary, and apply crossover and mutation to evolve solutions over generations. GAs are particularly useful in combinatorial optimization tasks and have been applied extensively in fields such as scheduling, routing, and feature selection (Goldberg & Holland, 1988).

Genetic Programming (GP) extends the concept of EAs to the evolution of entire computer programs (Koza, 1994). Instead of simple strings, candidate solutions in GP are represented as tree structures, with nodes in these trees corresponding to program functions or operations. This representation makes GP particularly well suited for evolving complex structures like Abstract Syntax Trees (ASTs), aligning closely with the objectives of this thesis. By manipulating ASTs, GP enables the generation of novel algorithms in the Search Algorithm Description Language (SADL), where each node in the AST corresponds to a component or parameter in a search algorithm. This approach allows FrankenSearch to autonomously generate adaptive search algorithms tailored to specific problem domains (see Subsection 4.1.2).[2]

[2]The source code for the SADL and FrankenSearch implementation can be found at https://github.com/lelhaag/FrankenSearch

Chapter 3

Framework Design and Implementation

This chapter provides details on the Search Algorithm Description Language (SADL) and its implementation. To ensure it is easily extendable and adaptable to different implementations or programming languages, the framework operates in three layers with carefully chosen limited interfaces. The decomposition of existing search algorithms and their representation in the newly proposed SADL are entirely independent of any specific implementation. These are detailed in Sections 3.1 and 3.2. The next step involves transforming the SADL representation of an algorithm into executable code. For this thesis, Java has been selected as the underlying programming language, but other implementations are also feasible. Section 3.3 describes the Java-based implementation of the SADL interpreter, which at this stage remains agnostic to any specific problem domain or implementation. Finally, the framework must be integrated with search problems. For this thesis, the Ludii GGP framework is used to apply and evaluate the framework. This integration is outlined in Section 3.4.[1]

3.1 Search Algorithm Decomposition

As discussed in Section 2.1, we initially focus on best-first search (BFS) algorithms for two-player, perfect-information games. Both Monte-Carlo Tree Search (MCTS) and Proof-Number Search (PNS), as described in Subsections 2.1.2 and 2.1.3, respectively, work by iteratively executing specific steps: *Selection*, *Expansion*, *Simulation* (in the case of MCTS), and *Backpropagation.* This provides a blueprint to construct BFS algorithms by combining these core elements.

3.1.1 Common Components

To construct BFS algorithms in this framework, we propose the following abstract components:

- **Selection:** In each iteration, starting from the root node, a child node is iteratively chosen until a leaf node is reached. The specific selection policy depends on the individual algorithm instance.

[1]The source code for the SADL and FrankenSearch implementation can be found at https://github.com/lelhaag/FrankenSearch

- **Evaluation:** Once a terminal state is reached, an evaluation is executed. This may include an external evaluation function. In MCTS, this involves performing a playout, while in PNS, it involves checking for a proof or disproof of the goal.
- **Backpropagation:** After the evaluation, values are backpropagated through the tree, from the leaf node back to the root.
- **Final Move Selection** (optional): A policy for final move selection can be defined. In MCTS, this often involves selecting the child node with the highest visit count. If not explicitly provided, the selection policy used in the *Selection* block can also be applied here.

The *Expansion* step is omitted in this definition. When a leaf node is reached, checking whether it is a terminal state, and expanding its children if it is not, can be done in the same way for any algorithm. Therefore, this process is part of the higher-level implementation (described in Subsection 3.3) and does not need to be explicitly defined in SADL, simplifying its structure.

3.2 Search Algorithm Description Language

The Search Algorithm Description Language (SADL) is designed to offer a flexible and modular way to encode search algorithms. SADL's syntax is inspired by Lisp-like languages, using symbolic expressions (S-expressions) to define the structure of algorithms. Subsection 3.2.1 provides more details on these design principles. Afterwards, Subsection 3.2.2 presents the syntax, as well as an explanation of the semantics of SADL. A comprehensive exmaple of a SADL representation can be found in Subsection 3.2.3

3.2.1 Design Principles

SADL draws inspiration from Lisp in its use of S-expressions for representing both code and data uniformly (see Subsection 2.3.3). The syntax consists of parenthesized expressions, where the first element is an operator or function name, followed by its arguments. This uniform and nested structure naturally represents the hierarchical components of many search algorithms.

While SADL uses Lisp-like syntax, it incorporates imperative constructs to make algorithm execution explicitly controllable. Declarative constructs are used for larger core elements to maintain conciseness and readability (e.g., `Selection`, `Backpropagation`), but imperative features such as variable assignments and conditional statements are included to allow precise control and modifications of the algorithm's behavior. With this hybrid approach SADL remains expressive and lean while providing the necessary control to easily define complex search algorithms.

3.2.2 Syntax and Semantics

The syntax of SADL is designed to be both expressive and easily parsable. It uses a minimal set of constructs to define the components of search algorithms using S-expressions. Following is an introduction of the syntax elements and an explanation of their semantics. Extended Backus-Naur Form (EBNF) (Scowen, 1993) is used to describe parts of the grammar. In EBNF notation, symbols are interpreted as follows:

- `<...>` enclose non-terminal symbols, which are syntactic categories (or grammar classes) that are defined in terms of other symbols.

- `"..."` (double quotes) enclose terminal symbols, which represent exact characters or strings that must appear in the language as specified.
- $\rightarrow$ is the defining symbol, indicating that the element on the left side is defined by the elements on the right side.
- | separates alternatives in a rule, meaning that any one of the listed options can satisfy the rule.
- {...} denotes zero or more repetitions of the enclosed item.
- `[...]` denotes an optional item, meaning the enclosed content may appear zero or one time.

Below each block of grammar classes is a brief explanation of their semantics (a cohesive overview of the entire grammar can be found in Appendix A.1).

Program Structure

```
            ⟨Program⟩ → ⟨AlgorithmDefinition⟩
⟨AlgorithmDefinition⟩ → "(" "SearchAlgorithm" ⟨String⟩ {⟨Declaration⟩} {⟨Component⟩} ")"
        ⟨Declaration⟩ → "(" "Define" ⟨Identifier⟩ ⟨Expression⟩ ")"
          ⟨Component⟩ → ⟨Selection⟩
                      | ⟨Evaluation⟩
                      | ⟨Backpropagation⟩
                      | ⟨FinalMoveSelection⟩
          ⟨Selection⟩ → "(" "Selection" [⟨String⟩] {⟨Statement⟩} ")"
         ⟨Evaluation⟩ → "(" "Evaluation" [⟨String⟩] {⟨Statement⟩} ")"
    ⟨Backpropagation⟩ → "(" "Backpropagation" [⟨String⟩] {⟨Statement⟩} ")"
 ⟨FinalMoveSelection⟩ → "(" "FinalMoveSelection" [⟨String⟩] {⟨Statement⟩} ")"
```

A SADL program consists of an algorithm definition, specified using the `SearchAlgorithm` keyword. `<String>` represents the name of the algorithm, which does not affect the execution, but can be used to identify and distinguish different algorithm instances.
Using the `Define` command, optional global variable can be declared, that are accessible throughout the algorithm.
The `<Component>` class represents the key stages of the searach algorithm (see Subsection 3.1.1). `<Selection>`, `<Evaluation>`, and `<Backpropagation>` are mandatory in a `SearchAlgorithm` definition, while `<FinalMoveSelection>` is optional. each component can also optionally be given a name, and otherwise consists of a list of statements.

Statements

```
 ⟨Statement⟩ → ⟨Assignment⟩ | ⟨Condition⟩ | ⟨SelectNode⟩
⟨Assignment⟩ → "(" "Set" ⟨Identifier⟩ ⟨Expression⟩ ")"
 ⟨Condition⟩ → "(" "Condition" ⟨Expression⟩ {⟨Statement⟩} ")"
⟨SelectNode⟩ → "(" "SelectNode" ("argmax" | "argmin") ⟨Expression⟩ ")"
```

Statements define the control flow within each core component:

- `Set`: Assigns a value to a variable. If the `<Identifier>` was declared as a global variable using `Define` in the outermost layer layer, then that global variable is updated, otherwise the `<Identifier>` will represent a local variable at the current node in the search tree.

- `Condition`: Evaluates a logical expression and executes a list of statements if true. The `<Expression>` must resolve to a boolean value.

- `SelectNode`: Returns the child node with the highest or lowest value of a given `<Expression>` (must resolve to a numeric value). Local variables in expressions usually relate to the node that the algorithm is currently processing. So it is noteworthy, that here the local variables would relate to all the child nodes.

Expressions

```
         ⟨Expression⟩ → ⟨Literal⟩ | ⟨FunctionCall⟩
       ⟨FunctionCall⟩ → “(” ⟨Operator⟩ {⟨Expression⟩} “)”
                      | “(” “Aggregate” ⟨AggregationFunction⟩ ⟨Expression⟩ “)”
                      | “(” “Parent” ⟨Expression⟩ “)”
                      | “(” “ExternalFunction” ⟨String⟩ ⟨Expression⟩ “)”
           ⟨Operator⟩ → “+” | “-” | “*” | “/” | “eq” | “neq” | “lt” | “gt” | “lte” | “gte”
                      | “and” | “or” | “not” | “log” | “sqrt”
⟨AggregationFunction⟩ → “min” | “max” | “sum” | “avg”
```

Expressions are used to compute and represent values, and consist of literals (see below) and function calls:

- `Operator`: These are basic logical or numerical operations that can be applied to expressions. Table 3.1 shows the semantically valid number and type(s) of input expression(s), as well as the type each operator resolves to. Most listed operators behave as commonly expected, merely the division operator (“/”) in SADL is modified as to not error on division by zero, instead returning the dividend (in other words, division by one). This allows SADL expressions to remain lean while being able to be processed in all situation, for instance, when nodes are visited for the first time, and variables have not been set before, defaulting to zero.

- `Aggregate`: Allows for aggregation of values of child nodes. It is separate from the operators, because, like with `SelectNode`, expressions in aggregations relate to the child nodes, as opposed to the currently executed node.

- `Parent`: Allows to access local variables of the parent node.

- `ExternalFunction`: Calls an external function with a name provided as a string and an expression as input, and resolves to the return value. In the context of tree search this meant for an evaluation function. The respective external function has to be provided to FrankenSearch as input in order for the SADL to use it.

Literals

```
           ⟨Literal⟩ → ⟨Number⟩ | ⟨Boolean⟩ | ⟨Constant⟩ | ⟨PredefinedVariable⟩ | ⟨Identifier⟩
          ⟨Constant⟩ → “inf” | “unknown” | “maxNode” | “minNode” | “orNode” | “andNode”
⟨PredefinedVariable⟩ → “node” | “nodeType” | “numChildren” | “depth”
        ⟨Identifier⟩ → ⟨Letter⟩ {⟨Letter⟩ | ⟨Digit⟩ | “_”}
            ⟨Number⟩ → [“-”] ⟨Digit⟩ {⟨Digit⟩} [“.” ⟨Digit⟩ {⟨Digit⟩}]
           ⟨Boolean⟩ → “true” | “false”
            ⟨Letter⟩ → “A” | “B” | ... | “Z” | “a” | “b” | ... | “z”
```

Table 3.1: Operators in SADL with Input and Result Types

Operator	Input Count	Input Type(s)	Result Type
`+`	2	Numeric	Numeric
`-`	2	Numeric	Numeric
`*`	2	Numeric	Numeric
`/`	2	Numeric	Numeric
`eq`	2	Numeric or Boolean	Boolean
`neq`	2	Numeric or Boolean	Boolean
`lt`	2	Numeric	Boolean
`gt`	2	Numeric	Boolean
`lte`	2	Numeric	Boolean
`gte`	2	Numeric	Boolean
`and`	2	Boolean	Boolean
`or`	2	Boolean	Boolean
`not`	1	Boolean	Boolean
`log`	1	Numeric	Numeric
`sqrt`	1	Numeric	Numeric

$\langle$Digit$\rangle \rightarrow$ "0" | "1" | ... | "9"
$\langle$String$\rangle \rightarrow$ """ {$\langle$Identifier$\rangle$} """

Literals can be numbers, booleans, or variables in the following form:

- **Constants**: Represent a constant value that is not a boolean or number.
  - `inf`: Represents infinity as commonly used for Proof-Number Search (PNS).
  - `unknown`: Used in Proof-Number Search as a proof value (next to `true`, and `false`).
  - `maxNode`, `minNode`, `orNode`, `andNode`: Represent node types. Note that `maxNode` and `orNode`, as well as `minNode` and `andNode` are equivalent, respectively. SADL allows for both notations in order to feel natural to both MCTS-based, and PNS-based algorithms.
- **PredefinedVariable**: Node-specific variables that exist for all nodes, and cannot be modified using the `Set` command:
  - `node`: References the entire node. Can be used, for instance, as an input for an external evaluation function.
  - `nodeType`: The node type (`maxNode`/`orNode`, `minNode`/`andNode`) is automatically assigned to each node during creation of the game tree.
  - `numChildren`: The number of children for each node.
  - `depth`: The depth of the respective node in the search tree.
- **Other Identifier**: Any other identifier can refer to either a global variable (if declared at the top of the algorithm), or else a local variable at each node. If a local variable was not previously set at a node, the value will default to zero, when called.

3.2.3 SADL Example: Monte-Carlo Tree Search

Figure 3.1 shows a SADL representation of Monte-Carlo Tree Search (MCTS). It begins with the `SearchAlgorithm` definition, naming the algorithm `"MCTS"`. Two global variables are defined: `C` (the exploration constant for UCT, see Subsection 2.1.2) and `value` (used to store results of an MCTS evaluation).

