Solving Konane Using Proof-Number Search and Extensions

Bachelorarbeit

Benedikt Siefke
471164

10.11.2025

(Slightly revised version as of 23.02.2026)

Supervisor: Prof. Dr. Benjamin Blankertz
Dr.-Ing. Stefan Fricke

Technische Universität Berlin
School of Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Neurotechnology

Kurzfassung

Konane, auch bekannt als Hawaiian Checkers, ist ein Zwei-Personen-Spiel mit perfekter Information, das auf einem rechteckigen Brett gespielt wird. Während es *schwache* Lösungen für einige kleinere Spielbrettgrößen gibt, sind größere Bretter eine Herausforderung. Diese Arbeit verfolgt zwei Ziele: Erstens sollen alle Brettgrößen bis einschließlich 8×8 *schwach* gelöst und dabei verschiedene Proof-Number-Search-Varianten miteinander verglichen werden. Zweitens sollen Bretter bis 6×6 durch die Konstruktion vollständiger Datenbanken mit Retrograde Analysis *stark* gelöst werden.

Um diese Ziele zu erreichen, wurde in C++ ein Solver implementiert, der Bitboards für die Zustandsrepräsentation, eine optimierte Zugerzeugung für einfache und mehrfache Sprünge sowie Transposition Tables mit Symmetriereduktion verwendet. Außerdem wurden PN^2 und df-pn parallelisiert und deren Leistung verglichen. Für die Retrograde Analysis haben wir eine neue effiziente Rangfunktion entwickelt und die Datenbank mittels einer Breitensuche auf den erreichbaren Zustandsraum beschränkt.

Die Implementierung konnte erfolgreich eine vollständige Datenbank für alle Spielbretter bis 6×6 konstruieren und alle Brettgrößen bis 7×7 *schwach* lösen. Die Versuche, die 8×7- und 8×8-Bretter *schwach* zu lösen, wurden nach einem Zeitlimit von 21 Tagen abgebrochen und bleiben daher ungelöst.

Abstract

Konane, also known as Hawaiian Checkers, is a two-player, perfect-information game played on a rectangular board. While *weak* solutions for some board sizes exist, larger board sizes remain a computational challenge. This thesis has two primary objectives. The first is to weakly solve board sizes up to 8×8 and to compare different Proof-Number Search variants in this setting. The second is to *strongly* solve boards up to 6×6 by constructing complete databases using Retrograde Analysis.

To achieve this, we implemented an efficient solver in C++ using bitboards for state representation, optimized move generation for single and multiple jumps, and Transposition Tables with symmetry reduction. We also parallelized PN^2 and df-pn, and evaluated their performance. For Retrograde Analysis, we introduced a new efficient ranking function for Konane and restricted the database to only include the reachable state space obtained by a breadth-first search from the starting position.

Our implementation successfully constructed a complete database for all boards up to 6×6 and *weakly* solved all board sizes up to 7×7. The attempts to solve 8×7 and 8×8 were terminated after a 21-day time limit and therefore remain unsolved.

Contents

List of Figures

List of Tables

1. Introduction

Solving two-player perfect-information games has been an important part of Artificial Intelligence research for over half a century [29].

However, the term "solved" can have several meanings. Allis [3] defines three distinct levels:

- **Ultra-weakly solved:** The game-theoretic value (win, loss, or draw) for the initial position has been determined.
- **Weakly solved:** From the initial position, a strategy is known that guarantees at least the game-theoretic value against any opponent.
- **Strongly solved:** For every legal position, a strategy is known that achieves the game-theoretic value for both players.

Over the years, this research has produced a growing list of solved games with increasing complexity. Early successes included **Qubic**, a $4 \times 4 \times 4$ Tic-Tac-Toe variant, which was *proven* to be a first-player win in 1980 by Patashnik [18]. This was followed by **Connect Four**, weakly solved independently by James D. Allen and Victor Allis in 1988, who both proved the first player has a winning strategy [29, 2, 1].

A significant leap in complexity was the weak solution for **Checkers**, a game with a vastly larger state space, which was achieved in 2007 by Schaeffer et al. [22].

A recent milestone is the solution of **Othello**, which was solved with a highly optimized version of Minimax with alpha-beta pruning ($\alpha\beta$) by Takizawa [25].

Another two-player perfect-information game is **Konane**, also known as Hawaiian Checkers. It is played on an $m \times n$ board that is filled with alternating black and white stones. Players take turns capturing the opponent's pieces by jumping over them, similar to Checkers, but horizontally and vertically instead of diagonally. The first player who is unable to move loses.

Solutions for Konane already exist for specific board sizes. According to Uiterwijk [27], a 2020 bachelor thesis by Kirk [14] determined the game-theoretic value for several boards, including 4×9, 5×7, and 6×6, using $\alpha\beta$ search. More recently, Wu [30] solved the 7×7 board in 2024 using an AND/OR search tree combined with Greedy Best-First Search.

While the 7×7 board has been solved, larger boards remain an open computational challenge. This thesis presents a new parallelized solver designed with the primary objective of weakly solving Konane for boards up to 8×8. As part of this development, we compare different variants of Proof-Number Search (PNS) to determine the most effective strategy. Furthermore, this thesis investigates a different solution type, aiming to strongly solve boards up to 6×6 using Retrograde Analysis.

1.1. Related Work

An early important paper on Konane was written by Ernst [9] in 1995, who analyzed the game in the context of combinatorial game theory (CGT). They showed how a position can be mathematically analyzed, simplified and decomposed into independent subgames.

Building on this work, several later papers studied specific Konane boards. In 2002, Chan and Tsai [7] investigated $1 \times n$ boards formed by three consecutive stone segments separated by one or two empty squares and computed their corresponding CGT values. More recently, Uiterwijk [27] focused on narrow rectangular boards, which he named Linear Konane ($1 \times n$), Double Konane ($2 \times n$), Triple Konane ($3 \times n$) and Quadruple Konane ($4 \times n$). He was able to completely solve Linear and Double Konane and provide partial solutions for the other two variants.

From a computational complexity perspective, Hearn [10] proved Konane to be PSPACE-complete, the same difficulty class as Hex [20].

A few works focused on creating strong playing agents. Thompson [26] trained a neural network to play Konane and proposed multiple evaluation functions for $\alpha\beta$. The most effective one was the ratio between the player's and the opponent's movable pieces. A bachelor thesis by Hendrick [11] compared Minimax and $\alpha\beta$ agents, while a thesis by Shokry [23] compared Minimax, $\alpha\beta$ and Monte Carlo Tree Search.

Only the two theses mentioned in the introduction concern solving Konane boards with search algorithms, and neither is publicly accessible. The public abstract[1] of the work by Wu [30] reports that Retrograde Analysis and backtracking were not ideal for solving Konane and that their approach of incorporating CGT was "ineffective". They were able to solve the 7×7 board in 25 hours, 45 minutes and 31 seconds using an AND/OR search tree combined with Greedy Best-First Search.

1.2. Structure

Chapter 2 introduces the rules of Konane and the theoretical foundations of the algorithms used. Chapter 3 details the implementation. Chapter 4 presents the results, and Chapter 5 discusses them in the context of related work. Finally, Chapter 6 concludes and outlines directions for future research.

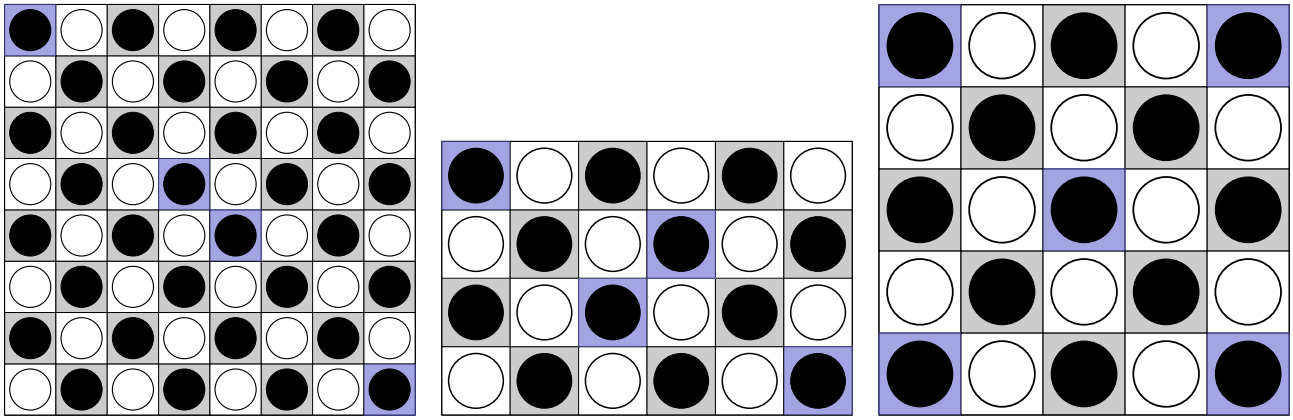
[1] https://www.airitilibrary.com/Article/Detail/U0021-NTNU46183

2. Background

This chapter introduces the game of Konane, detailing its rules, setup, and phases of play. It then explains the theoretical foundations of the algorithms used in this thesis, starting with Proof-Number Search (PNS) and its variants, and concluding with Retrograde Analysis.

2.1. Konane

Konane is played on a rectangular board that is filled with alternating black and white stones. The top left corner is always a black stone.



(a) 8×8 (b) 4×6 (c) 5×5

Figure 2.1.: Starting positions marked in blue for various board sizes.

The game proceeds in three phases:

1. **First Ply (Black):** The black player begins by taking a black stone from any corner or the center of the board. Figure 2.1 shows valid starting moves for various board sizes.
2. **Second Ply (White):** White removes a white stone that is orthogonally adjacent to the empty square created by Black's first move.
3. **Normal Play (Capturing Phase):** After the first two plies, stones of the opposing player are taken by jumping over them.
    - Jumps must be vertical or horizontal, landing in the empty space immediately behind the taken piece.

- A player can also take multiple enemy pieces in one round by jumping over multiple pieces in a line. Single and multiple jumps are shown in Figure 2.2.

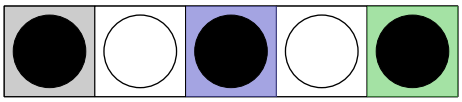


Figure 2.2.: Black can capture one stone and land on the square marked in blue or capture both white pieces and land on the green square.

