Parallel Solving of Two-Player Tierable Abstract Strategy Games

Robert Shi
Dan Garcia, Ed.
Justin Yokota, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-87
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-87.html

May 10, 2024

Acknowledgement

Thank you to Professor Dan Garcia for introducing me to GamesCrafters and shaping my academic journey. Thank you to Justin Yokota for guiding my first project and being my second reader of this report. Gratitude to Cameron Cheung for his support and development contributions. Acknowledgment to Max Fiero, Sameer Nayyar, Haonan Huang, Yifan Zhou, Jiachun Li, Benji Xu, Maria Rufova, Angela He, and Ryan Liu for their collaborative efforts in various projects.

Parallel Solving of Two-Player Tierable Abstract Strategy Games

by Robert Shi

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Teaching Professor Dan Garcia
Research Advisor

Date

* * * * * * *

Lecturer Justin Yokota
Second Reader

Date

Abstract

Many abstract strategy games have a large number of positions, with a significant fraction of games loopy in nature. Previous efforts to develop solvers in *GamesmanClassic* for these loopy tier games have utilized single-threaded approaches, which does not take advantage of the parallelism necessary to solve large games. This report provides a comprehensive overview of the design and development of a parallelized loopy solver for generic tierable games. It elaborates on the loopy solving algorithm, explores sources of parallelism, addresses synchronization issues, and details optimizations that enhance the solving process. Furthermore, it showcases the application of this algorithm in solving *Quixo* and endgames of *Chinese Chess*, demonstrating the effectiveness of our approach.

Acknowledgements

Thanks to Professor Dan Garcia, whose mentorship and passion for computational game theory not only brought me into the fascinating group of *GamesCrafters*, but also profoundly shaped my academic journey. I am deeply grateful for his guidance and encouragement, and for the opportunity to be part of this group.

Thanks to Justin Yokota, who guided me through my first project in GamesCrafters and kindly agreed to be my second reader of this report.

My appreciation also goes to Cameron Cheung for his extensive support over three semesters, and for his contributions to the development of *All Queens Chess*, move value visualization, and the integration of *GamesCraftersUWAPI* with the *GamesmanOne* project.

Acknowledgement is due to Max Fiero and Sameer Nayyar, with whom I collaborated on the development of a lookup table optimization for the bit-perfect database.

I am also thankful to Haonan Huang and Yifan Zhou, who helped implement the server for Chinese Chess Endgames in GamesCraftersUWAPI; and to Jiachun Li, who created the images for the Chinese Chess AutoGUI on GamesmanUni.

Gratitude is extended to Benji Xu, Maria Rufova, and Angela He for their joint development of Quixo; and to Ryan Liu, for his contribution to the development of Generalized Tic-tac-toe.

Lastly, my deepest thanks to my family for their unwavering support and for shaping me into the person I am today.

This research used the Savio computational cluster resource provided by the Berkeley Research Computing program at the University of California, Berkeley (supported by the UC Berkeley Chancellor, Vice Chancellor for Research, and Chief Information Officer).

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Contents

Chapter 1: Introduction

GamesCrafters is a computational game theory research and development group founded in 2001 by Teaching Professor Dan Garcia that has involved close to one thousand students since its founding. The group focuses on strongly solving two-player perfect information abstract strategy games and presenting the results through various game interfaces. For over 20 years, the group has completed various projects for solving, analyzing, and playing games under the GAMESMAN open-source architecture originally developed by Professor Dan Garcia in 1990 [1]. The original code base, which later became part of GamesmanClassic [2], was written in C and has gone through a series of improvements since 2001 by the members of GamesCrafters.

With the advances in computing technology, solving games with larger state spaces has become possible. Over the past few years, GamesCrafters have solved some of the biggest games in the group's history. Connect 4 was solved by Justin Yokota in 2022 [3]. Nine Men's Morris and Quarto were solved by Cameron Cheung in 2022 and 2023 respectively [4]. It is worth noting that two of the largest games, Quarto and Connect 4, were individually solved using custom parallel solvers and database systems. Connect 4, which has 4.53 trillion reachable positions, was solved in under 6 hours using 480 CPU cores (or 960 hyperthreads) on the Savio cluster. Quarto, which has over 11 quadrillion positions, was solved in 3.51 days using a single 12-core CPU. Nine Men's Morris, on the other hand, was solved on a single-threaded tier solver for loopy games in GamesmanClassic over the period of 9.4 days. To solve larger loopy games, parallelized loopy tier solvers needed to be created.

The research presented in this report is motivated by the growing need for parallel solvers for generic two-player games. The report consists of two parts.

The first part focuses on the theoretical background of computational game theory and parallel computing. In Chapter 2, we define terms that are used throughout this report, and provide the reader with an overview of the GamesCrafters' software infrastructure for solving and presenting games. In Chapter 3, we present the core algorithm that is used to solve a single tier of a loopy game in parallel. We also provide proof of the correctness of the algorithm and a scaling efficiency analysis in the same chapter. In Chapter 4, we introduce the tier solving algorithm that applies the single tier solving algorithm to all tiers in the game in an order that respects tier dependencies, while allowing multiple tiers to be solved in parallel.

The second part of this report discusses the application of the algorithms in various game-solving projects. In Chapter 5, we introduce the GamesmanOne parallel solving system for generic two-player games which utilizes the algorithms discussed in Chapters 3 and 4. In Chapters 6 and 7, we present the application of the algorithm on solving 6-piece Chinese chess endgames and the full original game of 5x5 Quixo.

Chapter 2: Background

In this chapter, we give the background of computational game theory and an overview of the software infrastructure used by the GamesCrafters Research and Development Group.

2.1 Definitions

Cameron Cheung's 2023 Technical Report "Techniques for Solving and Visualizing Large Games" [4] offers comprehensive definitions of many terms that are similarly employed in this document. Building upon his foundational work, we will only furnish definitions for terms that are newly introduced in this report. Specifically, we will reuse the following definitions provided in his report:

- Two-player perfect information abstract strategy games with alternating turns;
- Game as a two-tuple (G, V), where G is the game graph and V is the primitive value function;
- Game graph $G = (P, M)$ as a weakly connected directed graph with position/vertex set P and move/edge set M;
- Primitive positions as sink vertices in G;
- Loopy games and loop-free games;
- Child positions and parent positions;
- Value of a position, which can be one of win, $lose$, tie, or $draw$;
- Value of a move, which can be one of winning, losing, tying, or drawing move;
- Perfect play, or optimal strategy;
- Remoteness of a position;
- Position symmetry and canonical positions;
- Tiers, tier graph, trivial and non-trivial tier definitions, and tier definition validity;
- Trivial tier definitions;
- Child tiers and parent tiers;
- Tierability of a game;
- Tier symmetry and canonical tiers.
- Color convention for position value labeling, where green represents win, yellow represents tie or $draw$, and red represents $lose$.

Here, we underscore the importance of a valid tier definition for the existence of a tier graph, which must inherently be a *directed acyclic graph (DAG)*. Employing the definition of a game graph as $G = (P, M)$ and its corresponding tier graph G', we use the example game graph G as shown in Figure 2.1 and its tier graph G' in Figure 2.2 to illustrate the definitions we give in the following paragraphs.

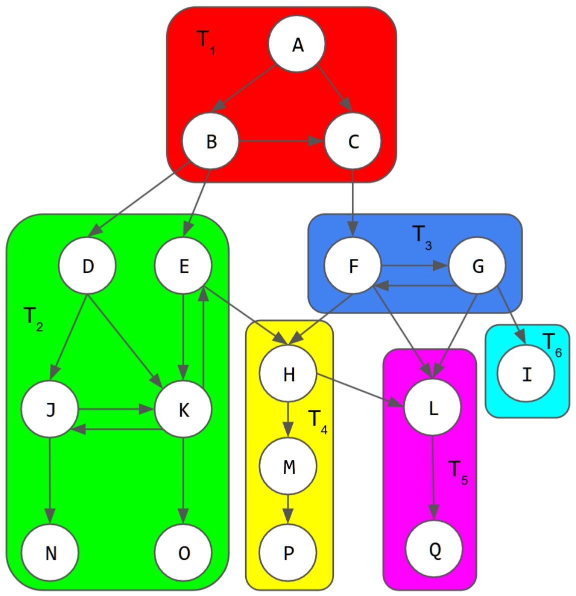

Figure 2.1: Example game graph G

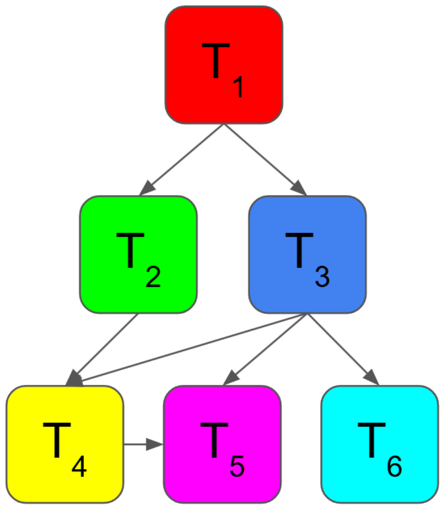

Figure 2.2: Tier graph G'

We define a **reverse game graph** as the transpose graph G^R of G. We also define a **reverse position graph of a tier** T to be a subgraph of G^R that contains only vertices in T and child tiers of T, and edges with one of the vertices in T as head. Figure 2.3 shows the reverse game graph of the example game graph G, and Figure 2.4 shows the reverse position graph of tier T_1.

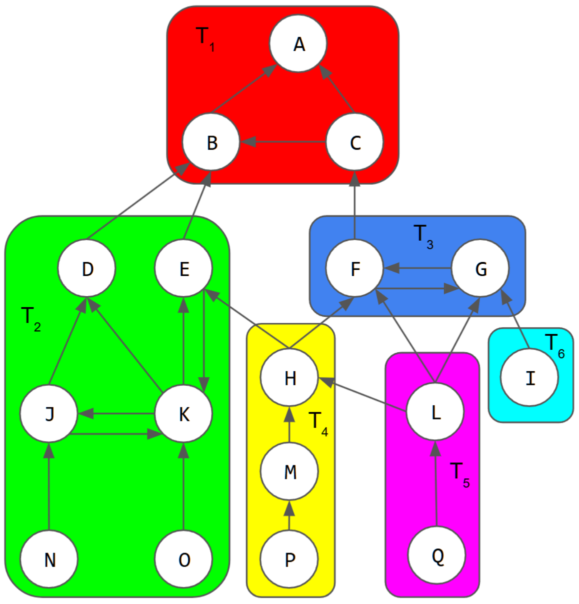

Figure 2.3: Reverse game graph G^R

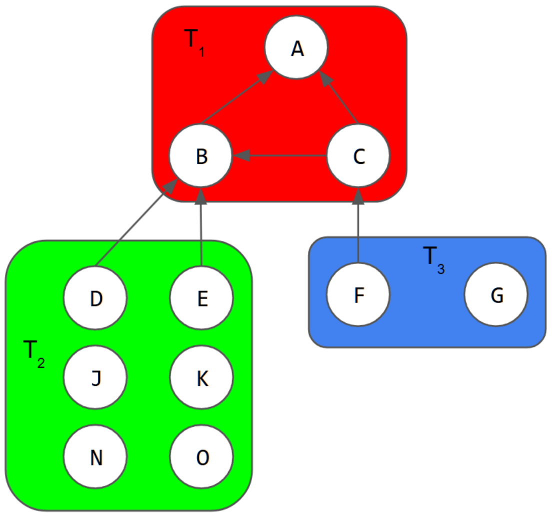

Figure 2.4: Reverse position graph of T_1

We also define a **reverse tier graph** as the transpose graph G'^R of G'. An example is shown in Figure 2.5.

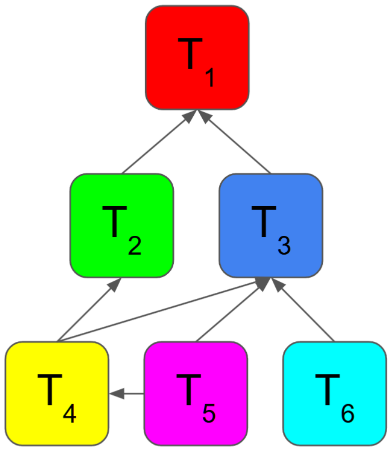

Figure 2.5: Reverse tier graph G'^R

We also define a **hash function** to be a bijective function $x \mapsto h(x)$ with a set of abstract elements X as its domain and a set of integers $I \subset \mathbb{Z}$ as its codomain. $h(x)$ is called the **hash value** or simply **hash** of element x. Therefore, a **position hash function** is a hash function with a set of positions P as its domain, and each value $h(p) \in I$ is called a hash value of position $p \in P$. We call h the **position hash function for game** G if P is the set of all positions in G. We call h the **position hash function for a tier** T if $P = T$. In this case, we define **the size of tier** T as the largest element in I plus 1. That is, if the largest element in I is m, then the size of tier T is $m + 1$. Unless otherwise specified, all hash functions in this report are *position hash functions for tiers*. Also, we define a **tier hash function** as a hash function with a set of tiers S as its domain and each value $h(T) \in I$ is called a **hash value of tier** $T \in S$.

For a game (G, V), a **strong solve** of the game is the discovery and storage of the value and remoteness of each position $p \in P$. The software used to carry out the solving process is called a **solver**. The ensemble of values and remotenesses of all positions is called the **results** (or **outcome**) of the solver. The solver results are typically stored in memory while the game is being solved and flushed to disk once the solving process is complete. The data structure that is used to store the positions in memory is called the **in-memory database**, and the files that are created as copies of the solver results are called the **database** for the game. One commonly used data structure for the in-memory database of tier T is an array of length equal to the size of T. The array is then compressed into database files and flushed to disk. We call this database an **array database**, and the in-memory database the **solver outcome array**. All database implementations discussed in this report are array databases unless otherwise specified.

If the game contains symmetric positions, the solver can reduce the amount of computation needed to solve the game by exploring only the canonical positions. We define the process of excluding non-canonical positions when solving the game as **position symmetry removal**. For an *array database implementation*, there are two models to achieve position symmetry removal:

Write-Once-Read-Many (**WORM**), and **Read-Once-Write-Many** (**ROWM**). In the WORM model, values and remotenesses are assigned only to the canonical positions, and spaces reserved for non-canonical positions remain all zeros, assuming they are zero initialized at the beginning of the solving process. The WORM model usually benefits database compression because it replaces seemingly random data with consecutive chunks of zero in all spaces reserved for all non-canonical positions. However, it comes at the cost of a slightly increased database access time because all positions must be converted to the canonical version before it is queried from the database. Figure 2.6 shows an example of first writing “win in 3” to the 0-th index of an array database and then reading indices 0 and 2 from the same database under the WORM model, assuming the positions that correspond to indices 0, 2, 4, and 6 are symmetric and 0 is the canonical position. Note that the entries at the non-canonical indices 2, 4, and 6 remain unchanged when W3 is written to index 0, and how the read at index 2 is redirected to index 0 using the `Canonical` function.

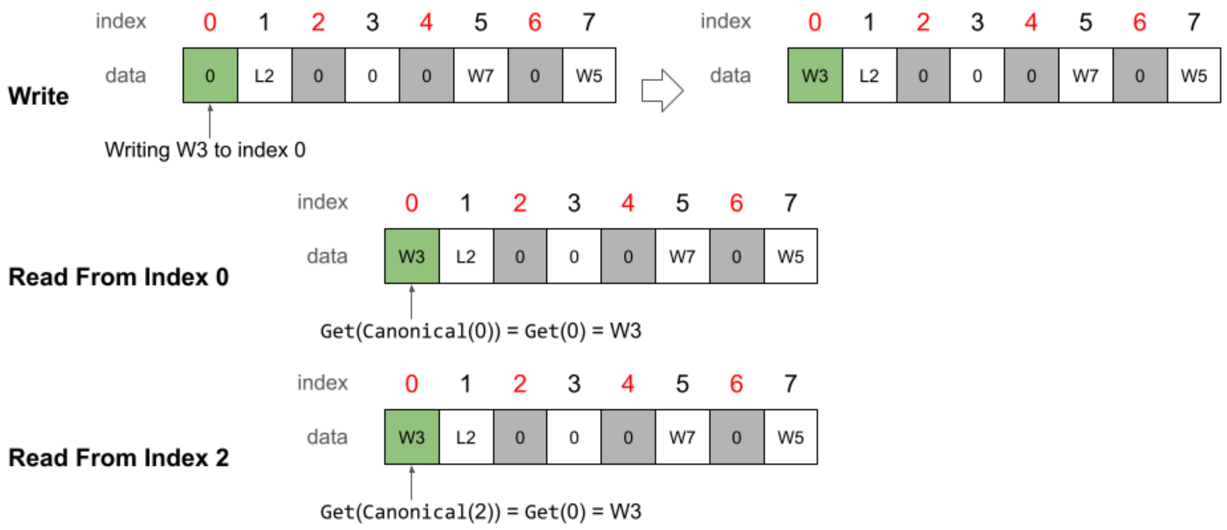

Figure 2.6: Writing to and reading from a database under the WORM model

On the other hand, an implementation that follows the ROWM model assigns values and remotenesses to all positions in the game. When a canonical position is solved, a ROWM database first generates all positions that are symmetric to the solved position, and then assign the same value and remoteness to all of them. Figure 2.7 shows the same example operations under the ROWM model. Note how the value written to position 0 is propagated to symmetric positions 2, 4, and 6.

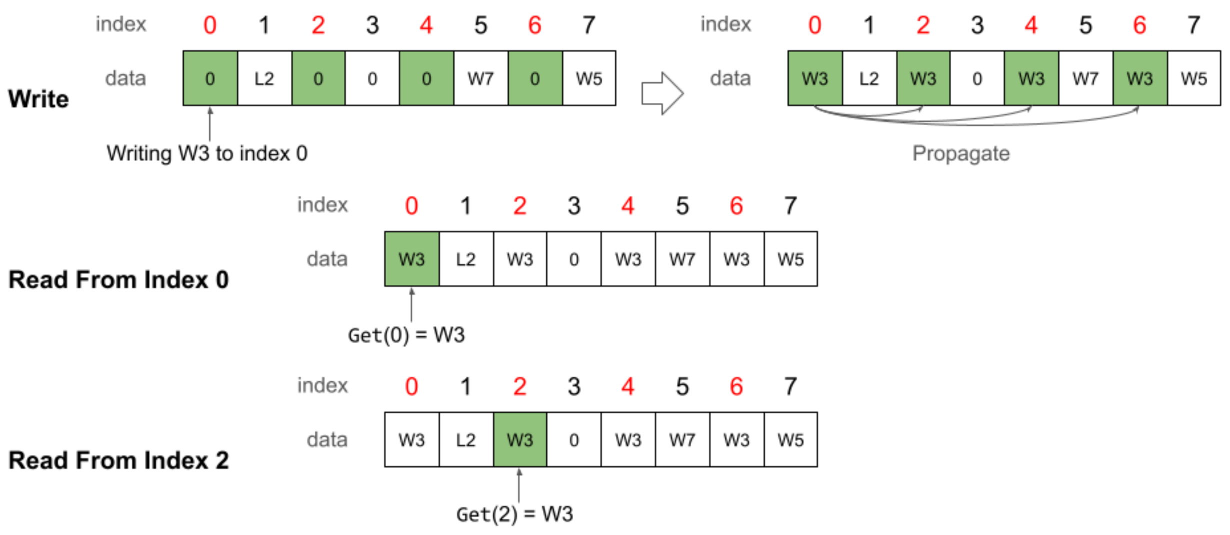

Figure 2.7: Writing to and reading from a database under the ROWM model

For most games, the WORM model gives much smaller database sizes with an acceptable overhead. Therefore, all databases mentioned in this report use the WORM model for symmetry removal.

We similarly define the process of excluding non-canonical tiers when solving a game as **tier symmetry removal**. When excluding the same number of positions from the array database, tier symmetry removal usually results in a greater reduction of the aggregate database size than position symmetry removal. This is because with tier symmetry removal, non-canonical tiers are completely ignored, whereas space is still reserved in the array for non-canonical positions in position symmetry removal.

2.2 Combinatorially Optimal Hash

The database that we use throughout this report is the array database. For this type of database, the solver outcome array has an $O(m)$ memory footprint, where m is the largest defined hash value of any position in a tier. Ideally, we would have $m = |I|$. In this case, the hash function h is called a **perfect hash function**. However, it is usually difficult, if at all possible, to come up with a perfect hash function. This is often due to the difficulty of filtering out unreachable positions from the hash definition without actually solving the game. For some board games with a constant-sized board and fixed sets of pieces, there exists an approach that balances the size of the solver outcome array and the difficulty of defining the hash function. In this approach, we consider the number of each type of piece on the board and map each possible rearrangement of pieces on the board to a unique integer from 0 to the number of rearrangements minus 1. A hash function generated with this approach is called a **combinatorially optimal hash function**. In this section, we discuss a way to automatically generate a combinatorially optimal hash function given information about the game.

Suppose we have the following knowledge of the board game:

- The size of the board is $\mathrm{board\_size}$.
- The game has k types of pieces, with blank slots included as a special type of "piece".
- For each type of piece p_i where $i \in [0, k)$, the minimum number of p_i pieces on the board is a_i, and the maximum number is b_i.

Then, we can follow the following steps to come up with a combinatorially optimal hash function:

1. Define a **valid piece configuration** as an assignment to the number of pieces of each type that respects the minimum and maximum numbers of each type of piece and the size of the board. Enumerate all valid piece configurations and sort them in any fixed order.
2. For each valid piece configuration:
    a. Calculate the number of piece rearrangements N on the board.
    b. Set the offset of the piece configuration as N plus the number of all piece rearrangements from configurations that are in front of the current configuration in the sorted list.
3. The hash function is then defined by mapping each position to an integer value equal to the offset of the piece configuration that the position observes plus the index of the piece rearrangement within that piece configuration. The indices of piece rearrangements in a configuration is defined by first sorting all rearrangements in some fixed order and then assigning each one to an integer between 0 and the number of rearrangements in that configuration minus 1.

This approach was first proposed by Professor Dan Garcia and implemented by Attila Gyulassy in GamesmanClassic as a specialized version that supports only 3 types of pieces. The algorithm was later generalized to an arbitrary number of types of pieces and implemented as the Generic Hash module of GamesmanClassic by Michel D'Sa. A more generalized version that supports multiple different maximum/minimum piece number definitions for tier games was later implemented by Scott Lindeneau. The module was recreated and further optimized in GamesmanOne, which is discussed in Chapter 5 of this report.

2.3 GamesCrafters Projects and Infrastructure

Figure 2.8 shows the interconnections between the various servers used by GamesCrafters.

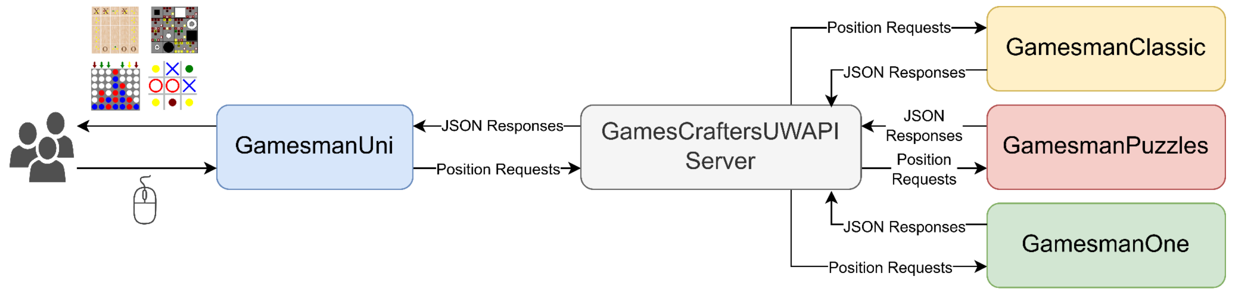

Figure 2.8: GamesCrafters servers

GamesmanUni is a single-page web application that allows users to explore games and puzzles solved in GamesCrafters. The application uses data fetched from the backend to render positions and provide values and remotenesses. In between GamesmanUni and the backend is a middleware called the **GamesCraftersUWAPI** server, where UWAPI stands for Universal Web API. It is an HTTP server written in Python that provides standard API calls used by GamesmanUni. The server contains hard-coded instructions on how to make API calls to backend servers and information on GUI layout for all games and puzzles. UWAPI is a separate server independent of GamesmanUni and publicly accessible. This means that people can make their own application by making API calls to the UWAPI server.

GamesmanOne is a new system created in this study. It is a parallel C solver and game generator system inspired by GamesmanClassic. The system is currently implemented with a special focus on solving tierable loopy games. However, the system is designed with extensibility in mind and has the ability to incorporate other types of solvers in the future. The system is also studied as an application of the parallel algorithms discussed in this report. A detailed description of the GamesmanOne system is provided in Chapter 5 of this report.

Chapter 3: The Single-Tier Loopy Solving Algorithm

In this chapter, we present the algorithm that is used to solve one tier from a game. The algorithm makes no assumption on the game graph and therefore can be applied to both loopy and loop-free games. For single-tier games, applying this algorithm on the only tier solves the entire game. For games with a valid non-trivial tier definition, we can apply the algorithm to all tiers in any topological order of the reverse tier graph. The algorithm for tier solving is covered in Chapter 4.

The original loopy solving algorithm was designed to solve single-tier games only and was proposed and implemented by Professor Dan Garcia as part of GamesmanClassic [2]. The algorithm was later improved and adapted to various different solvers by members of GamesCrafters. The algorithm presented in this chapter is an optimized version of the algorithm implemented by Max Delgadillo for the GamesmanClassic retrograde tier solver in 2006 [5].

The parallelized version of the algorithm described in this chapter is implemented as part of the Tier Solver module of GamesmanOne [6].

3.1 Algorithm

The algorithm presented in this section solves a loopy tier T under the following assumptions:

1. All child tiers of T have been solved.
2. The maximum remoteness of the game is no more than $\mathrm{REMOTENESS\_MAX}$, a value that is defined at compile time. Currently, this value is set to 1023. This assumption is safe to make if an error is reported when a position with remoteness greater than $\mathrm{REMOTENESS\_MAX}$ is discovered. When it happens, the solver should be reconfigured with a larger $\mathrm{REMOTENESS\_MAX}$ value. The value 1023 is empirically determined and is a valid assumption for all games that have been solved in the GamesCrafters group.[1]

We define a position p as an **illegal position** if p cannot be reached from the initial position of the game via a sequence of allowable moves. Recall from Section 2.1 that the position hash function is usually not perfect. This implies that many hash values reserved in the in-memory database are mapped to illegal positions. The solver is allowed to assign arbitrary values and remotenesses to illegal positions, because these positions are never encountered in normal play and therefore never queried from the database.

The following input is provided to the algorithm:

- n: the size of tier T.
- `IsLegalPosition`(T, p): a function that performs a *weak* test on the legality of a position p in tier T. It is a weak test because it may return false positives. If p is a legal position in

[1] The highest remoteness discovered by GamesCrafters so far is 264, which was found in the endgames of Chinese chess.

tier T, the function always returns `true`. However, if p is an illegal position in T, the function may return `true` or `false`. This function is only for optimization purposes and is not meant to provide accurate legality of positions.
- `Value`(C_i, p): a function that takes in a position p in child tier C_i, and returns the solved value of p. If p is an illegal position in C_i, the function returns an arbitrary value.
- `Remoteness`(C_i, p): a function that takes in a position p in child tier C_i, and returns the solved remoteness of p. If p is an illegal position in C_i, the function returns an arbitrary remoteness.
- `Primitive`(T, p): a function that takes in a tier T and a position p inside T, and returns the value of position p if p is a primitive position, or $undecided$ if p is not a primitive position.
- `GetChildPositions`(T, p): a function that takes in a tier T and a position p inside T, and returns an array of $(child\_tier, child\_position)$ tuples.
- `GetParentPositions`$(child\_tier, child\_position, parent\_tier)$: a function that returns an array of parent positions of $(child\_tier, child\_position)$ in $parent\_tier$.

We make the following definitions:

1. Let $\{C_0, C_1, \dots C_{m-1}\}$ be the set of child tiers of T, where $m \geq 0$.
2. Let the size of C_i be n_i, where $0 \leq i \leq m$.