The `Selection` phase follows UCT, using the `SelectNode` statement to select the child node with the highest UCT value. In the `Evaluation` phase, a leaf node is evaluated using the external `"mctsEval"` function, which performs a random playout to estimate the node's value.

In the `Backpropagation` phase, the `valueEstimate` of each node is updated based on the new data from the last simulation stored in `value`. Both `value` and `valueEstimate` are always interpreted from the maximizing player's perspective across all nodes, which is why the `Selection` phase conditionally inverts `valueEstimate` for `minNode` types.

The `FinalMoveSelection` phase selects the child with the highest `visitCount` as the final decision. If no `FinalMoveSelection` is provided, the strategy in `Selection` will be used to determine the final move choice.

```
(SearchAlgorithm "MCTS"
  (Define C 1.4)
  (Define value 0)
  (Selection "UCT"
    (Condition (eq nodeType maxNode)
      (SelectNode argmax
        (+ valueEstimate
            (* C (sqrt (/ (log (Parent visitCount)) visitCount)))
        )
      )
    )
    (Condition (eq nodeType minNode)
      (SelectNode argmax
        (+ (- 0 valueEstimate)
            (* C (sqrt (/ (log (Parent visitCount)) visitCount)))
        )
      )
    )
  )
  (Evaluation
    (Set value (ExternalFunction "mctsEval" node))
  )
  (Backpropagation
    (Set valueEstimate (+ valueEstimate (/ (- value valueEstimate) visitCount)))
  )
  (FinalMoveSelection
    (SelectNode argmax visitCount)
  )
)
```

Figure 3.1: SADL representation of Monte-Carlo Tree Search (MCTS)

3.3 General Search Algorithm Framework

The Search Algorithm Description Language (SADL) as an implementation-agnostic way to represent search algorithms is described in the previous Section 3.2. For this thesis, Java has been chosen to implement the SADL interpretation and execution. The object-oriented nature of Java aligns well with FrankenSearch's core idea of creating algorithms as combinations and modifications of existing building blocks. While not quite as fast as C++, Java still has sufficiently high performance (especially compared to Python (Pereira et al., 2017), which is increasingly popular in AI research), and is widely used. Finally, the Ludii General Game Playing (GGP) framework, which is used to test our implementation, is also written in Java, making the integration (see Section 3.4) simple.

Subsection 3.3.1 briefly describes the lexing, parsing, and compilation of the SADL into executable code. Subsection 3.3.2 details node representation in the search tree and other data handling, and, finally, Subsection 3.3.3 outlines the general algorithm execution.

3.3.1 SADL Compilation Pipeline

The process pipeline turning the SADL representation of a search algorithm into executable Java code can be generally divided into three process modules with respective input and output data: The *Lexer*, *Parser*, and *Compiler*. Figure 3.2 provides a simplified overview of the steps, which are explained below.

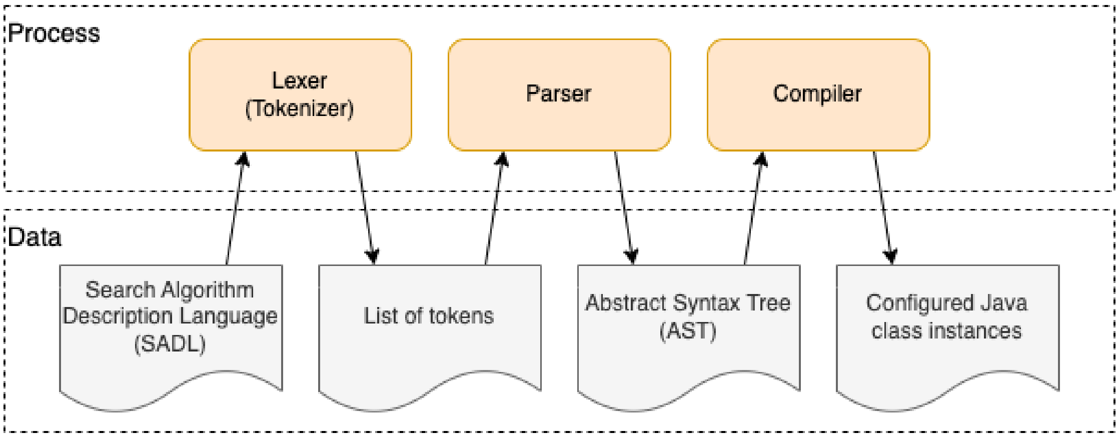

Process
Lexer (Tokenizer)
Parser
Compiler
Data
Search Algorithm Description Language (SADL)
List of tokens
Abstract Syntax Tree (AST)
Configured Java class instances

Figure 3.2: Pipeline steps turning the Search Algorithm Description Language (SADL) into executable code

Lexer

The *Lexer* or *Tokenizer* takes the SADL representation of an algorithm as a string-input and turns it into an ordered list of tokens in order to simplify subsequent parsing. A token in this implementation consists of a `value` (the original string representing the token from the SADL input), and a token `type`, which is one of: `NUMBER`, `BOOLEAN`, `NAME`, `SYMBOL`, `LEFT_PAREN`, `RIGHT_PAREN`, or `EOF`.

The token types largely match the atom types of the SADL syntax described in Subsection 3.2.2; the only addition are `LEFT_PAREN` and `RIGHT_PAREN` to identify the nested expressions, as well as `EOF` to indicate the end of the input stream.

At least one space-character is needed to separate tokens in the SADL (except parentheses), but beyond that all white-spaces are ignored/removed during the lexing process; line-breaks and indents are meaningless for SADL. Following is an example of the lexer processing. Input string of SADL component to the lexer:

```
(Selection "UCT"
  (SelectNode argmax
    (+ valueEstimate (* 1.4 (sqrt (/ (log (Parent visitCount))
        visitCount))))
  )
)
```

Output list of tokens:

```
(‘(’, LEFT_PAREN),
(‘Selection’, SYMBOL),
(‘UCT’, NAME),
(‘(’, LEFT_PAREN),
(‘SelectNode’, SYMBOL),
(‘argmax’, SYMBOL),
(‘(’, LEFT_PAREN),
(‘+’, SYMBOL),
(‘valueEstimate’, SYMBOL),
(‘(’, LEFT_PAREN),
(‘*’, SYMBOL),
(‘1.4’, NUMBER),
(‘(’, LEFT_PAREN),
(‘sqrt’, SYMBOL),
(‘(’, LEFT_PAREN),
(‘/’, SYMBOL),
(‘(’, LEFT_PAREN),
(‘log’, SYMBOL),
(‘(’, LEFT_PAREN),
(‘Parent’, SYMBOL),
(‘visitCount’, SYMBOL),
(‘)’, RIGHT_PAREN),
(‘)’, RIGHT_PAREN),
(‘visitCount’, SYMBOL),
(‘)’, RIGHT_PAREN),
(‘)’, RIGHT_PAREN),
(‘)’, RIGHT_PAREN),
(‘)’, RIGHT_PAREN),
(‘)’, RIGHT_PAREN),
(‘)’, RIGHT_PAREN),
(‘’, EOF)
```

Parser

The parser takes the list of tokens as an input and turns it into an abstract syntax tree (AST), thereby representing the nested hierarchical structure of the algorithm components in a natural tree structure. `LEFT_PAREN`, `RIGHT_PAREN` and `EOF` tokens guide the shape of the tree and disappear in the process, all other tokens turn into nodes of the AST. An AST node in turn consists of the `value` of the underlying token, an AST node `type` of `NUMBER`, `BOOLEAN`, `NAME`, or `SYMBOL`, as well as a list of `children`, consisting of other AST nodes.

Unlike the lexer, the parser also checks for consistency and enforces some grammar rules. There will be errors, for instance, when the parentheses mismatch, or when unexpected tokens are encountered.

Continuing the example from above, the parser translates the previous token list into the AST visualized in Figure 3.3.

Compiler

A *compiler* takes a source program (usually in a high-level programming language) and turns it into an executable machine-language program, which can then in a separate step be called by the user and, e.g., process some input during runtime. An *interpreter*, on the other hand, interprets

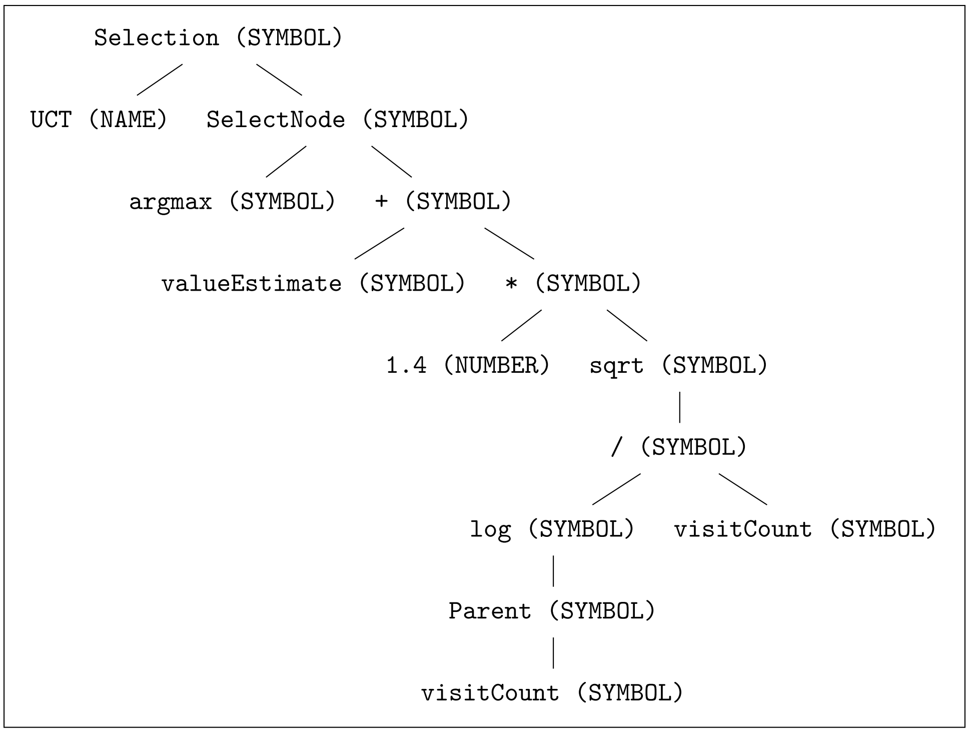


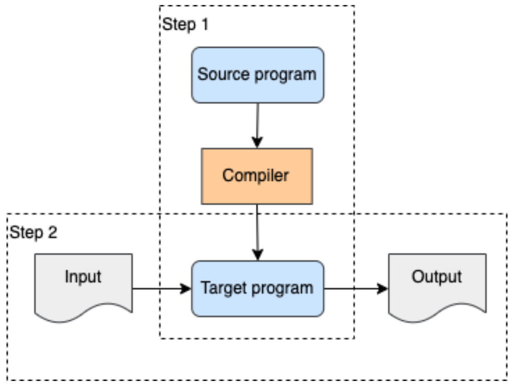
Figure 3.3: Visual representation of the abstract syntax tree (AST) of a SADL component

the source program during runtime and executes the instructions of the source program together with the user input (Aho et al., 2006). Figure 3.4 visualizes the different workflows.

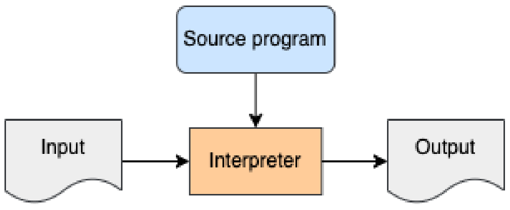
In FrankenSearch, the AST is turned into Java class instances during runtime, as opposed to a target program in an executable machine-language. It can therefore be argued that it is not technically a compiler, but an interpreter, since it is evaluated during runtime. For executing a search in a board game, when considering the current game state as input and the move to make as output, FrankenSearch does not – as a regular interpreter would – interpret the SADL (or its AST) in every move, but rather turns it into the Java classes once, during start-up, and then only executes those with the respective input of the search problem.

Because of the role it plays, and the two step process it follows (as in Figure 3.4a), we call this component simply *compiler*, when technically it is a hybrid-approach, similar to a *staged interpreter* (Wei et al., 2019).

The compiler follows the structure of the AST to build nested Java classes representing the different components of an algorithm. The top-level Java class is `ExecutableSearchAlgorithm`, representing an instance of any compilable search algorithm. It stores instances of the component classes (`ExecutableSelection`, `ExecutableEvaluation`, `ExecutableBackpropagation`, `ExecutableFinalMoveSelection`), each containing a list of (possibly nested) statements (`ExecutableSet`, `ExecutableCondition`, or `ExecutableSelectNode`). Finally, for expressions within these statements, instead of dedicated Java classes, the respective AST subtree is simply stored. See Subsection 3.3.3 for details about the execution of the algorithm in this structure.



(a) A compiler.



(b) An interpreter.

Figure 3.4: Comparison of compiler and interpreter workflows, adapted from Aho et al. (2006).

3.3.2 Data Representation

The core data object in FrankenSearch's SADL execution is the node in the search tree. The individual algorithm components are executed with a node as input and return a node as output (see Subsection 3.3.3). A Java class, `Node`, stores references to its parent node (except for the root node) and all child nodes (if applicable), effectively creating a search tree that is easily navigable in both directions. Furthermore, each node stores additional local node variables, including both predefined variables (such as depth and node type) and custom-defined variables in SADL (e.g., proof number and value estimates). Subsection 3.2.2 contains details on the variables used in SADL.