The game ends once a player is unable to make a move. The other player wins.

2.2. Proof-Number Search

Proof-Number Search (PNS) is an algorithm first introduced by Allis [3] in 1994. It uses best-first search to determine the game-theoretic value of the root node. A node is considered *proven* if the first player has a forced win, *disproven* if the second player can force a win and *unknown* otherwise.

2.2.1. AND/OR tree

PNS represents the game as an AND/OR search tree. This tree uses alternating OR and AND nodes at each depth. OR nodes represent the perspective of the first (root) player and AND nodes represent the perspective of the opposing player. A tree is considered solved once the root node has been *proven* or *disproven*.

A node is *terminal* if it has no children and is a win or a loss for the first player. A *non-terminal leaf* is unexpanded. An *internal node* has at least one child.

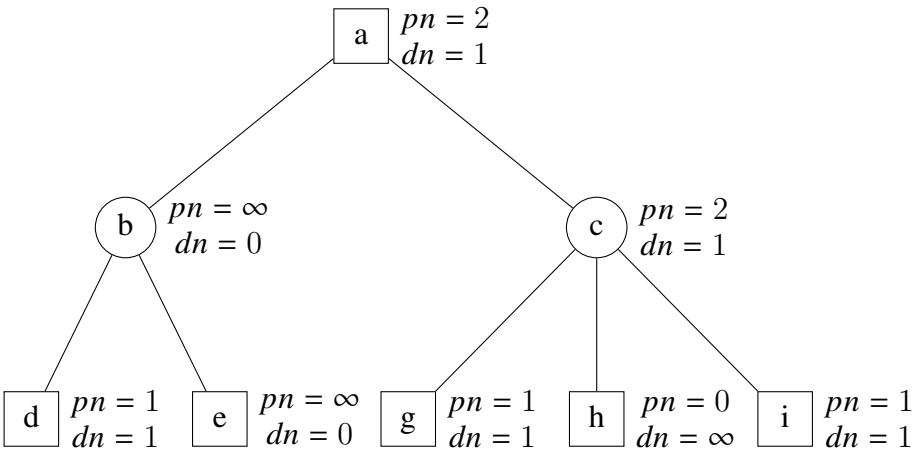


Figure 2.3.: AND/OR search tree: Squares are OR nodes and circles are AND nodes

2.2.2. Proof and disproof numbers

For each node v in the AND/OR tree, PNS maintains two nonnegative integers: the *proof number* $\mathrm{pn}(v)$ and the *disproof number* $\mathrm{dn}(v)$. The former represents the minimum number of leaf nodes that need to be proved in order to prove the node. The latter represents the minimum number of nodes that need to be disproved for the node to be considered disproved.

A *terminal* node that is a win for the first player has a $\mathrm{pn}(v) = 0$ and $\mathrm{dn}(v) = \infty$. If the position is a loss, the values are reversed.

An example of an AND/OR tree with realistic *proof* and *disproof numbers* is shown in Figure 2.3. Node **a** is the root node, nodes **b** and **c** are internal. Nodes **d**, **g** and **i** are non-terminal leaves that were initialized with $\mathrm{pn}(v) = 1$ and $\mathrm{dn}(v) = 1$. Nodes **e** and **h** are *terminal* nodes: **h** is a win for the first player, and **e** is a win for the second player.

2.2.3. Algorithm

PNS begins by creating the root node v_{root}. As an unexpanded, *non-terminal* leaf, this node is initialized with $\mathrm{pn}(v_{root}) = 1$ and $\mathrm{dn}(v_{root}) = 1$. The algorithm then enters its main loop. It stops once the root is proven or disproven or when a maximum number of nodes is reached.

The main loop consists of the four steps shown in Figure 2.4:

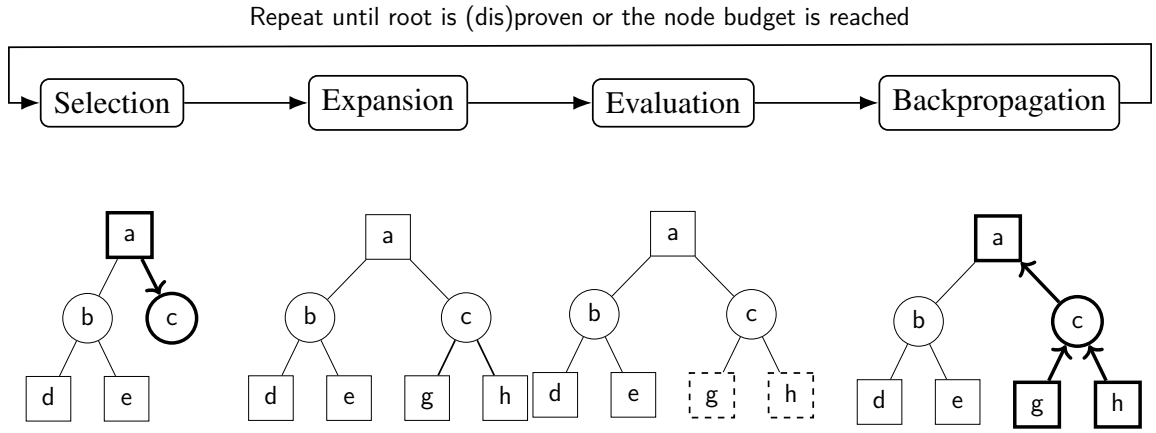


Figure 2.4.: The four stages of Proof-Number Search

1. **Selection**. Starting from the root node v_{root}, we descend until we reach an unexpanded leaf (or a *terminal*) of the search tree. This is the most proving node (MPN). At an OR node, we select the child with the smallest *proof* number; at an AND node, we select the child with the smallest *disproof* number.
2. **Expansion**. For each legal move from the selected node, we create a new node and evaluate it.
3. **Evaluation**. We evaluate a node as follows:
   - If it is a *terminal* node and the position is a win for the first player, initialize it with

$$\mathrm{pn}(v) = 0, \qquad \mathrm{dn}(v) = \infty.$$

- If it is a *terminal* node and the position is a loss for the first player, initialize it with

$$\mathrm{pn}(v) = \infty, \qquad \mathrm{dn}(v) = 0.$$

- Otherwise, initialize it with

$$\mathrm{pn}(v) = 1, \qquad \mathrm{dn}(v) = 1.$$

4. **Backpropagation**. For every parent of the node we recalculate $\mathrm{pn}(v)$ and $\mathrm{dn}(v)$:
   - If it is an OR node:

$$\mathrm{pn}(v) = \min_{v' \in \mathrm{Children}(v)} \mathrm{pn}(v')$$
$$\mathrm{dn}(v) = \sum_{v' \in \mathrm{Children}(v)} \mathrm{dn}(v')$$

   - Otherwise for an AND node:

$$\mathrm{pn}(v) = \sum_{v' \in \mathrm{Children}(v)} \mathrm{pn}(v')$$
$$\mathrm{dn}(v) = \min_{v' \in \mathrm{Children}(v)} \mathrm{dn}(v')$$

More detailed pseudo-code for PNS is shown in Algorithm 1.

A disadvantage of standard Proof-Number Search is that it requires storing the entire search tree in memory.

Algorithm 1 Proof-Number Search (PNS)

```
1: function PNS(root)
2:     Evaluate(root)
3:     while root.pn > 0 and root.dn > 0 do
4:         currentNode ← SelectMostProvingNode(root)
5:         Expand(currentNode)
6:         UpdateAncestors(currentNode)
7:     end while
8:     if root.pn == 0 then
9:         return PROVEN
10:    else
11:        return DISPROVEN
12:    end if
13: end function
```

2.2.4. Enhancements

Mobility initialization

Instead of initializing all *non-terminal* leaves with $(\mathrm{pn}(v) = 1, \mathrm{dn}(v) = 1)$, an enhancement known as **mobility initialization** uses n, the number of legal moves at that node [16]. The initialization values depend on the node type:

- OR nodes: $\mathrm{pn}(v) = 1, \mathrm{dn}(v) = n$,
- AND nodes: $\mathrm{pn}(v) = n, \mathrm{dn}(v) = 1$.

This enhancement significantly improved the performance of PNS on the game `Lines of Action` [28]. The algorithm generated only one-fifth the number of nodes and was able to solve substantially more test positions than the standard PNS.

Deleting solved subtrees

In PNS, once a node is proved or disproved, its children are no longer needed. Deleting these solved subtrees frees memory, reduces the chance of hitting memory limits, and yields a modest speedup in practice [28].

2.3. PN^2

Plain PNS often runs into memory limits on hard positions [28]. To mitigate this, Allis [3] introduced PN^2, a two-level algorithm: the first level (L1) runs proof-number search and, at its current most-proving node (MPN), invokes a second, bounded PNS (L2). The L2 search refines the evaluation of that child and returns updated proof/disproof information to L1; it terminates as soon as it proves or disproves the subproblem, or when its node budget is exhausted.

There are two ways proof-number search algorithms evaluate nodes [4]:

- **Immediate evaluation.** Each node is evaluated as soon as it is generated.
- **Delayed evaluation.** A node is only evaluated once it is selected as the MPN.

Standard PN^2 uses *delayed evaluation* at L1 by calling PNS on the MPN and *immediate evaluation* with the L2 search as described in Section 2.2. Pseudo-code for PN^2 is shown in Algorithm 2. It reuses the functions from PNS.