We also initialize the following data structures:

- Let V be a value array of size n storing the values of all positions in T. V is part of the in-memory database.
- Let R be an unsigned integer array of size n storing the remotenesses of all positions in T. R is also part of the in-memory database.
- Let $\mathrm{num\_undecided\_children}$ be an array of unsigned integers of length n storing the number of $undecided$ children of each position in T.
- Let F_{win}, F_{lose}, and F_{tie} be three 2D arrays called the winning, losing, and tying frontiers respectively. The first dimension of these 2D arrays is set to $\mathrm{REMOTENESS\_MAX}$ and the second dimension is dynamically adjusted during the execution. The frontiers are used to temporarily hold positions that are solved but not yet used to update the status of their parents.

The output of the algorithm is stored in the arrays V and R. The algorithm for solving a single loopy tier can be summarized as follows:

Algorithm 1: Algorithm for Solving a Single Loopy Tier

```
1: Load all winning, losing, and tying positions from all child tiers into the corresponding
   frontiers.
```

```
2: Scan tier $T$ for illegal positions as defined by the IsLegalPosition function, and set
   their number of undecided children to $0$.
3: Scan tier $T$ for the number of children of each legal position, and set the corresponding
   values in the $\mathrm{num\_undecided\_children}$ array.
4: Scan tier $T$ for all primitive winning, losing, and tying positions and load them into the
   corresponding frontiers at remoteness $0$.
5: Process $F_{lose}$[remoteness] and $F_{win}$[remoteness] for each possible remoteness from 0 to
   $\mathrm{REMOTENESS\_MAX}$.
6: Process $F_{tie}$[remoteness] for each possible remoteness from 0 to $\mathrm{REMOTENESS\_MAX}$.
7: Mark all positions that still have a positive number of undecided children as drawing.
8: Flush $V$ and $R$ to disk.
```

The algorithm above can be subdivided into 5 steps:

1. Load positions from child tiers (line 1).
2. Scan tier (lines 2-4).
3. Push the frontiers up using backward induction (lines 5-6).
4. Mark all drawing positions (line 7).
5. Save solver results (line 8).

Each of the first 4 steps is described below as an algorithm.

Algorithm 1.1: Algorithm for Loading Positions from Child Tiers

```
1:  for $i = 0$ to $m - 1$ do // for each child tier
2:    for $p = 0$ to $n_i - 1$ do // for each position in child tier $i$
3:      if Value($C_i$, $p$) == $win$ then
4:        Push ($C_i$, $p$) into $F_{win}$[Remoteness($C_i$, $p$)].
5:      else if Value($C_i$, $p$) == $lose$ then
6:        Push ($C_i$, $p$) into $F_{lose}$[Remoteness($C_i$, $p$)].
7:      else if Value($C_i$, $p$) == $tie$ then
8:        Push ($C_i$, $p$) into $F_{tie}$[Remoteness($C_i$, $p$)].
9:      end if
10:   end for
11: end for
```

Algorithm 1.2: Algorithm for Scanning a Tier

```
1:  for $p = 0$ to $n - 1$ do // for each position $p$ in the tier we are solving
2:    $V[p] \leftarrow undecided$. // initialize the value of $p$ to $undecided$.
3:    if NOT IsLegalPosition($T, p$) then
4:      $\text{num\_undecided\_children}[p] \leftarrow 0$. // ensure illegal positions are skipped.
5:      continue
6:    end if
7:    Let $\text{children} \leftarrow$ GetChildPositions($T$, $p$)
8:    $\text{num\_undecided\_children}[p] \leftarrow \text{children.size}$.
9:    if Primitive($T$, $p$) == $win$ then
10:     $V[p] \leftarrow win$.
11:     $R[p] \leftarrow 0$.
12:     Push ($T$, $p$) into $F_{win}[0]$.
13:   else if Primitive($T$, $p$) == $lose$ then
14:     $V[p] \leftarrow lose$.
15:     $R[p] \leftarrow 0$.
16:     Push ($T$, $p$) into $F_{lose}[0]$.
17:   else if Primitive($T$, $p$) == $tie$ then
18:     $V[p] \leftarrow tie$.
19:     $R[p] \leftarrow 0$.
20:     Push ($T$, $p$) into $F_{tie}[0]$.
21:   end if
22: end for
```

Algorithm 1.3: Algorithm for Pushing the Frontiers Up

```
1:  for $remoteness = 0$ to REMOTENESS_MAX do // for each possible remoteness
2:    for each position ($T'$, $p'$) in $F_{lose}[remoteness]$ do // for each position of $remoteness$ in
    the losing frontier
3:      for each parent position $p$ in GetParentPositions($T'$, $p'$, $T'$) do
4:        if $\text{num\_undecided\_children}[p] > 0$ then  // if the position has not been closed
5:          $\text{num\_undecided\_children}[p] \leftarrow 0$
6:          $V[p] \leftarrow win$.
7:          $R[p] \leftarrow remoteness + 1$.
8:          Push ($T$, $p$) into $F_{win}[remoteness + 1]$.
9:        end if  // otherwise, skip the position because it has been solved or is illegal.
```

```
10:     end for
11:   end for
12:   for each position ($T'$, $p'$) in $F_{win}[remoteness]$ do // for each position of $remoteness$ in
        the winning frontier
13:     for each parent position $p$ in GetParentPositions($T'$, $p'$, $T$) do
14:       if num_undecided_children[$p$] $> 0$ then // if the position has not been closed
15:         num_undecided_children[$p$] $\leftarrow$ num_undecided_children[$p$] $- 1$
16:         if num_undecided_children[$p$] $== 0$ then
17:           $V[p] \leftarrow lose$.
18:           $R[p] \leftarrow remoteness + 1$.
19:           Push ($T$, $p$) into $F_{lose}[remoteness]$.
20:         end if
21:       end if
22:     end for
23:   end for
24: end for
25: for $remoteness = 0$ to REMOTENESS_MAX do // process tying positions in a similar way.
26:   for each position ($T'$, $p'$) in $F_{tie}[remoteness]$ do
27:     for each parent position $p$ in GetParentPositions($T'$, $p'$, $T$) do
28:       if num_undecided_children[$p$] $> 0$ then
29:         num_undecided_children[$p$] $\leftarrow 0$
30:         $V[p] \leftarrow tie$.
31:         $R[p] \leftarrow remoteness + 1$.
32:         Push ($T$, $p$) into $F_{tie}[remoteness + 1]$.
33:       end if
34:     end for
35:   end for
36: end for
```

Algorithm 1.4: Algorithm for Marking Drawing Positions

```
1: for $p = 0$ to $n - 1$ do  // for each position $p$ in the tier we are solving
2:   if num_undecided_children[$p$] $> 0$ then
3:     $V[p] \leftarrow draw$.
4:   end if
5: end for
```

Note that in Algorithm 1.3, there exists an alternative approach that does not rely on the assumption of the validity of $\mathrm{REMOTENESS\_MAX}$, which is to keep iterating until we have

a. reached remoteness at least the maximum remoteness observed from any solved child tier positions, and
b. finished an iteration in which no new positions were pushed into the frontier.

This approach may potentially speed up the algorithm by eliminating the need to loop through the unused remotenesses and the proof of its correctness is similar to the one we give in Section 3.2 later. However, we did not adopt this approach to keep the implementation simple. Empirical tests also suggest that looping through 1023 frontiers corresponding to each remoteness takes less than a second to complete, which is negligible compared to the amount of time spent solving positions of each remoteness level.

Also note that in Algorithm 1.4, we did not specify the value assigned to $R[p]$. This is because drawing positions implicitly have infinite remoteness and $R[p]$ is ignored when we query the database on position p. In practice, $R[p]$ is set to zero for all drawing positions to improve compressibility of the database.

Figures 3.1.1 through 3.1.10 simulates using Algorithm 1 to solve a loopy tier. The counter array for undecided children and output arrays V and R are omitted for simplicity. Instead, we label the assignments of value and remoteness directly on the vertices to indicate positions that are solved. The values of positions and moves are represented using colors as defined in Section 2.1, and remotenesses are directly labeled as integers on top of the vertices. The execution is divided into 10 steps, where each step explores one remoteness level of winning and losing positions (Lines 1-24 of Algorithm 1.3), tying positions (Lines 25-36 of Algorithm 1.3), or drawing positions (Algorithm 1.4). Note that the values of moves (edges) are labeled as they are explored.

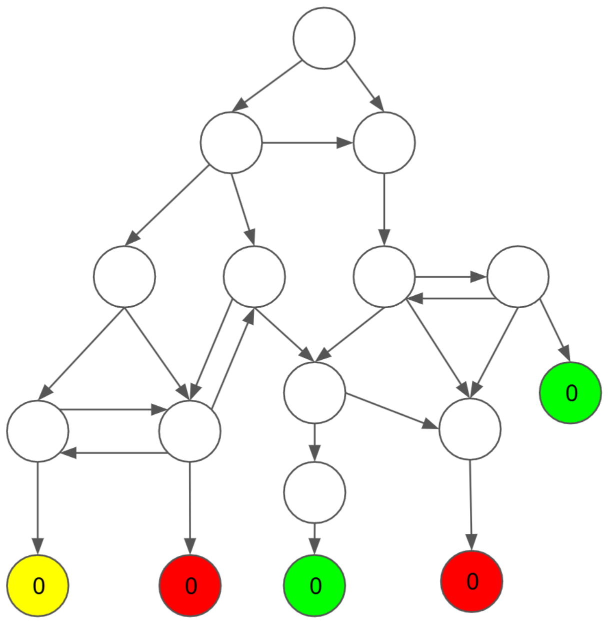

Figure 3.1.1: Initial state with all primitive positions scanned.

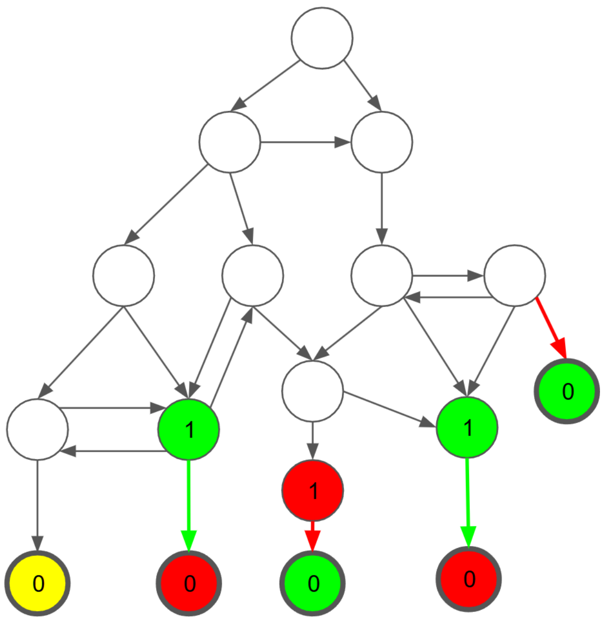

Figure 3.1.2: State after processing all winning and losing positions of remoteness 0

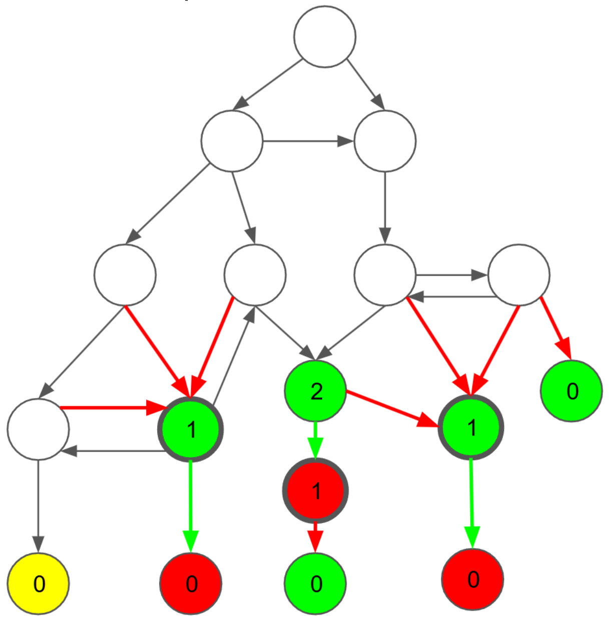

Figure 3.1.3: State after processing all winning and losing positions of remoteness 1

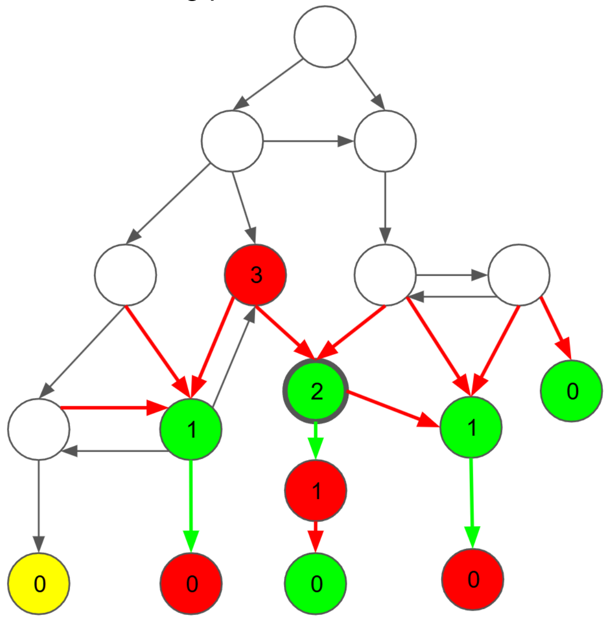

Figure 3.1.4: State after processing all winning and losing positions of remoteness 2

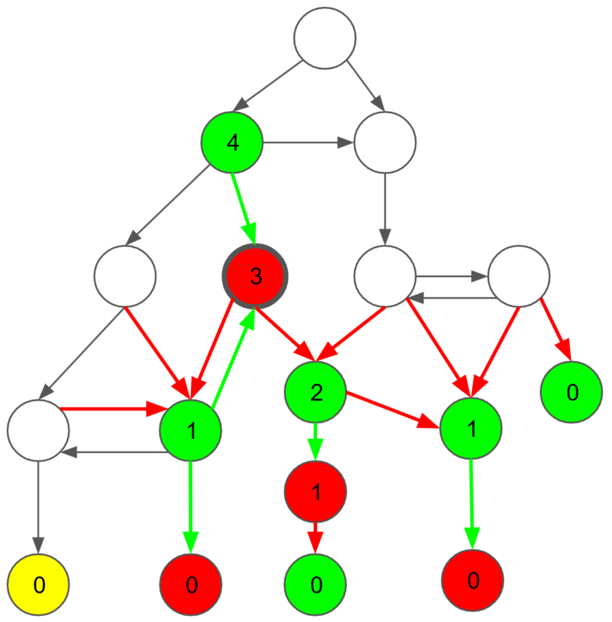

Figure 3.1.5: State after processing all winning and losing positions of remoteness 3

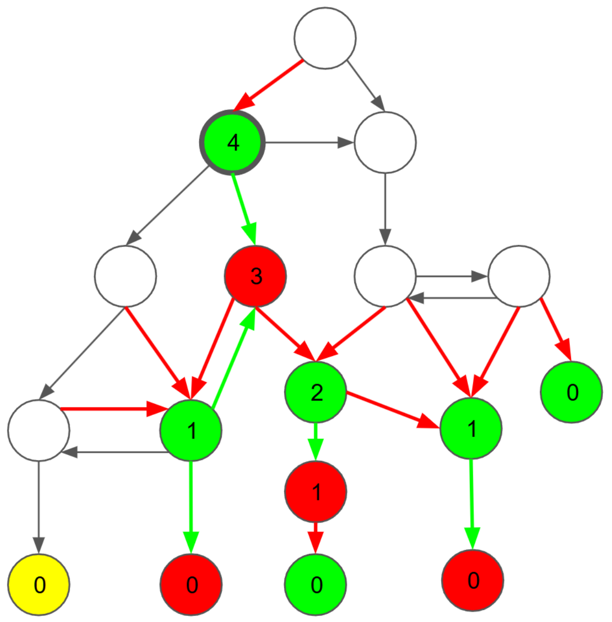

Figure 3.1.6: State after processing all winning and losing positions of remoteness 4

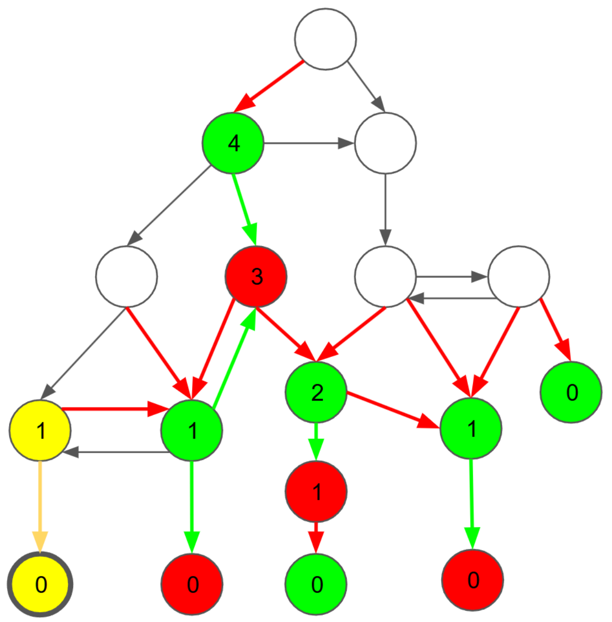

Figure 3.1.7: State after processing all tying positions of remoteness 0

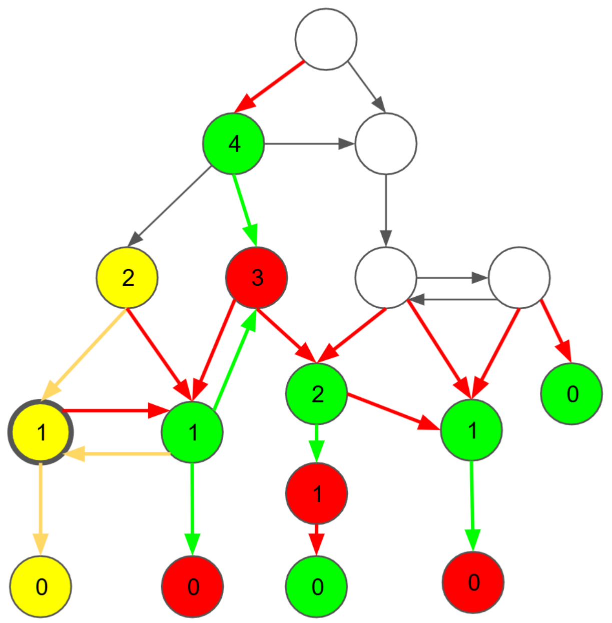

Figure 3.1.8: State after processing all tying positions of remoteness 1

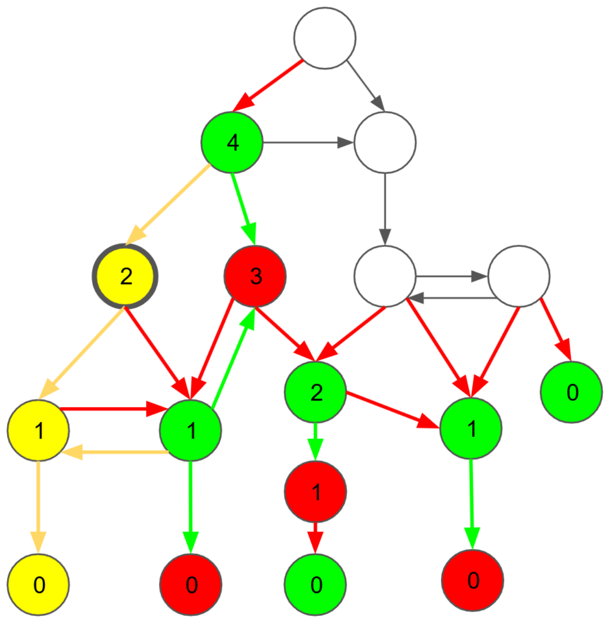

Figure 3.1.9: State after processing all tying positions of remoteness 2

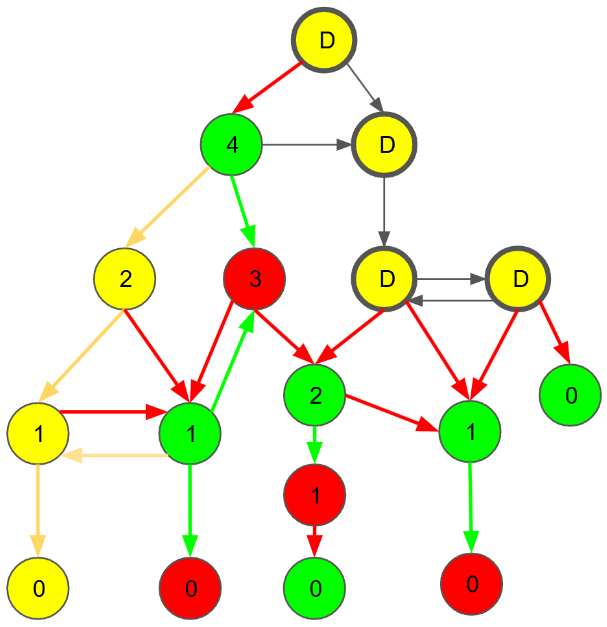

Figure 3.1.10: State after assigning all remaining positions to draws

3.2 Proof of the Loopy Solving Algorithm

To prove that the algorithm can be used to solve any tier of a loopy game, we need to show that

a. the algorithm terminates in a finite number of steps, and
b. the algorithm outputs the correct assignments of values and remotenesses to all positions in T. This includes
    i. the assignment of values and remotenesses to all winning and losing positions,
    ii. the assignment of values and remotenesses to all tying positions, and
    iii. the assignment of values to all drawing positions.

Proof of Part a

All loops in Algorithms 1.1, 1.2, and 1.4 have a finite number of iterations and will therefore terminate.

To see that Algorithm 1.3 terminates, notice that $\mathrm{num\_undecided\_children}[p]$ is always set to 0 at the same time when position p enters one of the frontiers. Furthermore, for all p in T, $\mathrm{num\_undecided\_children}[p]$ never increases throughout Algorithm 1.3. Therefore, by validating that $\mathrm{num\_undecided\_children}[p] > 0$ on lines 4, 12, and 24, each p in T is guaranteed to enter one of the frontiers no more than once. All other loops have a finite number of iterations and therefore terminate. □

Proof of Part b

Proof of the Correctness of Illegal Position Handling

Recall that the algorithm accepts an input function `IsLegalPosition`, which performs a weak test on the legality of positions. We first need to show that the existence of illegal positions in the solve process does not affect the outcome of legal positions.

By definition, illegal positions are not reachable from the initial position. In other words, all child positions of any legal position are legal positions. This implies that all parent positions of any illegal position must be illegal. Therefore, the assignment of arbitrary values and remotenesses to illegal positions is guaranteed not to affect the outcome of legal positions because no legal parent positions are generated on lines 3, 13, and 27 of Algorithm 1.3.

<u>Proof of Part b (i)</u>
We prove the correctness value and remoteness assignments using induction. Let r be remoteness, and r_{max} be the maximum remoteness of any winning or losing position in tier T. Proving the original statement is equivalent to showing that, for all possible r in $[0, r_{max}]$ and all winning or losing position p that has remoteness r in T, the algorithm assigns $R[p]$ to r and $V[p]$ to v, where v is the actual value of p.

Base case: For all winning or losing position p of remoteness $r = 0$, the correctness of $V[p] = v$ is guaranteed by the `Primitive` function, and $R[p] = r = 0$ is guaranteed by Algorithm 1.2.

Induction hypothesis: Assume the statement holds for all r in $[0, k]$, where $k \geq 0$.

Inductive step: We want to show that the statement holds for any winning or losing position p of remoteness $r = k + 1$.

We first consider winning positions of remoteness $k + 1$. Let p be any winning position of remoteness $k + 1$. By definition, p must have at least one losing child position of remoteness k. Without loss of generality, let p' be the first one of them in the losing frontier on line 2 of Algorithm 1.3. Also by definition of a winning position, p may have winning, tying, or drawing child positions, or more than one losing child position of remoteness k, but it must not have any losing child position of remoteness less than k.

Since in Algorithm 1.3, all tying and drawing positions are not processed until all winning and losing positions are, the tying and drawing child positions of p could not have affected the assignment of $V[p]$ and $R[p]$. Furthermore, the winning child positions of p may have reduced the number of undecided children of p by 1. However, they could not have reduced $\text{num\_undecided\_children}[p]$ to 0 because of the existence of p' and therefore could not have affected the assignment of $V[p]$ and $R[p]$ either. Since Algorithm 1.3 processes positions of smaller remotenesses first (as seen from line 1), no losing child positions of p could have been processed when $remoteness = k$. Therefore, they could not have affected the assignment of $V[p]$ and $R[p]$. Finally, when $remoteness = k$, the induction hypothesis and lines 2-7 of Algorithm 1.3 guarantee the correct assignment of $V[p] = lose$ and $R[p] = k + 1$ when p' is processed from the losing frontier. Line 5 guarantees that the assignments to $V[p]$ and $R[p]$ are never modified again throughout the execution of Algorithm 1.

Then, we consider all losing positions of remoteness $k + 1$. Let p be any losing position of remoteness $k + 1$. By definition, p has at least one winning child position of remoteness k. Furthermore, p can only have winning child positions of remoteness at most k. By the induction hypothesis, all of these child positions must have been processed by lines 12-15 of Algorithm 1.3. Since $\text{num\_undecided\_children}[p]$ is reduced to 0 only when the last winning child position of remoteness k is processed, $V[p]$ is set to lose and $R[p]$ is set to $k + 1$ on line 16 of Algorithm 1.3. $\text{num\_undecided\_children}[p]$ = 0 guarantees that these assignments are never modified again. □

Proof of Part b (ii)
The proof of correctness for part b (ii) is similarly done using induction as in the previous proof for part b (i).

Base case: For all tying position p of remoteness $r = 0$, the correctness of $V[p] = v$ is guaranteed by the `Primitive` function, and $R[p] = r = 0$ is guaranteed by Algorithm 1.2.

Induction hypothesis: Assume the statement holds for all r in $[0, k]$, where $k \geq 0$.

Inductive step: We want to show that the statement holds for all tying position p of remoteness $r = k + 1$. By definition of a tying position, p must have at least one tying child position of remoteness k and may also have winning child positions and other tying child positions of remotenesses greater than k. The processing of all winning child positions of p could not have assigned values to $V[p]$ or $R[p]$ because $\text{num\_undecided\_children}[p]$ is still greater than 0. On the other hand, all tying child positions of remotenesses greater than k have not been processed when $remoteness = k$ on line 25 of Algorithm 1.3. Therefore, the inductive hypothesis and lines 26-31 of Algorithm 1.3 guarantees that $V[p]$ is set to tie and $R[p]$ is set to $k + 1$ when the first tying child position p' of p is processed from the tying frontier. Line 29 guarantees that the above assignments to $V[p]$ and $R[p]$ are never modified again later.

Proof of Part b (iii)
Since all winning, losing, and tying positions are correctly solved, the remaining legal positions must be all drawing positions. Lines 1-3 of Algorithm 1.4 guarantees that $V[p] = draw$ for all p that are not winning, losing, or tying. Since the remoteness of drawing positions is infinite, there is no need to explicitly assign the values of $R[p]$ for these positions. □

3.3 Parallelization

3.3.1 Shared Memory Parallelization

The iterations of many loops in Algorithm 1 can be executed in an arbitrary order, making the loops sources of parallelism.

The for loop on line 2 of Algorithm 1.1 can be parallelized by assigning different iterations to different threads because the processing of positions inside each frontier does not need to follow a particular order. Note that line 1 of Algorithm 1.1 can also be parallelized if we were to implement the algorithm as given. However, further optimization (see Section 3.5.1) requires a modification to the frontier structure and that the child tiers be processed in a fixed order. Therefore, we omit the parallelization of the loop on line 1 of Algorithm 1.1.

The for loop on line 1 of Algorithm 1.2 can be parallelized by assigning different iterations to different threads because we are writing to distinct locations of the V, R, and $\mathrm{num\_undecided\_childen}$ arrays indexed by p.

Lines 2, 12, and 26 of Algorithm 1.3 can be parallelized because the proof in Section 3.2 still holds even if the positions within each frontier are processed in an arbitrary order. The correctness of the algorithm holds as long as the frontiers are processed in increasing remoteness order, and the losing and winning frontiers are processed before the tying frontier.

Line 1 of Algorithm 1.4 can be parallelized in a similar way as line 1 of Algorithm 1.2.