To apply SADL to a game (or any other search problem), state information is required within the nodes. However, this middle layer of SADL execution should remain independent from any specific application. Therefore, specific state representations are not included in the core `Node` class. Instead, for any particular use case, the framework requires the top layer to add this information, typically by extending the node into an inherited class containing domain-specific data (see Subsection 3.4 for details on this step in the Ludii integration).

In addition to local node variables, the algorithm can access global variables and external functions through a `GlobalVariableRegistry` and `FunctionRegistry`, respectively. Global variables can be declared using the `Define` statement in the outermost layer of the SADL code. Before execution, the compiler reads these declarations and creates the respective global variables in the registry, which is globally accessible from each node during execution. When a variable identifier is encountered during processing, the system first checks for its existence in the global registry before assuming it to be a local node variable. The external functions are possibly domain-specific (e.g., evaluation functions) and must be defined and added to the function registry in the outermost layer of the program architecture (see Figure 3.5).

3.3.3 Algorithm Execution

The main algorithm execution is controlled by the `GeneralBestFirstSearch` class, which takes as input the output of the SADL compilation pipeline (see Subsection 3.3.1), a fully configured `ExecutableSearchAlgorithm` instance. The `selectAction` method can be called with domain-specific information about state and context, as well as a resource constraint (maximum search time in seconds). The executor then creates a root node from the domain-specific input and enters the following loop until the search time is exhausted (or until the tree is fully solved, which is less relevant in practice):

Within a sub-loop, the current node is passed to the `ExecutableSearchAlgorithm`'s `Selection` component, which returns a child node based on the SADL-defined selection strategy. This process is repeated to navigate down the tree until a leaf node is reached. The found leaf node is then expanded by creating all its children based on the domain-specific state and context. The expanded node is passed to the `Evaluation` component, which executes the SADL-defined statements and then returns the updated same node. The node is subsequently passed to the `Backpropagation` component, which iteratively executes the SADL-defined statements while moving up the tree to the parent node and continues until reaching the root node. The output of `Backpropagation` is always the root node, which then re-enters the main loop. Once the time limit is reached, the root node is passed to the `FinalMoveSelection` component (or to the `Selection` component if no `FinalMoveSelection` is defined) for a single execution, which returns a child node of the root as the result of the overall action selection.

All these main components, as well as the statements they generally consist of (`ExecutableSet`, `ExecutableCondition`, and `ExecutableSelectNode`), extend a parent class, `ExecutableStatement`, which has an `execute` function that takes and returns a node. This design enables easy and flexible listing and nesting of different operations. The main components generally call the `execute` function on a list of child statements in order (and possibly perform additional processing, e.g., moving up the tree in the case of backpropagation). The `ExecutableSet` statement stores a variable identifier and an expression, and upon execution, assigns the evaluated expression to the variable (which might be global or local; see Subsection 3.3.2). The `ExecutableCondition` statement stores a boolean expression as a condition and a list of statements as its body. During execution, the body statements are executed if the condition is true. Finally, the `ExecutableSelectNode` statement stores a function (either `argmin` or `argmax`) and an expression. During execution, the expression is evaluated on all child nodes, and the "best" node, according to the specified minimization/maximization function, is returned.

Expressions, as described previously, are stored in the respective classes as abstract syntax trees (ASTs), which are subtrees of the original full AST created during compilation (see Subsection 3.3.1). An `ExpressionEvaluator` class takes an expression AST node as input and recursively resolves it to a final value (generally numeric or boolean), evaluating both numeric and logical operators as seen in Table 3.1, as well as any `Aggregate`, `Parent`, or `ExternalFunction` calls (see Subsection 3.2.2).

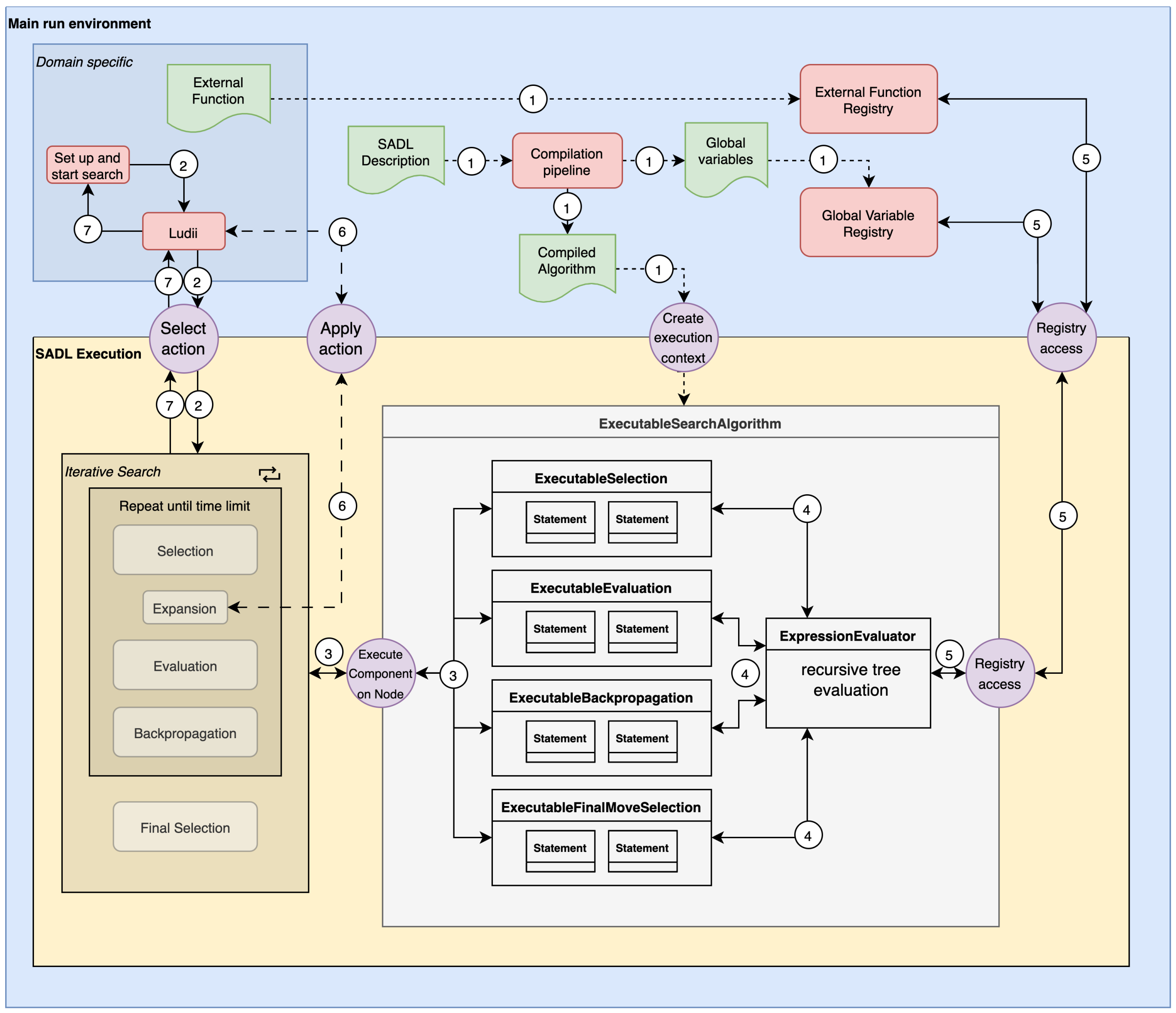
Figure 3.5 provides an overview of the process, highlighting the layered approach of the FRANKENSEARCH framework. The nested Java classes generated by the SADL compilation pipeline (see Subsection 3.3.1) are shown in gray. The fixed `GeneralBestFirstSearch` SADL execution context appears in the yellow middle layer, while the main run environment, including any domain-specific configuration, is represented in blue. The purple circles indicate the interfaces between the layers. This layered approach allows for individual layers to be easily replaced with alternative implementations while reusing the rest of the framework.

3.4 Integration with Ludii Framework

The descriptions of the general search algorithm framework in the previous sections are mostly domain-independent and designed to be easily integrable with any search problem context. The Ludii GGP framework (see Subsection 2.2.2) is used to apply FrankenSearch in this prototype. Agents can be registered in Ludii, by extending the `AI` class and overwriting the `selectAction` method (Soemers et al., 2020):

```
public abstract Move selectAction
(
    final Game game,
    final Context context,
    final double maxSeconds,
    final int maxIterations,
    final int maxDepth
);
```

For any domain specific application of the FrankenSearch framework, the `Node` class (as described in Subsection 3.3.2) can be extended. For the Ludii integration, the `LudiiNode` class extends `Node` and stores the game context, as well as the move from the parent in the search tree as Ludii specific `Context`, and `Move` objects, respectively. Leaf nodes are expanded by generating all legal moves in Ludii using the game context in the node, and adding children for each move, storing the new respective game context after applying the particular move. Custom evaluation functions (e.g., the random playout in MCTS) can be defined in the function registry, and can take the Ludii specific game context as input. Figure 3.5 visualizes the Ludii integration with the `selectAction` call as main interface.



1 Prepare for execution by setting up external functions and compiling the SADL description, which results in global variables and configures an *ExecutableSearchAlgorithm* instance within the SADL execution context.

2 Perform domain-specific preparations and initiate the search. Here, this involves setting up a game in the Ludii framework and registering the SADL execution as an agent to be called by the *selectAction* interface, with the game context and computational time limit as inputs.

3 The individual components of the compiled algorithm are called, each receiving a search tree node as input and returning a node. The first three components run in a loop until the time limit is reached; afterward, the final move selection is performed.

4 Expressions in statements within the algorithm components are passed to the *ExpressionEvaluator* as Abstract Syntax Trees (AST), where they are recursively evaluated.

5 During expression evaluation, global variables can be accessed and modified, and external function calls can be performed (with the game context passed to external functions as needed).

6 After Selection, the leaf node is expanded by generating all its children using domain-specific information. In this case, a game move is applied in Ludii, and game contexts are returned for each child.

7 Selected action returned to Ludii (which might repeatedly call *selectAction* within a game). Once the game is over, Ludii returns the final game result to the main call.

Figure 3.5: Overview of the FrankenSearch framework, executing a SADL-described search using Ludii.

Chapter 4

Experimental Setup and Evaluation

This chapter describes the evaluation of the FrankenSearch framework, in terms of both quantitative results when applied to games using the Ludii framework, as well as a qualitative assessment of the usability of SADL to describe algorithms. Section 4.1 details the experiment setup, defining general scopes and constraints, as well as describing how evolutionary algorithms are applied to generate new search algorithms. Afterwards, details for individual experiments are provided in Section 4.2.

4.1 Experimental Setup

The search algorithms described by SADL and implemented in the FrankenSearch framework, as detailed in Chapter 3, are applied and evaluated by playing board games using the Ludii framework. The general scope and constraints of these experiments are outlined in Subsection 4.1.1. Subsection 4.1.2 describes the implementation of evolutionary algorithms (EAs) used to generate and optimize search algorithms.

4.1.1 Scope and Constraints

As described in Section 1.2, the scope of the research is limited to two-player, deterministic, turn-taking, zero-sum, perfect-information games as the domain and focusing on best-first search algorithms.

Games

Lines of Action (LOA) is used for most of the experiments. It is a strategic board game where players aim to connect their pieces into a single group. LOA has a long history of use in AI research due to its strategic depth and balance between search and knowledge requirements, making it an ideal testbed for algorithmic studies (Winands, 2004; Kowalski et al., 2025). The typical board size is 8×8, but LOA can also be played on smaller boards, which is useful for obtaining faster results. Many experiments were conducted on a 6×6 board, with final verification typically performed on the 8×8 board. Matches generally consist of 1,000 games, alternating sides after half of the games to mitigate any potential first-mover advantage. The time constraint per move varies between 0.1 and 10 seconds. Specific parameters for each experiment are detailed in Subsection 4.2.

Implementation and Hardware

The FrankenSearch framework, as described in Section 3.3 and illustrated in Figure 3.5, is integrated with Ludii sources into a single `.jar` file for execution. A `main` function specifies the parameters for each experiment. Parallel processing is implemented using Java's concurrency utilities in order to run games in Ludii asynchronously. Experiments were conducted on two servers: one equipped with an AMD EPYC 7453 processor with 28 CPU cores and 250 GB memory, and another with an AMD EPYC-Rome processor offering 32 cores and 32 GB of memory.

4.1.2 Evolutionary Algorithm Implementation

The evolutionary algorithm implemented in the FrankenSearch framework uses Genetic Programming (Koza, 1994) to evolve search algorithms represented as Abstract Syntax Trees (ASTs), and follows the principles of EAs as described in Subsection 2.4.1.

The initial population is created by first loading predefined search algorithms from SADL files, a full list of which can be found in Appendix A.2. Mutated copies are added until the initial population size is reached. Individuals are evaluated based on their performance in a series of games using the Ludii framework. A Swiss-system tournament is employed to assess and rank individuals, ensuring that each individual competes against others with similar scores (for specific parameter values, see Subsection 4.2.3). The top half of the population is selected for the next generation, along with a subset of original algorithms to maintain diversity. Genetic operators are applied until the new population reaches the defined size:

- **Crossover:** This operator combines two parent ASTs to produce offspring. It involves selecting compatible nodes from each parent tree and swapping sub-trees to maintain syntactic correctness. Compatibility is determined based on the node type, e.g., numerical variables, numbers, main components, respectively, can be switched between ASTs. For operators, input and output types are checked to assess compatibility.
- **Mutation:** This operator introduces diversity by randomly altering parts of an individual's AST. Mutations can involve changing numerical values, replacing operators with compatible alternatives, or growing a sub-tree, by, e.g., replacing a numerical variable or number with an operator with numeric return type.