2.3.1. Budget

Allis [3] suggested setting the L2 node budget equal to the current L1 tree size N. This choice can make easy positions slower than plain proof-number search [28]. To address this, Breuker *et al.* [4] proposed allocating an L2 budget that is a logistic fraction of N. Let

$$f(N) = \frac{1}{1 + e^{(a-N)/b}}$$

Algorithm 1 Proof-Number Search (PNS) (continued)

```
1: function EVALUATE(node)
2:     if node is a terminal node then
3:         if node is PROVEN then
4:             node.pn ← 0; node.dn ← ∞
5:         else if node is DISPROVEN then
6:             node.pn ← ∞; node.dn ← 0
7:         end if
8:     else if node has children then
9:         if node is an OR node then
10:            node.pn ← min_{c∈children}(c.pn)
11:            node.dn ← Σ_{c∈children}(c.dn)
12:        else if node is an AND node then
13:            node.pn ← Σ_{c∈children}(c.pn)
14:            node.dn ← min_{c∈children}(c.dn)
15:        end if
16:    else
17:        node.pn ← 1; node.dn ← 1
18:    end if
19: end function

20: function SELECTMOSTPROVINGNODE(node)
21:    while node has children do
22:        if node is an OR node then
23:            node ← child c with minimum c.pn
24:        else if node is an AND node then
25:            node ← child c with minimum c.dn
26:        end if
27:    end while
28:    return node
29: end function

30: function EXPAND(node)
31:    Generate children for node
32:    for all child c of node do
33:        Evaluate(c)
34:    end for
35: end function

36: function UPDATEANCESTORS(node)
37:    while node ≠ null do
38:        old_pn ← node.pn
39:        old_dn ← node.dn
40:        Evaluate(node)
41:        if node.pn == old_pn and node.dn == old_dn then
42:            break
43:        end if
44:        node ← node.parent
45:    end while
46: end function
```

Algorithm 2 PN^2

```
1: function PN2(root)
2:     Evaluate(root)
3:     while root.pn > 0 and root.dn > 0 do
4:         v ← SelectMostProvingNode(root)
5:         B ← BUDGET(N)
6:         PNS(v, B)
7:         UpdateAncestors(v)
8:     end while
9:     if root.pn == 0 then
10:        return PROVEN
11:    else
12:        return DISPROVEN
13:    end if
14: end function
```

and define the L2 node budget as

$$B(N) \;=\; \lfloor f(N)\, N \rfloor.$$

Here, a is the inflection point (the L1 size at which $f = 0.5$), and $b > 0$ controls the slope of the transition. For small N, $f(N) \approx 0$ keeps L2 inexpensive; as N grows, $f(N) \to 1$, enabling deeper searches once the L1 tree is more established.

2.4. df-pn

While PN^2 reduces the memory requirements, it remains limited by memory and can exit prematurely. The standard PNS algorithm also suffers from a significant search inefficiency: it must re-select the Most-Proving Node (MPN) by traversing from the root on every single iteration [28]. To address both problems, Nagai [17] introduced **Depth-first Proof-number search (df-pn)**, which traverses the tree depth-first and is primarily limited by the size of the transposition table [16].

Df-pn controls its search using two thresholds for every node v: $\mathrm{pt}(v)$ for the *proof number* and $\mathrm{dt}(v)$ for the *disproof number*. These thresholds define the search bounds for the subtree at v and are used to check if the MPN is still within that subtree.

When df-pn examines a node v, it first recalculates $\mathrm{pn}(v)$ and $\mathrm{dn}(v)$ based on its children's current values the same way as in PNS. The search of the subtree at v continues until one of these thresholds is reached: $\mathrm{pn}(v) \geq \mathrm{pt}(v)$ or $\mathrm{dn}(v) \geq \mathrm{dt}(v)$. If this condition is met, the algorithm backtracks.

The search begins at the root node, v_{root}, with its thresholds set to infinity: $\mathrm{pt}(v_{root}) = \infty$ and $\mathrm{dt}(v_{root}) = \infty$. As the search descends, new thresholds are calculated and passed down to the selected child:

- At an OR node p, the algorithm selects the child c_1 with the smallest *proof number*. Let $\mathrm{pn}(v)_2$

be the second-smallest *proof number* among p's children. The new thresholds for c_1 are set as:

$$\mathrm{pt}(c_1) = \min(\mathrm{pt}(p), \mathrm{pn}(v)_2 + 1) \tag{2.1}$$
$$\mathrm{dt}(c_1) = \mathrm{dt}(p) - \mathrm{dn}(p) + \mathrm{dn}(c_1) \tag{2.2}$$

- At an AND node p, the algorithm selects the child c_1 with the smallest *disproof number*. Let $\mathrm{dn}(v)_2$ be the second-smallest *disproof number* among p's children. The new thresholds for c_1 are set as:

$$\mathrm{pt}(c_1) = \mathrm{pt}(p) - \mathrm{pn}(p) + \mathrm{pn}(c_1) \tag{2.3}$$
$$\mathrm{dt}(c_1) = \min(\mathrm{dt}(p), \mathrm{dn}(v)_2 + 1) \tag{2.4}$$

Pseudo-code for df-pn is shown in Algorithm 3. It reuses the functions EVALUATE and EXPAND from PNS.

2.4.1. Enhancements

A significant weakness of df-pn emerges when the search space is very large and exceeds the available transposition table memory. In this scenario, df-pn spends most of its time repeatedly rebuilding parts of the search tree that have been overwritten in the table [19].

To prevent this, Pawlewicz and Lew [19] suggested the $1 + \epsilon$ trick. It changes the formula for the threshold at OR nodes for the *proof number* to:

$$pt[c_1] = \min(pt[p], \lceil pn_2(1 + \epsilon) \rceil) \tag{2.5}$$

2.5. Parallelization

For many difficult problems, a single-threaded search is simply too slow to be practical. Parallelization addresses this by distributing the computational load across multiple threads, but this process introduces its own costs.

Brockington and Schaeffer [5] introduce three types of overhead: **Search overhead** is the additional amount of work being done, measured in nodes visited compared to a sequential implementation. **Synchronization overhead** is the time spent at synchronization points when one thread is idle waiting for another thread. **Communication overhead** is the time or cost created by exchanging information between threads.

An important property of parallel algorithms is scaling, which measures how efficiency improves as the number of threads (T) increases. The primary metric for this is speedup, expressed as t_1/t_T, where t_1 is the time it takes one thread to finish successfully and t_T is the time for T threads to finish successfully [21].

Several strategies have been developed for parallelizing Proof-Number Search algorithms.

Algorithm 3 df-pn

```
1: function DF-PN(root)
2:     EVALUATE(root)
3:     while root.pn > 0 and root.dn > 0 do
4:         SEARCH(root, ∞, ∞)
5:     end while
6:     if root.pn == 0 then
7:         return PROVEN
8:     else
9:         return DISPROVEN
10:    end if
11: end function
12: function SEARCH(node, pt, dt)
13:    EVALUATE(node)
14:    if node.pn ≥ pt or node.dn ≥ dt then
15:        return
16:    end if
17:    if node is a non-terminal leaf then
18:        EXPAND(node)
19:        for all child c of node do
20:            EVALUATE(c)
21:        end for
22:    end if
23:    EVALUATE(node)
24:    if node is OR then
25:        c_1 ← arg min_c c.pn
26:        pn(v)_2 ← secondMin{c.pn}
27:        SEARCH(c_1, min(pt, pn(v)_2 + 1), dt − node.dn + c_1.dn)
28:    else
29:        c_1 ← arg min_c c.dn
30:        dn(v)_2 ← secondMin{c.dn}
31:        SEARCH(c_1, pt − node.pn + c_1.pn, min(dt, dn(v)_2 + 1))
32:    end if
33:    EVALUATE(node)
34: end function
```

The first existing parallelization for PNS is ParaPDS by Kishimoto [15]. They used a distributed master-servant design where a master process coordinated several worker processors. The speedup was 3.6 on 16 processors [21].

RP-PNS and RP-PN^2 by Saito et al. [21] use a principal variation (PV) thread that behaves like sequential PNS and alternative threads that use a probability distribution during selection to explore near-PV branches. They achieved a scaling of 4.7 for 8 threads with RP-PN^2 and a search overhead of 33%.

Another approach is presented for df-pn by Kaneko [13]. They use a shared global transposition table and access locks to prevent multiple expansions of the same node. At the selection step for a node n, threads use virtual proof and disproof numbers that include a congestion term. At OR nodes:

$$vpn(n,c) = pn(c) + T(n,c) \tag{2.6}$$

and at AND nodes:

$$vdn(n,c) = dn(c) + T(n,c) \tag{2.7}$$

where $T(n,c)$ is the number of threads currently inside the subtree of the child node c. Since the smallest $vpn(n,c)$ is selected at OR nodes, $T(n,c)$ acts as a congestion factor. The algorithm does the same for disproof numbers at AND nodes. $vpn(n,c)$ and $vdn(n,c)$ are also used during back-propagation instead of $\mathrm{pn}(c)$ and $\mathrm{dn}(c)$. Once a node n is *proven* or *disproven*, a signal is sent to stop other threads working in the subtree of n, to prevent unnecessary extra work.

They reported a scaling factor of ~ 3.5 on 8 threads with a search overhead of 15%.

2.6. Retrograde Analysis

Another technique for solving games is Retrograde Analysis. Rather than exploring the game tree from the root node, Retrograde Analysis starts from *terminal* positions and propagates values backward through the game graph [29].

Solutions can be stored in an endgame database, which can later be used with forward search such as PNS or PN^2. To find a solution of a position in the database we need a bijection that maps every position to a unique integer (rank of the position). A **ranking function** maps from a permutation or combination to a rank and an **unranking function** does the inverse.

A complete database constitutes a strong solution, since every possible position, even if it is not reachable, is assigned its game-theoretic value.

While this approach is thorough, it wastes significant computation on solving positions that can never occur. A more targeted strategy is to first map out the reachable state space. This is done by performing a Breadth-first search from the root to discover every state that can actually occur in a game, and then applying the analysis only to those positions [8].

3. Methods

This chapter details the practical implementation and the search algorithms discussed in Chapter 2.

It first describes how the state is represented, how legal moves are generated and how the state space is reduced by taking into account symmetry. It then details how the Transposition Table is structured, and in which way the Proof-Number Search variants are implemented and how they are evaluated. Finally, it explains how the Endgame Database is built and used to solve boards strongly.

The solver was implemented in C++23 due to its speed and its low-level control over memory.

3.1. State Representation

The game state is represented internally using a bitboard representation. This approach uses two 64-bit integers, one for each player, where each bit corresponds to a single square on the board. The state also stores the player to move and the current game ply. The state is managed as a single, mutable object within the search algorithm, which is updated incrementally using the functions *doMove* and *undoMove*.