3.3.2 Distributed Memory Parallelization

Algorithm 1 requires a shared $O(n)$ memory space to store the values (array V) and remotenesses (array R) of all positions in the tier that is currently being solved. Since the algorithm is applied to generic games, the access pattern of the above arrays is game-dependent and random in the worst case. Therefore, parallelizing using distributed memory machines may require very careful implementation of complex algorithms. Although we have not attempted to implement such an algorithm, the algorithm may potentially allow us to overcome the limitations imposed by the memory capacity of a single machine and solve larger tiers. It is therefore left here as a future work.

3.4 Synchronizations

In shared memory parallel programs, threads communicate by accessing a shared memory space. Synchronization is therefore critical to the correctness and performance of such a program. Ideally, we want threads to communicate with each other as infrequently as possible to minimize the communication overhead. This can be achieved by creating local copies of data structures for each thread and delaying the gathering of results to the end of each parallel section. However, data structures such as the output arrays V and R, and the undecided child position counter array $\mathrm{num\_undecided\_children}$ grow linearly with the size of the tier T and therefore cannot be duplicated in a memory-efficient way. This section focuses on the synchronization techniques employed in the implementation of Algorithm 1 to ensure correctness and optimal performance.

3.4.1 Atomic Integers

The parallelization of Algorithm 1.2 and 1.4 described in Section 3.3.1 guarantees no race condition on the $\mathrm{num\_undecided\_children}$ array because each position p in T is processed by exactly one thread. However, more than one thread in the parallelized version of Algorithm 1.3 may write to the same index of the $\mathrm{num\_undecided\_children}$ array at the same time. As an example, consider the case where two threads t_1 and t_2 are processing positions (T_1', p_1') and (T_2', p_2') respectively on line 12 of Algorithm 1.3 at the same time. If the two positions happen to share the same parent position p in tier T, the two threads may try to decrement $\mathrm{num\_undecided\_children}[p]$ at the same time, hence creating a race condition. Therefore, we need to make sure that all accesses to the $\mathrm{num\_undecided\_children}$ array are atomic.

In GamesmanOne, which is discussed in detail in Chapter 5, the synchronization on the $\mathrm{num\_undecided\_children}$ array is achieved using the atomic operations library of C under the C17 standard [7]. The $\mathrm{num\_undecided\_children}$ array is declared as an `atomic_uchar` array of length n. The `atomic_uchar` type is an alias for `_Atomic unsigned char`, which behaves just like an `unsigned char` in C but allows atomic operations through functions in the `<stdatomic.h>` library.

Lines 4-5 and 28-29 of Algorithm 1.3 are implemented as

```c
unsigned char child_remaining =
    atomic_exchange_explicit(&num_undecided_children[parents.array[i]],
                             0, memory_order_relaxed);
if (child_remaining == 0) continue;
```

The first line atomically extracts the value stored in $\mathrm{num\_undecided\_children[parents.array[i]]}$ and replaces it with 0. If the value was 0 before it was swapped out, the parent position must have been solved already and is therefore skipped.

Lines 14-21 of Algorithm 1.3 are implemented as

```c
unsigned char child_remaining =
    DecrementIfNonZero(&num_undecided_children[parents.array[i]]);
if (child_remaining == 1) {
    // Process p...
}
```

where `DecrementIfNonZero` is defined as follows:

```
static unsigned char DecrementIfNonZero(atomic_uchar *obj) {
    unsigned char current_value = atomic_load_explicit(obj, memory_order_relaxed);
    while (current_value != 0) {
        bool success = atomic_compare_exchange_strong_explicit(
            obj, &current_value, current_value - 1, memory_order_relaxed,
            memory_order_relaxed);
        if (success) return current_value;
    }
    return 0;
}
```

We first need to make sure that each losing position is pushed into the losing frontier only once. This is ensured by the `DecrementIfNonZero` function, which atomically decrements $\mathrm{num\_undecided\_children[parents.array[i]]}$ and returns its original value if it was greater than 0, or does nothing and returns 0 otherwise.

To achieve the desired behavior of `DecrementIfNonZero`, we first atomically load the value and then test if the value is non-zero. If the value we loaded is 0, we return 0 immediately. Otherwise, we repeatedly attempt to exchange the value of `obj` with the value we previously held (stored as current_value) minus 1. The `atomic_compare_exchange_strong_explicit` function first tests if the value in `obj` is still equal to `current_value`. If so, we claim that the exchange is safe to be carried out because `current_value` is still valid and `current_value - 1` is the new value we want `obj` to have. In this case, the `success` flag is set to true, and we return the previous value of `obj`. If, for example, another thread modified the value of `obj` before the exchange operation, the test will fail and false will be stored in the `success` flag. In this case, `atomic_compare_exchange_strong_explicit` updates `current_value` to the new value of `obj` and the test and exchange process is repeated until `obj` is successfully updated or becomes 0.

3.4.2 Thread-Local Frontiers

Algorithm 1 describes an approach using 3 global arrays F_{win}, F_{lose}, and F_{tie} (called the frontiers) to store the positions that are solved but not yet processed. This approach is logically correct but requires explicit synchronization of all "push" operations in Algorithms 1.2 and 1.3 because multiple threads may be pushing new positions into the same frontier at the same time.

We proved in Section 3.2 that the algorithm does not rely on the order in which the positions of the same remoteness are pushed into the frontiers. Therefore, we come up with the following new design for the frontiers:

- Let $\mathrm{num\_threads}$ be the maximum number of threads used to process any parallel section in the algorithm.
- Redefine F_{win}, F_{lose}, and F_{tie} to be three 3D arrays. The first dimension of these 3D arrays is set to $\mathrm{num\_threads}$, which is determined at runtime. The second dimension and

the third dimensions are set to REMOTENESS_MAX and dynamically expanding respectively, as in the original design.

With the new design of the frontiers, each thread only pushes new positions into the frontiers that are reserved for it. For example, the thread with thread ID equal to 3 would push a new lose-in-4 position into $F_{lose}[3][4]$.

To read positions inside each frontier as we previously did on lines 2, 12, and 26 of Algorithm 1.3, we now need to read from all frontiers for all threads. Some additional steps would also be required to evenly redistribute the positions in all frontier arrays for each thread. However, the implementation of these steps is not logically complicated. With the new design, we effectively avoided communication and synchronization between threads while positions are processed from the frontiers, hence improving the performance of parallel execution.

3.4.3 Synchronizing Access to Solver Outcome Arrays

In Algorithm 1, we abstracted away the implementation complexity of the arrays storing the solver outcomes and simply referred to them as two arrays V and R. However, the actual implementation of such arrays depends on the implementation of the in-memory database.

A simple yet memory-expensive implementation of an in-memory database is to store the V and R as integer arrays. This approach is time-efficient as it requires no explicit synchronization for in-memory database accesses throughout the entire execution of Algorithm 1. This is because each position is guaranteed to have its value and remoteness assigned no more than once in Algorithm 1, and the values are never read from the in-memory database during the execution.

GamesmanOne employs a more optimized in-memory database implementation, which involves bit-level operations and maintaining a look-up table for all value-remoteness pairs that have appeared during the solving process. Since the value and remoteness records of each position are not stored in a byte-aligned fashion, the writing of one record may create a race condition with another access to a neighboring position in the in-memory database. The current implementation serializes all in-memory database operations using a single OpenMP lock.

3.5 Optimizations

3.5.1 Frontier Storage Optimization

In Algorithm 1, we described the frontiers as storing (T, p) tuples. For each position pushed into the frontiers, this approach requires us to store the hash value of the tier that the position is associated with. In actual implementation, the hash of each tier is stored as a 64-bit integer, as is the hash of a position.

However, we observe that the positions in each child tier C_i of T are pushed into the frontiers in increasing i order in Algorithm 1.1. This implies that the positions from each child tier are in consecutive chunks in each frontier. As we move on to the following subparts of the algorithm, all new positions pushed into the frontiers are from tier T. This suggests that if we know the number of positions pushed into the frontiers from each child tier, we can infer the tier of a position from its index. This allows us to store only the *positions* inside each frontier, reducing the memory usage by half.

3.5.2 The `GetParentPositions` Function

We abstracted away the implementation of the `GetParentPositions` function and simply referred to it as an input to Algorithm 1. In reality, this function needs to be generated at runtime using other functions that encode the rules of the game or implemented by the game developer according to the rules of the game.

The first way we can obtain this function is to dynamically generate it at runtime by constructing a reverse position graph of the tier we are solving. The advantage of this approach is that it shifts the logical burden away from the game developers as they now only need to implement logic for moves in the forward direction, which are typically well-defined by the game rules. However, this approach significantly increases the memory overhead due to the storage of the reverse position graph, which is of size $O(n + m)$, where m is the number of all possible moves from all positions in T.

The second way is to implement the `GetParentPositions` function as part of the game. The `GetParentPositions` function, assuming correctly implemented, implicitly defines the reverse position graph of any tier T in the game, hence eliminating the need to store the reverse graph in memory. The disadvantage of this approach is that the implementation of such a function often requires very careful reasoning of all possible moves that could lead to the current position. The implementation of the `GetParentPositions` function must be thoroughly tested in the following way to make sure that it matches the implementation of functions that define moves in the forward direction:

> Suppose the forward moves are defined by a function `GetChildPositions` which takes in a tier T and a position p and returns an array of child positions of p. Then, we must have that for all positions p and c,
>
> - if `GetParentPositions` returns p as a parent of c, then `GetChildPositions` must return c as a child of p, and
> - if `GetChildPositions` returns c as a child of p, then `GetParentPositions` must return p as a parent of c.

In GamesmanOne, both options are available for implementing the `GetParentPositions` function, and the design choice is up to the game developer. A number of tests to validate the above conditions for some randomly chosen game positions are also provided as part of the user interface.

3.5.3 Position Symmetry Removal

Many games have positions symmetric to each other. We can reduce the runtime of the algorithm by exploring only the *canonical* positions. To achieve this, we introduce the following new input to the algorithm

- `GetCanonicalPosition`(T, p): a function that takes in a position p in tier T and returns the canonical position p' in T.

Recall from Section 2.1 that the WORM model is preferred. The following modifications to Algorithms 1.2 and 1.3 implements position symmetry removal under the WORM database model:

Algorithm 1.2.1: Algorithm for Scanning a Tier w/ Position Symmetry Removal

```
1.  for $p = 0$ to $n - 1$ do
2.      $V[p] \leftarrow undecided.$
3.      if $p$ is not a legal position OR if GetCananicalPosition($T$, $p$) $!= p$ then // also
        skipping non-canonical positions as their values can be determined when their canonical
        counterparts are solved.
4.          $\mathrm{num\_undecided\_children}[p] \leftarrow 0.$
5.          continue
6.      end if
7.      Let $children \leftarrow$ GetChildPositions($T$, $p$)
8.      Let $S$ be an empty set of (tier, position) tuples. // for deduplication
9.      for each ($T'$, c) in $children$ do
10.         Let $c'$ ← GetCanonicalPosition($T'$, c)
11.         if $c'$ is not in $S$ then
12.             Add $c'$ to $S$
13.         end if
14.     end for
15.     $\mathrm{num\_undecided\_childen}[p] \leftarrow S.\mathrm{size}.$ // number of unique canonical child positions
16.     if Primitive($T$, $p$) == win then
17.         $V[p] \leftarrow win.$
18.         $R[p] \leftarrow 0.$
19.         Push ($T$, $p$) into $F_{win}[0]$.
20.     else if Primitive($T$, $p$) == lose then
21.         $V[p] \leftarrow lose.$
22.         $R[p] \leftarrow 0.$
23.         Push ($T$, $p$) into $F_{lose}[0]$.
24.     else if Primitive($T$, $p$) == tie then
25.         $V[p] \leftarrow tie.$
```

```
26.         $R[p] \leftarrow 0$.
27.         Push $(T, p)$ into $F_{tie}[0]$.
28.     end if
29. end for
```

Algorithm 1.3.1: Algorithm for Pushing the Frontiers Up w/ Position Symmetry Removal

```
1.  for $remoteness = 0$ to REMOTENESS_MAX do
2.      for each position $(T', p')$ in $F_{lose}[remoteness]$ do
3.          Let $S$ be an empty position set.
4.          for each parent position $p$ in GetParentPositions($T'$, $p'$, $T$) do
5.              $p \leftarrow$ GetCanonicalPosition($T'$, $p$).
6.              if $p$ is not in $S$ AND num_undecided_children[$p$] > 0 then
7.                  add $p$ to $S$.
8.                  num_undecided_children[$p$] $\leftarrow 0$.
9.                  $V[p] \leftarrow win$.
10.                 $R[p] \leftarrow remoteness + 1$.
11.                 Push $(T, p)$ into $F_{win}[remoteness + 1]$.
12.             end if
13.         end for
14.     end for
15.     for each position $(T', p')$ in $F_{win}[remoteness]$ do
16.         Let $S$ be an empty position set.
17.         for each parent position $p$ in GetParentPositions($T'$, $p'$, $T$) do
18.             $p \leftarrow$ GetCanonicalPosition($T$, $p$).
19.             if $p$ is not in $S$ AND num_undecided_children[$p$] > 0 then
20.                 add $p$ to $S$.
21.                 num_undecided_children[$p$] $\leftarrow$ num_undecided_children[$p$] - 1
22.                 if num_undecided_children[$p$] $== 0$ then
23.                     $V[p] \leftarrow lose$.
24.                     $R[p] \leftarrow remoteness + 1$.
25.                     Push $(T, p)$ into $F_{lose}[remoteness + 1]$.
26.                 end if
27.             end if
28.         end for
29.     end for
30. end for
31. for $remoteness = 0$ to REMOTENESS_MAX do
```

```
32.   for each position $(T', p')$ in $F_{tie}[remoteness]$ do
33.     Let $S$ be an empty position set.
34.     for each parent position $p$ in GetParentPositions($T'$, $p'$, $T$) do
35.       $p \leftarrow$ GetCanonicalPosition($T$, $p$).
36.       if $p$ is not in $S$ AND $\mathrm{num\_undecided\_children}[p] > 0$ then
37.         add $p$ to $S$.
38.         $\mathrm{num\_undecided\_children}[p] \leftarrow 0$.
39.         $V[p] \leftarrow tie$.
40.         $R[p] \leftarrow remoteness + 1$.
41.         Push $(T, p)$ into $F_{tie}[remoteness + 1]$.
42.       end if
43.     end for
44.   end for
45. end for
```

The added/modified lines of the two algorithms are highlighted in red in Algorithms 1.2.1 and 1.3.1 above.

With the new algorithm design, the $\mathrm{num\_undecided\_children}$ array now stores the number of undecided canonical positions instead. In addition, one can conclude that a position p is losing as soon as all canonical child positions of p have been solved and evaluated to win.

Note that a set is used each time we generate an array of canonical child or parent positions. This is necessary because it is possible that two different child (or parent) positions of the same position are symmetric to each other.

3.5.4 Workload Balancing

In the shared memory parallelization model, a necessary condition for achieving scalability is to ensure that each thread is allocated an approximately equal amount of work. Since all parallelization in this section is achieved by parallelizing the iterations of loops, we will only focus on the redistribution of loop iterations in this subsection. Furthermore, since the above algorithm is implemented in C and parallelized using the OpenMP library, we will only discuss workload balancing techniques that are available in OpenMP.

Loops with a relatively simple body can usually be handled well by a static scheduling policy. This is because under a static scheduling policy, each thread is assigned with the same number of iterations of a parallel for loop (with some exceptions[2]). If each iteration takes about the same amount of time to complete, the workload will almost be evenly distributed.

On the other hand, if each iteration of a loop takes a different amount of time to complete, a static scheduling policy may not work very well because some threads may finish early and stay idle until all other threads finish their work. In this case, using a dynamic or guided scheduling policy can lead to better performance. The OpenMP dynamic scheduling policy splits the number of iterations of a for loop into chunks of `chunk_size` and then distributes the chunks to threads on a first-come-first-serve basis. A large `chunk_size` is more suitable if cache locality is better utilized by processing many consecutive iterations in a row, whereas a small `chunk_size` improves workload balancing. Enabling dynamic scheduling also incurs a scheduler overhead at runtime, which may be reduced by using a larger `chunk_size`. Therefore, the optimal `chunk_size` depends on the implementation of the game and requires benchmarking. The OpenMP guided scheduling policy, on the other hand, first splits the number of iterations to large chunks and gradually decreases the `chunk_size` as it approaches the end of the loop execution.

The scheduling policies for the parallel for loops mentioned in Section 3.3 are listed below:

a. Line 2 of Algorithm 1.1 uses a guided scheduling policy. The most expensive operation in this loop involves loading positions into the frontier. The workload may become imbalanced if some threads receive a large number of consecutive drawing or illegal positions that are not loaded into the frontiers while other threads receive a large number of winning, losing, and tying positions. This is especially true if we enable position symmetry removal as described in Subsection 3.5.3. However, the loading of positions from child tiers involves reading data from disk, whose latency can be hidden only if we read data in large enough chunks. Therefore, the guided scheduling policy is chosen.[3]
b. The for loop on line 1 of Algorithm 1.2 uses a dynamic scheduling policy because of a similar reason as in the previous part. With position symmetry removal enabled, more canonical positions can be found in ranges of smaller position hashes than ranges of larger position hashes, therefore causing workload imbalance under a static scheduling policy.
c. Lines 2, 12, and 26 of Algorithm 1.3 all use the dynamic scheduling policy. The actual behavior of these parallel for loops are much harder to predict and are game-dependent. However, the dynamic scheduling policy with a reasonable `chunk_size`[4] generally results in better performance than a static scheduling policy.
d. Line 1 of Algorithm 1.4 similarly suffers from imbalanced workload for the same reasons as in part a. However, unlike part a, there are no disk operations involved when labeling

[2] If no chunk size is specified for the static scheduling policy, the last thread may receive fewer iterations if the number of threads does not divide the total number of iterations. If a chunk size is specified, the workload distribution depends on the number of threads, iterations, and the chunk size, and more than one thread may receive fewer iterations.
[3] Empirically, using a dynamic scheduling policy with a reasonably large chunk size (e.g., 4096) yielded a similar performance.
[4] The current implementation of GamesmanOne uses a hard-coded `chunk_size` of 1024.

draw positions. Therefore, a dynamic scheduling policy with a reasonable `chunk_size` is more appropriate as it provides slightly better load balancing.

3.6 Scaling Efficiency

To get an insight into how effectively the algorithm can handle increasing workloads and how it scales when additional computing resources are added, we need to measure the strong and weak scaling efficiencies of the algorithm. The scaling tests were conducted on the fully optimized implementation of Algorithm 1 in GamesmanOne's tier solver using a single CPU node of the Perlmutter supercomputer. Each CPU node uses 2x AMD EPYC 7763, 64 cores per socket, and 512 GiB memory [8].

3.6.1 Strong Scaling

The strong scaling tests were done by solving a single tier (5, 5, 6) of **4x4 Quixo**, which is one of the largest tiers of this game variant. The rules of the game and definitions for the tiers are given in Chapter 7 of this report.
The size of tier (5, 5, 6) is 4036032 positions. The test is conducted by solving the same tier multiple times using different numbers of OpenMP threads. We tested the program using 1, 2, 4, 8, 16, 32, and 64 threads. The number of threads are chosen to be no greater than the number of cores per socket to minimize the effect of hyperthreading and/or inter-socket communication on performance. The number of OpenMP threads is manually adjusted before each test run using the command

```
export OMP_NUM_THREADS=<n>
```

where n is the number of threads to use. The execution time is measured by taking the difference between the finish and the start time returned from the OpenMP `omp_get_wtime` function. The results are shown in Table 3.1.

Number of threads	Ideal (s)	Actual (s)
1	29.248698	29.248698
2	14.624349	15.3226
4	7.3121745	7.808866
8	3.65608725	4.009482
16	1.828043625	2.142348
32	0.9140218125	1.224848
64	0.4570109063	0.730756
128	0.2285054531	0.583174

Table 3.1: Strong scaling execution time of Algorithm 1

Under the assumption of perfect scaling, the speedup should be equal to the number of threads used. However, this is almost never achievable due to scheduling and synchronization overhead. The actual speed up of the program is plotted as Figure 3.1.

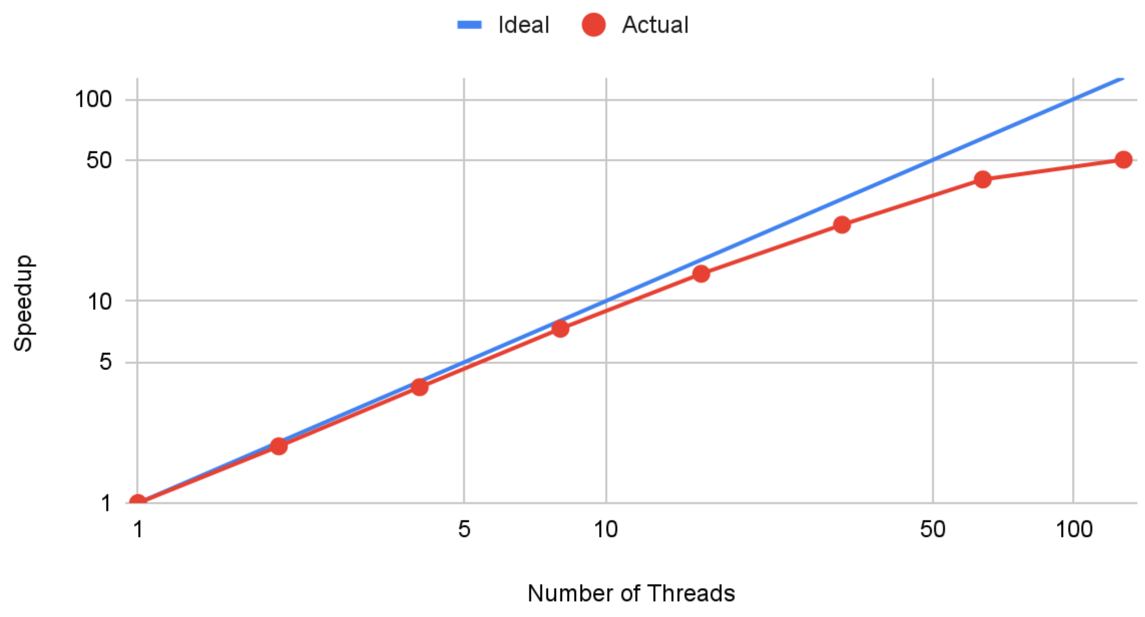

Figure 3.1: Strong scaling speedup of Algorithm 1

The algorithm demonstrates good scalability with the utilization of up to 32 threads. A potential explanation for this behavior is that Algorithm 1.3 incorporates a sequential process that handles each remoteness level from 0 to REMOTENESS_MAX in series. When the size of each tier is substantial, the impact of this sequential component is less significant, as the processing time for each remoteness level becomes the dominant factor.

3.6.2 Weak Scaling

Weak scaling is not quite applicable to game solving because the ultimate goal is to solve a particular game whose size remains constant regardless of the amount of resource being used. It is also difficult to perform the test by giving each thread the same workload because the total workload of solving a game depends on the number of edges in the game graph, which is unknown until the game is solved. Nevertheless, we include the result of weak scaling measurements for completeness. We also make the assumption that the total workload for each test run is linear to the number of positions.

The weak scaling tests were done by solving tiers (2, 11, 3), (1, 9, 6), (2, 10, 4), (2, 9, 5), (6, 8, 2), (5, 8, 3), (5, 7, 4), and (5, 5, 6) of 4x4 Quixo. The size of these tiers are 87360, 160160, 240240, 480480, 720720, 1441440, 2882880, and 4036032 positions respectively, and the ratio of their sizes roughly matches 1:2:3:6:9:18:36:50, which correspond the numbers of threads we used to conduct the tests. All tests are carried out using the same hardware as in Subsection 3.6.1. The results are shown in Table 3.2.

number of threads	Ideal (s)	Actual (s)
1	0.348333	0.348333
2	0.348333	0.418242
3	0.348333	0.386318
6	0.348333	0.46231
9	0.348333	0.603022
18	0.348333	0.658355
36	0.348333	0.800556
50	0.348333	0.907884

Table 3.2: Weak scaling execution time of Algorithm 1

The results are plotted in Figure 3.2.

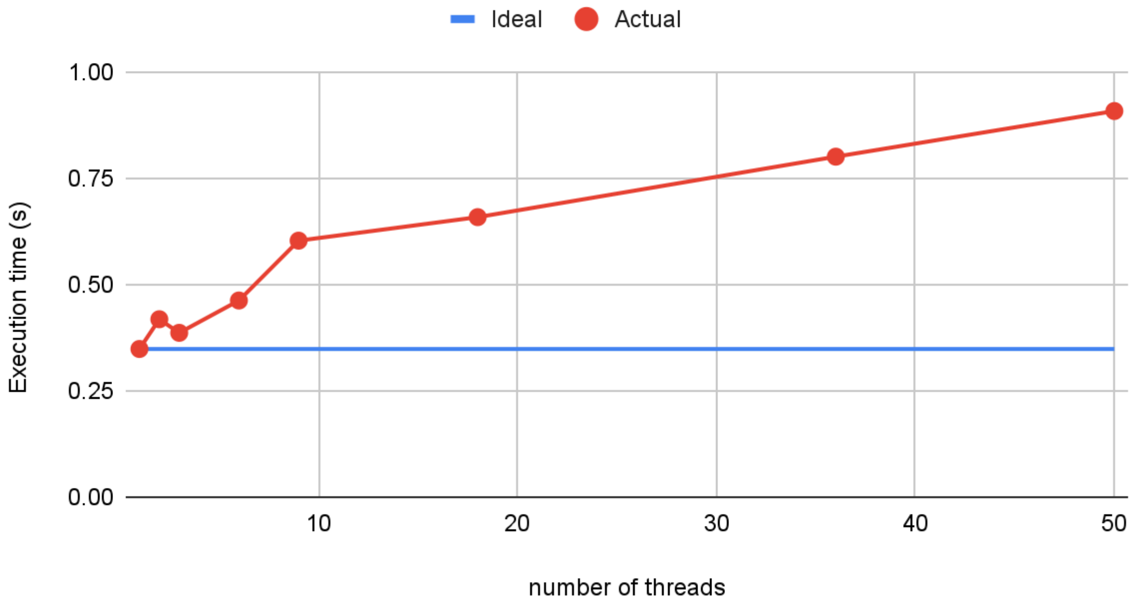

Figure 3.2: Weak scaling execution time of algorithm 1

This result is expected as our assumption on the linear relationship between workload and the number of positions is inaccurate. As we increase the number of positions, the average branching factor also increases because those tiers contain mostly middle-game positions where more moves are available for each player.

Chapter 4: The Tier Solving Algorithm and MPI

In Chapter 3, we discussed an algorithm that allows us to solve one tier. To solve a full tier game, we need to solve all tiers. Doing so requires that tiers be solved in such an order that each tier has all of its child tiers solved before it. In this chapter, we discuss an algorithm that allows us to create a reverse tier graph and then solve tiers in any valid topological order of the graph, hence maximizing the number of tiers that can be solved in parallel.

All algorithms in this chapter are implemented as part of the *Tier Solver module* of GamesmanOne in `tier_manager.c` and `tier_worker.c`.

4.1 Creating the Reverse Tier Graph

For all games with a valid non-trivial tier definition, the following functions are implemented as part of the game module:

1. `GetInitialTier()`: returns the initial tier of the game.
2. `GetChildTiers`(T): returns an array of child tiers of tier T, assuming T is a valid tier.
3. `GetTierSize`(T): returns the size of tier T, assuming T is a valid tier in the game.

The first two functions implicitly define a tier graph: `GetInitialTier` defines the root of the tier graph, whereas `GetChildTiers` defines the outgoing edges of any vertex in the tier graph. Therefore, we can perform a depth-first search starting from the root tier to create a reverse tier graph while we visit all vertices in the tier graph.

We create the following data structures for the algorithm:

- Let fringe be an initially empty tier stack.
- Let closed be an initially empty tier set.
- Let G be an initially empty map from tiers to tier arrays. G is used to store the reverse tier graph and is also the output of the algorithm.

The algorithm is shown summarized as Algorithm 2 below.