The core of the EA processing, including all relevant experiment parametrization is in the `EvolutionaryAlgorithm` class. Functions to perform crossover and mutations, including compatibility checks are outsourced into the `Mutation` class, while a `Utils` class contains operator metadata for compatibility checks as well as some helper functions for e.g., tracking individuals, logging results and file handling.

4.2 Experiments

The performed experiments broadly fall into two categories: experiments validating the general functionality and applicability of the SADL language and FrankenSearch framework can be found in Subsections 4.2.1 and 4.2.2; the application of evolutionary algorithms to generate and modify search algorithms is described in Subsection 4.2.3.

4.2.1 SADL Usability

This subsection describes qualitative evaluations of the usability of SADL to describe algorithms (independent of actual search). It starts of by probing the robustness of the error handling for SADL parsing and compiling, followed by an example of using SADL to describe a search algorithm variant.

SADL Error Handling

Based on the SADL description of MCTS (see A.2.1), slight variations that are syntactically or semantically invalid have been tried to parse, compile, and execute to probe the error handling. Following is a selection of cases, that showcase different layers of successful or failed error handling:

- **Example 1**
  - **SADL Part:** `(Define C 0.6 1.4)`
  - **Description:** Adding an extra parameter to the `Define` operator.
  - **Result:** Error in Parser: `java.lang.RuntimeException: Expected ), found 1.4`
- **Example 2**
  - **SADL Part:** `(Selection "UCT" ((Condition ...))`
  - **Description:** Adding extra parenthesis before Condition block.
  - **Result:** Error in Parser: `java.lang.RuntimeException: Unexpected token in Selection: Token{type=SYMBOL, value='('}`
- **Example 3**
  - **SADL Part:** `(Define C abc)`
  - **Description:** Assigning a string instead of a number to a global variable.
  - **Result:** Successful parsing, error in Compiler: `java.lang.NumberFormatException: For input string: "abc"`
- **Example 4**
  - **SADL Part:** `(Selection "UCT" (Condition ... ((SelectNode ...))`
  - **Description:** Adding extra parenthesis before `SelectNode` statement inside the `Condition` block.
  - **Result:** Successful parsing, error in Compiler: `java.lang.RuntimeException: Unexpected node: (`
- **Example 5**
  - **SADL Part:** `(Selection "UCT" (Condition ((eq ...))`
  - **Description:** Adding extra parenthesis before `eq` condition inside the `Condition` block.
  - **Result:** Successful parsing and compiling, error during execution: `java.lang.RuntimeException: Unexpected condition: (`

- **Example 6**
  - **SADL Part:** N/A
  - **Description:** Missing `Evaluation` block.
  - **Result:** Successful parsing and compiling, error during execution: `java.lang.NullPointerException: Cannot invoke "algos.ExecutableEvaluation.execute(algos.Node)" because the return value of "algos.ExecutableSearchAlgorithm.getEvaluation()" is null`
- **Example 7**
  - **SADL Part:** `(Set 123.456 (ExternalFunction "mctsEval" node))`
  - **Description:** Assigning numeric value to a number.
  - **Result:** Successful parsing, compiling, and execution; nodes get a variable named "123.456".

Ideally, invalid SADL descriptions should be caught as early as possible. Many checks, indeed, happen in the parser, Examples 1 and 2 from above show such instances. Examples 3 and 4 were successfully parsed, but failed in the Compiler, when the Java classes are configured. As described in Section 3.3, any mathematical expression within the SADL are stored as the pure parsed ASTs and then resolved only during execution in the `ExpressionEvaluator`. As a consequence, invalid expressions only lead to an error during execution, as seen in Example 3. While not ideal in terms of usability, this behavior was deliberately accepted to ease the implementation of this first proof-of-concept, a future version could include an expression validation step in the compiler. Examples 6 and 7, finally, are some oversights, as opposed to intentional design choices. A check for missing core components, as well as a check for numbers as variable names could easily be added in the parser or compiler.

PN-MCTS Description

SADL was designed specifically with MCTS and PNS in mind, so it is no surprise that those two search algorithms can be represented by SADL. As a next step, it was attempted to describe PN-MCTS (Doe et al., 2022; Kowalski et al., 2025) using SADL. PN-MCTS has three independent enhancements: *Final Move Selection*, *Solving Subtrees*, and *UCT-PN*.

As a first step, in order to use both PNS- and MCTS-based values, the `Evaluation` and `Backpropagation` components of both MCTS and PNS have to simply be combined, i.e., evaluating a node subsequently both using the PNS and MCTS external evaluation functions, and backpropagating all values. With that, the *Final Move Selection* can easily be described in SADL by just adding a `Condition` block, checking for a proven tree and following the lowest proof number (0) in that case, or selecting the most visited node (as per standard MCTS) otherwise:

```
(FinalMoveSelection
  (Condition (eq proofNumber 0)
    (SelectNode argmin proofNumber)
  )
  (Condition (neq proofNumber 0)
    (SelectNode argmax visitCount)
  )
)
```

The *Solving Subtrees* enhancement can similarly be implemented by having an extra nested `Condition` block, check for (dis-)proof numbers of 0, while also checking for the threshold T (Winands et al., 2008), which can be defined as a global variable at the top. The *UCT-PN* enhancement, finally, cannot immediately be represented by SADL, because of the ranking of children based on (dis-)proof numbers. While SADL supports operators such as min, max, argmin, argmax applied to all node children, it does not offer a full ranking. Such a function could easily be added, however. A.2.3 shows the full SADL description of the PN-MCTS with the first two enhancements.

4.2.2 Validation Experiments

Following are two sets of validation experiments for the two BFS algorithms evaluated in this thesis: Monte-Carlo Tree Search, and Proof-Number Search, respectively. A profiling of execution for both of them is discussed afterwards.

MCTS Validation

The validation of Monte-Carlo Tree Search within the FrankenSearch framework is demonstrated through two experiments: a performance comparison between SADL-MCTS (the SADL description can be found in Appendix A.2.1) and a plain Java implementation, and an analysis of the impact of the UCT exploration constant C.

The first experiment, illustrated in Figure 4.1, compares the win rates of SADL-MCTS and plain Java MCTS implementations on LOA for both 6×6 and 8×8 board sizes under various time-per-move constraints. The results indicate that SADL-MCTS performs almost equivalently to the plain Java MCTS across the majority of time settings between 0.5 and 10 seconds per move. This consistency demonstrates that the SADL framework does not introduce any significant inefficiencies or conceptual flaws in implementing MCTS. The sheer design of the framework as nested components (described in Section 3.3) is virtually guaranteed to add some overhead compared to a plain Java implementation, however, so it is worth pointing out that the usage of Ludii to play and evaluate the games is the biggest bottleneck. Handling the Ludii game state objects in each tree node adds significant overhead, so that the performance difference in actual search execution appears to be negligible in comparison.

Interestingly, for the shortest time constraint of 0.1 seconds per move, SADL-MCTS outperforms the plain Java implementation on both board sizes. However, this difference may not be particularly meaningful, as it could stem from minor overheads associated with initializing the classes in the plain Java implementation, or other implementation-specific nuances, given that 0.1 seconds is not enough time to build any substantial search tree. Overall, these results establish that MCTS implemented through the SADL framework in the FrankenSearch system functions effectively and conceptually mirrors the behavior of a standard Java implementation.

The second experiment, shown in Figure 4.2, examines the impact of varying the UCT exploration constant C on the performance of MCTS, as measured by win rates against the default setting of $C = 1.4$. The experiments were conducted on a 6×6 LOA board with time controls of 0.1 seconds, 1 second, and 5 seconds per move. The results reveal that the optimal value for C for LOA appears to be around $C = 0.6$, as MCTS consistently achieves its highest win rates at this setting across all tested time controls. The shape of the performance distribution is consistent for all time constraints, with win rates dropping drastically as C approaches zero. This behavior aligns with theoretical expectations, as $C = 0$ eliminates the exploration term in the UCT formula, leading to pure exploitation and suboptimal search behavior. For higher C values, performance gradually declines, likely due to an overemphasis on exploration at the

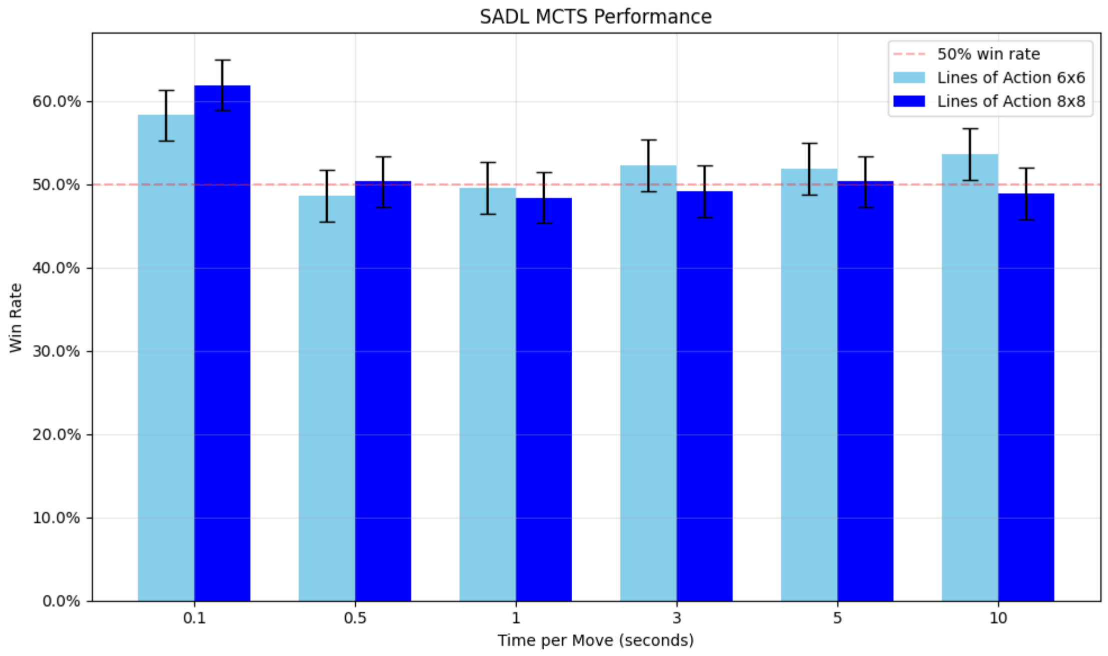


Figure 4.1: Win rate of SADL MCTS versus Ludii built-in plain Java implementation on LOA (6×6, and 8×8 board) for different time constraints on 1000 games (95% confidence interval indicated)

expense of exploitation. For shorter time controls (particularly 0.1 seconds per move), the drop in performance for C approaching zero is much less pronounced. This is expected since not much search can happen in this short time either way, so the results are closer to random. While flatter, the shape of the curve is still generally in line with the others, which gives confidence in using small time settings for further experiments, and still getting results that correlate with more meaningful results from experiments with longer time settings.

Based on this result, the value of $C = 0.6$ is also used for subsequent MCTS-based experiments that use the UCT formula. Trying different values for parameters such as C is standard procedure and can easily be tested in most MCTS implementations. Nonetheless, this experiment also serves as a validation of using SADL to experiment with algorithm versions, since algorithms with different C values have been created by changing the SADL description and then parsing and compiling it, as opposed to changing a variable in the compiled Java code. By manually or programmatically changing parts of the SADL description, more substantial changes than a simple parameter value can also be easily achieved.

PNS Validation

Next to MCTS, Proof-Number Search (PNS) was the other best-first search algorithm examined in detail in this thesis. Unlike MCTS, PNS is not designed to be used as an immediate game-playing engine from the start of the game. The core purpose of PNS is to prove or disprove specific positions, making it particularly useful for solving endgame situations. Applying PNS at the beginning of a game is either infeasible or prohibitively expensive due to the vast size of the search space. Furthermore, testing PNS against another implementation, as was done for MCTS,

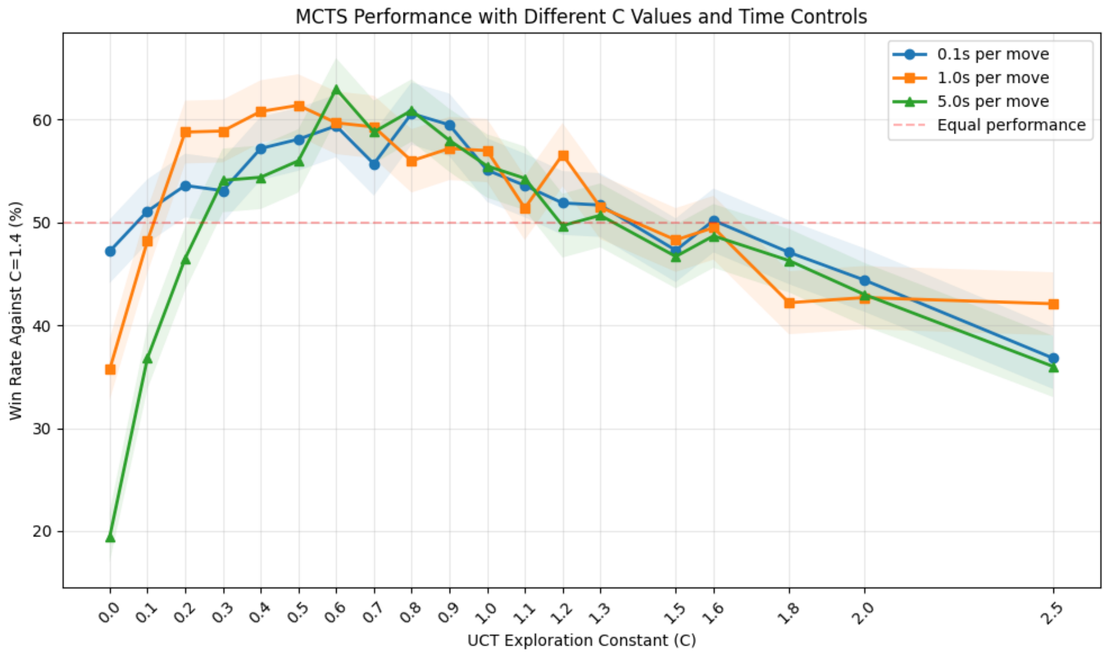


Figure 4.2: Win rate of MCTS with different values for the UCT exploration constant (C) against the default of $C = 1.4$ for different time controls in 1000 games of LOA (6×6 board) (95% confidence interval indicated)

is not meaningful since PNS does not rely on statistical performance or win rates. Instead, it provides definitive solutions to problems.