3.2. Move Generation

For the following section, we use the syntax introduced by Browne [6]:

$bits_w$ = pieces belonging to White

$bits_b$ = pieces belonging to Black

$bits_p$ = pieces of the current player

$bits_o$ = pieces of the opponent

$bits_m$ = the cell at which the last move was played

$mask$ = mask of all valid cells

$full$ = $bits_w \mid bits_b$ = the set of cells occupied by either player

$empty$ = $\sim full \,\&\, mask$ = the set of empty cells

The function `pop_lsb`(x) returns the least-significant bit as a mask and clears it from x.

We will use standard logical operations (&, |, ~) and the morphological operation *Dilation*, denoted by $\oplus$. As described in the paper, dilation is the expansion of on-bits by a given pattern, which for game tasks is typically an expansion to adjacent neighbors.

A dilation in a single direction d is given by Browne [6] as:

$$bits_d = bits \mid (bits >> shift_d) \tag{3.1}$$

where $shift_d$ is the number of bit positions to shift for that direction:

$$\begin{aligned} shift_N &= cols + 1 \\ shift_E &= 1 \\ shift_S &= -(cols + 1) \\ shift_W &= -1 \end{aligned}$$

3.2.1. Removals

For the first move (ply 1), where the black player removes a black stone from the center or the corners we use pre-calculated bit masks.

For the second move (ply 2), we use a one-step dilation on the removed black stone to detect neighboring white stones, as shown in Algorithm 4.

Algorithm 4 Second-ply removal

1: $moves \leftarrow (bits_m \oplus 1)\ \&\ bits_w$

3.2.2. Single-Jumps

After the initial two removals, players move by capturing an opponent's piece, which involves jumping over it into an adjacent empty square.

To calculate all possible single-jump moves, we first identify any opponent's pieces that are adjacent to one of our own. Then, for each of these adjacent opposing pieces, we check if the square immediately behind it (in the same direction) is empty. This process is detailed in Algorithm 5.

Algorithm 5 Finding Single-jump capture landing squares

1: $landings \leftarrow 0$
2: **for all** directions d **do**
3: $\quad adj_d \leftarrow (bits_p >> shift_d)\ \&\ bits_o$
4: $\quad land_d \leftarrow (adj_d >> shift_d)\ \&\ empty$
5: $\quad landings \leftarrow landings \mid land_d$
6: **end for**

3.2.3. Multi-Jumps

The rules permit multi-jumps, where a player captures multiple pieces in a single turn. This is performed by a series of consecutive jumps from the same piece, all in the same orthogonal line.

Algorithm 6 generates these multi-jump moves. It iterates through each direction d and finds all initial jumps ($single\_jumps$), just as in the single-jump algorithm. For each initial landing square ($landing$), it then enters an inner loop (line 10) to find subsequent jumps in the same direction. It uses an updated occupancy bitboard (occ) to track the piece's new position and the removed stones. This ensures that subsequent jumps only land on cells that are empty or have become empty during the current turn's sequence. This process continues until no further valid jumps can be found along that line.

Algorithm 6 Move generation with Multi-jumps

```
 1: for all directions d do
 2:     adj ← (bits_p >> shift_d) & bits_o
 3:     single_jumps ← (adj >> shift_d) & empty
 4:     while single_jumps ≠ 0 do
 5:         landing ← pop_lsb(single_jumps)
 6:         start ← landing << (2·shift_d)
 7:         captured ← landing >> shift_d
 8:         moves ← Move(start, landing)
 9:         occ ← (full & ~(start | captured)) | landing
10:         line ← landing
11:         while true do
12:             victims ← (line >> shift_d) & bits_o
13:             landings ← (victims >> shift_d) & ~occ
14:             if landings = 0 then break
15:             end if
16:             tmp ← landings
17:             while tmp ≠ 0 do
18:                 next_landing ← pop_lsb(tmp)
19:                 moves ← Move(start, next_landing)
20:             end while
21:             occ ← occ | victims | landings
22:             line ← landings
23:         end while
24:     end while
25: end for
```

3.3. Transposition Tables

We use Transposition Tables (TT) with Zobrist Hashing for detecting already encountered positions.

As shown in Figure 3.1, the layout for an entry is:

- The upper 16 bits of the position's Zobrist Key. (**tag16**)
- The best move (**move**)
- The search depth at which the entry was created (**depth**)
- A flag that denotes if the node is *proven*,*disproven* or *unknown* (**pns_flag**)
- The age of the entry for the replacement strategy (**age**)
- The node's *proof number* (**proof_number**)
- The node's *disproof number* (**disproof_number**)

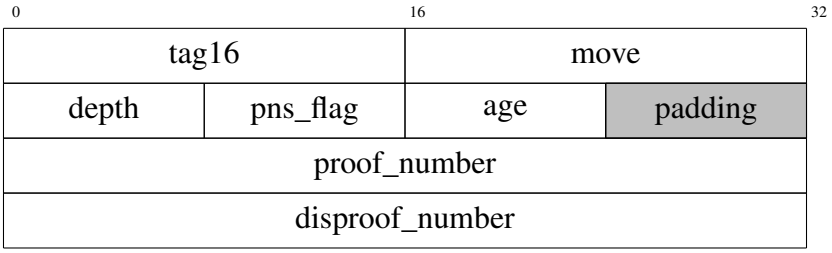


Figure 3.1.: Layout of a PNS-TT entry

To improve cache performance, the table is organized into 64-byte clusters. As shown in Figure 3.2, each cluster contains four 16-byte TTEntry objects.

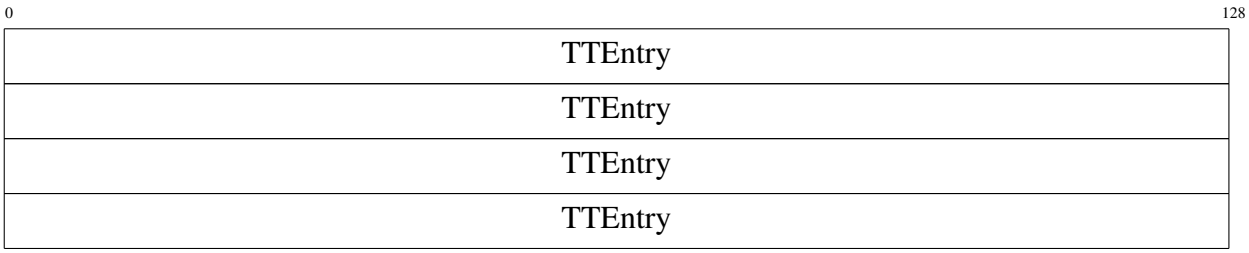


Figure 3.2.: A cluster consists of four TT entries

We use the lower 16 bits of the Zobrist hash to find the right cluster and the upper 16 bits to identify the right entry.

When storing a new entry, the replacement strategy prioritizes overwriting a matching key or an empty slot; otherwise, it replaces the entry with the lowest score, disfavoring entries that are already solved, have a greater depth, or are more recent.

3.4. Symmetry Reduction

In Konane, many positions are strategically equivalent even though the stones are in different places. A state might be identical to another by rotation or reflection. A standard implementation using Transposition Tables would treat all of these positions as unique states, as their hash keys would differ.

To solve this, we map all symmetrically equivalent positions to a single canonical representation. This is achieved by calculating a canonical Zobrist key for every position. This canonical key, rather than the position's direct key, is then used for Transposition Table probes.

Which symmetries are valid depends on the shape of the board:

- If the number of files F is odd: allow horizontal flip.
- If the number of ranks R is odd: allow vertical flip.
- If $R + F$ is even: allow 180° rotation.
- If $R = F$ (square board): allow main- and anti-diagonal flips.
- If $R = F$ is odd: also allow 90° and 270° rotations.

For fast transformations, we use precalculated lookup tables. We compute the canonical key of a state s as

$$K_{\text{can}}(s) = \min_{T \in \mathcal{S}(R,F)} Z(T(s)),$$

where $\mathcal{S}(R, F)$ is the set of valid symmetries for the current (R, F) and $Z(s)$ is the Zobrist hashing function.

We evaluate the impact of canonical keys on a 5×5 board by running a Breadth-first search (BFS) twice: (i) with canonical Zobrist keys and (ii) without. We count the number of new unique states that are first encountered at each ply.

3.5. PNS

The PNS algorithm is implemented as described in Section 2.2 including mobility initialization and deletion of solved subtrees.

To avoid the overhead of standard memory allocation, we use memory arenas. This method pre-allocates one large block of memory for the entire search. New nodes are then created by simply pointing to the next available unused spot within that block[1].

We use the following steps during evaluation:

1. **Transposition Table Probe:** The TT is checked first. If the position is found and marked as *proven* or *disproven*, this value is used immediately, and no further evaluation is needed.
2. **Endgame Database Probe:** Next the endgame database is queried. If the database returns a definitive win or loss, we treat it as a *terminal*. The result is then also immediately stored in the TT.
3. **Terminal Node Check:** If the position is *terminal* and was not found in the TT or database,

[1] https://medium.com/@sgn00/high-performance-memory-management-arena-allocators-c685c81ee338

the node is initialized based on the winner relative to the root player:

Win for root player:	$\mathrm{pn}(v) = 0,$	$\mathrm{dn}(v) = \infty$
Loss for root player:	$\mathrm{pn}(v) = \infty,$	$\mathrm{dn}(v) = 0$

4. **Mobility Initialization:** If the node is not *terminal*, it is a *non-terminal* leaf. The **mobility initialization** enhancement is applied, using the number of legal moves n to set the initial proof and disproof numbers:

OR node:	$\mathrm{pn}(v) = 1,$	$\mathrm{dn}(v) = n$
AND node:	$\mathrm{pn}(v) = n,$	$\mathrm{dn}(v) = 1$

3.6. df-pn

3.6.1. Parallelization

For parallelization we use the approach described by Kaneko [13], which was discussed in Section 2.5.

The original paper suggested using explicit locks and unlocks at nodes. We instead pack $\mathrm{pn}(v)$ and $\mathrm{dn}(v)$ into a single 128-bit atomic and use compare-and-swap (CAS) when updating nodes to achieve lock-free value updates [12]. Expansions use a spinlock.

In each node a thread sets a bit in a 64-bit unsigned integer to indicate that they are currently in the subtree of the node.

To avoid synchronization overhead, each thread uses its own thread-local memory arenas, allowing new nodes to be created without any locks or atomic operations.

3.7. PN^2

PN^2 is implemented with a logistic budget function as described in subsection 2.3.1. We set a minimum budget of 100 nodes for each L2 search. The evaluation function is the same as in PNS.