Algorithm 2: Algorithm for Creating the Reverse Tier Graph

```
1: Let $T_r$ ← GetInitialTier()
2: Push $T_r$ into $\mathrm{fringe}$.
3: repeat
4:     Pop out one tier from $\mathrm{fringe}$ and store it as $T$.
5:     if $T$ is not in $\mathrm{closed}$ then
6:         Let $children$ ← GetChildTiers($T$)
7:         for each $child$ in $children$ do
```

```
8:          Push $child$ into fringe
9:          Append $T$ to $G[child]$.
10:       end for
11:       Add $T$ to closed.
12:    end if
13: until fringe is empty
```

4.2 Solving the Reverse Tier Graph

Algorithm 2 provides us with a reverse tier graph G for the game that we are looking to solve. To solve the game, we need to apply algorithm 1 to each tier in G in a topological order. We introduce the following algorithm to achieve this.

Define the following data structures:

1. Let Q be an initially empty tier queue.
2. Let num_child_tiers be an integer array of length equal to the number of tiers in G.

Algorithm 3: Algorithm for Solving All Tiers

```
1:  for each tier $T$ in $G$ do
2:     Let $children \leftarrow$ GetChildTiers($T$).
3:     num_child_tiers[$T$] $\leftarrow$ $children$.size.
4:     if $children$.size $== 0$ then
5:        Add $T$ to $Q$
6:     end if
7:  end for
8:  repeat
9:     Pop the tier at the front of $Q$ and store it as $T$.
10:    Let $n \leftarrow$ GetTierSize($T$).
11:    Call Algorithm 1 on $T$ with $n$ as input for tier size.
12:    for each $T'$ in $G[T]$ do
13:       num_child_tiers[$T'$] $\leftarrow$ num_child_tiers[$T'$] $- 1$.
14:       if num_child_tiers[$T'$] $== 0$ then
15:          Add $T'$ to $Q$.
16:       end if
17:    end for
18: until $Q$ is empty
```

To see that Algorithm 3 calls Algorithm 1 on all tiers in G in a topological order, notice that each tier T only enters Q once throughout the entire execution when $\mathrm{num\_child\_tiers}[T]$ is set to 0.

4.3 Parallelization

4.3.1 Distributed Memory Parallelization Using MPI

The key observation in Algorithm 3 is that all tiers in Q can be solved in parallel. This is because the solving of each tier T using Algorithm 1 only requires that the child tiers of T have been solved. Since Algorithm 3 guarantees that all tiers in Q have all their child tiers solved, we are safe to invoke Algorithm 1 on any number of them at the same time. Furthermore, we observe that each invocation of Algorithm 1 never communicates with another. This allows us to efficiently allocate each tier in Q to a separate process as a task, which may reside on a different machine connected through a network layer. Therefore, one idea of parallelization is to use multiprocessing implemented by the *Message Passing Interface (MPI)* and employ a manager-worker model [3] for task delegation.

A group of MPI processes working under the manager-worker model is divided into two subgroups:

1. The manager process: a single process that is responsible for creating the reverse tier graph, maintaining the queue of tiers ready to be solved, and delegating tiers in the queue to the worker processes.
2. The worker processes: all other processes that are not the manager; responsible for receiving tasks from the manager, solving individual tiers, and reporting back to the manager.

We parallelize Algorithm 3 by combining the following two algorithms[5] using the MPI and the manager-worker model.

Algorithm 4.1: Algorithm for the Manager Process

```
1: Let $\mathrm{task\_list}$ be an initially empty tier array of length equal to the total number of
   processes including the manager.
2: repeat
3:    Block until a message is received from any one of the workers.
4:    Let $rank$ be the rank of the worker process from which message was received.
5:    if message is “solved” then
6:       mark the $tier$ allocated to process rank as solved.
7:       update the number of unsolved child tiers of all parents of $\mathrm{task\_list}[rank]$.
8:    end if
```

[5] For simplicity, all error checkings have been omitted from the algorithms.

```
9:      if there currently are tiers that can be solved then
10:         Pop the next $tier$ from the front of $Q$.
11:         Send $tier$ to the worker process of rank equal to $rank$.
12:         $\text{task\_list}[rank] \leftarrow tier$.
13:     else
14:         Send a “sleep” message to the worker process of rank equal to $rank$.
15:     end if
16: until all tiers have been solved
17: Send a “terminate” message to all worker processes.
```

Algorithm 4.2: Algorithm for a Worker Process

```
1:  Send a “check” message to the manager process.
2:  while TRUE do
3:      Block until a message is received from the manager process.
4:      if message is “terminate” then
5:          break
6:      else if message is “sleep” then
7:          Sleep for 1 second.
8:          Send a “check” message to the manager process.
9:      else
10:         Let $n \leftarrow$ GetTierSize($T$)
11:         Call Algorithm 1 on $T$ with $n$ as input for tier size.
12:         Send a “solved” message to the manager process.
13:     end if
14: end while
```

In summary, the manager process remains in a worker-dispatching loop until all tiers are solved. On the other hand, all worker processes remain in a serving loop to solve tiers delegated by the manager until a “terminate” message is received.

Figures 4.1.1 through 4.1.8 show an example of solving the tier graph as shown in Figure 2.2 in parallel using two worker processes under the manager-worker model.

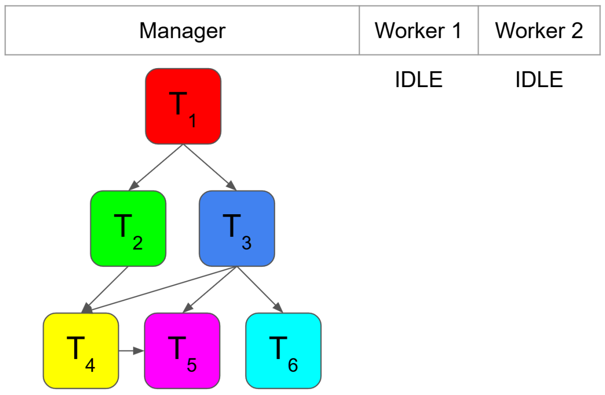

Figure 4.1.1: Initial state of the tier solver with both workers idle

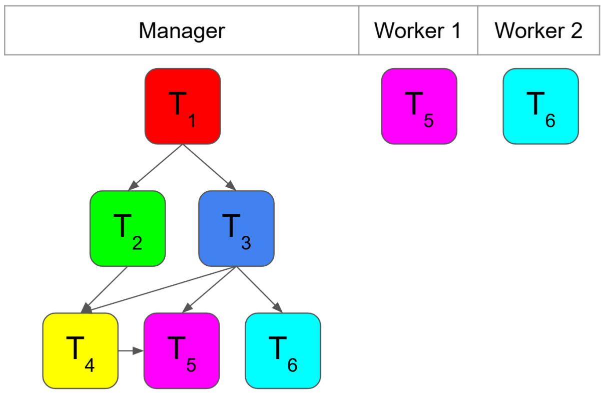

Figure 4.1.2: State of the tier solver after the manager assigning T_5 to worker 1 and T_6 to worker 2

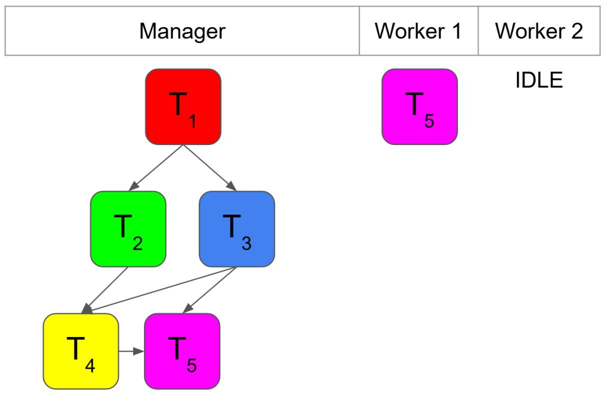

Figure 4.1.3: State of the tier solver after worker 2 finishes solving T_6

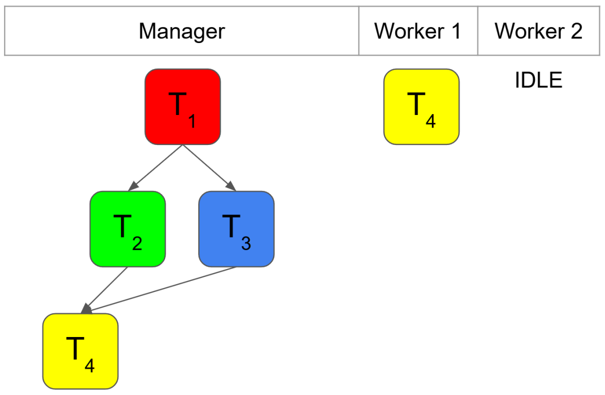

Figure 4.1.4: State of the tier solver after worker 1 finishes solving T_5 and receiving T_4 from the manager

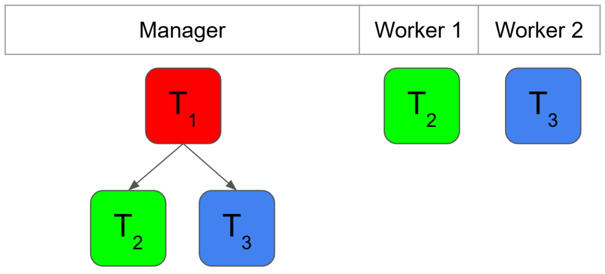

Figure 4.1.5: State of the tier solver after worker 1 finishes solving T_4. The manager then assigns T_2 and T_3 to the workers

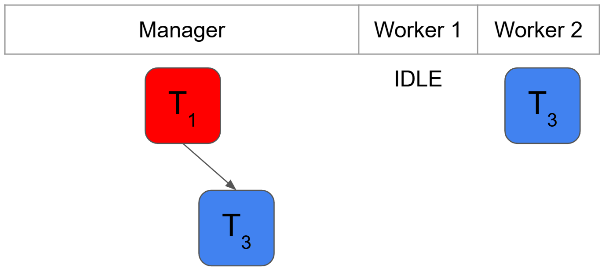

Figure 4.1.6: State of the tier solver after worker 1 finishes solving T_2

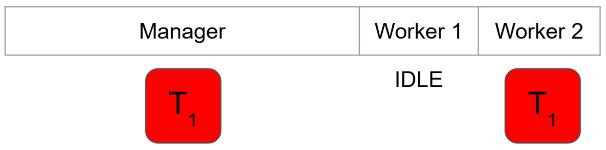

Figure 4.1.7: State of the tier solver after worker 2 finishes solving T_3 and receiving T_1 from the manager

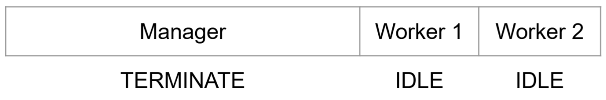

Figure 4.1.8: Final state of the tier solver after worker 2 finishes solving T_1. The manager then signals "terminate" to all workers

4.3.2 Shared Memory Parallelization

It is possible to generalize algorithm 4.1 and 4.2 to shared memory parallelization by replacing all processes with threads. Message passing can then be achieved between threads by accessing shared global memory. However, we did not implement tier-level parallelism using the shared memory model for the following reasons:

a. Each machine has a limited amount of memory. Solving multiple tiers on the same machine may limit the maximum size of the largest tier that we can solve.
b. Algorithm 1 scales reasonably well with the number of threads. Solving multiple tiers on the same machine at the same time may not significantly improve overall performance.
c. We can achieve the same goal of solving multiple tiers on the same machine using MPI with a slightly bigger communication overhead.

4.4 Synchronizations

One significant benefit of tier-solving is that tiers being solved at the same time do not need to communicate. Hence no synchronization is needed besides messages sent to and from the manager.

In Algorithms 4.1 and 4.2, all sends and receives are blocking, meaning that the function calls to send and receive will not return until the messages have actually gone through. Implementation-wise, the send and receive operations are achieved using the MPI standard `MPI_Send` and `MPI_Recv` functions that allow the caller to specify the target rank of the sender and receiver process. Therefore, all synchronizations are handled internally by MPI and no explicit synchronization is required.

4.5 The Tier Symmetry Removal Optimization

Similar to position symmetry removal, one can also leverage symmetry of tiers to reduce the number of positions explored during solve. Suppose two tiers are symmetric to each other, then solving either one of them allows one to infer the evaluations of all positions in the other.

To implement tier symmetry removal, we require two additional inputs for the new algorithm:

- `GetCanonicalTier`(T): a function that takes in a tier T and returns the canonical tier in the set of tiers symmetric to T.

- `GetPositionInSymmetricTier`(T, p, T'): a function that takes in a position p in tier T, and returns the position p' in tier T' that is symmetric to p, assuming T' is symmetric to T.

The first function, `GetCanonicalTier`, allows us to find the canonical tier and ignore the non-canonical tiers during solve. The second function, `GetPositionInSymmetricTier`, allows us to read the evaluations of positions of a non-canonical tier from the database of its canonical counterpart as if we have solved the non-canonical tier.

Then, we can modify Algorithm 3 to achieve tier symmetry removal in a similar way as we modified Algorithm 1.2 and 1.3 for position symmetry removal. The key modifications include the following points:

a. Instead of counting the number of child tiers of each tier, count the number of unique canonical child tiers. The uniqueness of child tiers can be guaranteed using a tier set for deduplication.
b. For each tier T popped out of the queue of pending tiers Q, skip the solving of T if T is not a canonical tier.

Chapter 5: GamesmanOne

The algorithms discussed in the Chapters 3 and 4 laid the groundwork for the implementation of a parallel loopy tier game solver. In this chapter we introduce **GamesmanOne**, a parallel solver and analyzer system for generic two-player abstract strategy games. In Section 5.1, we give an overview of the system and discuss various design choices that were made to modularize the system. In Sections 5.1 and 5.2, we introduce the various APIs that a game module must implement for it to be recognized by GamesmanOne. Section 5.4 explains the integration of game modules with GamesCrafters Universal Web API, which then allows the game to be presented on the web frontend provided by GamesmanUni. Section 5.5 discusses an optimization implemented for the Bit-Perfect Database, which is a database implementation in GamesmanClassic. Section 5.6 discusses the steps to compile GamesmanOne on a HPC platform that supports the Message Passing Interface (MPI). Finally, Section 5.7 summarizes all possible future work.

5.1 Overview

GamesmanOne is a parallel solver, analyzer, and game-playing system for generic two-player abstract strategy games written in C. It is implemented with special emphasis on parallelization and modularization. The system uses the OpenMP library for shared memory parallelization, and MPI for distributed memory parallelization. The components of the system are designed to be as self-contained as possible so that the implementation of new components, such as new solvers and databases, will not interfere with any existing component or game.

Figure 5.1 shows an overview of GamesmanOne's component structure.

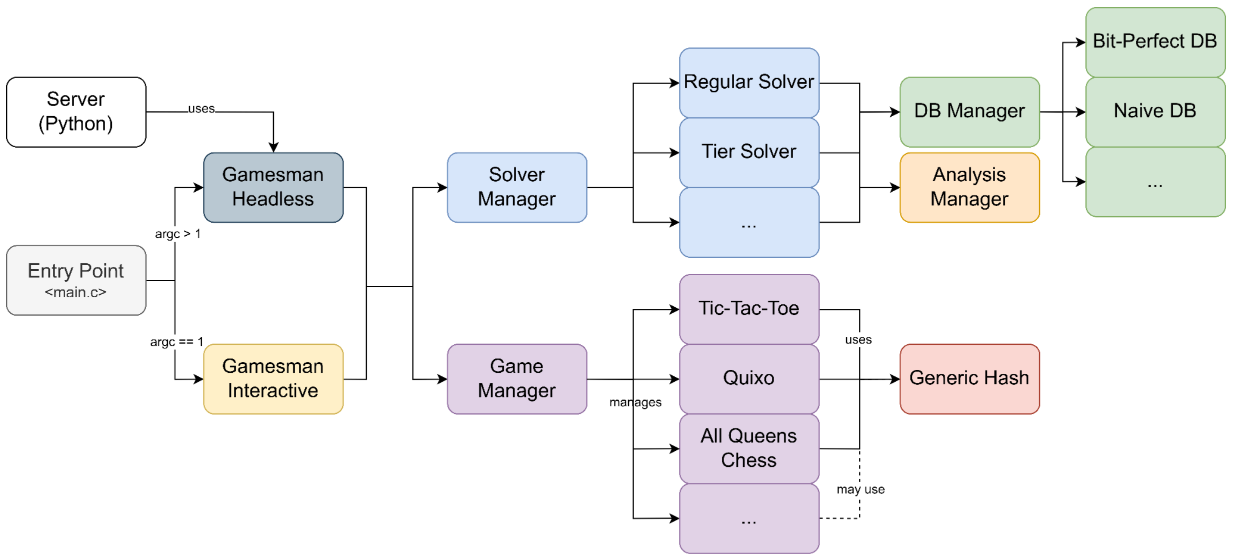

Figure 5.1: System overview of GamesmanOne

The system builds into a single `gamesman` binary capable of solving and analyzing all implemented games and has the ability to query the database of a game if it has been solved. The program has a single entry point, but can be used in two different ways depending on the number of command line arguments passed in.

The most common way to use `gamesman` is to use it as an interactive command line program. This can be done by launching the binary without any arguments; i.e. by running `bin/gamesman` under the root project directory, assuming by default the binary is compiled and stored in a subdirectory named `bin`. This mode provides the user with a command-line interface that allows the user to interactively switch games, specify game options, solve the game, analyze the game, and play the game with or without solving it. Since no system can catch errors in the implementation of game rules, it is important to have the ability to play the game before solving it to make sure that the game rules are correctly implemented.

If any option is passed to the program at launch time, the program will launch in headless mode. This mode accepts various commands such as `solve`, `analyze`, `query`, and `getstart`. The program then executes the command, typically with parameters that specify the name and variant of the game, generates an output, and exits. For position queries, a third parameter for the position string must also be specified. The default output path for databases and game statistics is `data` under the working directory in which the `gamesman` binary is invoked. For all commands above, the user can optionally change the output path by specifying the

```
--data-path=<path>
```

option. This is useful when the command is invoked from a different directory as the one containing `data`. The examples in Table 5.1 demonstrate the typical usage of each command above.

Command	**Effect**
`bin/gamesman solve quixo 0`	Solve variant 0 of Quixo (internally named “`quixo`”) and store the results in the default `data` path, assuming the current working directory is the root project directory and the `gamesman` binary is in `bin`.
`bin/gamesman -f -v solve quixo 0`	Force re-solve (and overwrite existing database of) variant 0 of Quixo with verbose output.
`bin/gamesman analyze quixo 0 --data-path=~/data`	Analyze variant 0 of Quixo, assuming the existing database is in `~/data`.

`bin/gamesman query -- mtttier 0 --x----o-`[6]	Get a UWAPI-compliant JSON response containing detailed position information on position formally encoded[7] as "`--x----o-`" in variant 0 of the tiered tic-tac-toe game (internally named "`mtttier`").
`bin/gamesman getstart mtttier 0`	Get a UWAPI-compliant JSON response containing information about the starting position for the tiered tic-tac-toe game.
`bin/gamesman --help`	Print help message on how to use the `gamesman` binary.

Table 5.1: Example GamesmanOne commands

The headless mode can be used in two different ways. Users can skip the interactive interface and solve or analyze a game directly if they already know the name and variant index of the target game. This is especially helpful for wrapping the solve/analyze command as part of another command. For example, the user can time the solve of the default variant (variant 0) of the tic-tac-toe game (internally named "`mttt`") using the following command:

```
/usr/bin/time bin/gamesman solve mttt 0
```

This behavior also allows us to submit a solving task to the Slurm scheduler of an HPC platform such as the Berkeley Savio Cluster using the srun command. The following example command submits a solving task to the Slurm scheduler, informing it to solve variant 0 of Quixo by spawning 6 processes with 28 CPUs assigned to each process:

```
srun -n 6 -c 28 --cpu_bind=cores bin/gamesman solve quixo 0
```

This enables us to solve Quixo with 6 MPI processes, assuming the gamesman binary was built with MPI. The details about building GamesmanOne are given in Section 5.2 of this chapter.

The headless mode of GamesmanOne is also used by the GamesmanOne Server, which is a simple Python HTTP server that can provide raw JSON responses for position queries implemented using the Flask framework.

The description of each component is given in the list below:

- **Entry Point (`main.c`), Gamesman Headless, and Gamesman Interactive**: these components together serve as the core of the system. GamesmanOne will run in different modes and perform different actions depending on the number of arguments provided when the binary is launched.
    - **Solver Manager**: a unified abstraction layer to invoke different solvers. Each solver must implement a standard set of required functions to qualify as a solver and be recognized by the solver manager. When a game is loaded, the core

[6] The double dash ("`--`") after the `query` command signifies the end of options. It is not optional in this case and in many other cases because the formal position string may contain dashes ('`-`') that confuses the command line argument parser.

[7] Definition for a formal position string is given in subsection 5.3.4.

system informs the Solver manager to check and load the solver specified by the game.

        - **Regular Solver**: solver for single-tier games.
        - **Tier Solver**: solver for tier games.
            - **DB Manager**: a unified abstraction layer to invoke different database implementations. Each database must implement a standard set of required functions to qualify as a database and be recognized by the DB manager. Solvers that require permanent storage of solver results in the format of databases will inform the Database manager to check and load the database implementation it needs.
                - **Bit-Perfect DB**: a database implementation that uses a minimal number of bits to represent the value and remoteness of each position.
                - **Naive DB**: a naive database implementation using integer arrays.
            - **Analysis Manager**: responsible for saving and loading analysis results of games.
    - **Game Manager**: responsible for loading games. GamesmanOne makes the assumption that no more than one game can be loaded at a time. To solve, analyze, or play multiple games at the same time, the user should invoke multiple instances of the `gamesman` binary.
        - **Tic-Tac-Toe**: the Tic-tac-toe game.
        - **Quixo**: the Quixo game.
        - **All Queens Chess**: the All Queens Chess game.
- **Generic Hash**: an optional helper module for game developers to easily manage position hashing for their games.
- **Server (Python)**: an HTTP server to generate JSON responses for position queries.

5.2 Building and Running GamesmanOne

The current implementation allows the user specify the following options while compiling GamesmanOne:

- Compiler flags including compiler optimization level, debugging symbols, and level for output of warning messages.
- OpenMP support.
- MPI support.

The user can use different configurations for different use cases. Some typical use cases are as follows:

1. Debugging on a single thread without OpenMP by turning off compiler optimizations and including debugging symbols and all warning messages. This includes
    a. running a debugger on the program, and
    b. playing the game through a command-line interface without solving the game.

2. Debugging with OpenMP by using the same configuration as in 1 but with OpenMP enabled.
3. Solving on a single thread without OpenMP by using the maximum compiler optimization level.
4. Solving with OpenMP support by using the same configuration as in 3 but with OpenMP enabled.
5. Solving with MPI enabled by manually specifying the compiler using `CC=mpicc`.
6. Playing the game with a database available after the game is solved. This can be achieved under any configuration so long as MPI is not enabled.

Users can find detailed instructions on how to configure and compile the project in the README.md file under the root project directory.

The most common use case is to compile GamesmanOne for solving games with OpenMP support. A bash script `gamesman_install.sh` for this purpose is provided at the users' convenience. Running `./gamesman_install.sh` under the root project directory automatically installs all dependencies and compiles GamesmanOne for use case 4.

GamesmanOne also has MPI support, which allows the user to parallelize the solving of their game using distributed memory machines as long as the solver of their choice supports it. Currently, the tier solver is the only solver that supports MPI. To configure and compile GamesmanOne with MPI support, the user should supply `CC=mpicc` to the `configure` script and then recompile their project by running `make clean` and `make`.

5.3 Implementing Games in GamesmanOne

The implementation of a new game in GamesmanOne can be divided into five steps:

1. Create a Game instance and choose a solver for the game based on game properties.
2. Implement the API for the chosen solver.
3. Implement the Gameplay API.
4. Implement the GamesCrafters Universal Web API (UWAPI). This step is optional and only required if the user wants to publish the game on GamesmanUni.
5. Add the game to the global list of games in `src/games/game_list.c`.

Instructions for each step are detailed in a subsection below.

5.3.1 Creating a New Game Instance

The process of adding a new game to the system is abstracted as implementing a new instance of the Game type and providing all the necessary information that defines a game.

The following code block defines the abstract Game type. In-line documentation for each field is provided in `src/core/types/game/game.h` but is omitted here for brevity.

```
typedef struct Game {
    char name[kGameNameLengthMax + 1];
```

```
        char formal_name[kGameFormalNameLengthMax + 1];
        const Solver *solver;
        const void *solver_api;
        const GameplayApi *gameplay_api;
        const Uwapi *uwapi;
        int (*Init)(void *aux);
        int (*Finalize)(void);
        const GameVariant *(*GetCurrentVariant)(void);
        int (*SetVariantOption)(int option, int selection);
    } Game;
```
As the name of each field suggests, all games should have a name for internally representation in GamesmanOne, a formal name that is presented to the user of the system, a solver that the game developer wish to use to solve the game, an implementation of the solver API that matches the solver chosen, an implementation of the Gameplay API which enables gameplay through the command line interface, and several other functions for setting up the game. The initialization, finalization[8], and variant selection functions are provided as function pointers that need to be implemented. In addition, game developers may implement the optional UWAPI if they want to make the game available through the web frontend provided by GamesmanUni.

The following code block shows the implementation of the tiered version of the Tic-tac-toe game as an example. The source code can be found in `src/games/mtttier/mtttier.c`.
```
    const Game kMtttier = {
        .name = "mtttier",
        .formal_name = "Tic-Tac-Tier",
        .solver = &kTierSolver,
        .solver_api = (const void *)&kSolverApi,
        .gameplay_api = (const GameplayApi *)&kMtttierGameplayApi,
        .uwapi = &kMtttierUwapi,
        .Init = &MtttierInit,
        .Finalize = &MtttierFinalize,
        .GetCurrentVariant = &MtttierGetCurrentVariant,
        .SetVariantOption = &MtttierSetVariantOption,
    };
```
The goal of implementing a game in GamesmanOne is to have the game solved using one of the existing solvers. Therefore, the next step in game implementation is to get familiar with the solvers available in the system and pick the one that best suits the game that we wish to solve. As of this writing (May 2024), two solvers are available in the system:

1. Tier Solver: The solver for tier games. It is capable of leveraging tier parallelism and supports multi-process solving using MPI.
2. Regular Solver: The solver for single-tier games. It uses the same algorithm for single-tier solving and abstracts away the tier concept from the game developer.

[8] Memory needs to be manually managed in C. A finalization step makes sure that all heap memory is properly deallocated.

To use a solver, the game must first include the header file in which the solver is declared. Here we choose the tier solver, and make the following include statement at the beginning of the source file:

```
#include "core/solvers/tier_solver/tier_solver.h"
```

5.3.2 Implementing a Solver API

The solvers in GamesmanOne are modularized and self-contained. Each one of them provides an application programming interface (API) for the game developer to implement. A solver API usually includes a standard set of functions that defines the game graph. The solver can then use the functions provided by the game to understand and solve the game.

The API of a solver is provided to the game developer as a collection of C function pointers. The following code block shows the API of the tier solver. In-line documentation for each function is provided in `tier_solver.h` but is omitted here for brevity.

```
typedef struct TierSolverApi {
    Tier (*GetInitialTier)(void);
    Position (*GetInitialPosition)(void);
    int64_t (*GetTierSize)(Tier tier);
    MoveArray (*GenerateMoves)(TierPosition tier_position);
    Value (*Primitive)(TierPosition tier_position);
    TierPosition (*DoMove)(TierPosition tier_position, Move move);
    bool (*IsLegalPosition)(TierPosition tier_position);
    Position (*GetCanonicalPosition)(TierPosition tier_position);
    int (*GetNumberOfCanonicalChildPositions)(TierPosition tier_position);
    TierPositionArray (*GetCanonicalChildPositions)(TierPosition tier_position);
    PositionArray (*GetCanonicalParentPositions)(TierPosition child, Tier parent_tier);
    Position (*GetPositionInSymmetricTier)(TierPosition tier_position, Tier symmetric);
    TierArray (*GetChildTiers)(Tier tier);
    Tier (*GetCanonicalTier)(Tier tier);
    int (*GetTierName)(char *name, Tier tier);
} TierSolverApi;
```

To implement a function, the game developer should first define a function in the game's source file that matches the function pointer signature, and then pass the address of that function to a solver API instance. The address of the solver API instance is then included as part of the Game instance to make the implementation visible to the solver.