To validate the implementation of PNS in the FrankenSearch framework, we attempted to solve a curated set of 122 endgame positions in LOA (8×8 board). These positions were compiled by Winands (2004) and Winands and Uiterwijk (2001) as part of their early work on PNS. It contains positions created by him, as well as positions by other researchers.[1] The SADL description of PNS used in this experiment is provided in Appendix A.2.2. For each endgame position, PNS was applied with unlimited time constraint and hence was tasked with proving or disproving the outcome, and the following performance metrics were collected:

- **Time to solve**: The total time required to prove the endgame position.
- **Maximum search depth**: The deepest level reached in the search tree.
- **Nodes per second**: The average number of nodes evaluated per second during the search.
- **Memory usage**: The total memory required to store the search tree with all the game states.
- **Number of expansions**: The total number of PNS iterations required to solve the position.

[1]The full set of endgame positions can be found at https://dke.maastrichtuniversity.nl/m.winands/loa/endpos.htm

The experiments revealed that memory usage was a significant factor due to the integration with the Ludii framework. Each node in the search tree stores the full Ludii game context, leading to substantial memory consumption, particularly for deeper and more complex searches. Out of the 122 endgame positions, 80 have been successfully solved, whereas 42 failed due to memory error. The Java heap space was limited to 230 GB for those experiments. Average values for all 80 solved positions, as well as a selection of individual positions with particularly high or low values in any column are shown in Table 4.1.

The selected endgame positions have additionally been solved with PNS using the GameMaster 2.0 board-game playing environment developed by Winands (2004), which is also implemented in Java.[2] Table 4.2 shows the time to solve, and nodes per second, as well the speed-up factor of nodes per second between the GameMaster 2.0 and FrankenSearch solvings of the selected positions. The GameMaster 2.0 searches through up to 26 times more nodes per second compared to the FrankenSearch implementation, the factor varies significantly, however, between endgame positions. For FrankenSearch it can be observed that the nodes/s go down when the search complexity (indicated by solve time and max depth) goes up. For GameMaster 2.0, however, the nodes/s go up with increasing search time. This is likely due to one time computational costs for initialization, which are higher relative to total runtime for GameMaster 2.0, due to the much shorter overall runtime and efficient tree implementation. In FrankenSearch, start-up costs are negligible, but handling an increasingly big search tree, especially with the Ludii objects in memory seems to decrease nodes/s for bigger searches.

Apart from the difference in game handling (Ludii versus custom implementation), the GameMaster 2.0 also uses an optimized version of PNS, improving performance over the base plain version used in FrankenSearch (Winands & Schadd, 2011).

Table 4.1: Selected results for endgame positions and overall average of all 80 solved positions

Position Name	**Time (s)**	**Max Depth**	**Nodes/s**	**Memory (GB)**	**Expansions**
dulcichvscohen_31_c4c3	177	136	118,297	64.56	850,922
kokvscohen_48_f6e7	7	9	160,058	10.01	52,920
miavsyl_0_27_e8e7	893	16	68,606	220.43	2,365,249
miavsyl_1_35-d3e2	< 1	5	133,311	0.26	973
pntest_2001-03-08_35_2	472	36	114,954	184.48	2,458,520
pntest_2001-05-15_35_153	11	7	156,095	15.68	73,036
pntest_2001-05-15_35_195	565	20	101,447	199.83	2,547,307
wikmanvsmona_33_d5f5	< 1	5	127,338	0.76	2,997
Average (80 positions)	*87.6*	*14.4*	*129,096*	*44.5*	*425,280.3*

Execution Profiling

The previous experiments already indicate that the Ludii framework causes significant overhead in the FrankenSearch execution. As visualized in Figure 3.5, Ludii is used as an external domain-specific entity to manage the game states and game execution with well defined interfaces to the actual SADL-based search within FrankenSearch. To investigate the Ludii overhead further and also identify other potential bottlenecks, the execution for both MCTS (playing a game of LOA with three seconds per move) and PNS (solving an endgame position) have been analyzed using a Java profiler. Figure 4.3 shows flamegraphs of the of CPU time for both

[2]The GameMaster 2.0 including the PNS implementation can be downlaoded at https://dke.maastrichtuniversity.nl/m.winands/loa/

Table 4.2: Comparison of PNS performance in GameMaster 2.0 implementations for selected endgame positions, showcasing nodes per second and the relative speed-up (based on nodes/s) compared to FrankenSearch in Table 4.1

Position Name	**Time (ms)**	**Nodes/s**	**Speed-Up Factor**
dulcichvscohen_31_c4c3	319	1,241,921	10.5
kokvscohen_48_f6e7	100	492,450	3.1
miavsyl_0_27_e8e7	2,387	1,780,743	26
miavsyl_1_35-d3e2	5	142,000	1.1
pntest_2001-03-08_35_2	842	1,397,384	12.2
pntest_2001-05-15_35_153	53	656,283	4.2
pntest_2001-05-15_35_195	853	1,535,532	15.1
wikmanvsmona_33_d5f5	10	254,200	2

executions. In both cases it is immediately visible that the vast majority of CPU time is spend in Ludii sources (gray bars), whereas only a small fraction is actually spend in the FrankenSearch code (gold bars), confirming Ludii as a major bottleneck for search execution.

For MCTS (Figure 4.3a), most time (75%) is spend in the playout phase, which is executed in Ludii, and part of the external evaluation function, called by the evaluation component of SADL. 12.8% of CPU time is spend on node expansion, which again is entirely in Ludii, generating and applying all possible moves, when a non-terminal leaf node reached. The remaining 12.2% of CPU time distributes between all other parts of the search but is dominated by the selection component of SADL (8.5% of total time), which in turn spends most time in the "evaluateExpression" function. This makes sense, since in the MCTS SADL the UCT equation for node selection has the most complexity and is evaluated using many recursive calls of the expression evaluator.

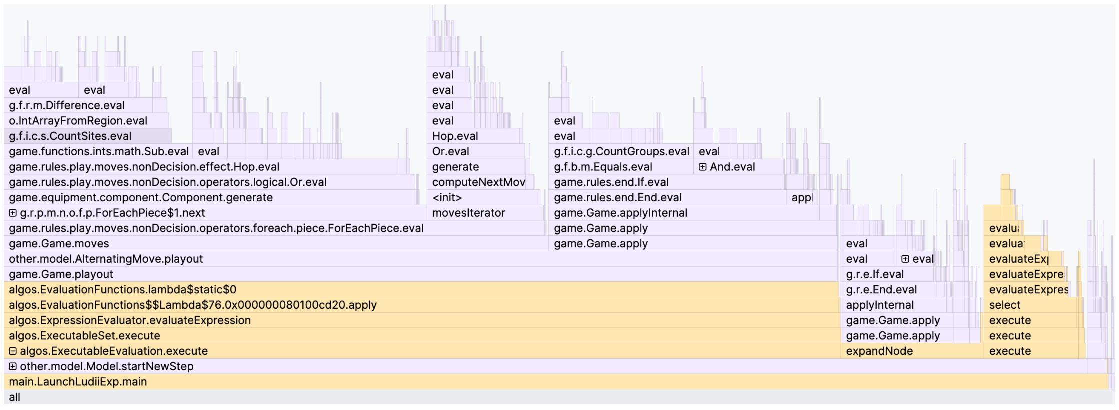
For PNS (Figure 4.3b), 79% of all CPU time is spend on the aforementioned node expansion using Ludii. While the backpropagation component was negligible for MCTS, it is the biggest non-Ludii component for PNS (6.2% of total CPU time). This is also reasonable, since calculating the min and sum of (dis-)proof numbers for all children per node is the most complex task in the PNS SADL description. The selection component, having to calculate an argmin over childern per node, takes up 4% of total CPU time.

4.2.3 Algorithm Generation

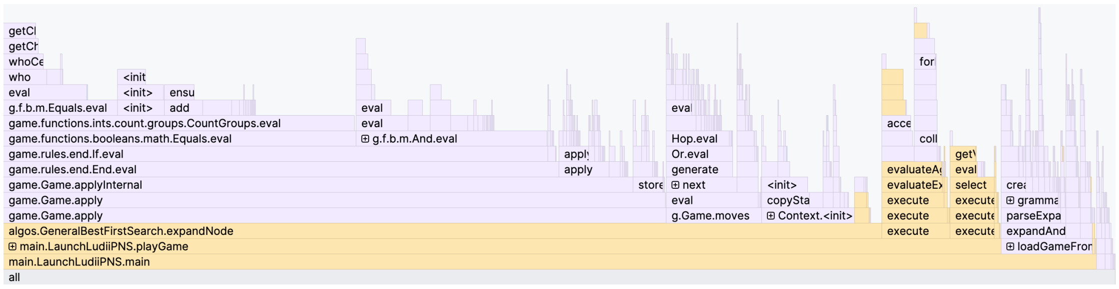
The process of algorithm generation in the FrankenSearch framework involves iteratively evolving search algorithms using a configurable evolutionary framework. Various parameters and mechanisms were explored and refined to balance diversity, efficiency, and performance while addressing computational constraints.

Early experiments highlighted challenges in runtime and diversity management. For example, round-robin tournaments provided comprehensive pairwise evaluations but proved computationally prohibitive for larger populations. To address this, a Swiss-system tournament format was introduced, significantly reducing runtime while maintaining evaluation quality. To prevent premature convergence to a single dominant strategy, mechanisms were introduced to reintroduce predefined base algorithms, such as MCTS, into each generation when necessary, ensuring a consistent level of diversity.

Another key improvement was the evolutionary check mechanism. Before a newly mutated individual is added to the population, it plays a small number of games against its original ancestor. This step filters out poorly performing mutations early, saving computational resources. To



(a) Flamegraph of CPU time for FrankenSearch execution using MCTS SADL.



(b) Flamegraph of CPU time for FrankenSearch execution using PNS SADL.

Figure 4.3: Profiler flamegraphs of CPU time for two search algorithms, MCTS (a) and PNS (b), using Ludii. Gold bars represent user-defined code (FrankenSearch), while gray bars indicate library or external methods (primarily Ludii). These visualizations help identify where the processing time is spent in the codebase.

ensure the validity of evolved algorithms, the crossover and mutation processes use compatibility checks based on operator metadata, such as return types, input types, and structural constraints.

Parameters such as mutation rates, population size, and tournament rounds were iteratively refined through experimentation. The framework was designed to be flexible, allowing for adaptation to different research contexts. Below is a summary of the key parameters used in the framework:

Final Parameters

The following parameters define the configuration of the framework and can be tuned for specific domains or computational environments:

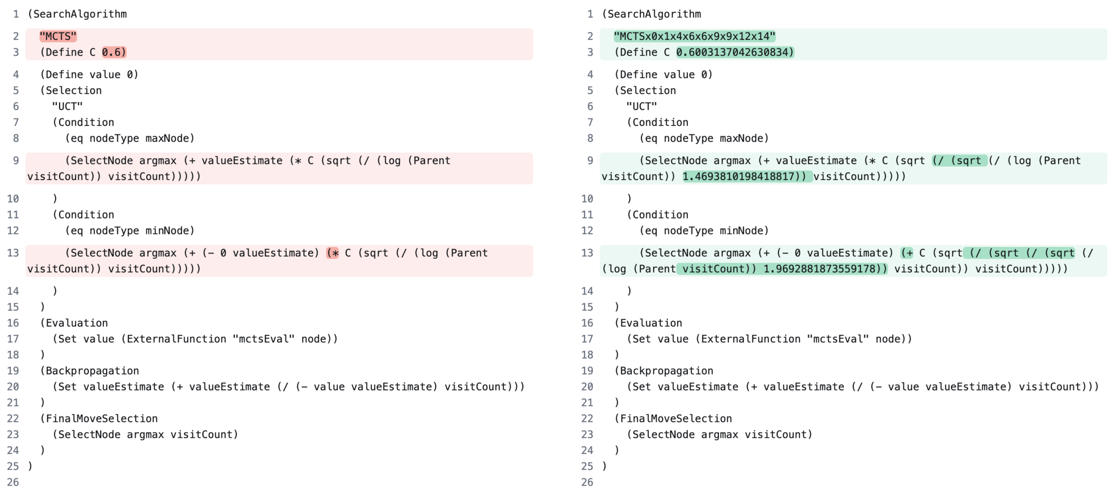
- **Number of Threads:** Number of CPU cores used for parallel execution of games. Up to 30 threads were utilized in these experiments.
- **Population Size:** Initially set to 20 in early experiments but increased to 50 with the introduction of Swiss-system tournaments to support greater diversity.