3.7.1. Parallelization

Our parallelization strategy for PN^2 is also based on the method by Kaneko [13], which we apply specifically to the PN^2's L1 search. The core implementation details from Section 3.6.1 are reused here.

However, there are two major differences:

1. $vpn(n, c)$ and $vdn(n, c)$ are not used during backpropagation.
2. No stop signals are sent to threads operating within the subtree of a node that has just been *proven* or *disproven*.

Stop signals are not necessary for PN^2 since selection always starts from the root compared to df-pn.

3.8. Evaluation Metrics

We compare the performance of PNS, PN^2, df-pn and $\alpha\beta$ on the starting position of the 6×6 board, using their completion time and a node count. Completion time is the total time from the start of the search until it successfully terminates, measured in seconds.

Similar to Allis [3] and Van Den Herik and Winands [28], we use $\alpha\beta$ with depth-first iterative-deepening and Transposition Tables and count nodes as follows:

- For $\alpha\beta$ with iterative deepening, we count nodes at depth i only during the first iteration in which that level is reached.
- For PNS and df-pn, we count nodes once they are first evaluated.
- For PN^2, we count nodes when they are first evaluated, and keep separate counters for the L1 search and for all L2 searches.

For the evaluation function for $\alpha\beta$, we use the ratio between the player's movable pieces and the opponent's movable pieces, as suggested by Thompson [26].

Additionally, we evaluate the parallel scaling performance of PN^2 and df-pn on the 6×6 board. We use the scaling metrics introduced in Section 2.5. t_1 denotes the completion time with one thread and t_n the completion time with n threads. The speedup S_n and efficiency E_n for n threads are defined as

$$S_n = \frac{t_1}{t_n}, \qquad E_n = \frac{S_n}{n}.$$

Search overhead is measured as $\mathrm{nodes}_n/\mathrm{nodes}_1$, where nodes_n is the number of evaluated nodes with n threads. We also report the node throughput nodes_n/t_n in nodes per second.

3.9. Endgame Database

We construct the database layer by layer. Layer n contains all positions with exactly n stones. Layer 0 is the empty board. To compute it, we loop through every possible position with exactly n stones and generate its legal moves M and evaluate it as follows:

1. If M is empty, we mark the position as a LOSS for the player whose turn it currently is.
2. Otherwise, we look up each move in the previous layers: If any move leads to a LOSS, we mark the position as a WIN, otherwise all positions lead to a WIN and we mark the position as a LOSS.

Only a single pass over layer n is required, since every legal move in Konane removes at least one stone. Consequently, every successor of a position in layer n lies in some layer $k < n$ and has already been evaluated.

3.9.1. Ranking

For Retrograde Analysis we need a bijection that maps every position to a unique number. We cannot reuse the Zobrist hashes since they are non-injective. Instead, we define a combinatorial ranking.

Instead of constructing a single index over all N squares on the board, we exploit an invariant of Konane. The starting board is constructed with white and black stones in a "checkerboard" pattern, where each stone's neighbors are all of the opposite color. Since pieces can only move by jumping an even number of squares, a piece on a square occupied by a black stone can only ever land on a square of the same color. The same holds true for white.

This allows us to treat the board as two independent, disjoint sets of squares and we only need to compute two smaller, independent ranks: rank_B for the N_B black squares and rank_W for the N_W white squares.

To calculate these ranks we use **combinadic ranking**. It maps a unique integer to every possible combination of k items chosen from a set of n items using the binomial function [24].

We first map all pieces in a position P to their local indices, creating two sets:

- $P_B = \{\beta_1, \ldots, \beta_b\} \subset B$, the sorted set of local indices for the b black pieces.
- $P_W = \{\omega_1, \ldots, \omega_w\} \subset W$, the sorted set of local indices for the w white pieces.

The ranking for each set is given by:

$$\text{rank}_B(P_B) = \sum_{j=1}^{b} \binom{\beta_j}{j} \qquad \text{rank}_W(P_W) = \sum_{j=1}^{w} \binom{\omega_j}{j}$$

These two independent ranks, $\text{rank}_B(P_B)$ and $\text{rank}_W(P_W)$, are then combined to produce the final, unique rank $R(P)$ for the entire position. The index for the current layer with T pieces is split into sub-layers, each containing all positions with a specific combination of w white pieces and b black pieces such that $w + b = T$.

The ranking function is:

$$R(P) = \underbrace{\sum_{i=0}^{w-1} \left(\binom{N_B}{T-i} \times \binom{N_W}{i} \right)}_{\text{Base Offset}} + \underbrace{\left(\text{rank}_B(P_B) \times \binom{N_W}{w} \right)}_{\text{Black Piece Offset}} + \underbrace{\text{rank}_W(P_W)}_{\text{White Piece Offset}}$$

The ranking formula consists of these parts:

- **Base Offset:** The first term calculates the total size of all sub-layers that come before the current position's combination.
- **Black Piece Offset:** The second term finds the position for this combination of black pieces within that sub-layer. The value $\binom{N_W}{w}$ is the total number of ways to arrange the w white pieces.
- **White Piece Offset**: Lastly, we locate the white combination inside that block.

Figure 3.3 shows how an index for a layer with $T = 4$ pieces would be structured. It also shows how a position with three white stones and one black stone would be accessed.

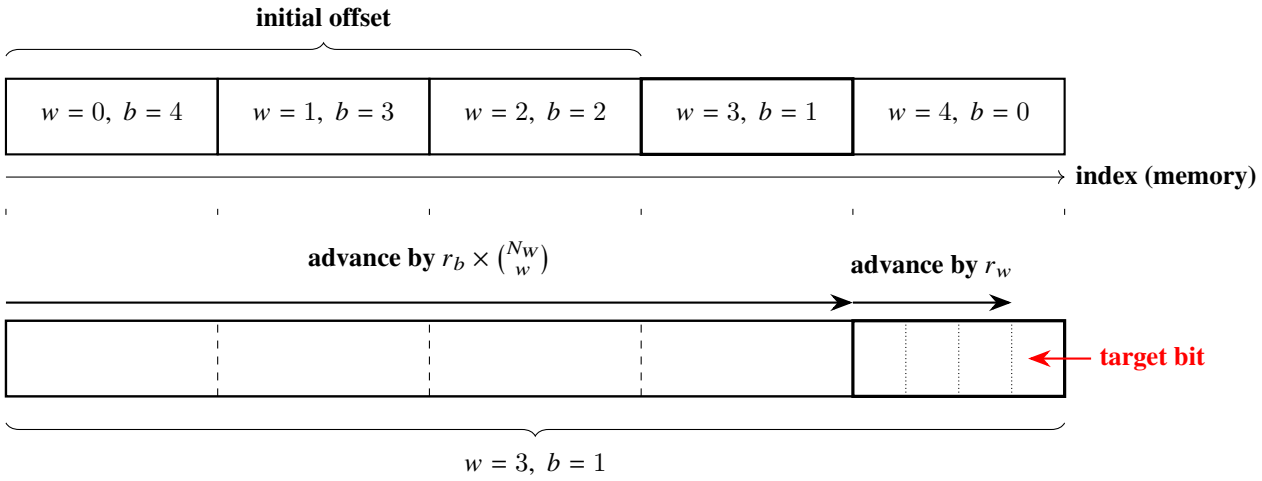


Figure 3.3.: Index layout for Layer 4 of a 4×4 board and an example access of sub-layer (w=3, b=1)

3.9.2. Reachable State Space

To strongly solve a board, a complete database is required. However, building this for the entire state space is infeasible for larger board sizes. Therefore, we first run a Breadth-first search (BFS) from the root node. During this search, we use Symmetry Reduction as described in Section 3.4 and maintain a set of already encountered positions to obtain the symmetry-reduced reachable state space. We cannot use transposition tables here, since entries may be replaced. After running BFS, we construct the database layer by layer only including positions inside the set.

For every board size we were able to strongly solve, we report the following:

- The total number of unique reachable states.
- The number and percentage of states where Black wins
- The number and percentage of states where White wins
- The number and percentage of *terminal* positions

4. Results

This chapter presents the experimental findings. It begins by quantifying the effectiveness of symmetry reduction. It then compares the single-threaded performance of all implemented search algorithms, evaluates the parallel scaling of PN^2 and df-pn, and concludes by presenting the final game-theoretic outcomes for both weakly and strongly solved Konane boards.

The parameters $a = 150000$ and $b = 30000$ are used for PN^2's budget function and were found by a grid search on 6×6. df-pn uses the parameter $\epsilon = 0.15$.

The experiments were carried out on a dual-socket Linux server with two Intel Xeon Gold 6326 CPUs (16 cores per socket; 64 threads) and 510 GB RAM.

4.1. Symmetry Reduction

The results of the Breadth-first search with and without symmetry reduction for 5×5 are shown in Table 4.1. At every ply, the search with symmetry reduction recorded a lower number of new unique states compared to the search without symmetry reduction.

Figure 4.1 shows all unique states of 5×5 for all 22 plies of the game with and without canonical keys.

Table 4.1.: New unique states at plies 1–10 on a 5×5 board, with and without symmetry reduction.

	Ply 1	**Ply 2**	**Ply 3**	**Ply 4**	**Ply 5**	**Ply 6**	**Ply 7**	**Ply 8**	**Ply 9**	**Ply 10**
With symmetry reduction	2	2	3	9	34	138	440	1492	4166	10784
Without symmetry reduction	5	12	20	72	268	1096	3512	11912	33244	86132

4.2. Algorithm Comparison

Table 4.2 shows the performance of the Proof-Number Search algorithms PNS, PN^2 and df-pn compared to a standard $\alpha\beta$ implementation with Transposition Tables on a 6×6 board.

$\alpha\beta$ took 4432.66 seconds and explored 1546.77 million nodes. Of the Proof-Number Search variants, df-pn recorded the shortest search time at 39.73 seconds and evaluated 46.36 million nodes. PN^2 created the most nodes at 385.93 million and its search time of 99.03 seconds was the longest of the three. PNS recorded a time of 70.96 seconds and a node count of 136.83 million nodes.