5.3.3 Implementing the Gameplay API

Being able to play the game through the command line interface is important for testing the game and querying positions once the game is solved. The Gameplay API defines a set of functions that each game must implement to allow the system to generate an interactive play loop. Since the database querying process for single-tier games and tier games are slightly different, the Gameplay API is split into three parts as follows:

```
typedef struct GameplayApi {
        const GameplayApiCommon *common;    /**< Common API collection. */
        const GameplayApiRegular *regular;  /**< API for single-tier games. */
        const GameplayApiTier *tier;        /**< API for tier games. */
    } GameplayApi;
```
where `GameplayApiCommon` is defined as
```
    typedef struct GameplayApiCommon {
        Position (*GetInitialPosition)(void);
        int position_string_length_max;
        int move_string_length_max;
        int (*MoveToString)(Move move, char *buffer);
        bool (*IsValidMoveString)(ReadOnlyString move_string);
        Move (*StringToMove)(ReadOnlyString move_string);
    } GameplayApiCommon;
```
`GameplayApiRegular` is defined as
```
    typedef struct GameplayApiRegular {
        int (*PositionToString)(Position position, char *buffer);
        MoveArray (*GenerateMoves)(Position position);
        Position (*DoMove)(Position position, Move move);
        Value (*Primitive)(Position position);
    } GameplayApiRegular;
```
and `GameplayApiTier` is defined as
```
    typedef struct GameplayApiTier {
        Tier (*GetInitialTier)(void);
        int (*TierPositionToString)(TierPosition tier_position, char *buffer);
        MoveArray (*GenerateMoves)(TierPosition tier_position);
        TierPosition (*DoMove)(TierPosition tier_position, Move move);
        Value (*Primitive)(TierPosition tier_position);
    } GameplayApiTier;
```
Again, most in-line documentation is omitted here for brevity, but the game developer should read it in the header files and use it as guidelines for function implementation.

All games should implement `GameplayApiCommon`, which defines the set of gameplay functions that is common to both single-tier and tier games. Then, depending on the game type, the game developer should implement either `GameplayApiRegular` or `GameplayApiTier` and set the other API to `NULL`.

Some functions, such as `GetInitialPosition` and `GenerateMoves`, of the Gameplay API may also be found in the solver API for some solvers. In this case, the game developer can pass the addresses of those functions to both APIs. In fact, they should always do so to ensure the logic used during gameplay is consistent with the logic used to solve the game. The system is designed in this way to decouple the solvers from the gameplay logic and to allow solvers to provide unconventional interfaces. For example, a solver may directly ask the game to provide a function `GetChildPositions` that returns a list of child positions of any valid position in the game, instead of relying on first generating moves and then performing each possible move at the given position.

5.3.4 Implementing the GamesCrafters Universal Web API

This is an optional step that bridges the gap between the backend database querying system in GamesmanOne and the GamesmanUni web frontend. As earlier mentioned in Chapter 2, the connection between the two systems is established through a middleware separately maintained as the GamesCraftersUWAPI project. As the name suggests, it provides a standard API that all backend game-hosting servers (e.g., the server for GamesmanClassic) must implement to serve value and remoteness information in the format that GamesmanUni understands. The current standard uses the JSON format.

GamesmanOne has built-in JSON response formatting functionality and a Python server that is capable of handling HTTP GET requests from the UWAPI server. The game developer is only responsible for implementing the following structure that contains the functions needed for converting positions (or tier positions for tier games) and moves from the internal hash representations to the AutoGUI string representations:

```
typedef struct Uwapi {
    const UwapiRegular *regular; /**< API for regular (non-tier) games. */
    const UwapiTier *tier;       /**< API for tier games. */
} Uwapi;
```

where `UwapiRegular` is defined as

```
typedef struct UwapiRegular {
    MoveArray (*GenerateMoves)(Position position);
    Position (*DoMove)(Position position, Move move);
    Value (*Primitive)(Position position);
    bool (*IsLegalFormalPosition)(ReadOnlyString formal_position);
    Position (*FormalPositionToPosition)(ReadOnlyString formal_position);
    CString (*PositionToFormalPosition)(Position position);
    CString (*PositionToAutoGuiPosition)(Position position);
    CString (*MoveToFormalMove)(Position position, Move move);
    CString (*MoveToAutoGuiMove)(Position position, Move move);
    Position (*GetInitialPosition)(void);
    Position (*GetRandomLegalPosition)(void);
} UwapiRegular;
```

and `UwapiTier` is defined as

```
typedef struct UwapiTier {
    Tier (*GetInitialTier)(void);
    Position (*GetInitialPosition)(void);
    TierPosition (*GetRandomLegalTierPosition)(void);
    MoveArray (*GenerateMoves)(TierPosition tier_position);
    TierPosition (*DoMove)(TierPosition tier_position, Move move);
    Value (*Primitive)(TierPosition tier_position);
    bool (*IsLegalFormalPosition)(ReadOnlyString formal_position);
    TierPosition (*FormalPositionToTierPosition)(ReadOnlyString formal_position);
    CString (*TierPositionToFormalPosition)(TierPosition tier_position);
    CString (*TierPositionToAutoGuiPosition)(TierPosition tier_position);
    CString (*MoveToFormalMove)(TierPosition tier_position, Move move);
    CString (*MoveToAutoGuiMove)(TierPosition tier_position, Move move);
} UwapiTier;
```

Similar to the Gameplay API, the functions required for UWAPI are also separated for single-tier games and tier games. The game developer should implement exactly one of the two APIs in their game-specific `Uwapi` instance and set the other one to `NULL`.

We provide definitions to some of the terms in the above API function names:

- A **formal position string** is a string that unambiguously describes a position that is easy to read or construct by a human being. This string is used by GamesmanUni to query positions values from the web frontend. It also allows users to specify a custom starting position for their game by entering the formal position string of the position from which they wish to start playing. Therefore, it is important to use a format that facilitates typing. Note that there is no common standard for formal position strings as long as they are self-consistent. As an example, if chess were to be implemented in GamesmanOne with UWAPI support, the game developer can use the FEN notation [9] as the formal position string for the game.
- An **AutoGUI position string** is a position-describing string that follows the AutoGUI standard of GamesmanUni. As of the time of this report, the standard format of an AutoGUI position string is “`<turn>_<board><extra>`”, where turn is either ‘`1`’ or ‘`2`’ indicating whose turn it is at the current position, board is a one-dimensional representation of the game board, and extra contains optional extra information that facilitates the creation of image AutoGUI. A full explanation of the standard is outside the scope of this report. The reader is encouraged to read the documented source code of the GamesmanUni project [10].
- A **formal move string** is a string that unambiguously describes a move in a human-readable way. The GamesmanUni web frontend uses this string to record the move history of a match.
- An **AutoGUI move string** is the internal move representation used by GamesmanUni’s AutoGUI system. Under the current standard, two of the most commonly used AutoGUI move string formats are “`A_<char>_<center>_<soundChar>`” and “`M_<center1>_<center2>_<soundChar>`”. They correspond to moves with a destination but without a source and moves with a source and a destination, respectively. The `char` substring consists of a single character used to map piece character tokens in the board string to images. A `center` substring is a numerical string that indicates the source or destination of the move. The `soundChar` substring is optionally used to bind sound effects to the move. Again, the reader is encouraged to read the full documentation from the GamesmanUni project on the latest standard specifications.

Some functions such as `GetInitialTier` and `GenerateMoves` may also be found as part of the solver API, but are reproduced here for the same reason as mentioned in Subsection 5.3.3. Definitions to the same functions can again be reused by passing the same function pointers to them.

As an important note on security, the game developer should pay close attention to the implementation of the `UwapiRegular::IsLegalFormalPosition` (or the `UwapiTier::IsLegalFormalPosition` for tier games) function. This function serves as the security check for external formal position inputs that come from unsanitized user input. The minimum requirement for this function is that ill-formed formal position strings should not cause any of the other string conversion functions or position querying functions including `GenerateMoves` and `DoMove` to crash. For instance, consider an implementation of the Tic-tac-toe game where the formal position string is formatted as a flattened array of the 2D board. Then, this function should check if the input string size is of length 9 and only contains characters in `{'-', 'x', 'o'}`.

In addition, the developer may incorporate the logic from the `IsLegalPosition` function as discussed in Section 3.1 to filter out as many illegal positions as possible. Accurately filtering out as many illegal positions as possible can help the frontend generate error messages that are more informative. However, it is very difficult to implement a position validation function that filters out all illegal positions. To see why this is the case, consider the game of Tic-tac-toe, which is one of the games with the simplest set of rules. To filter out all positions that are illegal, one needs to verify that:

1. The number of X's on the board is either equal to the number of O's, or one greater than the number of O's when no player has 3 pieces in a row;
2. The number of X's on the board is equal to the number of O's if O has at least one 3-in-a-row;
3. The number of X's on the board is one greater than the number of O's if X has at least one 3-in-a-row; and
4. There cannot be two types of 3-in-a-rows on the board.

For games with more complex rule sets, it is almost impossible to enumerate all the requirements for a position to be legal. Recall from Chapter 3 that we allow arbitrary value and remoteness assignments to illegal positions in our database. Therefore, the response one gets from querying an illegal position is currently dependent on the implementation of the `IsLegalFormalPosition` function.

5.3.5 Adding a Game to the List of All Games

The last step in adding a new game to GamesmanOne is to include the game in the global list of all implemented games. The list is defined in `src/games/game_list.c` as

```
const Game *const kAllGames[] = {...};
```

The file contains instructions on how to include the header file for the new game and how to add the game to the list by providing a pointer to the game object.

Whenever new source files are added, the project environment needs to be updated and the binaries must be recompiled. The instructions on how to do so can be found in the README at the root directory of the GamesmanOne repository.

5.4 From GamesmanClassic to GamesmanOne

GamesmanOne drew significant inspiration from GamesmanClassic. Initially conceived as an enhancement to GamesmanClassic, it evolved into a standalone project due to the extensive modifications required across all games and solvers linked to previous design decisions that GamesmanOne aimed to supersede. Designed with future maintenance and upgrades in mind, GamesmanOne incorporates a modular architecture with thoroughly documented module header files. While preserving many of GamesmanClassic's features, we maintained a similar programming interface to facilitate a smooth transition for users already familiar with GamesmanClassic. This section details the modifications and considerations implemented in GamesmanOne relative to GamesmanClassic.

5.4.1 Adjustments in System Structure

Both GamesmanClassic and GamesmanOne consist of two components: the core system and the games. In GamesmanClassic, each game module implements a standardized set of global variables and API functions, which the core system utilizes to select the appropriate solver automatically. Each game module is then compiled into its own binary, linked with the same core system. This design ensures uniformity across APIs and eliminates the need for game developers to manually select solvers. However, a significant drawback is that modifications to the core system can disrupt existing games that depend on older versions. For instance, a redesign of the `GenerateMoves` function might require the addition of a new API function (e.g., `GenerateMovesAlt`), or require rewriting the old `GenerateMove` function definitions across over 100 games.

GamesmanOne addresses these challenges by modularizing solvers and requiring each solver to declare its API function requirements within its module. This approach facilitates the future replacement or addition of solvers simply by integrating a new solver module. Consequently, game developers must specify which solver they intend to use during game development. This is to ensure that they implement the correct set of API functions. Although this demands more upfront clarity from developers about the solvers they use, we view this as a beneficial tradeoff. For example, if a game employs a CUDA-based GPU solver, the API function signatures will indicate that the functions are called on the GPU device-side, prompting developers to minimize branching in their code. Even under the old system in GamesmanClassic, developers needed to be aware of the required API functions and possess some understanding of their chosen solver such as defining optional API functions for tier management. Given the absence of fully automatic solver selection at this stage, we believe these design choices are well-founded.

5.4.2 Data Structure Optimizations

The most computationally demanding step of game solving is the traversal of the game graph. The functions responsible for generating parent and child positions are invoked most frequently. Therefore, the efficiency of these functions is critical to solver performance.

In GamesmanClassic's shared solver API, a linked list is used to store the moves returned by the `GenerateMoves` function, defined in `src/core/gamesman.h` as follows:

```
MOVELIST* GenerateMoves (POSITION position);
```

These moves are subsequently used to generate child positions via the `DoMove` function. Likewise, the retrograde tier solver in GamesmanClassic employs a linked list to manage "undo moves[9]," which are then used to generate parent positions. The function pointer's declaration is located in `src/core/globals.h`:

```
extern UNDOMOVELIST* (*gGenerateUndoMovesToTierFunPtr)(POSITION, TIER);
```

A key drawback of using linked lists is the requirement to allocate heap memory for each new entry, potentially leading to memory fragmentation and decreased cache locality. As there is no need to insert in the middle of these move lists, using arrays would be more efficient.

Two design alternatives are proposed. The first is employing a dynamic array to store the moves. This method is both flexible and straightforward, and is currently implemented by the tier solver in GamesmanOne (refer to subsection 5.3.2). The second approach, suggested by Professor Dan Garcia, involves delegating the responsibility of space allocation to the function caller. The caller would then provide a buffer sufficiently large to accommodate all possible moves through a pointer to the buffer's start. The modified API function prototype would be:

```
int (*GenerateMoves)(Move *moves, TierPosition tier_position);
```

where the first parameter is an output buffer supplied by the caller, and the output size (or an error code) can be returned as an integer value. This method could be more efficient if the output buffers are small enough to remain on stack memory. This is a reasonable assumption for most games since the number of available moves at any position generally does not exceed a few hundred, equating to no more than 8 KiB of stack memory if each move is represented as a 64-bit integer. This approach significantly reduces heap memory allocations and deallocations throughout the solving process. However, if a game position exceeds the maximum number of specified moves, the solver would require reprogramming.

We conducted timing tests on both solutions using an Intel Core i9-12900K CPU with 16 cores and 24 threads. In our experiment, we solved 4x4 Quixo using the two different solutions and recorded the completion times using all 24 threads. The first solution completed in 44.3 seconds, while the second finished in 43.8 seconds, marking a performance improvement of 1.1%. Given the marginal gain, we opted to retain the first implementation. Nonetheless, the second solution could be advantageous for GPU solving in the future, where memory allocation is typically more costly.

[9] Moves that can lead to the current position. The tier solver in GamesmanOne does not utilize this concept, instead requiring game modules to supply a function that directly yields parent positions. This approach is adopted because it is generally more efficient and undo moves are not employed elsewhere in the system.

5.5 Future Work

5.5.1 Adding Endgame Solving to the Tier Solver

The existing tier solver implementation can be slightly modified to enable endgame solving. Since endgame positions are usually defined as positions with a smaller set of pieces on the board, one way to make the modification is to solve only those tiers that fit the endgame definition. This idea is proven possible in the creation of a 6-piece endgame tablebase for Chinese chess in a separate project discussed in Chapter 6.

5.5.2 Implementing Specialized Solvers

The main focus of this report is on solving loopy games. The loopy game solving algorithm is the most generalized algorithm for solving two-player abstract strategy games and *can be applied to loop-free games as well*. However, there exist algorithms that solve loop-free games in a much more efficient way. For example, if tiers in a loop-free game are defined in such a way that every move in the game leads to a child position in one of the child tiers, then we can simplify the algorithm by solving all positions in a given tier in one pass without generating parent positions in the same tier. This algorithm was implemented as part of the Retrograde Tier Solver in GamesmanClassic but has not been adapted to GamesmanOne.

In addition, the solution of games such as “10 to 0 by 1 or 2”[10] are simple enough to be encoded as a function that describes the best strategy. These games can be categorized as “games with closed-form solutions” that require no database storage for the solver result.

GamesmanOne is designed to allow developers to implement any number of new solvers in the system. Hence an open-ended future project could be to implement specialized solvers for games with special properties.

5.5.3 Implementing Specialized Analyzers

The current design requires each solver to provide its own analyzer. The reason for this design is that different solvers may take different approaches to analyze the game. For example, the current implementation of the Tier Solver analyzes the game by traversing the game graph from the initial position after the game is solved. It is designed this way to make sure that illegal positions introduced by the imperfect `IsLegalPosition` function (see Section 3.1) are not included in the value-remoteness statistics. However, suppose another solver performs a full traversal of the game graph before the game is solved (called a discovery phase), it is possible, and perhaps more efficient, for the solver to analyze the game while the game is being solved. Then, the solver may simply return the information collected in the solving phase when the user passes in the “analyze” command.

[10] 10 to 0 by 1 or 2 is a simple combinatorial game in which two players take turns taking either 1 or 2 items from a pile of 10. The player who takes the last item wins. Under the perfect strategy, the first player can win by taking 1 item first and then leaving a multiple of 3 items for the second player each turn.

In addition to the examples above, games with closed-form solutions may alternatively calculate the number of positions at each remoteness level without having to traverse the full game graph. For combinatorial games, a specialized analyzer might be capable of providing more information about positions in the game then a simple value-remoteness count.

5.5.4 Exploring Puzzles

Puzzles are single-player games where the objective is to reach a specific set of terminal positions known as the solved states. Prior research within the GamesCrafters group has explored systems for both solving and presenting puzzles. Notably, GamesmanPuzzles, created by Anthony Ling in 2021 [11], stands out as the sole system capable of solving and presenting puzzles via the GamesmanUni web interface at the time of this document. This Python-based framework facilitates the implementation of puzzles, and provides single-threaded puzzle solvers. However, due to the inherent limitations of the Python language, the system can only handle puzzles with no more than a few million positions. Additionally, ongoing research in puzzle solving is being conducted under the GamesmanNova project, led by Max Fierro.

Puzzle solvers differ significantly from game solvers in that they do not operate within the conventional framework of winning and losing. Instead, they categorize puzzles as either solvable in a certain number of moves or unsolvable. Additionally, because puzzles are inherently single-player, the concept of alternating turns, an assumption made throughout this report, does not apply. It is feasible to incorporate puzzle solvers into the existing GamesmanOne framework to allow fast parallel solving of puzzles. However, this would require a careful revision of many existing modules within the system to address the issues outlined above.

Chapter 6: Solving Chinese Chess Endgames

Chinese chess, also known as Xiangqi, has a rich history spanning over a millennium, making it one of the oldest board games still widely played today. The game reflects Chinese military strategy and philosophy, with each piece representing different units in an army, such as generals, advisors, elephants, horses, chariots, cannons, and soldiers. The game is played on a board that symbolizes a battlefield divided by a river, which adds a unique strategic element to the game.

The rules of Chinese chess are surprisingly similar to those of Western chess. However, Chinese chess is played on the 10 by 9 intersections of a 9 by 8 grid, making the state space of the game even larger than chess [12]. In this chapter, we discuss an approach that allowed us to solve all 6-piece endgames of Chinese chess.

All 6-piece endgames of Chinese chess were solved as part of the GamesmanXiangqi project[11]. A Chinese chess interactive gameplay interface is also available through GamesmanUni[12].

6.1 Rules

6.1.1 Game Board

The game is played on a board as shown in Figure 6.1. All pieces are placed on the intersections of the squares, forming a 10 by 9 grid. Since the pieces are always placed on the intersections, all mentions of m-by-n grids later in this chapter refer to grids of intersections.

On either side of the board, there is a 3 by 3 “palace” signified by the two diagonal lines for each player. The palace restricts the move of the kings (generals) and the advisors. A river spans horizontally across the middle of the board. The movements of two types of pieces are restricted by the river: a. the bishops (elephants) cannot cross the river, and b. the pawns (soldiers) cannot make horizontal moves before they cross the river. There are also 14 alignment marks on the board that signifies the initial positions of the pawns and cannons, but have no additional meanings once the game starts.

[11] https://github.com/GamesCrafters/GamesmanXiangqi.
[12] https://nyc.cs.berkeley.edu/uni/games/chinesechess/variants/regular.

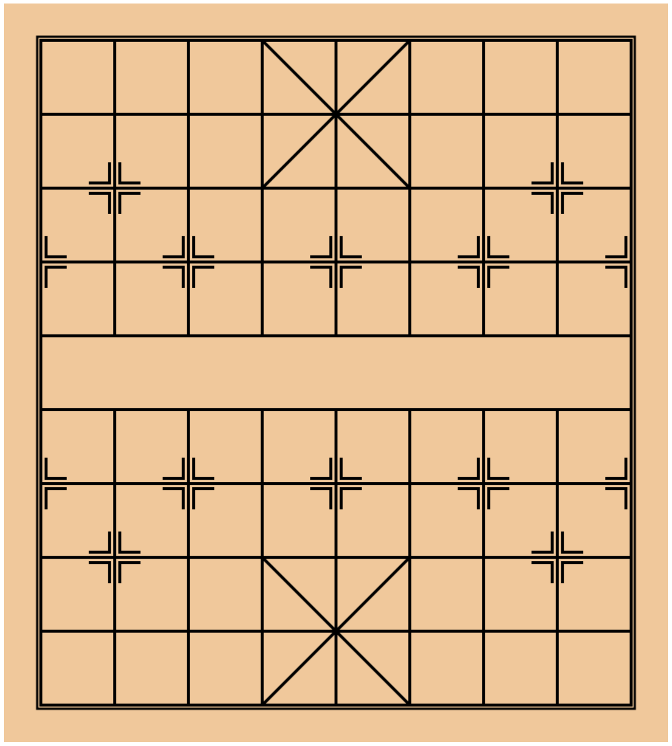

Figure 6.1: Chinese chess board

6.1.2 Pieces

The rules and pieces of Chinese chess share a lot of similarities with those of Western chess. Since there is no official translation for the names of the pieces, we prefer the use of terms that appear in chess when similar rules are available. They are also consistent with the naming conventions we used in the endgame tablebases. For each type of piece, we also include its alternative translations in parenthesis, and the single-character representations for both colors inside the square brackets.

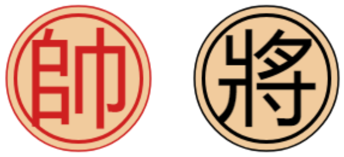

Figure 6.2: Chinese chess red and black kings (generals)

Kings (or generals, Red: 帥 [K]; Black: 將 [k]): the most important piece of each player. The goal of the game is to checkmate or stalemate the opponent's king. The kings may move one step horizontally or vertically inside the palace but may not move diagonally. The rule also forbids any moves that place play-to-move's king under immediate attack of any opponent pieces. Furthermore, the two kings are not allowed to be in the same column without any pieces in between. Therefore, any moves that make the two kings "see" each other are also forbidden.

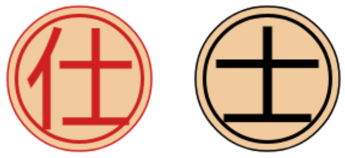

Figure 6.3: Chinese chess red and black advisors

Advisors (Red: 仕 [A]; Black: 士 [a]): each player owns two of these defensive pieces. The advisors may move one step along one of the four diagonal lines inside the palace. As a result, the advisors also cannot leave the palace.

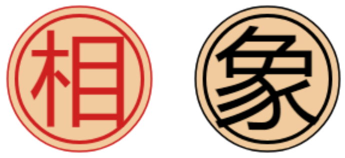

Figure 6.4: Chinese chess red and black bishops (elephants)

Bishops (or elephants, Red: 相 [B]; Black: 象 [b]): each player owns two of these defensive pieces. The bishops can move two spaces diagonally in a 3x3 grid. However, they are not allowed to jump over any pieces. If there is any piece in the next diagonal space of the bishop, the bishop cannot move in that direction. In addition, the bishops are not allowed to cross the river. As a result, each bishop can only ever appear in 7 different spaces on one side of the river.

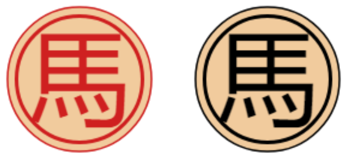

Figure 6.5: Chinese chess red and black knights (horses)

Knights (or horses, Red: 馬 [N][13], Black: 馬 [n]): each player owns two of these attack pieces. The knights may move one space horizontally or vertically and another space diagonally to either forward direction. The knights' moves are similar to those from Western chess but with an additional rule that forbids them to jump over a piece if they are blocked in the horizontal or vertical direction.

[13] We use the character "N" here to distinguish knights from the kings.

Figure 6.6: Chinese chess red and black cannons

Cannons (Red: 炮 [C]; Black: 砲 [c]): each player owns two of these attack pieces. The cannons have two types of moves: sliding and jump-capturing. By sliding, a cannon may move any number of spaces horizontally or vertically without capturing. In jump-capturing, a cannon must first jump over a piece and land on an opponent's piece and capture it. The piece jumped over may belong to either side. The cannons are not allowed to capture with sliding and they cannot jump without capturing.

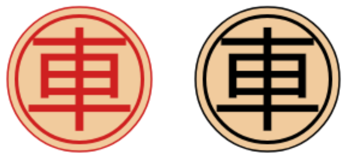

Figure 6.7: Chinese chess red and black rooks (chariots)

Rooks (or chariots, Red: 車 [R], Black: 車 [r]): each player owns two of these attack pieces. The rooks may move any number of spaces horizontally or vertically, and may capture the piece they encounter along the way if it's an opponent's piece. Their moves are exactly the same as the rooks in Western chess, except they move along the grid lines like all other pieces in Chinese chess.

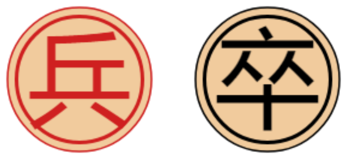

Figure 6.8: Chinese chess red and black pawns (soldiers)

Pawns (or Soldiers, Red: 兵 [P]; Black: 卒 [p]): each player owns five of these attack pieces. The pawns may only move one space forward horizontally in each turn before they cross the river. After crossing the river, they may also move one space horizontally in either direction. However, they are never allowed to move backwards. Unlike in Western chess, pawns in Chinese chess are never promoted, and they can only move horizontally once they reach the last row on the opponent's side.

6.1.3 Goal of the Game

The game starts at the position as shown in Figure 6.9 with red being the first player to move. The goal of the game is to checkmate or stalemate the opponent's king, and the first player to do so wins the game immediately.

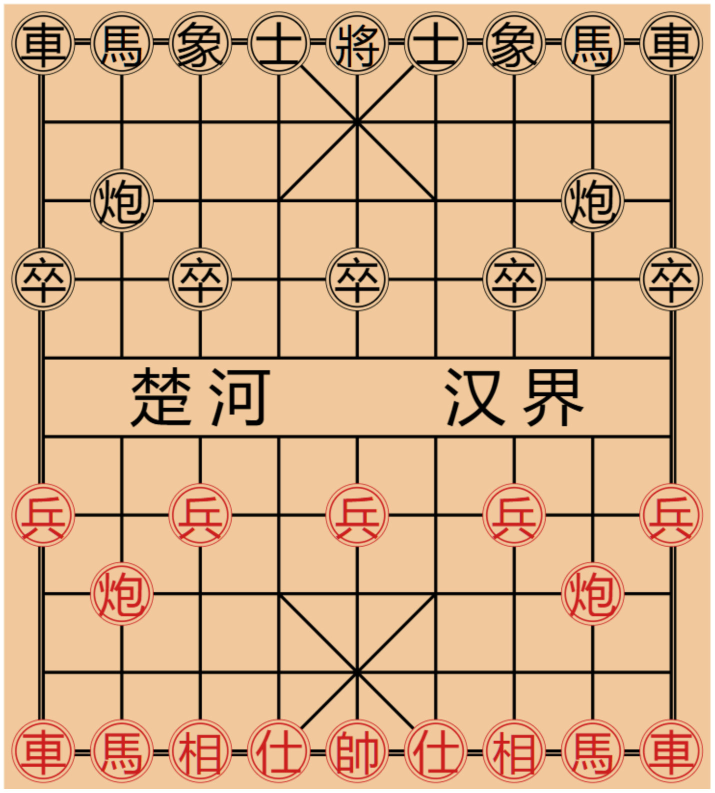

Figure 6.9: Initial position of Chinese chess

A checkmate position is defined as a position at which the player-to-move's king is under direct attack of at least one opponent's piece and there are no available moves that can help the king escape. Hence, a checkmate position is always a primitive losing position for the player whose turn it is.

A stalemate position is defined as a position at which the player-to-move's king is not under direct attack of any opponent's pieces, but all moves that are otherwise legal will place the king under attack. Note that unlike the rules of Western chess in which stalemates are defined as tie games, the rules of Chinese chess defines stalemates also as losing for the player who runs out of moves. Therefore, all stalemate positions are also primitive losing positions.

Note that the game ends as soon as the game reaches a checkmate or stalemate position. Therefore, no valid moves can lead to the capturing of kings. Additionally, players are not allowed to capture their own pieces, move pieces off the board, or skip their turn.