- **Maximum Generations:** Limited to 100, although most experiments terminated earlier due to convergence or resource constraints.
- **Mutation and Crossover Rates:** Both set at 0.95 to ensure frequent exploration of the search space through AST modifications.
- **Elite Pool Ratio:** Defines the proportion of the population eligible for crossover. Set to 0.25, allowing the top 25% to reproduce, while the top 50% survives and may undergo mutation.
- **Games Per Match:** For tournaments, typically 150 games per match. During evolutionary checks, this was reduced to 50 to save computation time.
- **Swiss Rounds:** Set to six for a population size of 50, providing an efficient balance between evaluation accuracy and runtime.
- **Maximum Seconds Per Move:** Significantly impacts computational cost and evaluation quality. Typical values ranged from 0.5 to 3 seconds per move, with evolutionary checks using 0.2 seconds.
- **Original Algorithm Threshold:** A win rate threshold of 0.3 used during evolutionary checks to determine whether a mutation is added to the population.
- **Maximum Mutation and Crossover Attempts:** Capped at 30 to prevent excessive retries for invalid individuals.
- **Predefined SADL Files:** A set of initial algorithms used as the starting point for evolution, including MCTS, PNS, and PN-MCTS variants.
- **Reload from Checkpoint:** A mechanism allowing experiments to resume from a saved generation file (`generation.txt`), facilitating long-running experiments.

Observations and Example

The experiments demonstrate the framework's ability to evolve and evaluate search algorithms. However, computational overhead and time constraints limited the results. Shorter move times resulted in less meaningful evaluations, as the outcomes were closer to random (as shown in Figure 4.2). Figure 4.4 shows an example of a mutated MCTS variant after 15 generations. The algorithm includes modified numerical constants, altered operators, and expanded AST structures, such as new nodes for division and square root operations.

In this example, the individual ranked highest in the Swiss tournament after 15 generations with a time constraint of 0.5 seconds per move. However, when evaluated against the baseline MCTS algorithm on 1000 games with 3 seconds per move, it only had a win rate of 37.6%.



```
(SearchAlgorithm
  "MCTS"
  (Define C 0.6)
  (Define value 0)
  (Selection
    "UCT"
    (Condition
      (eq nodeType maxNode)
      (SelectNode argmax (+ valueEstimate (∗ C (sqrt (/ (log (Parent visitCount)) visitCount)))))
    )
    (Condition
      (eq nodeType minNode)
      (SelectNode argmax (+ (− 0 valueEstimate) (∗ C (sqrt (/ (log (Parent visitCount)) visitCount)))))
    )
  )
  (Evaluation
    (Set value (ExternalFunction "mctsEval" node))
  )
  (Backpropagation
    (Set valueEstimate (+ valueEstimate (/ (− value valueEstimate) visitCount)))
  )
  (FinalMoveSelection
    (SelectNode argmax visitCount)
  )
)
```

```
(SearchAlgorithm
  "MCTSx0x1x4x6x6x9x9x12x14"
  (Define C 0.6003137042630834)
  (Define value 0)
  (Selection
    "UCT"
    (Condition
      (eq nodeType maxNode)
      (SelectNode argmax (+ valueEstimate (∗ C (sqrt (/ (sqrt (/ (log (Parent visitCount)) 1.4693810198418817)) visitCount)))))
    )
    (Condition
      (eq nodeType minNode)
      (SelectNode argmax (+ (− 0 valueEstimate) (+ C (sqrt (/ (sqrt (/ (sqrt (/ (log (Parent visitCount)) 1.9692881873559178)) visitCount)) visitCount)))))
    )
  )
  (Evaluation
    (Set value (ExternalFunction "mctsEval" node))
  )
  (Backpropagation
    (Set valueEstimate (+ valueEstimate (/ (− value valueEstimate) visitCount)))
  )
  (FinalMoveSelection
    (SelectNode argmax visitCount)
  )
)
```

Figure 4.4: Example of a constructed algorithm after 15 generations. Crossovers and mutations have been applied to the SADL of MCTS. Whenever a mutation occurs, the respective generation number is added to the SADL name.

Chapter 5

Conclusion and Future Work

This chapter offers conclusions, by first answering and discussing the research questions in Section 5.1. Afterwards, Section 5.2 discusses limitations and possible shortcomings of this thesis, as well as proposing some future research directions.

5.1 Research Questions

The following subsections discuss the outcomes of the research questions outlined in Section 1.2.

5.1.1 Algorithm Decomposition

How can complex search algorithms such as Monte-Carlo Tree Search and Proof-Number Search be effectively decomposed into building blocks for the design of search algorithms?

As previously outlined, the scope for this thesis has been limited in various ways in order to present a proof-of-concept that can be expanded on. As part of the scope limitations, only best-first search (BFS) algorithms have been considered, with focus on Monte-Carlo Tree Search and Proof-Number Search. For this scope, the decomposition of search algorithms as described in Section 3.1 proved successful through the identification of the fundamental components (Selection, Expansion, Evaluation, Backpropagation and Final Move Selection as detailed in Subsection 3.1.1). These high-level components provide a fundamental skeleton to create a new BFS algorithm.

Six distinct building blocks alone do not allow for much versatility in creating new algorithms, however. So in addition to decomposing algorithms into their main components, the decomposition follows a nested approach, where each main component can itself be described as a collection of lower-level operations, e.g., conditional executions and value assignments based on mathematical expressions.

By approaching algorithm decomposition as a group of components on various levels, the process can also easily expanded, allowing to add new high-level or lower-level building blocks, when a wider scope of search algorithms should be considered, without compromising the existing elements.

5.1.2 Language Design Implementation

How can a structured language be developed to effectively represent search algorithms?

The Search Algorithm Description Language (SADL), described in Section 3.2 constitutes the core of this thesis. SADL's lisp-like structure of nested lists (s-expressions) aligns well with the decomposition approach discussed for the previous research question (Subsection 5.1.1). At all times, the balance between expressiveness and simplicity is considered. Resulting design choices for simplicity include, for instance, not providing an explicit `Expansion` step in SADL, since that can implicitly be handled by the framework, or having the command `SelectNode` loop over all child nodes without needing explicit instructions in the language. Design choices for expressiveness include the `Set` and `Condition` commands, allowing for fine-grained execution control.

Multilayer nesting as a design approach is not only reflected in the SADL syntax itself, but also in the wider execution framework, which consists of three layers with carefully designed interfaces:

1. **SADL:** Implementation-agnostic algorithm description
2. **Interpretation and Execution:** Java implementation to compile and execute SADL instances (domain-agnostic)
3. **Domain application:** Domain-specific execution environment (Ludii is used in this thesis)

Designing not only the language but also its surrounding execution framework in this way allows for easy extension or modification of the framework for different search domains, or even SADL implementations in languages other than Java.

The validation experiments demonstrated SADL's robustness and flexibility. Error-handling tests (Subsection 4.2.1) confirmed that the parser and compiler reliably detect syntactic and semantic issues, though more checks during parsing as well as even more user-friendly error messages could be added. The ability to at least partly describe complex algorithms like PN-MCTS further showcased SADL's modularity and its capacity to combine features from different search algorithms. While certain operations, such as ranking child nodes by proof numbers, are beyond SADL's current capabilities, these limitations highlight specific directions for extending its functionality.

Experiments with MCTS and PNS (Subsection 4.2.2) validated SADL's practical utility. SADL-based MCTS matched the performance of a plain Java implementation and enabled rapid experimentation with algorithm variants, such as tuning the UCT exploration constant C. SADL-based PNS successfully solved a number of complex endgame positions in LOA, though memory overhead from the Ludii framework posed challenges, resulting in significantly slower search speed (nodes/s) compared to a handcrafted implementation. These results affirm that SADL can effectively represent and execute complex search algorithms within the FrankenSearch framework.

5.1.3 Evolutionary Algorithm Application and Evaluation

How can evolutionary algorithms be employed to generate, optimize, and evaluate new search algorithm variants effectively?

To apply evolutionary algorithms (EAs) to generate and optimize new search algorithm instances, the Genetic Programming (Koza, 1994) approach proved useful. By using the abstract syntax tree (AST) representation of a parsed SADL instance as a foundation, AST nodes can be treated as "genes". This allows for operations such as crossover, where individual nodes or entire subtrees are swapped between individuals, and mutation, where nodes are modified or newly generated (see Subsection 4.1.2).

As described in Subsection 2.1.1, games are a well-established testbed for search algorithm research because of their structured rules and inherent scoring mechanism. For evolutionary algorithms, a fitness function usually evaluates a single individual. In this case, a search algorithm can only ever be evaluated in comparison to another algorithm as opposed to a static scale. The question on how to evaluate search algorithms splits into two categories for this research:

1. Evaluating final generated algorithm for overall performance
2. Evaluating candidates during EA processing

For the first category of evaluations, benchmarking the performance of a new algorithm against existing and established algorithms (such as plain MCTS) in a selected domain is a natural way. In order to get meaningful results, a large enough number of games have to be played, and domain constraints such as computational time budget should also be varied to get a better picture of the performance of an algorithm in different situations.

This approach is quite computationally expensive, and therefore not feasible for fitness evaluation during EA processing. Therefore, in that second category, different variants of tournament selection have been tried, where in each generation the individuals play games against each other to determine an overall ranking. The best results can be achieved with round-robin tournaments, but the computational requirements are also enormous. Instead, a Swiss-system tournament format emerged as the most useful evaluation schema during EA processing, but the runtime is still quite long. Evaluating the mutating population in EA just amongst themselves, however, carries the risk that the overall population quality converges and ultimately deteriorates to parts of the search space that are far from optimal. Therefore, the benchmarking approach also finds its way into the EA fitness evaluation by ensuring that some base algorithms such as plain MCTS are present in each population.

5.2 Limitations and Future Research Directions

One key limitation of this work lies in the constrained scope of the SADL language and the FrankenSearch framework. Currently, the focus is limited to best-first search (BFS) algorithms applied to two-player, turn-based, deterministic, perfect-information games. While this narrow scope served as a manageable starting point for demonstrating the feasibility of the framework, it restricts the applicability of FrankenSearch to a small subset of potential search algorithms and problem domains. Expanding the scope of SADL to support non-BFS algorithms, as well as games beyond the aforementioned limitations, would significantly broaden the utility and versatility of the framework.

The use of Ludii to evaluate the algorithms, while convenient in terms of implementation, proved to be a significant bottleneck, both in terms of computational requirements to run the games, as well as memory requirements to store game state in the tree nodes. The overhead that the Ludii framework introduced were particularly troublesome for the evolutionary algorithm experiments, where a very large number of games need to run in order to achieve any meaningful results. In future work, FrankenSearch could be applied to a dedicated, lean and efficient implementation of LOA, or any other search problem, enabling much more efficient research.

A related limitation lies in the evolutionary component of FrankenSearch. While evolutionary algorithms were successfully employed to generate new search algorithm candidates, the results did not construct any algorithmic variants meaningfully better than the originals within the time-frame. This suggests that the evolutionary process, particularly the crossover and mutation operations, could benefit from further refinement. For example, mutations should be guided more intelligently to ensure meaningful changes. By either improving the efficiency, or simply taking much more time to run these experiments, even with the current EA implementation, better results can likely be achieved by running the experiments for more generations, and with more seconds per move.

Another shortcoming is the decision to fully externalize the evaluation step and treat the simulation of MCTS as an evaluation function. While this approach seems justified for classic evaluation functions that are highly domain-specific and independent of the actual search process, any attempts to improve MCTS by guiding the simulation phase cannot be described by SADL alone. Future research on SADL could therefore explore ways to make the evaluation step more flexible and integrative.

The current framework is limited to two-player games, and this limitation influenced the design choices for evaluation and backpropagation components in SADL. In the current implementation, the SADL descriptions frequently include conditional checks to determine the node type (MIN/MAX node or AND/OR node). This design makes the evaluation and backpropagation steps transparent and adaptable, as the behavior can be explicitly defined for different node types. However, this approach perhaps also makes the SADL descriptions unnecessarily long and complicated, and it restricts the framework to two-player games. A potential future improvement would be to adopt a tuple-based backpropagation method, where tuples represent evaluations for all players. This would allow the framework to support multiplayer games seamlessly by eliminating the need for conditional logic tied to two-player assumptions.

Another promising direction for future research lies in leveraging large language models (LLMs) to generate SADL descriptions. The natural language processing capabilities of LLMs could be used to generate SADL code directly from high-level descriptions of search algorithms or games. Additionally, LLMs could be trained to analyze game descriptions and generate tailored SADL candidates for solving specific problems. These LLM-generated candidates could then be refined using evolutionary algorithms.

In conclusion, while FrankenSearch and SADL provide a working proof-of-concept for automating search algorithm generation, significant opportunities remain to expand their capabilities. The current framework demonstrates the feasibility of describing and implementing search algorithms within a constrained scope, validating the potential of SADL for this purpose. By addressing limitations in scope, refining evolutionary techniques, and enabling multiplayer support, as well as exploring novel approaches like LLM-assisted generation, future research can build upon this foundation to unlock the full potential of the framework.