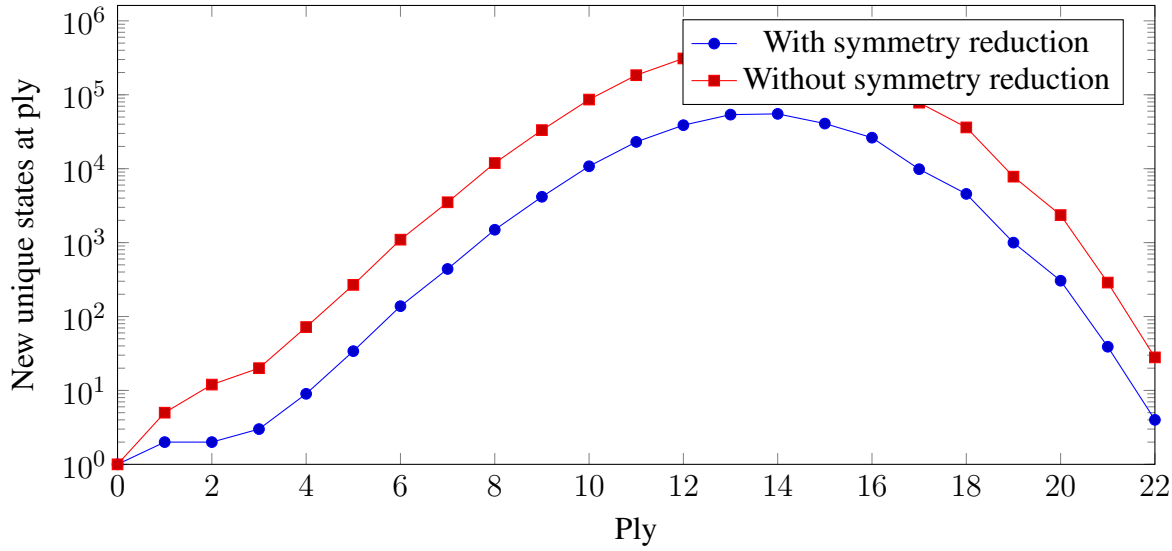


Figure 4.1.: Per-ply state size on a 5×5 board with and without symmetry reduction. The y axis is log-scaled.

Table 4.2.: Search performance with completion times in seconds and nodes created scaled to millions on a 6×6 board.

Algorithm	**Time (s)**	**Nodes (M)**
$\alpha\beta$	4432.66	1546.77
PNS	70.96	136.83
PN^2	99.03	385.93
df-pn	39.73	46.36

4.3. Parallel Scaling Performance

We tested the scaling performance of df-pn and PN^2. The results regarding time, speedup, efficiency and search overhead for 1, 2, 4, 8 and 12 threads are shown in Table 4.3. The speedup and node throughput are also visualized in Figure 4.2.

Table 4.3.: Scaling comparison of PN^2 and df-pn on a 6×6 board. Eff. is efficiency (speedup/threads) and Over. is search overhead.

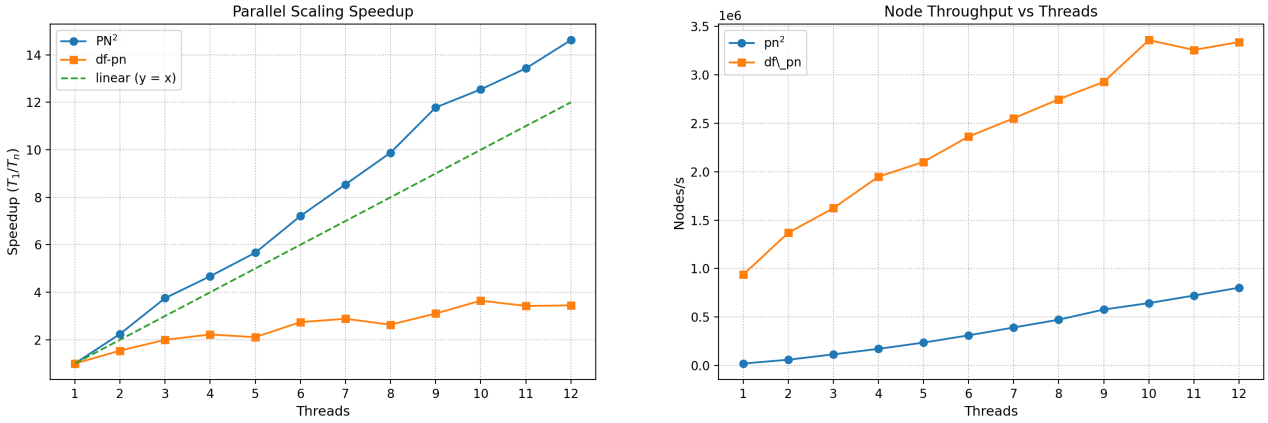
PN^2

Threads	**Time (s)**	**Speedup**	**Eff.**	**Over.**
1	99.77	1.00	1.00	1.00
2	44.44	2.25	1.12	1.35
4	21.34	4.68	1.17	1.89
8	10.10	9.88	1.23	2.47
12	6.83	14.62	1.22	2.84

df-pn

Threads	**Time (s)**	**Speedup**	**Eff.**	**Over.**
1	52.24	1.00	1.00	1.00
2	33.86	1.54	0.77	0.95
4	23.49	2.22	0.56	0.94
8	19.81	2.64	0.33	1.11
12	15.15	3.45	0.29	1.03

In the single-thread case, df-pn completes the search in 52.24 seconds, as shown in Table 4.3. As more threads are added, the execution time decreases, reaching 15.15 seconds at 12 threads. Figure 4.2a shows the speedup increasing to a maximum of 3.45 at 12 threads. This corresponds to a decrease in efficiency, which falls to 0.29. The search overhead for df-pn remains near 1.0 for all thread counts, fluctuating between 0.94 and 1.11.



(a) Observed speedup vs. linear scaling.

(b) Node search throughput (nodes per second).

Figure 4.2.: Parallel scaling performance of PN^2 and df-pn on a 6×6 board.

PN^2 starts with a single-thread time of 99.77 seconds. Its execution time decreases as threads are added, finishing at 6.83 seconds on 12 threads.

As seen in Figure 4.2a, PN^2 achieves **superlinear speedup** [**?**], which is a speedup greater than the number of threads. It reaches a maximum speedup of 14.62 at 12 threads. The efficiency is at its maximum of 1.23 at 8 threads. This scaling occurs alongside an increasing search overhead, which reaches 2.84 at 12 threads. Figure 4.2b shows that PN^2's node search throughput increases near-linearly as threads are added.

4.4. Weakly Solved Boards

All board sizes up to and including 7×7 were successfully weakly solved. The attempts for 8×7, 7×8 and 8×8 were terminated after 21 days. The game-theoretic outcomes for all board sizes investigated are summarized in Figure 4.3. 1×1, 5×1, 1×5, 5×5, 7×5, 5×7 and 7×7 were a win for Black, while all other sizes with determined outcomes were wins for White.

While the 8×7 board was not fully solved, it was determined that the central opening move (illustrated in Figure 4.4) results in a loss for Black.

Table 4.4 shows how long it took to solve each square board and how many nodes PN^2 has created during L1 search and L2 searches.

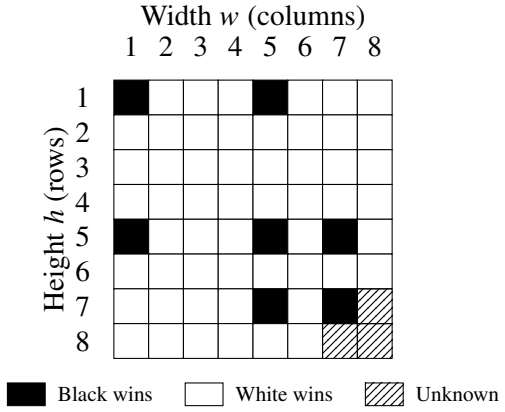


Figure 4.3.: Grid of winners by board size. Black cells denote a win for Black, white cells denote a win for White, and dashed cells denote unknown outcomes.

Table 4.4.: Measured completion times, L1 nodes, and L2 nodes for solving square boards of increasing size with PN^2.

Board	**Time (ms)**	**L1 nodes**	**L2 nodes**
1×1	49.00	2	3
2×2	55.00	3	9
3×3	62.00	6	59
4×4	69.00	5	165
5×5	117.00	5,959	163,765
6×6	6,630.00	7,592,521	247,048,448
7×7	4,596,031.05	33,827,638	518,131,212,527

4.5. Strongly Solved Boards

We were able to strongly solve all boards up to 6×6. The initial Breadth-first search for 6×6 took 428 seconds and the construction of the database took 628 seconds. Table 4.5 reports the size of the symmetry-reduced state space, as well as the number of Black wins, White wins, and *terminal* positions. As the board size increases, the size of the state space increases exponentially. From only 9 states on the 3×3 board to roughly 3.7×10^9 states on the 6×6 board.

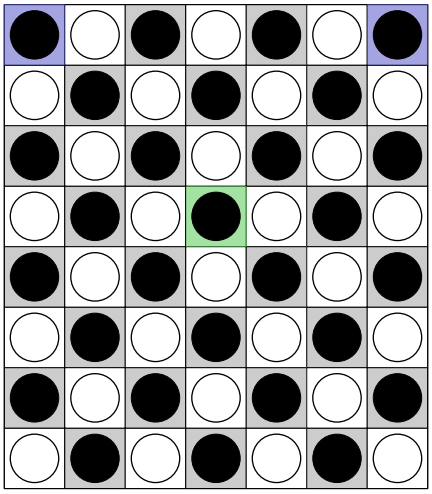


Figure 4.4.: The opening move in the center marked in green leads to a loss for Black. The other two openings marked in blue were not solved in time.

Table 4.5.: Symmetry-reduced reachable state space of boards of increasing size.

Board	**Unique Nodes**	**Black Wins**	**White Wins**	**Terminals**
3×3	9	2 (22.22%)	7 (77.78%)	3 (33.33%)
4×3	95	52 (54.74%)	43 (45.26%)	25 (26.32%)
4×4	978	457 (46.73%)	521 (53.27%)	104 (10.63%)
5×4	28,202	12,987 (46.05%)	15,215 (53.95%)	2,104 (7.46%)
5×5	270,692	205,797 (76.03%)	64,895 (23.97%)	8,458 (3.12%)
6×5	57,012,291	26,179,451 (45.92%)	30,832,840 (54.08%)	651,033 (1.14%)
6×6	3,739,049,319	1,766,388,662 (47.24%)	1,972,660,657 (52.76%)	11,635,291 (0.31%)

5. Discussion

This chapter analyzes and interprets the results from Chapter 4. We will discuss the performance of the symmetry reduction, compare the different search algorithms, investigate the parallel scaling of PN^2 and df-pn, and place the newly solved boards in the context of prior work.