6.1.4 Move Restrictions

The rules outlined in sections 6.1.1 to 6.1.3 are universal across all variants of Chinese chess rulesets. These sections collectively define what we refer to as **the simplified ruleset of Chinese chess**. In formal Chinese chess competitions, a more complex ruleset is typically used. This advanced ruleset includes additional regulations designed to prevent perpetual checks and/or chasing of unprotected pieces. Moreover, a match is declared a draw if neither player captures any pieces within 60 moves. Various competitions may implement different rulesets, with many held in Mainland China adopting the most stringent version. This comprehensive ruleset includes prohibitions on tactics such as "perpetual threatening", where a **threatening move** is defined as one that could result in checkmate if the player were allowed to move again immediately. For the purposes of this project, we have chosen to use only the simplified ruleset for clarity and ease of implementation.

6.2 Tier Definition

Chinese chess has two types of irreversible moves: capturing and forward pawn moves. This is evident from the fact that captured pieces are never placed back onto the board, and pawns can never make a backward move.

Capturing moves affect the number of remaining pieces on the board. Since the number of each type of piece is fixed at the beginning of the game, one valid tier definition is to have each tier include all positions that share the same set of pieces on the board. For example, `{K, R, k}` can be defined as a tier that contains all positions with a red king (represented by capital K), a red rook (represented as capital letter R), and a black king (represented by lowercase `k`).

Forward pawn moves affect the number of pawns on each row. Another valid tier definition is to have each tier include all positions that have the same pawn configuration. We defined the indices of the pawn rows as follows: for red pawns, the rows are indexed from `0`, which corresponds to the bottom row of black’s side, to `6`, which corresponds to the initial row where red pawns are placed at the beginning of the game. For both colors, three additional rows are not labeled because the pawns are not allowed to move backwards and therefore can never appear on those rows. The definition for black pawn rows are defined in a similar way. The definitions are illustrated in Figure 6.10.

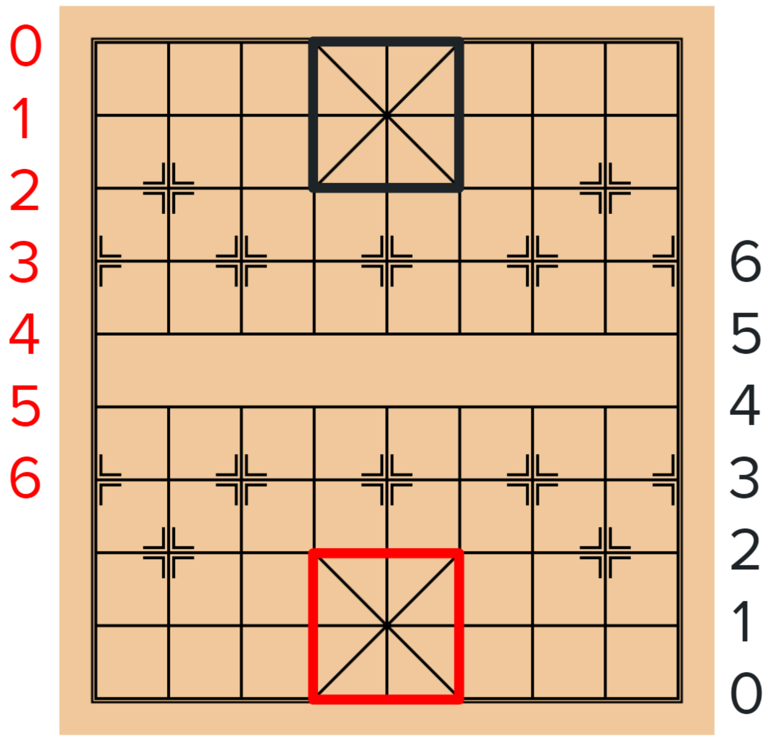

Figure 6.10: Chinese chess palaces and pawn row definitions

Then, an example tier using the second tier definition would be "`{4,2,2}_{5,3}`", with the first set of digits representing the indices of the rows where the red pawns are and the second set of digits digits representing the rows on which the black pawns are placed. Hence, the tier "`{4,2,2}_{5,3}`" includes all positions with 1 red pawn on row 4, 2 red pawns on row 2, one black pawn on row 5, and one black pawn on row 3. Note that all row indices here refer to pawn rows for either red or black.

Both tier definitions above are valid. However, we can combine the two definitions to get even finer granularity of tier parallelism. We define a **tier of Chinese chess** to be the set of all positions that has the same set of pieces on the board, supplemented by two sets of row indices for red and black pawns.

To show that this is a valid tier definition, one needs to show that (a) every position in the game is contained by one and only one tier, and (b) the resulting tier graph is a directed acyclic graph.

As a proof of part (a), notice that the set of remaining pieces and the pawns of both colors must fit the definition of one of the tiers defined. Suppose we pick any position in any defined tier, it must be a valid position because all tiers are valid by the game rules. Furthermore, tiers do not overlap because any two positions having different sets of pieces or the same set of pieces but different rows for the pawns are guaranteed to be different.

As a proof of part (b), notice that moves from one tier to another are either capturing moves or forward pawn moves, which are irreversible according to the game's rules. Suppose there exists a cycle $T_A \to T_B \to T_C \to \cdots \to T_A$ in the tier graph. Recall from Chapter 2 that T_A cannot have an edge to itself. Therefore, the sequence of moves required to traverse this cycle to and from T_A would necessarily have to be reversible, contradicting the tier definition.

We can uniquely represent each tier using a variable-length string consisting of 3 parts:

1. A substring of length 12 consisting of digits that represent the number of remaining pieces of each type. We represent it as "`{A}{a}{B}{b}{P}{p}{N}{n}{C}{c}{R}{r}`", where each letter is a placeholder for a single numeric character. Since the remaining number of each type of piece is no more than 5, we can safely use one character to represent each number. Note that we are not including the count for kings because they cannot be legally captured.
2. A variable-length substring consisting of the row indices for red pawns sorted in descending order. We represent it as `{red_pawn_rows}`, which consists of at least 0 and at most 5 numerical characters ranging from 0 to 6. The string is empty if no red pawns remain on the board.
3. A variable-length substring consisting of the row indices for black pawns sorted in descending order. We represent it as {black_pawn_rows}, which consists of at least 0 and at most 5 numerical characters ranging from 0 to 6. The string is empty if no black pawns remain on the board.

In addition, we must also exclude the tiers which has all 10 pawns on the same row. Also note that for parts 2 and 3, the descending order of pawn rows must be enforced to avoid giving two equivalent definitions to the same tier by rearranging the pawn rows.

Following this definition, we can uniquely represent each tier using the following string format:

```
{A}{a}{B}{b}{P}{p}{N}{n}{C}{c}{R}{r}_{red_pawn_rows}_{black_pawn_rows}
```

For example,

- the initial position of the game is in tier "`222255222222_66666_66666`", which is also by definition the initial tier of the game;
- any position with only two kings on the board is in tier "`000000000000__`";
- any position with two kings, two red pawns on the 3rd and the 4th red pawn row, and a black rook is in tier "`000010000001_43_`".

Note that the custom solver in the GamesmanXiangqi project did not use tier hashes. Instead, the tiers are directly represented using the strings above.

6.3 Hashing

In this section, we introduce the hashing algorithm that was used to calculate the hash values for Chinese chess endgame positions. Implementation-wise, the hash values are represented as 64-bit unsigned integers. Note that an assumption made by the algorithm is that no integer overflow can happen for all positions in the given tier. This is guaranteed by a tier-enumeration algorithm that only lists tiers that contain no more than 6 pieces.

6.3.1 Hashing Overview

To come up with a position hash definition for each tier, we can apply the idea of a combinatorially optimal hash as we discussed in Chapter 2 by rearranging the set of pieces remaining on the board. However, a naive implementation that allows all types of pieces to be rearranged on all 10x9 intersections may result in an inefficient use of memory if we are solving the game using a hash-indexed in-memory database. This is because many pieces can only reach a certain number of locations on the board due to the restrictions on their moves. For example, the kings and the advisors can never be outside of the palace, and a bishop can only appear in one of the 7 intersections on its side of the river. As a result, many hash values are mapped to invalid positions under this position hash definition.

We make the following observations:

1. The two kings are always on the board.
2. The kings are confined to their palaces and can only reach 9 intersections on the board.
3. The advisors are confined to the palaces and can only reach 5 intersections on the board.
4. The bishops cannot cross the river and can only reach 7 intersections.
5. The pawns cannot go backwards, which means that they can never appear in any of the three rows behind their initial row.
6. The pawns cannot move horizontally before crossing the river, and therefore cannot reach 8 additional intersections.
7. Knights, cannons, and rooks can reach any intersection on the board.

A combinatorially optimal hash for Chinese chess would therefore take into account all observations above and rearrange each type of pieces only within the set of locations they can reach.

We hence define the 16-step hash for any position within a tier as follows:

1. Hash the red king and red advisors.
2. Hash the black king and black advisors.
3. Hash the red bishops.
4. Hash the black bishops.
5. Hash the pawns on row 0.
6. Hash the pawns on row 1.
7. Hash the pawns on row 2.
8. Hash the pawns on row 3.
9. Hash the pawns on row 4.
10. Hash the pawns on row 5
11. Hash the pawns on row 6.
12. Hash the pawns on row 7.
13. Hash the pawns on row 8.
14. Hash the pawns on row 9.
15. Hash all remaining pieces.
16. Append a turn bit as the least significant bit of the hash value.

Note that the row indices here do not refer to pawn rows and are labeled for the entire board from 0 to 9 starting from the bottom-most row of black's side (see Figure 6.11 for an illustration of this definition). Without loss of generality, we also define the board to have the black pieces occupy the top half of the board and red pieces occupy the bottom half so that we follow a consistent execution sequence for all hashes. A description of each hashing step is detailed in the next subsection.

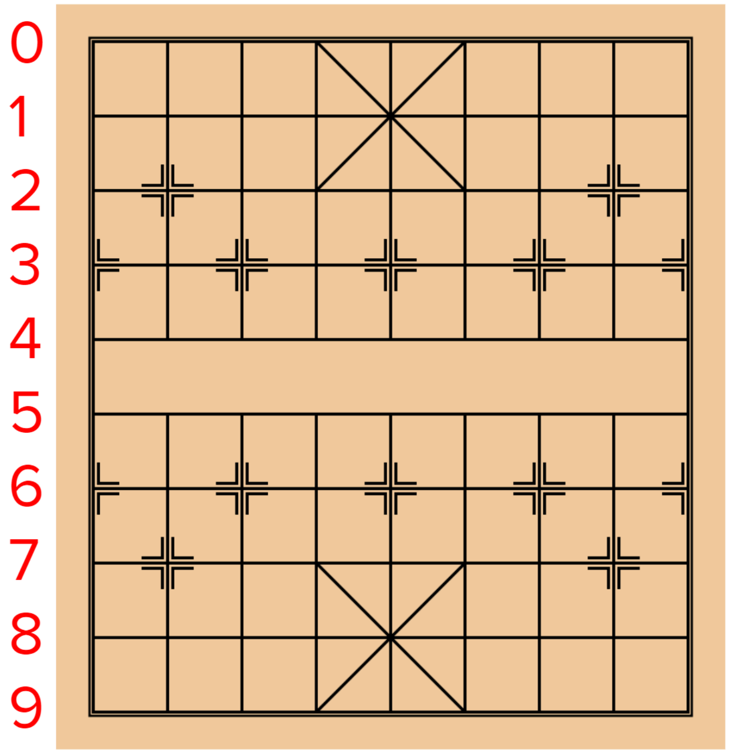

Figure 6.11: Chinese chess row definitions

6.3.2 Hashing Steps

In this subsection we introduce the hashing of each step in the above algorithm either by counting the number of rearrangements N for each possible piece configuration, or deferring the counting to the Generic Hash algorithm if the calculation is straightforward. The actual hashing algorithm also involves mapping each rearrangement of the pieces on the board to an integer between 0 and N-1. For brevity, we omit the description of this process. A high-level description of the algorithm can be found in Section 7.3 of the next Chapter.

Steps 1-2: Kings and Advisors

There are three possible combinations of the numbers of remaining kings and advisors:

1. One king + zero advisor: 9 possible rearrangements by moving the king inside the palace.
2. One king + one advisor: 40 possible rearrangements in total consisting of the following two possible cases:

    a. King does not occupy any of the 5 advisor intersections (illustrated by Figure 6.12): $4 \times 5 = 20$.
    b. King occupies one of the advisor intersections: $5 \times 4 = 20$.
3. One king + two advisors: 70 possible rearrangements in total consisting of the following three possible cases:
    a. King does not occupy any of the 5 advisor intersections: $4 \times nCr(5, 2) = 40$.
    b. King occupies one of the advisor intersections: $5 \times nCr(4, 2) = 30$

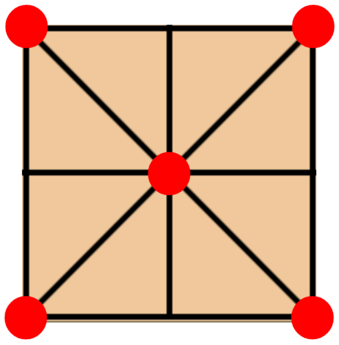
Figure 6.12: Chinese chess palace with possible advisor locations highlighted

Steps 3-4: Bishops
There are three possible numbers of remaining bishops on the board:
1. Zero bishop: 1 possible rearrangement with no pieces rearranged.
2. One bishop: 7 possible rearrangements of bishops: $nCr(7, 1) = 7$.
3. Two bishops: 21 possible rearrangements of bishops: $nCr(7, 2) = 21$.

Figure 6.13 shows the 7 reachable intersections of a red bishop.

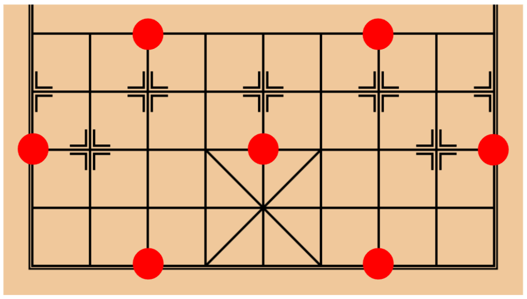
Figure 6.13: Chinese chess half board with possible bishop locations highlighted

Steps 5-7: Red pawns on rows 0-2
We know from the rules that only red pawns can ever reach these rows. Therefore, the hash is the rearrangement of red pawns over blank spaces.

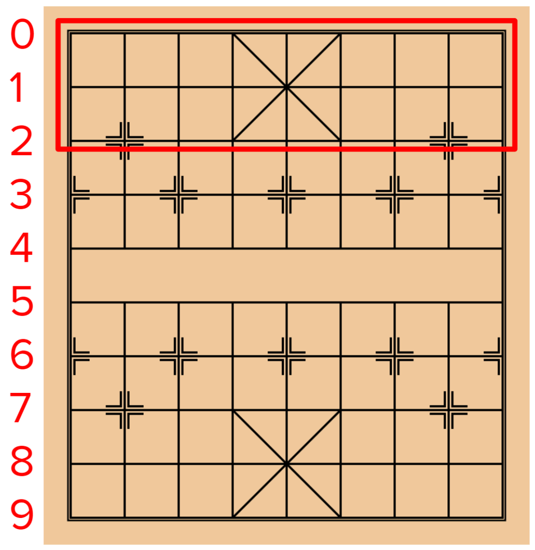

Figure 6.14: Rows hashed in steps 5-7

Steps 8-9: Red and black pawns on rows 3-4
We know from the rules that a black pawn on these two rows can only reach one of the 10 intersections marked with a black dot in Figure 6.15. Since the black pawns are more restricted, we locate them first in the rearrangement process. We then rearrange the red pawns over the remainder of the blank spaces.

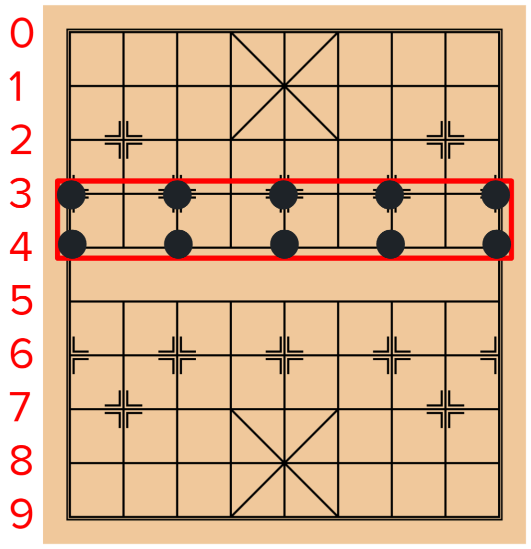

Figure 6.15: Rows hashed in steps 8-9 with possible black pawn locations highlighted

Steps 10-11: Red and black pawns on rows 5-6
We know from the rules that a red pawn on these two rows can only reach one of the 10 intersections marked with a red dot in Figure 6.16. Since the red pawns are more restricted, we locate them first in the rearrangement process. We then rearrange the black pawns over the remainder of the blank spaces.

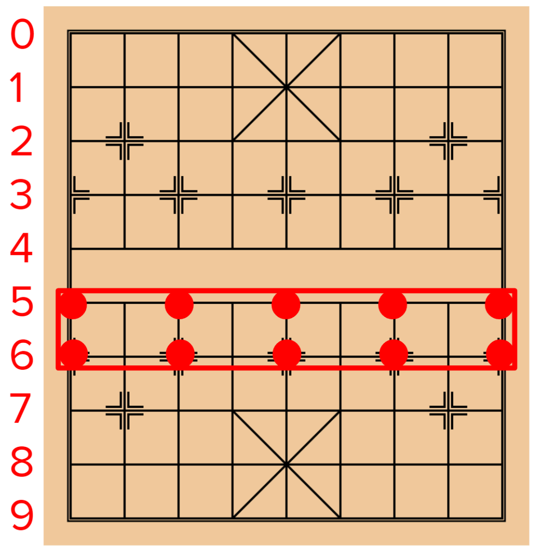

Figure 6.16: Rows hashed in steps 10-11 with possible red pawn locations highlighted

Steps 12-14: Black pawns on rows 7-9
We know from the rules that only red pawns can ever reach these rows. Therefore, the hash is the rearrangement of black pawns over blank spaces.

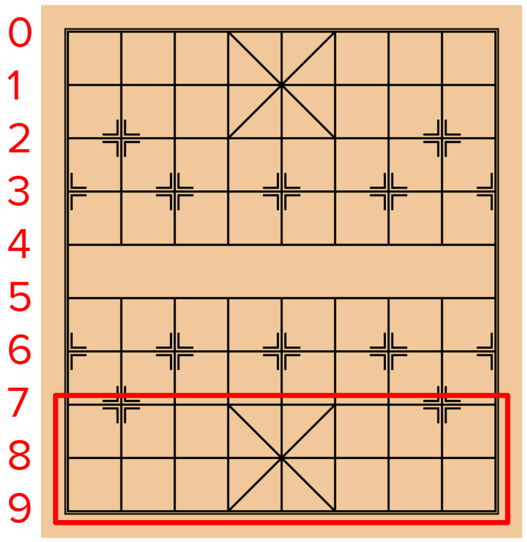

Figure 6.17: Rows hashed in steps 12-14

Step 15: Knights, cannons, and rooks
We first count the number of blank spaces left on the board by subtracting the pieces that we have placed so far in all previous steps. We then rearrange the remaining pieces over these blank spaces to get the hash for this step.

Step 16: The turn bit

Without loss of generality, we may define red's turn to be represented by binary value 0 and black's turn to be represented by 1. Then, we return for this step 0 if the current position is red's turn, or 1 if it is black's turn.

6.3.3 Combining Hash Values From Each Step

Since we need a single integral hash value for each position, we must combine the hash values we get from each step in the previous subsection. We first introduce a way of combining multiple hash values given their upper bounds. Here, we define hash upper bounds as the number of unique hash values defined. For example, the upper bound of the hash for step 1 is 9 if there is only one king and no advisors in the palace.

Suppose we have two hash values $h_1 = 3$ and $h_2 = 2$, where the hash upper bound of h_1 is $N_1 = 6$, and the upper bound of h_2 is $N_2 = 5$. The total number of combined hash values is $N_1 \times N_2 = 30$. This suggests that we can label the combined hash values from 0 to 29 in the combinatorially optimal way. Furthermore, we can take the cartesian product of the two sets of hash values and create a grid of dimensions N_1 by N_2, and use one hash as row index and the other as column index to find the value for the combined hash. Figure 6.18 illustrates this process using the above example.

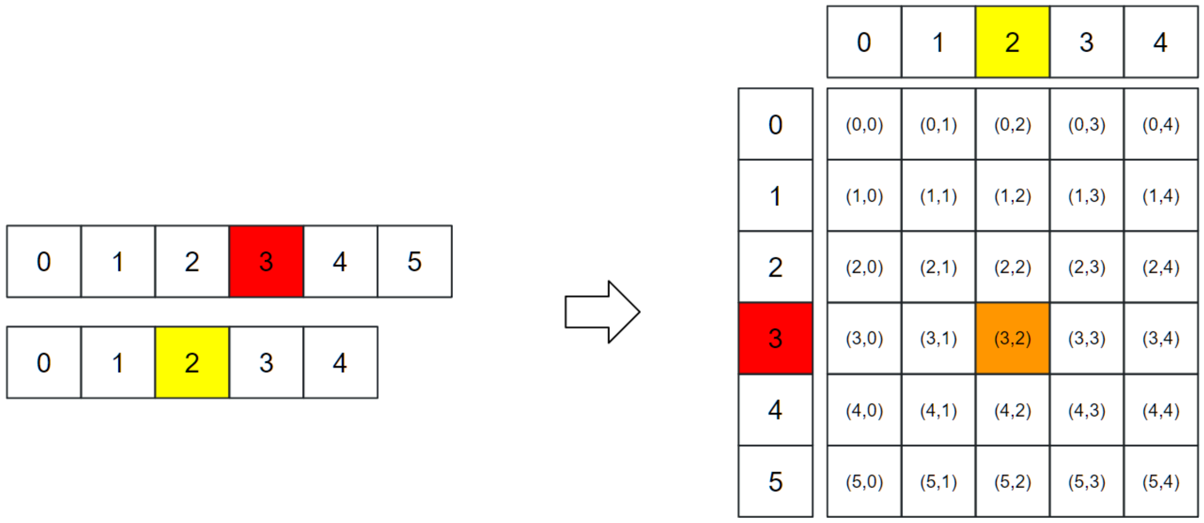

Figure 6.18: Combining two hash values

The combined hash value can be calculated as $h_1 \times N_2 + h_2 = 17$.

Now, suppose we want to combine more than two hash values. In this case, we can first perform the procedure above for an arbitrary pair of hash values, and then apply the same algorithm again to bring in yet another hash value. For example, suppose we want to combine the value 17 we got in the above example with another hash $h_3 = 4$ with upper bound $N_3 = 7$.

To do that, we simply flatten the first 2D matrix into a 1D array of length 30 and apply the same algorithm as shown in Figure 6.19.

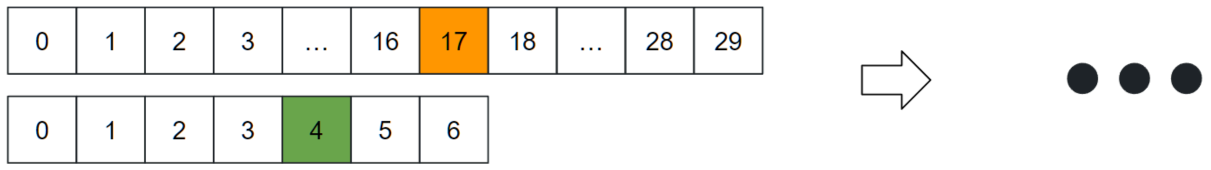

Figure 6.19: Combining the result of two hash values with a third one

Using the method above, we can combine the 16 hash values from each step into a single hash value. Given the hash value h_1 and upper bound N_i for each step i, we have the following formula for the combined hash H of the position:

$$H = ((((0 \cdot N_1 + h_1) \cdot N_2 + h_2) \cdot N_3 + h_3) \cdots) \cdot N_{16} + h_{16}$$

6.4 Optimizations

6.4.1 Tier Symmetry Removal

From the tier definition given in Section 6.2, we observe that any two tiers with piece colors swapped are symmetric to each other. For example, a tier with two kings and a red pawn is symmetric to a tier with two kings and a black pawn. Formally, we have that the tier "`AaBbPpNnCcRr_rows1_rows2`" is symmetric to the tier "`aAbBpPnNcCrR_rows2_rows1`". This allows us to only solve for one of the two tiers from each symmetric pair of tiers.

Without loss of generality, we define the canonical tier to be the one with a larger tier string value in any pair of symmetric tiers. For example, tier "`200200000000__`" is the canonical tier of the pair of symmetric tiers `{"022000000000__", "200200000000__"}`.

6.4.2 Encoding Value-Remoteness Records

Chinese chess has no tying positions because the only way to end a game is for one player to run out of moves. Therefore, the only possible values of positions are win, lose, and draw. This allows us to encode the value and remoteness of a position as a single integer using the following mapping:

```
0: RESERVED – UNREACHEABLE POSITION
1: lose in 0
2: lose in 1
3: lose in 2
…
32767: lose in 32766
32768: draw
32769: win in 32766
32770: win in 32765
```

```
   32771: win in 32764
   …
   65535: win in 0
```
Note that the values are assigned in a way that a higher encoded value is always preferred to a lower value, allowing a database querying algorithm to make faster decisions on which move gives the best anticipated result.

6.5 Results

All tiers with less than or equal to 6 pieces have been solved under the simplified ruleset that allows any number of repetitions of the same position and any number of moves without captures. The total size of the database for all 6-piece endgames is 98.49 GB.

There are a total of 1,148,364,635,860 6-piece endgame positions defined under the position hash definition we gave in Section 6.3. 715,160,403,432 of them appears to be legal under the rules described in Section 6.1 and were solved either explicitly if they belong to a canonical tier or implicitly if they belong to a non-canonical tier. The hash efficiency is therefore 0.6227642. A This efficiency is not surprising as we did not exclude illegal positions (such as those with two generals in the same column without pieces in between and those with one general overlapping with a bishop) from our hash definition. Excluding those illegal positions might be extremely difficult and computationally expensive.

We also scanned the database for a value-remoteness count for all positions in the endgames that we solved. Figure 6.20 shows one of the positions with the largest remoteness discovered in all 6-piece endgames. It is player red’s turn and it is a loss for the red player in 264 moves.

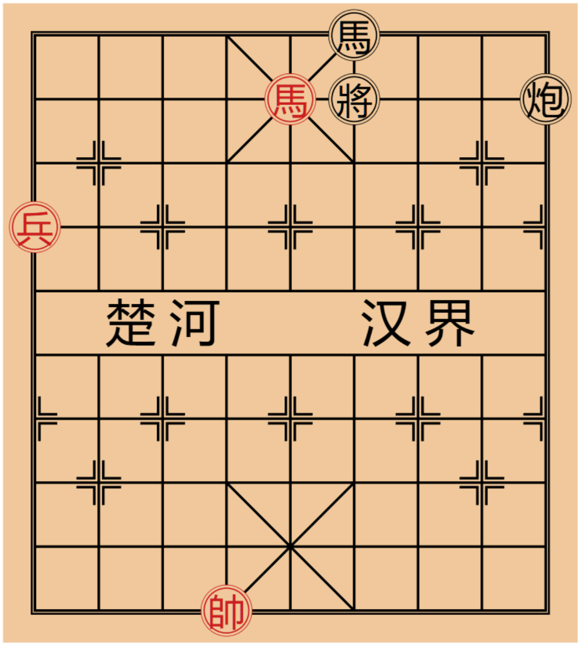

Figure 6.20: A position with the largest remoteness in all 6-piece Chinese chess endgames

Figure 6.21 shows one of the positions with the largest remoteness discovered in all 5-piece endgames. It is player black's turn and it is a loss for the black player in 154 moves.