Bibliography

Abelson, H., Sussman, G. J., & with Julie Sussman. (1996). *Structure and Interpretation of Computer Programs* (2nd Editon). MIT Press/McGraw-Hill, Cambridge, MA, USA.

Abramson, B. D. (1987). *The Expected-Outcome Model of Two-Player Games* [Doctoral dissertation, Department of Computer Science, Columbia University, New York, NY, USA].

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, Boston, MA, USA.

Allis, L., van der Meulen, M., & van den Herik, H. (1994). Proof-Number Search. *Artificial Intelligence*, *66*(1), 91–124. doi: 10.1016/0004-3702(94)90004-3.

Bontrager, P., Khalifa, A., Mendes, A., & Togelius, J. (2021). Matching Games and Algorithms for General Video Game Playing. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, *12*(1), 122–128. doi: 10.1609/aiide.v12i1.12884.

Breuker, D. (1998). *Memory versus Search in Games* [Doctoral dissertation, Department of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands]. doi: 10.26481/dis.19981016db.

Browne, C. (2011a). The Ludi System. In *Evolutionary Game Design* (pp. 11–21). Springer, London, UK. doi: 10.1007/978-1-4471-2179-4˙3.

Browne, C. (2011b). Yavalath. In *Evolutionary Game Design* (pp. 75–85). Springer, London, UK. doi: 10.1007/978-1-4471-2179-4˙7.

Browne, C., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1–43. doi: 10.1109/TCIAIG.2012.2186810.

Campbell, M., Hoane, A., & Hsu, F.-h. (2002). Deep Blue. *Artificial Intelligence*, *134*(1), 57–83. doi: 10.1016/S0004-3702(01)00129-1.

Cardoen, B., Demeulemeester, E., & Beliën, J. (2010). Operating room planning and scheduling: A literature review. *European Journal of Operational Research*, *201*(3), 921–932. doi: 10.1016/j.ejor.2009.04.011.

Cazenave, T. (2024). Monte Carlo Search Algorithms Discovering Monte Carlo Tree Search Exploration Terms. doi: 10.48550/arXiv.2404.09304.

Chaslot, G. M. J.-B., Winands, M. H. M., & van den Herik, H. J. (2008). Parallel Monte-Carlo Tree Search. In H. J. van den Herik, X. Xu, Z. Ma, & M. H. M. Winands (Eds.), *Computers and Games* (pp. 60–71). Springer, Berlin, Heidelberg, Germany. doi: 10.1007/978-3-540-87608-3˙6.

Chaslot, G. M. J.-B., Winands, M. H. M., van den Herik, H. J., Uiterwijk, J. W. H. M., & Bouzy, B. (2008). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, *4*(3), 343–357. doi: 10.1142/S1793005708001094.

Chomsky, N. (1957). *Syntactic Structures*. Mouton and Co., The Hague, The Netherlands.

Church, A. (1941). *The Calculi of Lambda-Conversation*. Princeton University Press, Princeton, NJ, USA.

Churchill, D., & Buro, M. (2021). Hierarchical Portfolio Search: Prismata's Robust AI Architecture for Games with Large Search Spaces. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, *11*(1), 16–22. doi: 10.1609/aiide.v11i1.12787.

Clocksin, W. F., & Mellish, C. S. (2003). *Programming in Prolog: Using the ISO Standard* (5th ed.) [Published: 25 July 2003, eBook Published: 06 December 2012]. Springer, Berlin, Heidelberg, Germany. doi: 10.1007/978-3-642-55481-0.

Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In H. J. van den Herik, P. Ciancarini, & H. H. L. M. Donkers (Eds.), *Computers and games* (pp. 72–83). Springer, Berlin, Heidelberg, Germany. doi: 10.1007/978-3-540-75538-8˙7.

Doe, E., Winands, M. H. M., Soemers, D. J. N. J., & Browne, C. (2022). Combining Monte-Carlo Tree Search with Proof-Number Search. *2022 IEEE Conference on Games (CoG)*, 206–212. doi: 10.1109/CoG51982.2022.9893635.

Dwivedi, Y. K., Hughes, L., Ismagilova, E., Aarts, G., Coombs, C., Crick, T., Duan, Y., Dwivedi, R., Edwards, J., Eirug, A., Galanos, V., Ilavarasan, P. V., Janssen, M., Jones, P., Kar, A. K., Kizgin, H., Kronemann, B., Lal, B., Lucini, B., . . . Williams, M. D. (2021). Artificial Intelligence (AI): Multidisciplinary perspectives on emerging challenges, opportunities, and agenda for research, practice and policy. *International Journal of Information Management*, *57*, 101994. doi: 10.1016/j.ijinfomgt.2019.08.002.

Eiben, A., & Smith, J. (2015). *Introduction to Evolutionary Computing* (2nd). Springer, Berlin, Heidelberg, Germany. doi: 10.1007/978-3-662-44874-8.

Franz, C., Mogk, G., Mrziglod, T., & Schewior, K. (2022). Completeness and Diversity in Depth-First Proof-Number Search with Applications to Retrosynthesis. In L. D. Raedt (Ed.), *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22* (pp. 4747–4753). International Joint Conferences on Artificial Intelligence Organization. doi: 10.24963/ijcai.2022/658.

Gelly, S., & Silver, D. (2007). Combining online and offline knowledge in UCT. *Proceedings of the 24th International Conference on Machine Learning*, 273–280. doi: 10.1145/1273496.1273531.

Gelly, S., & Silver, D. (2011). Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, *175*(11), 1856–1875. doi: 10.1016/j.artint.2011.03.007.

Genesereth, M., Love, N., & Pell, B. (2005). General Game Playing: Overview of the AAAI Competition. *AI Magazine*, *26*(2), 62. doi: 10.1609/aimag.v26i2.1813.

Gobble, M. M. (2019). The Road to Artificial General Intelligence. *Research-Technology Management*, *62*(3), 55–59. doi: 10.1080/08956308.2019.1587336.

Goldberg, D. E., & Holland, J. H. (1988). Genetic Algorithms and Machine Learning. *Machine Learning*, *3*(2), 95–99. doi: 10.1023/A:1022602019183.

Graham, P. (1993). *On LISP: Advanced Techniques for Common LISP*. Prentice Hall, Upper Saddle River, NJ, USA. doi: 10.5555/1095593.

Guerriero, F., & Guido, R. (2011). Operational research in the management of the operating theatre: a survey. *Health Care Management Science*, *14*(1), 89–114. doi: 10.1007/s10729-010-9143-6.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

Hudak, P. (1989). Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, *21*(3), 359–411. doi: 10.1145/72551.72554.

Kaneko, T. (2010). Parallel Depth First Proof Number Search. *Proceedings of the AAAI Conference on Artificial Intelligence*, *24*(1), 95–100. doi: 10.1609/aaai.v24i1.7551.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H., & Leyton-Brown, K. (2016). SATenstein: Automatically building local search SAT solvers from components. *Artificial Intelligence*, *232*, 20–42. doi: 10.1016/j.artint.2015.11.002.

Kishimoto, A., Winands, M. H. M., Müller, M., & Saito, J.-T. (2012). Game-Tree Search Using Proof Numbers: The First Twenty Years. *ICGA Journal*, *35*, 131–156. doi: 10.3233/ICG-2012-35302.

Knuth, D. E. (1964). Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, *7*(12), 735–736. doi: 10.1145/355588.365140.

Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, *6*(4), 293–326. doi: 10.1016/0004-3702(75)90019-3.

Kocsis, L., & Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. In J. Fürnkranz, T. Scheffer, & M. Spiliopoulou (Eds.), *Machine Learning: ECML 2006* (pp. 282–293). Springer, Berlin, Heidelberg, Germany. doi: 10.1007/11871842˙29.

Kowalski, J. (2016). *General Game Description Languages* [Doctoral dissertation, Faculty of Mathematics and Computer Science, University of Wrocław, Wrocław, Poland].

Kowalski, J., Doe, E., Winands, M. H. M., Górski, D., & Soemers, D. J. N. J. (2025). Proof Number Based Monte-Carlo Tree Search [Accepted for publication]. *IEEE Transactions on Games*. doi: 10.1109/TG.2024.3403750.

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, *4*(2), 87–112. doi: 10.1007/BF00175355.

Marsland, T. A. (1986). A Review of Game-Tree Pruning. *ICGA Journal*, *9*(1), 3–19. doi: 10.3233/ICG-1986-9102.

McCarthy, J. (1959). LISP: A Programming System for Symbolic Manipulations. *Preprints of Papers Presented at the 14th National Meeting of the Association for Computing Machinery*, 1–4. doi: 10.1145/612201.612243.

Nagai, A. (1999). A New Depth-First-Search Algorithm for AND/OR Trees. *ICGA Journal*, *22*(1), 35–36. doi: 10.3233/ICG-1999-22106.

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. doi: 10.1016/C2009-0-27773-7.

Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. doi: 10.1016/C2009-0-27663-X.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate? *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, 256–267. doi: 10.1145/3136014.3136031.

Perez-Liebana, D., Liu, J., Khalifa, A., Gaina, R. D., Togelius, J., & Lucas, S. M. (2019). General Video Game AI: A Multitrack Framework for Evaluating Agents, Games, and Content Generation Algorithms. *IEEE Transactions on Games*, *11*(3), 195–214.

Piette, É., Soemers, D. J. N. J., Stephenson, M., Sironi, C. F., Winands, M. H. M., & Browne, C. (2020). Ludii – The Ludemic General Game System. In G. De Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarín, & J. Lang (Eds.), *ECAI 2020 : 24th European Conference on Artificial Intelligence* (pp. 411–418, Vol. 325). IOS Press. doi: 10.3233/FAIA200120.

Pilditch, T. D. (2024). The Reasoning Under Uncertainty Trap: A Structural AI Risk. doi: 10.48550/arXiv.2402.01743.

Pitrat, J. (1968). Realization of a General Game-Playing Program. In A. J. H. Morrell (Ed.), *Information Processing 68, Proceedings of IFIP Congress 1968, Edinburgh, UK, 5-10 August 1968, Volume 2* (pp. 1570–1574). North-Holland.

Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th). Pearson, New York, NY, USA.

Saffidine, A. (2012). Minimal Proof Search for Modal Logic K Model Checking. In L. F. del Cerro, A. Herzig, & J. Mengin (Eds.), *Logics in Artificial Intelligence* (pp. 346–358). Springer, Berlin, Heidelberg, Germany. doi: 10.1007/978-3-642-33353-8˙27.

Saffidine, A., Finnsson, H., & Buro, M. (2012). Alpha-Beta Pruning for Games with Simultaneous Moves. *Proceedings of the AAAI Conference on Artificial Intelligence*, *26*(1), 556–562. doi: 10.1609/aaai.v26i1.8148.

Saito, J.-T., Chaslot, G., Uiterwijk, J. W. H. M., & van den Herik, H. J. (2006). Monte-Carlo Proof-Number Search for Computer Go. In H. J. van den Herik, P. Ciancarini, & H. H. L. M. Donkers (Eds.), *Computers and Games* (pp. 50–61). Springer, Berlin, Heidelberg, Germany. doi: 10.1007/978-3-540-75538-8˙5.

Saito, J.-T., Winands, M. H. M., & van den Herik, H. J. (2009). Randomized Parallel Proof-Number Search. In H. J. van den Herik & P. Spronck (Eds.), *Advances in Computer Games* (pp. 75–87). Springer, Berlin, Heidelberg, Germany. doi: 10.1007/978-3-642-12993-3˙8.

Schaeffer, J., & van den Herik, H. (2002). Games, computers, and artificial intelligence. *Artificial Intelligence*, *134*(1), 1–7. doi: 10.1016/S0004-3702(01)00165-5.

Schaul, T., Togelius, J., & Schmidhuber, J. (2011). Measuring Intelligence through Games. doi: 10.48550/arXiv.1109.1314.

Scowen, R. S. (1993). Generic base standards. *Proceedings 1993 Software Engineering Standards Symposium*, 25–34. doi: 10.1109/SESS.1993.263968.

Sebesta, R. W. (2012). *Concepts of Programming Languages* (10th). Pearson, New York, NY, USA.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, *529*(7587), 484–489. doi: 10.1038/nature16961.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, *362*(6419), 1140–1144. doi: –10.1126/science.aar6404″.

Sironi, C. F., & Winands, M. H. M. (2021). Adaptive General Search Framework for Games and Beyond. *2021 IEEE Conference on Games (CoG)*, 1–8. doi: 10.1109/CoG52621.2021.9619115.

Soemers, D., Piette, E., Stephenson, M., & Browne, C. (2020). *Ludii User Guide.* Department of Data Science and Knowledge Engineering, Maastricht University, Maastricht, The Netherlands. https://ludii.games/downloads/LudiiUserGuide.pdf

Stroustrup, B. (1997). *The C++ Programming Language* (3rd). Addison-Wesley.

Świechowski, M., Godlewski, K., Sawicki, B., & Mańdziuk, J. (2023). Monte Carlo Tree Search: a review of recent modifications and applications. *Artificial Intelligence Review*, *56*(3), 2497–2562. doi: 10.1007/s10462-022-10228-y.