5.1. Symmetry Reduction

As detailed in Section 3.4, the 5×5 board has a full set of eight symmetries:

- The identity
- Three rotations (90°,180°, 270°)
- Four reflections (horizontal, vertical, main-diagonal and anti-diagonal)

The results from the experiment directly reflect this property. The ratio of nodes between the search with and without symmetry reduction is approximately 8 across the search space. For instance at ply 10 the ratio is $86132/10784 \approx 7.99$.

5.2. Algorithm Comparison

All Proof-Number Search algorithms were faster and more efficient in finding a solution than $\alpha\beta$. Since we were able to strongly solve the 6×6 board we can also estimate how much of the total search space each algorithm explored: $\alpha\beta$ searches about 41% of the state space, whereas PNS only needs around 3.7%, PN^2 around 10.3%, and df-pn as little as 1.2% of all 3,739,049,319 states.

In terms of completion time this corresponds to large speedups over $\alpha\beta$ on 6×6: df-pn is roughly $4432.66/39.73 \approx 111\times$ faster, PNS about $4432.66/70.96 \approx 62\times$, and PN^2 about $4432.66/99.03 \approx 45\times$.

$\alpha\beta$ effectively searches almost every second position in the state space and does not manage to prune aggressively. In Konane the branching factor first increases and then decreases over the course of the game. $\alpha\beta$ might not be able to find tactical traps or decompositions that might lead to a faster solution.

In contrast the Proof-Number Search algorithms are able to find a minimal proof tree relatively efficiently, with df-pn being the most efficient variant.

Although PN^2 explored a comparatively large 10.3% of the state space, its memory footprint (measured by L1 nodes created) was minimal, corresponding to only 0.2% of the total states (see Table 4.4). It still performs a large amount of repeated work.

Compared to the implementation in "Lines of Action" by Van Den Herik and Winands [28], the completion time of PN^2 is much longer than that of PNS, while the number of evaluated nodes is similar. Most surprising is the efficiency of df-pn, which finds the solution fastest while evaluating the fewest nodes.

5.3. Parallel Scaling Performance

While df-pn achieved the strongest single-threaded performance, it did not scale well on Konane. The parallel efficiency declines sharply from 0.77 (at two threads) to 0.29 (at 12 threads) as the number of threads increases, while the search overhead remains approximately constant.

The scaling performance of df-pn was worse than that reported by Kaneko [13] for Shogi. They achieved a speedup of 2.71 and 3.58 for 4 and 8 threads, respectively.

In stark contrast, PN^2 scaled super-linearly, despite having the worst single-threaded performance among the PNS algorithms. It achieved a speedup of 14.62 with 12 threads and a faster completion time of 6.83 seconds compared to the 15.15 seconds for df-pn.

Our method of parallelizing PN^2 with a shared Transposition Table also outperforms existing methods such as RP-PN^2 by Saito et al. [21]. On "Lines of Action" positions, RP-PN^2 achieves speedups of 1.6, 2.5, and 3.5 with 2, 4, and 8 threads, respectively, whereas our approach achieves speedups of 2.25, 4.68, and 9.88 on the same number of threads.

The results suggest that df-pn has high lock contention since the search overhead is nearly constant and the efficiency is decreasing sharply as the number of threads increases. It is less effective at distributing work between threads than PN^2.

5.4. Weakly Solved Boards

A direct comparison of game-theoretic outcomes with the works of Kirk [14] and Wu [30] is not possible, as neither thesis is publicly accessible.

While we cannot compare outcomes, we can compare our completion time with the public completion time reported by Wu [30] on the 7×7 board. Our parallelized PN^2 with 64 threads took 1.28 hours compared to the 25.76 hours reported by Wu [30], who used a Greedy Best-first search on an AND/OR search tree. This result shows the efficiency of our current implementation even though the used hardware is not directly comparable.

As shown in the winner matrix in Figure 4.3, the most noticeable feature is how rare wins for Black are. Almost all board sizes favor White .The symmetry of the board family is also reflected, as swapping the number of ranks and files does not change the result (e. g. 5×7 and 7×5).

Despite the solver's efficiency, we were not able to solve board sizes larger than 7×7 within the 21-day time limit. The data in Table 4.4 shows exponential growth in complexity from the 6×6 board which finished in 6.6 seconds to the 7×7 board which finished in 1.28 hours. This is a **693-fold** increase in time, which corresponds to searching over **2100 times** more L2 nodes than the 6×6 board.

While we were not able to solve the 8×7 board completely, we did determine the game-theoretic value of the central opening move. On this board, horizontal reflection is a valid symmetry as described in Section 3.4. The other two opening moves, in the top-left and top-right corners (see Figure 4.4), are mapped onto each other by this horizontal flip and are therefore strategically equivalent. As a result, PN^2 maps both to the same canonical position. Further analysis only needs to analyze one of these moves to determine the game-theoretic value for both.

5.5. Strongly Solved Boards

To the best of our knowledge, no strong solutions for Konane boards have been published so far. This work therefore provides the first complete perfect-play databases for all boards up to 6×6. Detailed results of each state space are shown in Appendix A.

For every board size up to 6×6, the database created by Retrograde Analysis agrees with the weak solutions reported in the winner matrix in Figure 4.3. In this sense, the strong solution validates the results found by PN^2.

The databases give a more detailed view of Konane than the weak solutions alone. Table 4.5 shows that the symmetry-reduced reachable state space grows very quickly with board size, from only 9 states on the 3×3 board to 3.7×10^9 states on the 6×6 board. As the search tree grows, the percentage of *terminal* states decreases monotonically.

The distribution of winning positions for Black and White also shows the advantage the winning player has. On 5×5, where the initial position is a win for Black, the number of positions where Black wins dominates the state space with about 76% of all states. For all other boards in the table White wins and the distribution is in favor of White except for 4×3. This indicates that the player who wins from the start does not rely on a single narrow winning line, but controls a larger region of the reachable state space on that board size.

6. Conclusion

The main goals of this thesis were (i) to design and evaluate a parallel Proof-Number Search implementation for weakly solving Konane boards up to 8×8, and (ii) to build endgame databases that strongly solve smaller boards up to 6×6 using Retrograde Analysis.

To achieve these goals, we implemented an efficient state representation and a fast move generator, compared different Proof-Number Search variants and their enhancements, and reduced the state space by symmetry detection.

The main contributions of this thesis are:

1. Weakly solving all Konane boards up to 7×7 from the starting position, and partially solving the 8×7 board.
2. Strongly solving all boards up to 6×6 by constructing endgame databases with Retrograde Analysis and Breadth-first search.
3. Adapting the parallelization strategy originally proposed for df-pn by Kaneko [13] to PN^2 and achieving super-linear scaling on the 6×6 board.
4. Developing an efficient ranking function for Konane.

Future work could follow several directions:

1. Test the PN^2 parallelization strategy described in subsection 3.7.1 on other games, especially on "Lines of Action", which has been used extensively for testing Proof-Number Search algorithms [28].
2. Investigate and compare different strategies for parallelizing Retrograde Analysis and Breadth-first search for Konane.
3. Explore the feasibility of using a proof-cost network, as proposed by Wu et al. [31]. Such a neural network could be integrated into Proof-Number Search to guide the search towards smaller proof trees.

Code Availability

The complete source code used in this thesis is available at `https://git.tu-berlin.de/bene/konane`.

Hilfsmittel

Für die Prints und Tests im Code wurde Copilot von Microsoft mit GPT-4.1 verwendet. Zur Überprüfung der Satzstruktur und Grammatik wurde Gemini 2.5 Pro von Google verwendet.

Bibliography

[1] J Allen. A note on the computer solution of Connect-Four. *The First Computer Olympiad*, pages 134–135, 1989. Publisher: Ellis Horwood.

[2] Louis Victor Allis. A knowledge-based approach of connect-four. *J. Int. Comput. Games Assoc.*, 11(4):165, 1988.

[3] Louis Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. Ponsen & Looijen, 1994. ISBN 978-90-90-07488-7. Google-Books-ID: c7FTAgAACAAJ.

[4] Dennis Michel Breuker, Joseph Willem Hubertus Marie Uiterwijk, and Hendrik Jacob Herik. *The PN2-search algorithm*. Universiteit Maastricht, Department of Computer Science, 1999.

[5] Mark G. Brockington and Jonathan Schaeffer. APHID game-tree search. *Advances in Computer Chess*, 8:69–91, 1997. Publisher: Elsevier.

[6] Cameron Browne. Bitboard methods for games. *ICGA journal*, 37(2):67–84, 2014. ISSN 1389-6911. Publisher: SAGE Publications Sage UK: London, England.

[7] Alice Chan and Alice Tsai. 1 x n Konane: A Summary of Results. In Richard Nowakowski, editor, *More Games of No Chance*, Mathematical Sciences Research Institute Publications, pages 331–340. Cambridge University Press, Cambridge, 2002. ISBN 978-0-521-80832-3. doi: 10.1017/9781316135167.023. URL https://www.cambridge.org/core/product/9CC54BD6F919F5CA9788C7EFA186529E.

[8] Stefan Edelkamp and Damian Sulewski. Parallel state space search on the GPU. In *International Symposium on Combinatorial Search (SoCS 2009)*, 2009.

[9] Michael D Ernst. Playing Konane mathematically: A combinatorial game-theoretic analysis. *UMAP Journal*, 16(2):95–121, 1995. Publisher: Citeseer.

[10] Robert A Hearn. Amazons, Konane, and cross purposes are PSPACE-complete. pages 287–306. Citeseer, 2005.

[11] Kaitlin Hendrick. Analysis of Artificial Intelligence Techniques for Konane. 2018.

[12] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. ISBN: 0164-0925 Publisher: ACM New York, NY, USA.

[13] Tomoyuki Kaneko. Parallel depth first proof number search. volume 24, pages 95–100, 2010. ISBN 2374-3468. Issue: 1.