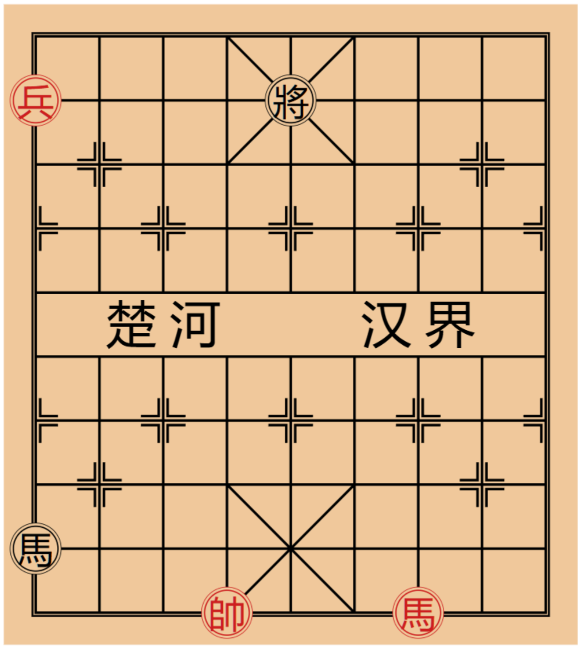

Figure 6.21: A position with the largest remoteness in all 5-piece Chinese chess endgames

The value-remoteness count of all positions in 6-piece and 5-piece endgames is summarized in Tables A.1 and A.2 respectively in Appendix A. It is worth noting that the statistics also count positions that appear to be legal but are in fact unreachable from the initial position of the game. Unfortunately, these positions could not be filtered out because doing so would require finding a path from each position to the initial position in the reverse position graph, and the amount of time it takes to do so is linear to the size of the full game graph. Figure 6.22 shows an example of a position that appears to be rule-abiding but is actually not reachable because all of its parent positions are illegal.

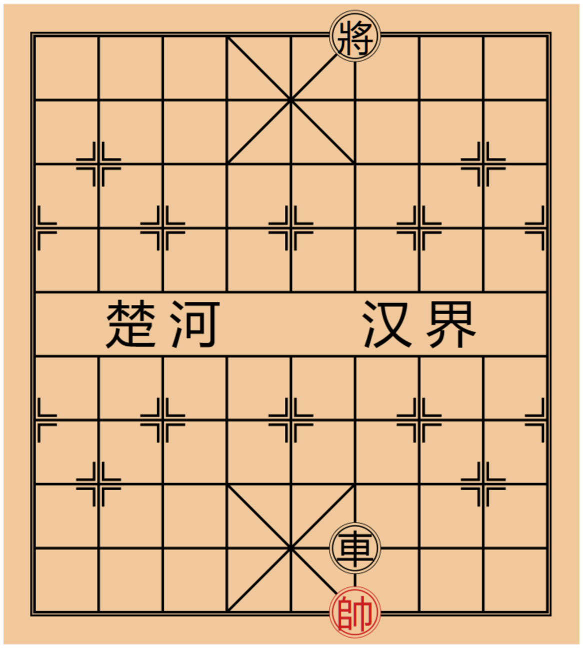

Figure 6.22: An unreachable position in Chinese chess that cannot be easily tested

It is currently red’s turn, and the position appears to be a losing position in 2 moves for red. However, the only possible moves that black could have made in the previous turn are horizontal rook moves, because otherwise red would have made an illegal move and placed the red general under the attack of the black rook two moves ago. But even if the previous move was a horizontal rook move, the move that red made two moves ago is still illegal because the two generals would be in the same column with no pieces between them.

Chapter 7: Solving Quixo

Quixo is a board game designed by Thierry Chapeau and published by Gigamic [13]. The game shares some similar rules with tic-tac-toe but is played on a 5x5 board and is loopy. The game can be played by either 2 or 4 players with slightly different rules. In this Chapter, we only discuss the 2-player version of the game.

Quixo has previously been solved by Tanaka et al [14]. The initial position was shown to be a drawing position for the original 5x5 version. The paper also discussed other variants of the game including those played on 4x4 and 3x3 boards with four or three pieces in a row wins, respectively. The paper shows that the initial position is a win for the first player in 21 moves for the “4x4 four-in-a-row wins” variant, or win for the first player in 7 moves for the “3x3 three-in-a-row wins” variant. We successfully reproduced the results with an alternative solving approach detailed in Chapters 3 and 4 of this report. The total size of the database for the original “5x5 five-in-a-row” game is 79.9 GiB.

The game is implemented in GamesmanOne and solved with the built-in Tier Solver. The source code for Quixo is located under the `src/games/quixo/` directory. The results of the game are published as an interactive gameplay interface on GamesmanUni.[14]

7.1 Rules

In this section and all subsequent sections, all definitions are given for the original “5x5 five-in-a-row” version of the game unless otherwise specified.

The game is played on a 5x5 grid of pieces with the blank side of all pieces initially facing up. Player one controls the X pieces, and player two controls the O pieces. Player one goes first. At each turn, the player can make two different types of moves:

1. Flip a piece on the border with the blank side facing up to their own symbol and move it to one of the ends of the same row or column, sliding all pieces in between to fill up the gap.
2. Move a piece on the border with their own symbol facing up to one of the ends of the same row or column, and slide all pieces in between to fill up the gap.

For both of the two cases, the player is not allowed to keep the piece in place by claiming to have moved it towards the direction of the border. For example, suppose it is O’s turn to make the next move at the position as shown in Figure 7.1, the player is not allowed to move the only piece with the O symbol facing up upwards.

[14] https://nyc.cs.berkeley.edu/uni/games/quixo/variants.

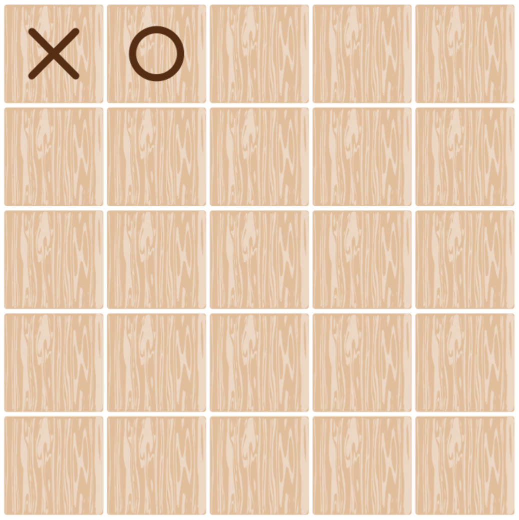

Figure 7.1: An example position in Quixo with O being the player to make the next move

The game is over once a five-in-a-row of the same symbol is formed. If only one type of five-in-a-row is formed, then it is a win for the player who controls that symbol. If both types of five-in-a-row are formed at the same time, it is a loss for the player who made the last move. For example, consider the position shown Figure 7.2.1. Suppose it is now X's turn to make the next move. If X moves the red X piece to the right, O would win the game because a column of X and a column of O are formed at the same time as shown in Figure 7.2.2.

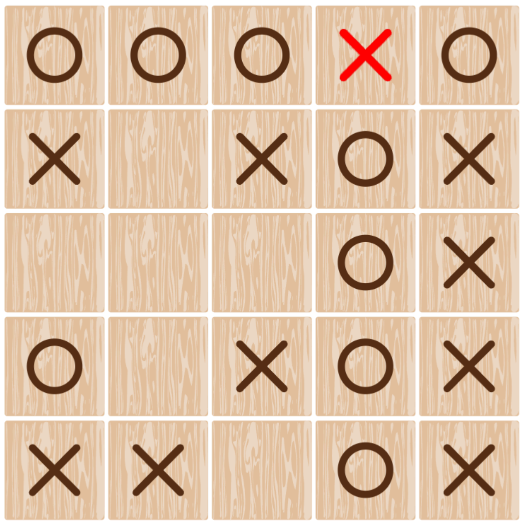

Figure 7.2.1: Position before player X moving the red X piece to the right

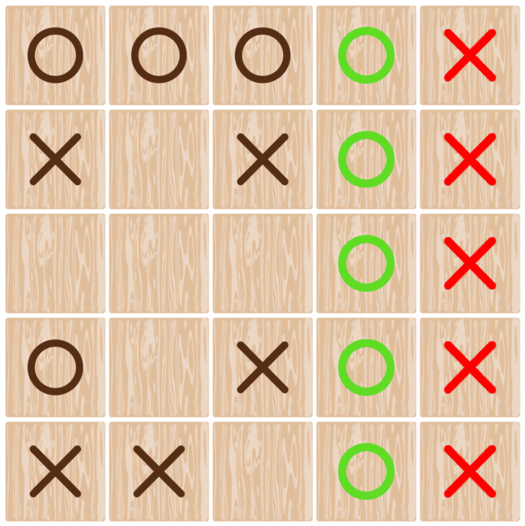

Figure 7.2.2: Position after the move - a winning position for player O

Note that the rules guarantee that players will never run out of moves unless the game is over. This is because the only way for a player to run out of moves is to have all pieces on the border owned by the opponent. But if that is the case, the game must have come to an end in the previous turn due to the formation of at least one five-in-a-row of the opponents pieces on the border.

7.2 Tier and Position Hash Definitions

The only type of irreversible moves in Quixo is the flipping of pieces. The rules state that pieces with either an X or an O symbol facing up can never be flipped again. As a result, the number of blank pieces cannot increase, and the number of X and O pieces can only go up throughout a game.

A natural tier definition is to have each tier contain all positions that share the same set of pieces on the board. Each tier can therefore be uniquely represented as a 3-tuple `(blank, x, o)`, where `blank`, `x`, and `o` are the number of blanks pieces, X's, and O's on the board, respectively. As an example, the position shown in Figure 7.3 is in tier `(6, 10, 9)` because there are 6 blanks, 10 X's, and 9 O's on the board.

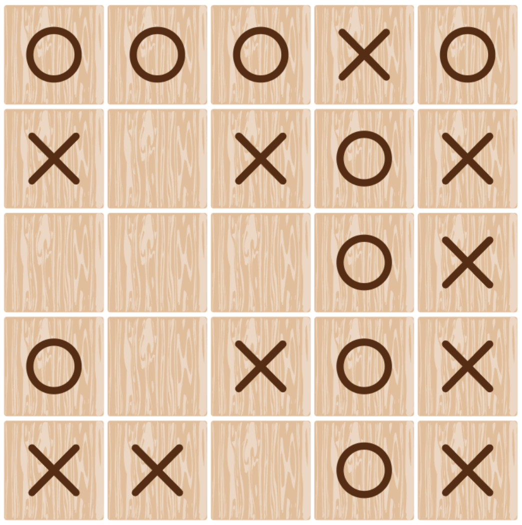

Figure 7.3: An example position in tier `(6, 10, 9)` of 5x5 Quixo

The initial tier of the game is `(25, 0, 0)`. The largest tiers of the game are `(8, 8, 9)`, `(8, 9, 8)`, and `(9, 8, 8)`. Each one of these tiers has

$$nCr(25, 9) \times nCr(16, 8) \times nCr(8, 8) = 26,293,088,250$$

positions defined by rearranging the pieces.

Given the above tier definition, we can apply the combinatorially optimal hash algorithm to obtain a position hash definition. The hash space for positions within a tier is therefore defined to be all possible rearrangements of the pieces specified by the tier. Using the same example above, tier `(9, 8, 8)` would have 26,293,088,250 hash values defined. Every integer between 0 and 26,293,088,249 is uniquely mapped to a possible rearrangement of 9 blanks, 8 X's, and 8 O's. Implementation-wise, the hashing of positions is handled automatically by the Generic Hash module adapted from GamesmanClassic and reimplemented for GamesmanOne. An overview of the combinatorially optimal hash algorithm is given in Chapter 2.

7.3 Optimizations

Since the game uses a square board, symmetric positions can be found by rotating the board 90, 180, and 270 degrees, or mirroring the board along the central row or column and then applying the same rotations above. There are at most 8 positions in a set of symmetric positions. For some positions such as the initial position with all pieces' blank side facing up, the number of unique symmetric positions may be fewer than 8. The 8 positions shown in Figures 7.4.1 through 7.4.8 form a set of symmetric positions.

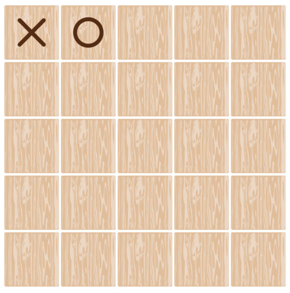
Figure 7.4.1: Original position

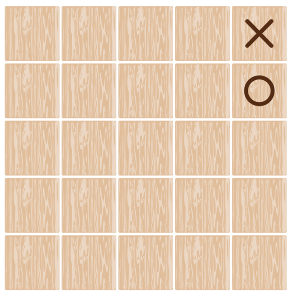
Figure 7.4.2: Rotated 90° clockwise

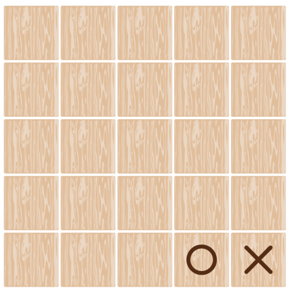
Figure 7.4.3: Rotated 180°

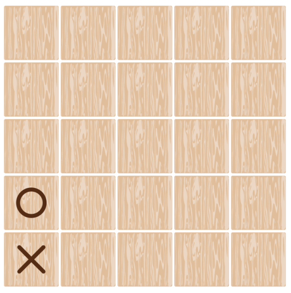
Figure 7.4.4: Rotated 270° clockwise

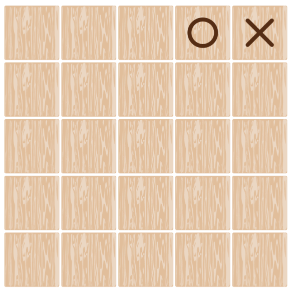
Figure 7.4.5: Mirrored

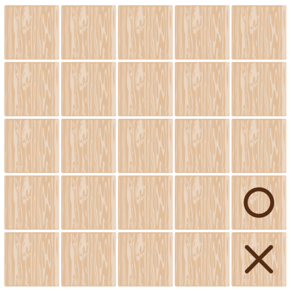
Figure 7.4.6: Mirrored and rotated 90° clockwise

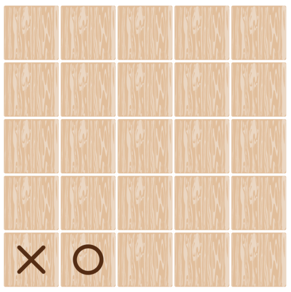
Figure 7.4.7: Mirrored and rotated 180°

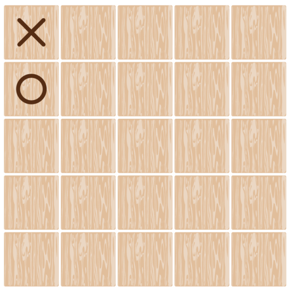
Figure 7.4.8: Mirrored and rotated 270° clockwise

Unfortunately, tier symmetry removal is not possible for Quixo. As a counter-example, consider the following two positions shown in Figures 7.5.1 and 7.5.2:

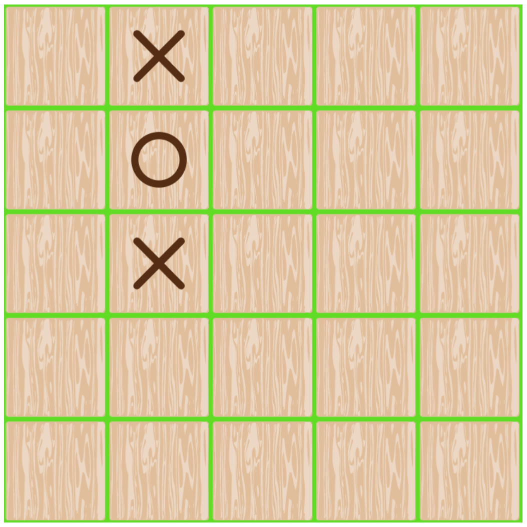

Figure 7.5.1: A reachable position in tier `(22, 2, 1)`, player O's turn

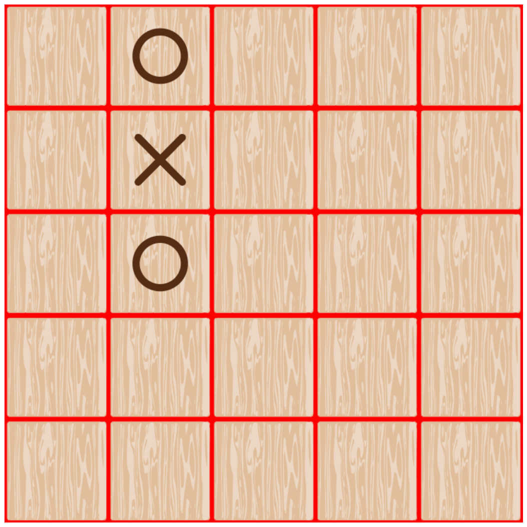

Figure 7.5.2: An unreachable position in tier `(22, 1, 2)`, player X's turn

The position in Figure 7.5.1 on the left is a reachable position in tier `(22, 2, 1)` with O being the player to make the next move. By tier symmetry, this position is symmetric to the position in Figure 7.5.2 in tier `(22, 1, 2)` with X being the player to make the next move. However, the position on the right is not reachable via any valid sequence of moves from the initial position. This shows that the two tiers `(22, 2, 1)` and `(22, 1, 2)` are not symmetric to each other by the definition of tier symmetry.

Further analysis shows that most of the tiers with more symbols facing up are symmetric to each other. However, we found no simple and verifiable way of identifying them without solving them. Hence the solve for all Quixo variants in this report was carried out without tier symmetry removal.

7.4 Results

We solved three different variants of the game: 5x5, 4x4, and 3x3, in which the number of pieces in a row to win the game is adjusted to 5, 4, and 3 respectively. The 3x3 variant takes less than a second to solve whereas the 4x4 variant is solved in 43 seconds with an Intel i9-12900K. The 5x5 variant was mostly solved on the Savio cluster and partially on the Perlmutter Supercomputer. Approximately 16,500 core hours were spent on the `savio4_htc` partition of the Savio cluster, and approximately 24 hours were spent on Perlmutter using 4 compute nodes. At the time this report is written, the cost of each core hour on the `savio4_htc` partition is 3.67 service units. An approximate total of 60,555 Savio service units were spent solving the 5x5 variant of Quixo. Both the 4x4 and the 3x3 variants were then analyzed using the built-in tier game analyzer in GamesmanOne, which performs a breadth-first traversal of the game graph from the initial position. The output of the analyzer includes a value-remoteness count of all positions, canonical positions, and one example position with the largest remoteness discovered.

7.4.1 5x5 Five-In-A-Row

5x5 is the original version of the game. The solver results show that the initial position is a drawing position, which means that both players have a strategy not to lose the game. The total database size is 85.8 GB.

The number of positions defined by the hash function is 1,694,577,218,886 (1.69 trillion). A detailed analysis of game positions is still being constructed by the time this report is written.

7.4.2 4x4 Four-In-A-Row

A total of 82,497,861 positions were discovered in 4x4 Quixo, which gives a total database size of 3.9 MB. The game is a win for the first player in 21 moves.

One of the longest wins is from the initial position (blank board), which has a remoteness of 21. One longest loss is from the position shown in Figure 7.6 with O being the player to make the next move. This position has a remoteness of 22, which means that player two will lose in 22 moves if both players follow their optimal strategies.

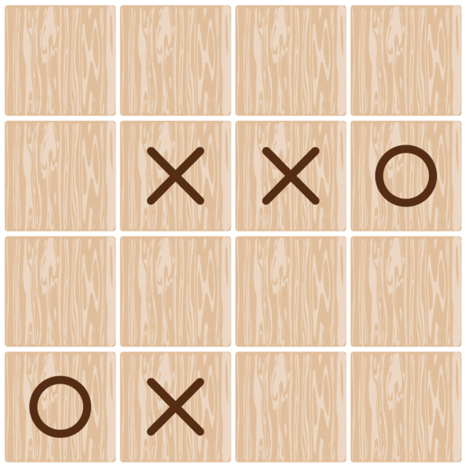

Figure 7.6: A position with the largest remoteness in 4x4 Quixo

The value-remoteness count of all positions and canonical positions of 4x4 Quixo is summarized in Tables A.3 and A.4 respectively in Appendix A.

7.4.3 3x3 Three-In-A-Row

A total of 32,027 positions were discovered in 3x3 Quixo, and the total database size is 8.6 KB. This variant is a win for the first player in 7 moves. One of the longest wins is the initial position (blank board), which has a remoteness of 7. One of the longest losses is the position shown in Figure 7.7 with O being the player to make the next move. This position has a remoteness of 8.

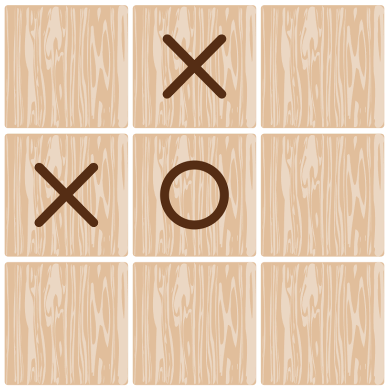
Figure 7.7: A position with the largest remoteness in 3x3 Quixo

The value-remoteness count of all positions and canonical positions of 3x3 Quixo is summarized in Tables A.5 and A.6 respectively in Appendix A.

Chapter 8: Conclusion

In this report, we introduced an algorithm for solving loopy tier games, demonstrating its application in various games and endgames. We successfully applied this algorithm to all 6-piece Chinese chess endgames and three distinct variants of Quixo, including the original 5x5 version. The games that we solved are summarized in Table 8.1.

Game	No. Reachable Positions	Database Size	Approx. Solve Time
Chinese chess (6-piece endgames)	< 715,160,403,432	98.5 GB	5,500 core hours
Chinese chess (5-piece endgames)	< 4,738,622,370	515 MB	21.3 core hours
Quixo (5x5)	< 1,694,577,218,886	85.8 GB	16,500 core hours
Quixo (4x4)	82,497,861	3.9 MB	0.191 core hour
Quixo (3x3)	32,027	8.6 KB	< 0.001 core hour

Table 8.1: Summary of solved games

Notably, we unveiled GamesmanOne, a new parallel game-solving system designed for generic two-player abstract strategy games. Featuring the parallel tier solver, GamesmanOne leverages modern features of the C programming language, offering a system architecture that is both easy to maintain and akin to the design of the original GamesmanClassic. This marks the first instance within GamesCrafters of creating a scalable, parallel solving system capable of solving both loopy and loop-free games at the level of high-performance computing. Game developers acquainted with GamesmanClassic can now effortlessly transition to programming within GamesmanOne, utilizing the Berkeley Savio computing cluster to solve games that previously required custom solvers to finish in a reasonable amount of time.

While GamesmanOne has significantly enhanced solver performance and maintenance efficiency, it is still in the process of incorporating some of the robust features available in GamesmanClassic. In particular, specialized solving features such as *WinBy*[15] and Grundy numbers for combinatorial games are yet to be adapted. Currently, as discussed in Section 5.4, GamesmanOne primarily utilizes the generic loopy tier solver, which has not yet harnessed the unique properties of loop-free games. We are excited about the potential to integrate more of GamesmanClassic's features into GamesmanOne in upcoming developments.

Finally, we presented the results of solving Chinese chess endgames and Quixo. We implemented a custom endgame solver for Chinese chess as part of the GamesmanXiangqi project. The position with the largest remoteness that we discovered in all 6-piece Chinese chess endgames under a simplified ruleset without move restrictions is 264. Quixo was implemented as part of GamesmanOne and was solved using the built-in tier solver. The

[15] An alternative measure of player utility. Following a perfect strategy defined by maximizing WinBy at a winning position, players would choose the path that allows them to win by as much as possible as opposed to as quickly as possible. A perfect strategy at losing positions is similarly defined.

original Quixo game is a draw game at the starting position, whereas the 4x4 and 3x3 variants are both wins for the first player in 21 and 7 moves, respectively.

Bibliography

[1] Garcia, Daniel Dante. “GAMESMAN - A finite, two-person, perfect-information game generator.” M.S. Technical Report, EECS Department, University of California, Berkeley, June 1995. https://people.eecs.berkeley.edu/~ddgarcia/software/gamesman/GAMESMAN.pdf. Accessed 25 April 2024.

[2] GamesCrafters Computational Game Theory Research and Development Group. “GamesCrafters/GamesmanClassic: The classic C solver and tcl/tk frontend.” *GitHub*, https://github.com/GamesCrafters/GamesmanClassic. Accessed 14 April 2024.

[3] Yokota, Justin. “High Efficiency Computation of Game Tree Exploration in Connect 4,” M.S. Technical Report, EECS Department, University of California, Berkeley, Aug 2022. *EECS-2022-219.pdf*, http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-219.pdf. Accessed 14 April 2024.

[4] Cheung, Cameron. “Techniques for Solving and Visualizing Large Games.” M.S. Technical Report, EECS Department, University of California, Berkeley, May 2023. *EECS-2023-186.pdf*, http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-186.pdf. Accessed 14 April 2024.

[5] Delgadillo, Max. “The Retrograde Solver.” *solveretrograde.c*, 11 Oct 2006, https://github.com/GamesCrafters/GamesmanClassic/blob/master/src/core/solveretrograde.c. Accessed 14 April 2024.

[6] GamesCrafters Computational Game Theory Research and Development Group. "GamesmanOne." *GamesCrafters/GamesmanOne: Parallel C solver*, 2024, https://github.com/GamesCrafters/GamesmanOne. Accessed 14 April 2024.

[7] Technical Committee ISO/IEC JTC 1/SC 22. "IEC 9899:2018." June 2018, https://www.iso.org/standard/74528.html. Accessed 28 April 2024.

[8] NERSC. "Perlmutter Architecture." *NERSC Documentation*, 2024, https://docs.nersc.gov/systems/perlmutter/architecture/#system-specifications. Accessed 14 April 2024.

[9] Wikipedia. "Forsyth–Edwards Notation." *Wikipedia*, https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation. Accessed 14 April 2024.

[10] GamesCrafters Computational Game Theory Research and Development Group. "GamesCrafters/GamesmanUni." *GitHub*, https://github.com/GamesCrafters/GamesmanUni. Accessed 14 April 2024.

[11] Ling, Anthony. "GamesmanPuzzles: A Leap Into the Puzzles Domain." M.S. Technical Report, EECS Department, University of California, Berkeley, May 2021, http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-146.pdf. Accessed 29 April 2024.

[12] Yen, Shi-Jim, et al. "Computer Chinese Chess." *ICGA journal*, vol. 27, no. 1, 2004, pp. 3-18.

[13] BoardGameGeek. "Quixo | Board Game." *BoardGameGeek*, https://boardgamegeek.com/boardgame/3190/quixo. Accessed 14 April 2024.

[14] Tanaka, Satoshi, et al. "Quixo Is Solved." *Advances in Computer Games*, vol. 13262, 2022, pp. 85-95.

Appendix A: Game Statistics

A.1 Value-Remoteness Counts for 6-Piece Endgames of Chinese chess[16]

Remoteness	Win	Lose	Draw	Total
Inf	0	0	108,689,963,230	108,689,963,230
264	0	4	0	4
263	4	0	0	4
262	0	16	0	16
261	4	0	0	4
260	0	116	0	116
259	64	0	0	64
258	0	192	0	192
257	244	0	0	244
256	0	828	0	828
255	644	0	0	644
254	0	1,716	0	1,716
253	1,348	0	0	1,348
252	0	3,980	0	3,980
251	2,792	0	0	2,792
250	0	9,188	0	9,188
249	6,660	0	0	6,660
248	0	17,936	0	17,936
247	11,464	0	0	11,464
246	0	27,376	0	27,376
245	16,652	0	0	16,652
244	0	34,568	0	34,568
243	20,864	0	0	20,864
242	0	32,844	0	32,844
241	18,372	0	0	18,372
240	0	32,148	0	32,148
239	19,632	0	0	19,632
238	0	34,952	0	34,952
237	21,552	0	0	21,552
236	0	45,920	0	45,920
235	28,116	0	0	28,116
234	0	66,112	0	66,112
233	38,384	0	0	38,384
232	0	80,672	0	80,672

[16] Under a simplified ruleset that allows any number of repetitions of the same position; also contains unreachable positions that appear to be legal.