Świechowski, M., & Mańdziuk, J. (2014). Self-Adaptation of Playing Strategies in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*, 367–381. doi: 10.1109/TCIAIG.2013.2275163.

von Neumann, J., & Morgenstern, O. (1944). *Theory of Games and Economic Behavior.* Princeton University Press, Princeton, NJ, USA.

Wang, X., Qian, Y., Gao, H., Coley, C. W., Mo, Y., Barzilay, R., & Jensen, K. (2020). Towards efficient discovery of green synthetic pathways with Monte Carlo tree search and reinforcement learning. *Chemical Science*, *11*, 10959–10972. doi: 10.1039/d0sc04184j.

Wei, G., Chen, Y., & Rompf, T. (2019). Staged Abstract Interpreters: Fast and Modular Whole-Program Analysis via Meta-programming. *Proceedings of the ACM on Programming Languages*, *Volume 3.* doi: 10.1145/3360552.

Winands, M. H. M. (2004). *Informed Search in Complex Games* [Doctoral dissertation, Department of Computer Science, Maastricht University]. Universitaire Pers Maastricht, Maastricht, The Netherlands. doi: 10.26481/dis.20041201mw.

Winands, M. H. M., Björnsson, Y., & Saito, J.-T. (2008). Monte-Carlo Tree Search Solver. In H. J. van den Herik, X. Xu, Z. Ma, & M. H. M. Winands (Eds.), *Computers and Games* (pp. 25–36, Vol. 5131). Springer, Berlin, Heidelberg, Germany. doi: 10.1007/978-3-540-87608-3˙3.

Winands, M. H. M., & Schadd, M. P. D. (2011). Evaluation-Function Based Proof-Number Search. In H. J. van den Herik, H. Iida, & A. Plaat (Eds.), *Computers and Games* (pp. 23–35). Springer, Berlin, Heidelberg, Germany. doi: 10.1007/978-3-642-17928-0˙3.

Winands, M. H. M., Uiterwijk, J. W. H. M., & van den Herik, J. (2003). PDS-PN: A New Proof-Number Search Algorithm. In J. Schaeffer, M. Müller, & Y. Björnsson (Eds.), *Computers and Games* (pp. 61–74). Springer, Berlin, Heidelberg, Germany. doi: 10.1007/978-3-540-40031-8˙5.

Winands, M., & Uiterwijk, J. (2001). PN, PN^2 and PN^* in Lines of Action. In J. Uiterwijk (Ed.), *The CMG Sixth Computer Olympiad Computer-Games Workshop Proceedings.* Technical Reports in Computer Science CS 01-04, Universiteit Maastricht, Maastricht, The Netherlands.

Xu, L., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2008). SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, *Volume 32*, 565–606. doi: 10.1613/jair.2490.

Xu, Z., Jain, S., & Kankanhalli, M. (2024). Hallucination is Inevitable: An Innate Limitation of Large Language Models. doi: 10.48550/arXiv.2401.11817.

Yannakakis, G., & Togelius, J. (2018). *Artificial Intelligence and Games.* Springer International Publishing. doi: 10.1007/978-3-319-63519-4.

Zhao, L., Zhang, L., Wu, Z., Chen, Y., Dai, H., Yu, X., Liu, Z., Zhang, T., Hu, X., Jiang, X., Li, X., Zhu, D., Shen, D., & Liu, T. (2023). When brain-inspired AI meets AGI. *Meta-Radiology*, *1*(1), 100005. doi: 10.1016/j.metrad.2023.100005.

Appendix A

SADL Grammar and Examples

A.1 Grammar Specification

```
              ⟨Program⟩ → ⟨AlgorithmDefinition⟩
  ⟨AlgorithmDefinition⟩ → “(” “SearchAlgorithm” ⟨String⟩ {⟨Declaration⟩} {⟨Component⟩} “)”
          ⟨Declaration⟩ → “(” “Define” ⟨Identifier⟩ ⟨Expression⟩ “)”
            ⟨Component⟩ → ⟨Selection⟩
                        | ⟨Evaluation⟩
                        | ⟨Backpropagation⟩
                        | ⟨FinalMoveSelection⟩
            ⟨Selection⟩ → “(” “Selection” [⟨String⟩] {⟨Statement⟩} “)”
           ⟨Evaluation⟩ → “(” “Evaluation” [⟨String⟩] {⟨Statement⟩} “)”
      ⟨Backpropagation⟩ → “(” “Backpropagation” [⟨String⟩] {⟨Statement⟩} “)”
   ⟨FinalMoveSelection⟩ → “(” “FinalMoveSelection” [⟨String⟩] {⟨Statement⟩} “)”

            ⟨Condition⟩ → “(” “Condition” ⟨Expression⟩ {⟨Statement⟩} “)”
           ⟨SelectNode⟩ → “(” “SelectNode” (“argmax” | “argmin”) ⟨Expression⟩ “)”
           ⟨Assignment⟩ → “(” “Set” ⟨Identifier⟩ ⟨Expression⟩ “)”
            ⟨Statement⟩ → ⟨Assignment⟩
                        | ⟨Condition⟩
                        | ⟨SelectNode⟩

           ⟨Expression⟩ → ⟨Literal⟩ | ⟨FunctionCall⟩
         ⟨FunctionCall⟩ → “(” ⟨Operator⟩ {⟨Expression⟩} “)”
                        | “(” “Aggregate” ⟨AggregationFunction⟩ ⟨Expression⟩ “)”
                        | “(” “Parent” ⟨Expression⟩ “)”
                        | “(” “ExternalFunction” ⟨String⟩ ⟨Expression⟩ “)”
             ⟨Operator⟩ → “+” | “-” | “*” | “/” | “eq” | “neq” | “lt” | “gt” | “lte” | “gte”
                        | “and” | “or” | “not” | “log” | “sqrt”
  ⟨AggregationFunction⟩ → “min” | “max” | “sum” | “avg”

              ⟨Literal⟩ → ⟨Number⟩ | ⟨Boolean⟩ | ⟨Constant⟩ | ⟨PredefinedVariable⟩ | ⟨Identifier⟩
             ⟨Constant⟩ → “inf” | “unknown” | “maxNode” | “minNode” | “orNode” | “andNode”
   ⟨PredefinedVariable⟩ → “node” | “nodeType” | “numChildren” | “depth”
           ⟨Identifier⟩ → ⟨Letter⟩ {⟨Letter⟩ | ⟨Digit⟩ | “_”}
```

$\langle$String$\rangle$ $\rightarrow$ “"” {$\langle$Identifier$\rangle$} “"”
$\langle$Number$\rangle$ $\rightarrow$ [“-”] $\langle$Digit$\rangle$ {$\langle$Digit$\rangle$} [“.” $\langle$Digit$\rangle$ {$\langle$Digit$\rangle$}]
$\langle$Boolean$\rangle$ $\rightarrow$ “true” | “false”
$\langle$Letter$\rangle$ $\rightarrow$ “A” | “B” | ... | “Z” | “a” | “b” | ... | “z”
$\langle$Digit$\rangle$ $\rightarrow$ “0” | “1” | ... | “9”

A.2 SADL Examples

A.2.1 Monte-Carlo Tree Search

```
(SearchAlgorithm "MCTS"
  (Define C 1.4)
  (Define value 0)
  (Selection "UCT"
    (Condition (eq nodeType maxNode)
      (SelectNode argmax
        (+ valueEstimate
            (* C (sqrt (/ (log (Parent visitCount)) visitCount)))
        )
      )
    )
    (Condition (eq nodeType minNode)
      (SelectNode argmax
        (+ (- 0 valueEstimate)
            (* C (sqrt (/ (log (Parent visitCount)) visitCount)))
        )
      )
    )
  )
  (Evaluation
    (Set value (ExternalFunction "MCTSeval" node))
  )
  (Backpropagation
    (Set valueEstimate (+ valueEstimate (/ (- value valueEstimate)
       visitCount)))
  )
  (FinalMoveSelection
    (SelectNode argmax visitCount)
  )
)
```

A.2.2 Proof-Number Search

```
(SearchAlgorithm "PNS"
  (Selection "MostProvingNode"
    (Condition (eq nodeType orNode)
      (SelectNode argmin proofNumber)
    )
    (Condition (eq nodeType andNode)
      (SelectNode argmin disproofNumber)
    )
  )
  (Evaluation
    (Set proofValue (ExternalFunction "pnsEval" node))
    (Condition (eq proofValue true)
      (Set proofNumber 0)
      (Set disproofNumber inf)
    )
    (Condition (eq proofValue false)
      (Set proofNumber inf)
      (Set disproofNumber 0)
    )
    (Condition (eq proofValue unknown)
      (Condition (eq nodeType orNode)
        (Set proofNumber 1)
        (Set disproofNumber numChildren)
      )
      (Condition (eq nodeType andNode)
        (Set proofNumber numChildren)
        (Set disproofNumber 1)
      )
    )
  )
  (Backpropagation
    (Condition (eq nodeType orNode)
      (Set proofNumber (Aggregate min proofNumber))
      (Set disproofNumber (Aggregate sum disproofNumber))
    )
    (Condition (eq nodeType andNode)
      (Set proofNumber (Aggregate sum proofNumber))
      (Set disproofNumber (Aggregate min disproofNumber))
    )
  )
)
```

A.2.3 PN-MCTS

```
(SearchAlgorithm "PN-MCTS"
  (Define C 0.6)
  (Define T 5)
  (Define value 0)

  (Selection "UCT"
    (Condition (eq nodeType maxNode)
      (SelectNode argmax
        (Condition (or (lte visitCount T) (and (neq proofNumber 0) (neq
            disproofNumber 0)))
          (+ valueEstimate (* C (sqrt (/ (log (Parent visitCount))
              visitCount))))
        )
      )
    )
    (Condition (eq nodeType minNode)
      (SelectNode argmax
        (Condition (or (lte visitCount T) (and (neq proofNumber 0) (neq disproofNumber 0)))
          (+ (- 0 valueEstimate) (* C (sqrt (/ (log (Parent visitCount)) visitCount))))
        )
      )
    )
  )
  (Evaluation
    (Set value (ExternalFunction "mctsEval" node))
    (Set proofValue (ExternalFunction "pnsEval" node))
    (Condition (eq proofValue true)
      (Set proofNumber 0)
      (Set disproofNumber inf)
    )
    (Condition (eq proofValue false)
      (Set proofNumber inf)
      (Set disproofNumber 0)
    )
    (Condition (eq proofValue unknown)
      (Condition (eq nodeType orNode)
        (Set proofNumber 1)
        (Set disproofNumber numChildren)
      )
      (Condition (eq nodeType andNode)
        (Set proofNumber numChildren)
        (Set disproofNumber 1)
      )
    )
  )
  (Backpropagation
    (Set valueEstimate (+ valueEstimate (/ (- value valueEstimate) visitCount)))
    (Condition (eq nodeType orNode)
      (Set proofNumber (Aggregate min proofNumber))
      (Set disproofNumber (Aggregate sum disproofNumber))
    )
    (Condition (eq nodeType andNode)
      (Set proofNumber (Aggregate sum proofNumber))
      (Set disproofNumber (Aggregate min disproofNumber))
    )
  )
  (FinalMoveSelection
    (Condition (eq proofNumber 0)
      (SelectNode argmin proofNumber)
    )
    (Condition (neq proofNumber 0)
      (SelectNode argmax visitCount)
    )
  )
```

```
)
```

A.2.4 PN-MCTSdepth

```
(SearchAlgorithm "PN-MCTSdepth"
  (Define C 0.6)
  (Define value 0)
  (Define switchDepth 5)

  (Selection "UCTPN"
    (Condition (lte depth switchDepth)
      (Condition (eq nodeType maxNode)
        (SelectNode argmax
          (+ valueEstimate (* C (sqrt (/ (log (Parent visitCount))
              visitCount))))
        )
      )
      (Condition (eq nodeType minNode)
        (SelectNode argmax
          (+ (- 0 valueEstimate) (* C (sqrt (/ (log (Parent visitCount))
              visitCount))))
        )
      )
    )
    (Condition (gt depth switchDepth)
      (Condition (eq nodeType orNode)
        (SelectNode argmin proofNumber)
      )
      (Condition (eq nodeType andNode)
        (SelectNode argmin disproofNumber)
      )
    )
  )
  (Evaluation
    (Set value (ExternalFunction "mctsEval" node))
    (Set proofValue (ExternalFunction "pnsEval" node))
    (Condition (eq proofValue true)
      (Set proofNumber 0)
      (Set disproofNumber inf)
    )
    (Condition (eq proofValue false)
      (Set proofNumber inf)
      (Set disproofNumber 0)
    )
    (Condition (eq proofValue unknown)
      (Condition (eq nodeType orNode)
        (Set proofNumber 1)
        (Set disproofNumber numChildren)
      )
      (Condition (eq nodeType andNode)
        (Set proofNumber numChildren)
        (Set disproofNumber 1)
      )
    )
  )
  (Backpropagation
    (Set valueEstimate (+ valueEstimate (/ (- value valueEstimate)
        visitCount)))
    (Condition (eq nodeType orNode)
      (Set proofNumber (Aggregate min proofNumber))
      (Set disproofNumber (Aggregate sum disproofNumber))
    )
    (Condition (eq nodeType andNode)
      (Set proofNumber (Aggregate sum proofNumber))
      (Set disproofNumber (Aggregate min disproofNumber))
    )
  )
  (FinalMoveSelection
    (Condition (eq proofNumber 0)
      (SelectNode argmin proofNumber)
    )
    (Condition (neq proofNumber 0)
      (SelectNode argmax visitCount)
    )
  )
)
```