[14] O. Kirk. *Solving Konane: A blended approach*. Bachelor's thesis, Maastricht University, 2020. Department of Data Science and Knowledge Engineering.

[15] Akihiro Kishimoto. Parallel AND/OR tree search based on proof and disproof numbers. ゲームプログラミングワークショップ*1999* 論文集, 1999(14):24–30, 1999. Publisher: 情報処理学会.

[16] Akihiro Kishimoto, Mark HM Winands, Martin Müller, and Jahn-Takeshi Saito. Game-tree search using proof numbers: The first twenty years. *ICGA journal*, 35(3):131–156, 2012. ISSN 1389-6911. Publisher: SAGE Publications Sage UK: London, England.

[17] Ayumu Nagai. Df-pn algorithm for searching AND/OR trees and its applications. *Ph. D. thesis, Department of Information Science, University of Tokyo*, 2002.

[18] Oren Patashnik. Qubic: 4× 4× 4 tic-tac-toe. *Mathematics Magazine*, 53(4):202–216, 1980. ISSN 0025-570X. Publisher: Taylor & Francis.

[19] Jakub Pawlewicz and Łukasz Lew. Improving depth-first pn-search: 1+ ε trick. pages 160–171. Springer, 2006.

[20] Stefan Reisch. Hex ist PSPACE-vollständig. *Acta Informatica*, 15(2):167–191, June 1981. ISSN 1432-0525. doi: 10.1007/BF00288964. URL https://doi.org/10.1007/BF00288964.

[21] Jahn-Takeshi Saito, Mark H. M. Winands, and H. Jaap van den Herik. Randomized Parallel Proof-Number Search. In H. Jaap van den Herik and Pieter Spronck, editors, *Advances in Computer Games*, pages 75–87, Berlin, Heidelberg, 2010. Springer. ISBN 978-3-642-12993-3. doi: 10.1007/978-3-642-12993-3_8.

[22] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science (New York, N.Y.)*, 317(5844): 1518–1522, September 2007. ISSN 1095-9203. doi: 10.1126/science.1144079.

[23] Mohamed Shokry. Comparing extended Minimax and Monte Carlo Tree Search for Hawaiian Checkers. 2023.

[24] Abu Bakar Siddique, Saadia Farid, and Muhammad Tahir. Proof of bijection for combinatorial number system. *arXiv preprint arXiv:1601.05794*, 2016.

[25] Hiroki Takizawa. Othello is solved. *arXiv preprint arXiv:2310.19387*, 2023.

[26] Darby Thompson. Teaching a Neural Network to Play Konane. *Bachelor Thesis, Department of Computer Science, Bryn Mawr College*, 2005.

[27] Jos Uiterwijk. Solving narrow Konane boards. *ICGA Journal*, 43(3):162–183, January 2021. ISSN 1389-6911. doi: 10.3233/ICG-220198. URL https://journals.sagepub.com/action/showAbstract. Publisher: SAGE Publications.

[28] H Jaap Van Den Herik and Mark HM Winands. Proof-number search and its variants. In *Oppositional Concepts in Computational Intelligence*, pages 91–118. Springer, 2008.

[29] H Jaap Van Den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1-2):277–311, 2002. ISSN 0004-3702. Publisher: Elsevier.

[30] Sheng-Yu Wu. 夏威夷跳棋較小盤面的分析與破解. Master's thesis, National Taiwan Normal University, Taipei, Taiwan, 2024. Publication Title: Analysis and Solving of the Small-Scale Board in Hawaiian Checkers (Konane).

[31] Ti-Rong Wu, Chung-Chin Shih, Ting Han Wei, Meng-Yu Tsai, Wei-Yuan Hsu, and I.-Chen Wu. AlphaZero-based proof cost network to aid game solving. In *International Conference on Learning Representations*, 2022.

A. Strongly Solved Boards

Table A.1.: 3×3 state space

Ply	Unique Nodes	Black Wins	White Wins	Terminals
0	1	0	1	0
1	2	0	2	0
2	2	0	2	1
3	1	0	1	0
4	2	1	1	1
5	1	1	0	1
Total	9	2	7	3

Table A.2.: 4×3 state space

Ply	Unique Nodes	Black Wins	White Wins	Terminals
0	1	0	1	0
1	2	0	2	0
2	5	1	4	1
3	4	1	3	1
4	8	3	5	3
5	10	4	6	1
6	16	10	6	5
7	16	9	7	0
8	20	13	7	7
9	8	7	1	4
10	3	2	1	1
11	2	2	0	2
Total	95	52	43	25

Table A.3.: 4×4 state space

Ply	Unique Nodes	Black Wins	White Wins	Terminals
0	1	0	1	0
1	2	0	2	0
2	3	1	2	0
3	4	1	3	0
4	11	7	4	0
5	24	9	15	0
6	61	34	27	2
7	102	31	71	3
8	183	98	85	8
9	189	75	114	8
10	190	99	91	27
11	117	56	61	17
12	63	30	33	23
13	20	13	7	10
14	7	2	5	5
15	1	1	0	1
Total	978	457	521	104

Table A.4.: 5×4 state space

Ply	Unique Nodes	Black Wins	White Wins	Terminals
0	1	0	1	0
1	2	0	2	0
2	5	2	3	0
3	7	2	5	0
4	21	13	8	0
5	62	20	42	0
6	210	121	89	1
7	528	161	367	7
8	1347	733	614	9
9	2496	820	1676	18
10	4491	2405	2086	100
11	5487	2030	3457	166
12	6010	3171	2839	444
13	4083	1796	2287	436
14	2343	1164	1179	500
15	819	428	391	268
16	245	91	154	123
17	42	30	12	29
18	3	0	3	3
Total	28202	12987	15215	2104

Table A.5.: 5×5 state space

Ply	Unique Nodes	Black Wins	White Wins	Terminals
0	1	1	0	0
1	2	2	0	0
2	2	2	0	0
3	3	3	0	0
4	9	9	0	0
5	34	30	4	0
6	138	133	5	0
7	440	347	93	1
8	1492	1404	88	0
9	4166	3192	974	2
10	10784	9850	934	29
11	23033	17152	5881	139
12	38788	34109	4679	199
13	53846	38113	15733	1100
14	55154	45766	9388	844
15	40770	26501	14269	2351
16	26276	19770	6506	1332
17	9848	5734	4114	1299
18	4559	2921	1638	730
19	999	576	423	286
20	305	152	153	122
21	39	27	12	23
22	4	3	1	1
Total	270692	205797	64895	8458

Table A.6.: 6×5 state space

Ply	Unique Nodes	Black Wins	White Wins	Terminals
0	1	0	1	0
1	2	0	2	0
2	5	1	4	0
3	9	1	8	0
4	37	10	27	0
5	144	13	131	0
6	702	273	429	0
7	2650	450	2200	2
8	11777	5841	5936	0
9	42510	10285	32225	4
10	164170	89032	75138	42
11	509542	145844	363698	227
12	1538242	858424	679818	1034
13	3569245	1117822	2451423	3608
14	7489542	4190103	3299439	16787
15	11131040	3753900	7377140	35995
16	13789973	7692287	6097686	112603
17	10585969	3939980	6645989	141909
18	5925002	3358874	2566128	191431
19	1836856	775420	1061436	103520
20	368849	218767	150082	36379
21	42168	19766	22402	6776
22	3644	2259	1385	662
23	204	94	110	54
24	8	5	3	0
Total	57012291	26179451	30832840	651033

Table A.7.: 6×6 state space

Ply	Unique Nodes	Black Wins	White Wins	Terminals
0	1	0	1	0
1	2	0	2	0
2	3	1	2	0
3	7	1	6	0
4	37	7	30	0
5	154	9	145	0
6	855	261	594	0
7	3624	509	3115	0
8	19061	8313	10748	0
9	81482	18580	62902	1
10	392354	200259	192095	19
11	1559106	431331	1127775	41
12	6416366	3497985	2918381	497
13	21456073	6575000	14881073	1457
14	69467818	39135526	30332292	12537
15	175660981	58172965	117488016	34199
16	400085391	229255225	170830166	210261
17	666778949	235546443	431232506	475121
18	905007577	523440574	381567003	1561813
19	801039361	298985594	502053767	2415979
20	490734498	286872861	203861637	3438927
21	164326111	63391481	100934630	2266041
22	32645193	19503323	13141870	998869
23	3156984	1224397	1932587	193648
24	211143	125701	85442	24967
25	6026	2195	3831	909
26	162	121	41	5
Total	3739049319	1766388662	1972660657	11635291

B. Additional Tables

Table B.1.: 8×8 Perft results

Depth	Nodes	Elapsed (ms)
1	4	0.02
2	12	1.10
3	28	1.15
4	172	1.72
5	892	1.37
6	7124	1.43
7	52044	2.56
8	508088	12.51
9	4633660	107.35
10	51883544	996.47
11	552955020	10781.59
12	6862224804	126508.37

Board	Ply 1		Ply 2		Ply 3		Ply 4		Ply 5		Ply 6		Ply 7		Ply 8		Ply 9		Ply 10	
	With	*Without*	*With*	*Without*	*With*	*Without*	*With*	*Without*	*With*	*Without*	*With*	*Without*	*With*	*Without*	*With*	*Without*	*With*	*Without*	*With*	*Without*
1×1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2×2	1	2	1	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3×3	2	5	2	12	1	8	2	8	1	8	0	0	0	0	0	0	0	0	0	0
4×4	2	4	3	12	4	16	11	42	24	92	61	244	102	408	183	720	189	742	190	760
5×5	2	5	2	12	3	20	9	72	34	268	138	1096	440	3512	1492	11912	4166	33244	10784	86132
6×6	2	4	3	12	7	28	37	146	154	612	855	3420	3624	14496	19061	76206	81482	325856	392354	1569364
7×7	2	5	2	12	3	20	14	104	57	448	350	2788	1719	13720	10935	87392	56937	455252	346570	2771904
8×8	2	4	3	12	7	28	43	170	215	856	1537	6148	9268	37072	68512	274000	438166	1752520	3190054	12760120

Table B.2.: Unique states at plies 1–10 with and without symmetry reduction for square boards 1×1 through 8×8. Each ply appears as a pair of columns (*With*, *Without*).