231	50,472	0	0	50,472
230	0	96,688	0	96,688
229	63,144	0	0	63,144
228	0	100,064	0	100,064
227	67,240	0	0	67,240
226	0	99,996	0	99,996
225	70,900	0	0	70,900
224	0	113,340	0	113,340
223	82,052	0	0	82,052
222	0	132,060	0	132,060
221	98,676	0	0	98,676
220	0	153,976	0	153,976
219	116,132	0	0	116,132
218	0	195,980	0	195,980
217	150,740	0	0	150,740
216	0	239,324	0	239,324
215	187,708	0	0	187,708
214	0	285,024	0	285,024
213	216,480	0	0	216,480
212	0	324,368	0	324,368
211	243,300	0	0	243,300
210	0	370,772	0	370,772
209	274,612	0	0	274,612
208	0	426,592	0	426,592
207	315,620	0	0	315,620
206	0	513,520	0	513,520
205	388,446	0	0	388,446
204	0	654,980	0	654,980
203	486,904	0	0	486,904
202	0	818,084	0	818,084
201	598,230	0	0	598,230
200	0	991,444	0	991,444
199	693,576	0	0	693,576
198	0	1,124,206	0	1,124,206
197	776,014	0	0	776,014
196	0	1,315,632	0	1,315,632
195	882,928	0	0	882,928
194	0	1,498,450	0	1,498,450
193	995,278	0	0	995,278
192	0	1,658,692	0	1,658,692
191	1,091,232	0	0	1,091,232
190	0	1,786,442	0	1,786,442

189	1,159,692	0	0	1,159,692
188	0	1,941,452	0	1,941,452
187	1,240,686	0	0	1,240,686
186	0	2,101,014	0	2,101,014
185	1,347,910	0	0	1,347,910
184	0	2,359,042	0	2,359,042
183	1,511,536	0	0	1,511,536
182	0	2,735,008	0	2,735,008
181	1,729,150	0	0	1,729,150
180	0	3,118,920	0	3,118,920
179	1,923,044	0	0	1,923,044
178	0	3,529,848	0	3,529,848
177	2,128,568	0	0	2,128,568
176	0	3,868,438	0	3,868,438
175	2,347,128	0	0	2,347,128
174	0	4,247,086	0	4,247,086
173	2,576,306	0	0	2,576,306
172	0	4,634,856	0	4,634,856
171	2,838,146	0	0	2,838,146
170	0	5,156,476	0	5,156,476
169	3,171,098	0	0	3,171,098
168	0	5,695,920	0	5,695,920
167	3,503,888	0	0	3,503,888
166	0	6,442,086	0	6,442,086
165	3,929,008	0	0	3,929,008
164	0	7,384,988	0	7,384,988
163	4,456,958	0	0	4,456,958
162	0	8,474,580	0	8,474,580
161	5,093,372	0	0	5,093,372
160	0	9,725,758	0	9,725,758
159	5,833,100	0	0	5,833,100
158	0	10,986,328	0	10,986,328
157	6,581,028	0	0	6,581,028
156	0	12,163,310	0	12,163,310
155	7,384,092	0	0	7,384,092
154	0	13,326,196	0	13,326,196
153	8,146,742	0	0	8,146,742
152	0	14,457,668	0	14,457,668
151	8,919,718	0	0	8,919,718
150	0	15,599,050	0	15,599,050
149	9,842,842	0	0	9,842,842
148	0	16,586,928	0	16,586,928

147	10,644,244	0	0	10,644,244
146	0	17,310,602	0	17,310,602
145	11,197,984	0	0	11,197,984
144	0	17,696,728	0	17,696,728
143	11,421,890	0	0	11,421,890
142	0	17,869,814	0	17,869,814
141	11,502,352	0	0	11,502,352
140	0	17,970,150	0	17,970,150
139	11,584,104	0	0	11,584,104
138	0	18,046,788	0	18,046,788
137	11,627,768	0	0	11,627,768
136	0	17,992,772	0	17,992,772
135	11,641,504	0	0	11,641,504
134	0	17,881,132	0	17,881,132
133	11,771,920	0	0	11,771,920
132	0	18,201,142	0	18,201,142
131	12,132,676	0	0	12,132,676
130	0	18,809,138	0	18,809,138
129	12,736,104	0	0	12,736,104
128	0	19,493,014	0	19,493,014
127	13,412,532	0	0	13,412,532
126	0	20,676,294	0	20,676,294
125	14,397,412	0	0	14,397,412
124	0	22,060,596	0	22,060,596
123	15,413,704	0	0	15,413,704
122	0	23,724,326	0	23,724,326
121	16,738,974	0	0	16,738,974
120	0	25,490,194	0	25,490,194
119	17,975,716	0	0	17,975,716
118	0	27,573,008	0	27,573,008
117	19,566,968	0	0	19,566,968
116	0	29,972,052	0	29,972,052
115	21,495,676	0	0	21,495,676
114	0	32,508,920	0	32,508,920
113	23,568,544	0	0	23,568,544
112	0	34,921,768	0	34,921,768
111	25,435,204	0	0	25,435,204
110	0	37,437,406	0	37,437,406
109	27,238,716	0	0	27,238,716
108	0	40,245,740	0	40,245,740
107	29,613,626	0	0	29,613,626
106	0	44,265,818	0	44,265,818

105	32,838,878	0	0	32,838,878
104	0	49,411,034	0	49,411,034
103	36,897,676	0	0	36,897,676
102	0	55,585,880	0	55,585,880
101	41,154,282	0	0	41,154,282
100	0	63,343,666	0	63,343,666
99	46,566,832	0	0	46,566,832
98	0	74,798,992	0	74,798,992
97	53,933,866	0	0	53,933,866
96	0	88,683,668	0	88,683,668
95	62,965,472	0	0	62,965,472
94	0	100,012,026	0	100,012,026
93	71,267,244	0	0	71,267,244
92	0	107,597,220	0	107,597,220
91	78,330,972	0	0	78,330,972
90	0	113,994,258	0	113,994,258
89	85,170,562	0	0	85,170,562
88	0	117,876,972	0	117,876,972
87	89,178,770	0	0	89,178,770
86	0	121,701,982	0	121,701,982
85	93,086,600	0	0	93,086,600
84	0	126,473,708	0	126,473,708
83	96,952,940	0	0	96,952,940
82	0	134,342,064	0	134,342,064
81	102,219,848	0	0	102,219,848
80	0	144,431,998	0	144,431,998
79	108,380,054	0	0	108,380,054
78	0	154,413,924	0	154,413,924
77	115,812,224	0	0	115,812,224
76	0	164,193,532	0	164,193,532
75	124,586,918	0	0	124,586,918
74	0	172,506,448	0	172,506,448
73	132,008,814	0	0	132,008,814
72	0	182,777,102	0	182,777,102
71	139,792,294	0	0	139,792,294
70	0	199,487,938	0	199,487,938
69	154,463,678	0	0	154,463,678
68	0	224,437,022	0	224,437,022
67	174,638,152	0	0	174,638,152
66	0	249,715,562	0	249,715,562
65	197,323,438	0	0	197,323,438
64	0	273,327,158	0	273,327,158

63	221,660,100	0	0	221,660,100
62	0	300,142,220	0	300,142,220
61	248,529,722	0	0	248,529,722
60	0	337,201,984	0	337,201,984
59	283,830,126	0	0	283,830,126
58	0	395,259,138	0	395,259,138
57	333,647,650	0	0	333,647,650
56	0	478,024,514	0	478,024,514
55	397,966,918	0	0	397,966,918
54	0	584,591,368	0	584,591,368
53	489,322,908	0	0	489,322,908
52	0	702,590,774	0	702,590,774
51	601,359,198	0	0	601,359,198
50	0	808,985,936	0	808,985,936
49	731,576,742	0	0	731,576,742
48	0	901,628,934	0	901,628,934
47	880,141,000	0	0	880,141,000
46	0	986,631,852	0	986,631,852
45	1,015,942,558	0	0	1,015,942,558
44	0	1,082,386,540	0	1,082,386,540
43	1,155,081,690	0	0	1,155,081,690
42	0	1,207,944,708	0	1,207,944,708
41	1,302,644,356	0	0	1,302,644,356
40	0	1,366,999,344	0	1,366,999,344
39	1,467,214,384	0	0	1,467,214,384
38	0	1,570,792,616	0	1,570,792,616
37	1,688,516,138	0	0	1,688,516,138
36	0	1,823,655,036	0	1,823,655,036
35	1,991,580,846	0	0	1,991,580,846
34	0	2,131,897,642	0	2,131,897,642
33	2,453,266,140	0	0	2,453,266,140
32	0	2,611,323,006	0	2,611,323,006
31	2,963,492,588	0	0	2,963,492,588
30	0	3,268,554,492	0	3,268,554,492
29	3,419,605,054	0	0	3,419,605,054
28	0	4,064,915,646	0	4,064,915,646
27	3,938,085,710	0	0	3,938,085,710
26	0	5,059,776,454	0	5,059,776,454
25	4,594,868,340	0	0	4,594,868,340
24	0	6,395,869,382	0	6,395,869,382
23	5,579,815,464	0	0	5,579,815,464
22	0	7,903,021,218	0	7,903,021,218

21	6,842,575,510	0	0	6,842,575,510
20	0	9,502,322,114	0	9,502,322,114
19	8,277,998,474	0	0	8,277,998,474
18	0	11,459,528,160	0	11,459,528,160
17	10,023,031,346	0	0	10,023,031,346
16	0	13,602,205,976	0	13,602,205,976
15	12,216,063,394	0	0	12,216,063,394
14	0	15,846,000,568	0	15,846,000,568
13	15,211,528,586	0	0	15,211,528,586
12	0	19,445,656,352	0	19,445,656,352
11	18,730,750,170	0	0	18,730,750,170
10	0	24,267,860,064	0	24,267,860,064
9	21,829,205,722	0	0	21,829,205,722
8	0	28,912,963,268	0	28,912,963,268
7	27,928,089,414	0	0	27,928,089,414
6	0	36,355,592,446	0	36,355,592,446
5	41,597,868,224	0	0	41,597,868,224
4	0	45,783,130,992	0	45,783,130,992
3	53,457,032,748	0	0	53,457,032,748
2	0	32,355,827,704	0	32,355,827,704
1	54,891,584,468	0	0	54,891,584,468
0	0	12,113,813,732	0	12,113,813,732
Totals	309,236,263,312	297,234,176,890	108,689,963,230	715,160,403,432
Total size of database: 98.49 GB				

Table A.1: Value-remoteness counts for 6-piece endgames of Chinese chess

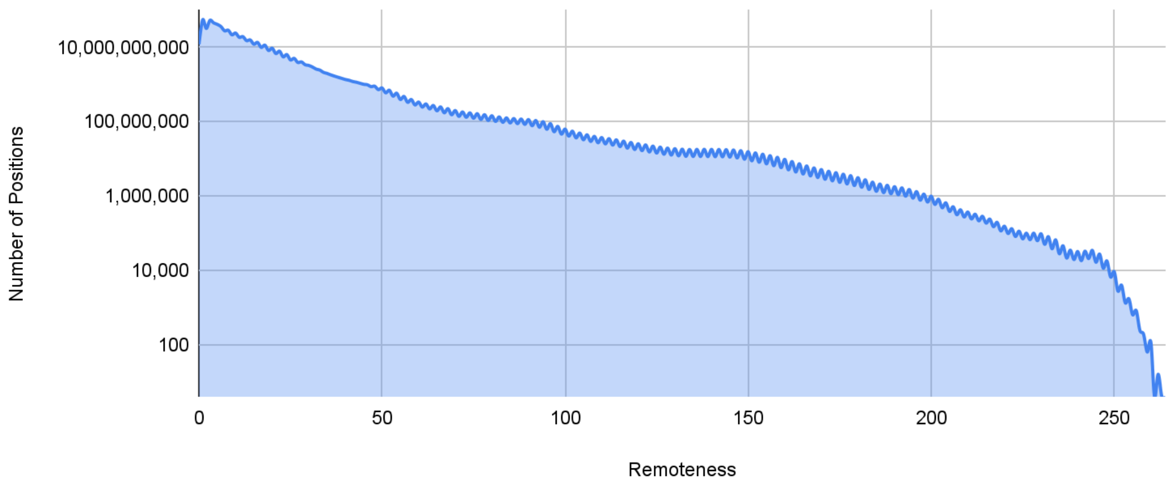

Figure A.1: Number of positions vs. remoteness in Chinese chess 6-piece endgames

A.2 Value-Remoteness Counts for 5-Piece Endgames of Chinese chess[17]

Remoteness	Win	Lose	Draw	Total
Inf	0	0	821,847,928	821,847,928
154	0	32	0	32
153	8	0	0	8
152	0	156	0	156
151	40	0	0	40
150	0	372	0	372
149	100	0	0	100
148	0	1,296	0	1,296
147	368	0	0	368
146	0	1,864	0	1,864
145	616	0	0	616
144	0	4,236	0	4,236
143	1,452	0	0	1,452
142	0	5,952	0	5,952
141	2,076	0	0	2,076
140	0	7,448	0	7,448
139	2,700	0	0	2,700
138	0	8,368	0	8,368
137	2,924	0	0	2,924
136	0	9,004	0	9,004
135	3,256	0	0	3,256
134	0	9,112	0	9,112
133	3,120	0	0	3,120
132	0	9,536	0	9,536
131	3,672	0	0	3,672
130	0	10,784	0	10,784
129	4,184	0	0	4,184
128	0	13,472	0	13,472
127	4,648	0	0	4,648
126	0	13,728	0	13,728
125	4,768	0	0	4,768
124	0	16,632	0	16,632
123	5,880	0	0	5,880
122	0	18,500	0	18,500
121	7,052	0	0	7,052
120	0	25,432	0	25,432
119	9,372	0	0	9,372

[17] Under a simplified ruleset that allows any number of repetitions of the same position; also contains unreachable positions that appear to be legal.

118	0	27,016	0	27,016
117	11,144	0	0	11,144
116	0	31,476	0	31,476
115	14,168	0	0	14,168
114	0	37,108	0	37,108
113	16,856	0	0	16,856
112	0	43,664	0	43,664
111	20,564	0	0	20,564
110	0	47,940	0	47,940
109	21,852	0	0	21,852
108	0	50,936	0	50,936
107	24,056	0	0	24,056
106	0	59,432	0	59,432
105	29,188	0	0	29,188
104	0	70,018	0	70,018
103	38,332	0	0	38,332
102	0	88,014	0	88,014
101	47,360	0	0	47,360
100	0	103,576	0	103,576
99	53,212	0	0	53,212
98	0	121,380	0	121,380
97	61,662	0	0	61,662
96	0	170,128	0	170,128
95	73,830	0	0	73,830
94	0	229,770	0	229,770
93	87,020	0	0	87,020
92	0	274,860	0	274,860
91	98,848	0	0	98,848
90	0	294,396	0	294,396
89	109,376	0	0	109,376
88	0	331,964	0	331,964
87	124,342	0	0	124,342
86	0	320,472	0	320,472
85	130,692	0	0	130,692
84	0	320,268	0	320,268
83	142,952	0	0	142,952
82	0	309,528	0	309,528
81	152,894	0	0	152,894
80	0	319,844	0	319,844
79	161,882	0	0	161,882
78	0	303,588	0	303,588
77	157,344	0	0	157,344

76	0	258,782	0	258,782
75	152,842	0	0	152,842
74	0	259,338	0	259,338
73	143,824	0	0	143,824
72	0	246,400	0	246,400
71	140,520	0	0	140,520
70	0	233,220	0	233,220
69	140,990	0	0	140,990
68	0	296,254	0	296,254
67	156,988	0	0	156,988
66	0	374,200	0	374,200
65	179,474	0	0	179,474
64	0	393,624	0	393,624
63	201,862	0	0	201,862
62	0	449,190	0	449,190
61	234,218	0	0	234,218
60	0	522,176	0	522,176
59	294,384	0	0	294,384
58	0	595,954	0	595,954
57	391,082	0	0	391,082
56	0	816,184	0	816,184
55	519,498	0	0	519,498
54	0	963,814	0	963,814
53	651,674	0	0	651,674
52	0	1,330,654	0	1,330,654
51	824,804	0	0	824,804
50	0	1,972,266	0	1,972,266
49	1,127,634	0	0	1,127,634
48	0	2,998,068	0	2,998,068
47	1,627,882	0	0	1,627,882
46	0	5,424,882	0	5,424,882
45	2,671,364	0	0	2,671,364
44	0	7,972,312	0	7,972,312
43	4,147,626	0	0	4,147,626
42	0	9,975,044	0	9,975,044
41	5,784,408	0	0	5,784,408
40	0	11,219,758	0	11,219,758
39	7,180,862	0	0	7,180,862
38	0	12,690,314	0	12,690,314
37	8,837,370	0	0	8,837,370
36	0	14,961,618	0	14,961,618
35	10,726,440	0	0	10,726,440

34	0	18,684,216	0	18,684,216
33	13,070,356	0	0	13,070,356
32	0	26,971,560	0	26,971,560
31	17,324,788	0	0	17,324,788
30	0	33,907,268	0	33,907,268
29	22,051,058	0	0	22,051,058
28	0	35,868,820	0	35,868,820
27	26,069,168	0	0	26,069,168
26	0	37,581,648	0	37,581,648
25	30,516,104	0	0	30,516,104
24	0	40,065,274	0	40,065,274
23	36,320,024	0	0	36,320,024
22	0	45,316,598	0	45,316,598
21	42,680,326	0	0	42,680,326
20	0	51,206,758	0	51,206,758
19	46,104,728	0	0	46,104,728
18	0	55,464,432	0	55,464,432
17	50,343,052	0	0	50,343,052
16	0	61,542,590	0	61,542,590
15	59,856,364	0	0	59,856,364
14	0	76,010,992	0	76,010,992
13	77,140,512	0	0	77,140,512
12	0	123,191,232	0	123,191,232
11	111,418,058	0	0	111,418,058
10	0	163,020,926	0	163,020,926
9	142,284,216	0	0	142,284,216
8	0	189,893,024	0	189,893,024
7	191,681,308	0	0	191,681,308
6	0	233,470,974	0	233,470,974
5	284,042,826	0	0	284,042,826
4	0	373,476,452	0	373,476,452
3	386,924,158	0	0	386,924,158
2	0	249,568,120	0	249,568,120
1	362,490,704	0	0	362,490,704
0	0	75,780,898	0	75,780,898
Totals	1,948,087,306	1,968,687,136	821,847,928	4,738,622,370
Total size of database: 515.4 MB				

Table A.2: Value-remoteness counts for 5-piece endgames of Chinese chess

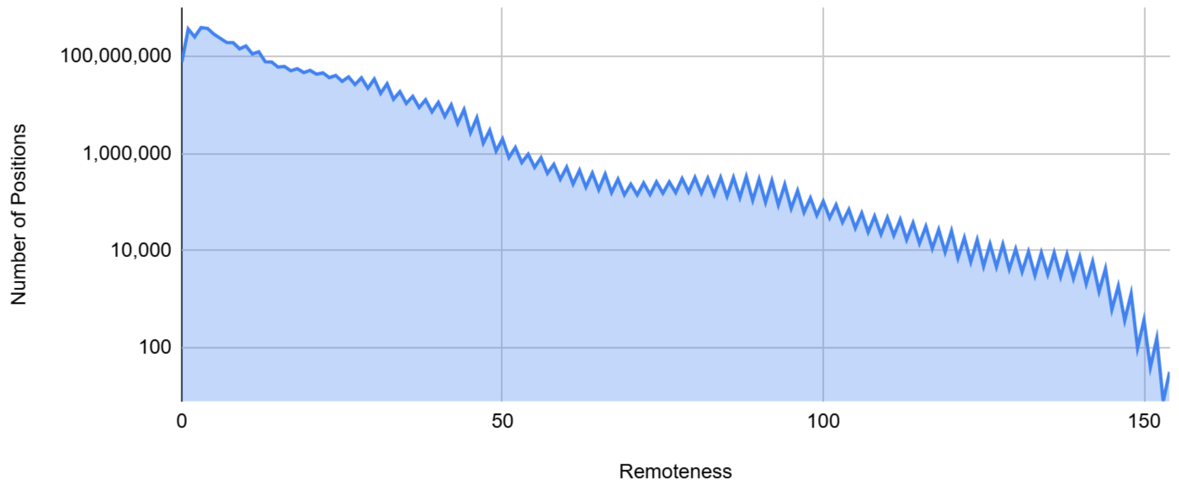

Figure A.2: Number of positions vs. remoteness in Chinese chess 5-piece endgames

A.3 Value-Remoteness Counts for All Positions in 4x4 Quixo

Remoteness	Win	Lose	Draw	Total
Inf	0	0	3,213,236	3,213,236
22	0	16	0	16
21	113	0	0	113
20	0	900	0	900
19	1,960	0	0	1,960
18	0	7,648	0	7,648
17	13,096	0	0	13,096
16	0	35,196	0	35,196
15	57,480	0	0	57,480
14	0	130,128	0	130,128
13	198,104	0	0	198,104
12	0	362,596	0	362,596
11	570,664	0	0	570,664
10	0	870,196	0	870,196
9	1,306,060	0	0	1,306,060
8	0	2,053,168	0	2,053,168
7	2,406,956	0	0	2,406,956
6	0	3,937,940	0	3,937,940
5	3,449,514	0	0	3,449,514
4	0	5,897,464	0	5,897,464
3	4,771,444	0	0	4,771,444
2	0	7,474,656	0	7,474,656
1	29,911,388	972	0	29,912,360
0	7,267,200	8,559,766	0	15,826,966
Totals	49,953,979	29,330,646	3,213,236	82,497,861

Hash space: 86093442	**Hash efficiency**: 0.958236	
Total size of database: 3.9 MB		

Table A.3: Value-remoteness counts for all positions in 4x4 Quixo

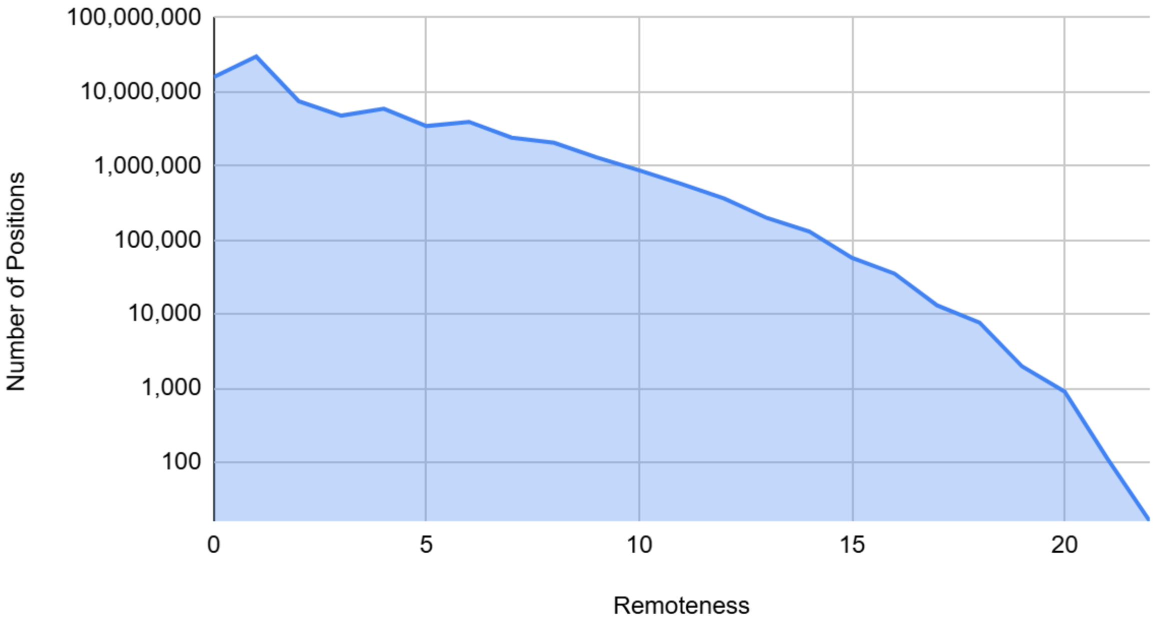

Figure A.3: Number of positions vs. remoteness in 4x4 Quixo

A.4 Value-Remoteness Counts for Canonical Positions in 4x4 Quixo

Remoteness	Win	Lose	Draw	Total
Inf	0	0	402,690	402,690
22	0	2	0	2
21	15	0	0	15
20	0	115	0	115
19	248	0	0	248
18	0	961	0	961
17	1,654	0	0	1,654
16	0	4,422	0	4,422
15	7,227	0	0	7,227
14	0	16,314	0	16,314
13	24,859	0	0	24,859
12	0	45,457	0	45,457
11	71,575	0	0	71,575
10	0	109,142	0	109,142
9	163,717	0	0	163,717
8	0	257,248	0	257,248

7	301,729	0	0	301,729
6	0	493,344	0	493,344
5	432,297	0	0	432,297
4	0	738,742	0	738,742
3	599,421	0	0	599,421
2	0	938,430	0	938,430
1	3,743,934	168	0	3,744,102
0	911,576	1,073,957	0	1,985,533
Totals	6,258,252	3,678,302	402,690	10,339,244

Table A.4: Value-remoteness counts for canonical positions in 4x4 Quixo

A.5 Value-Remoteness Counts for All Positions in 3x3 Quixo

Remoteness	**Win**	**Lose**	**Draw**	**Total**
Inf	0	0	1,532	1,532
8	0	16	0	16
7	37	0	0	37
6	0	98	0	98
5	294	0	0	294
4	0	868	0	868
3	1,038	0	0	1,038
2	0	3,376	0	3,376
1	12,978	20	0	12,998
0	5,080	6,690	0	11,770
Totals	19,427	11,068	1,532	32,027
Hash space: 39366		**Hash efficiency**: 0.813570		
Total size of database: 8.6 KB				

Table A.5: Value-remoteness counts for all positions in 3x3 Quixo

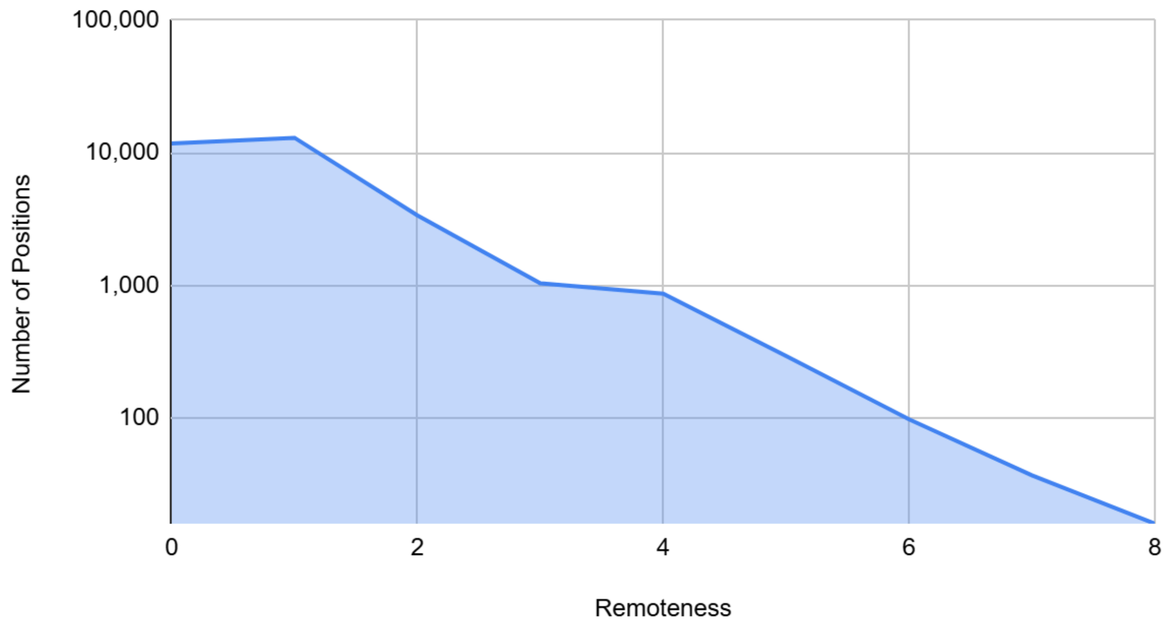

Figure A.4: Number of positions vs. remoteness in 3x3 Quixo

A.6 Value-Remoteness Counts for Canonical Positions in 3x3 Quixo

Remoteness	**Win**	**Lose**	**Draw**	**Total**
Inf	0	0	218	218
8	0	3	0	3
7	7	0	0	7
6	0	14	0	14
5	41	0	0	41
4	0	121	0	121
3	152	0	0	152
2	0	482	0	482
1	1,745	6	0	1,751
0	706	938	0	1,644
Totals	2,651	1,564	218	4,433

Table A.6: Value-remoteness counts for canonical positions in 3x3 Quixo