# Optimizing Time For Eigenvector Calculation

**[to find better moves during the Fuseki of Go]**
Bachelor-Thesis von Simone Wälde aus Würzburg
Tag der Einreichung:

1. Gutachten: Manja Marz
2. Gutachten: Johannes Fürnkranz

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowlegde Engineering Group

Optimizing Time For Eigenvector Calculation
[to find better moves during the Fuseki of Go]

Vorgelegte Bachelor-Thesis von Simone Wälde aus Würzburg

1. Gutachten: Manja Marz
2. Gutachten: Johannes Fürnkranz

Tag der Einreichung:

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-12345
URL: http://tuprints.ulb.tu-darmstadt.de/1234

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Namensnennung – Keine kommerzielle Nutzung – Keine Bearbeitung 2.0 Deutschland
http://creativecommons.org/licenses/by-nc-nd/2.0/de/

# Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den November 30, 2016

(S. Wälde)

## Contents

## Acknowledgements

I want to express my sincere gratitude to Joseph Leydold who helped me understand his ideas and the mathematical background. Furthermore, I want to thank Martin Hölzer and Florian Mock for their time and and help. In general, I want to thank all the people who helped with critique and improvement suggestions. At last, I also want to thank Manja Marz and Johannes Fürnkranz who made it possible for me to write and learn more about Go and Computer Go in general.

## 1 Abstract

The game of Go has been of major relevance in the field of artificial intelligence for the last decades [11, 18] and now, there has been a major breakthrough just recently with AlphaGo beating Lee Sedol, one of the strongest Go players in the World [84]. In this thesis we introduce the *big move* heuristic. This heuristic approximates moves to play in the Beginning stage of Go. The algorithm examines the natural vibration of the board and finds the intersection farthest away from the border and from the other stones. It therefore needs to compute the eigenvector representing the desired mode shape [23]. The *big move* heuristic is constructed to modify the widely known *Monte Carlo Tree Search* [18, 20].

A major part of this thesis consists of testing different methods and functions for eigenvector calculation and approximation to improve the heuristic timewise. The goal is to compute eigenvectors as fast as possible because they are required to be called often during the *tree policy* of *Monte Carlo Tree Search*. We discovered that approximating a single eigenvector tends to be much faster when using the right iterative methods. The best result we got was using methods from Python's SciPy [60] package which implement variants of the Arnoldi Method [91]. After optimizing the algorithm, we incorporated the *big move* heuristic as a *Prior* into the game engine of Pachi, one of the best open source Go programs [3]. The *Prior* is a starting worth for a particular state action pair $(s, a)$ consisting of an action value $Q_{prior}(s, a)$ which is claimed to be achieved after $n_{prior}(s, a)$ simulations [19]. The results fluctuated and are not very precise, but in the end the best result we measured had resulted in a winning expectancy of 59% against regular Pachi.

# 2 Introduction

## 2.1 Introduction to Go

Go strategic board game where all information about the current game state is available to the players. Furthermore, it is purely deterministic and therefore has no random elements [74]. Two players play against each other in order to maximize their own territory. We will explain the rules in section 2.1.2. Around the world, Go is played by around 60 000 000 people [33]. It is mostly played in three countries: Japan, China, and Korea. There are about a thousand professional Go players in these three countries [33] and a few outside of those countries, but the number is very small in comparison. Recently, there have been Pro Qualification tournaments to establish a professional system in Europe [42]. It is difficult to construct an artificial intelligence that plays Go on a professional level. The reason for this is the large decision tree due to the relatively large board size, among other things [11]. Compared to Chess which has a board size of $8 \times 8$, Go has a board size of $19 \times 19$ which gives much more room for possible moves on the board [92]. Not long ago AlphaGo [84], a Go program which uses neural networks, was the first Go program that ever beat a human professional without handicap on a $19 \times 19$ Go board.

### 2.1.1 History of the Game

The exact age of Go is unclear but its origin is dated far back in ancient China. According to legend, it is around 4000 years old, but this claim is lacking proof [17,29]. The first notion of Go is by Confucius around 500 BC who mentioned Go in his *Analects* [17,29,32,83]. He called the game *Yih*. Furthermore, a $17 \times 17$ Go board was found which was used prior to 200 AD in China as well as a silk painting of a Tang Lady from around 750 AD who also played on a $17 \times 17$ board [17]. Therefore, it seems reasonable to assume that an older version of the game existed that was played on a smaller board. Even though Go was invented in China, important contributions to its growth and popularity were made by the Japanese. In the beginning of the 17th century, Tokugawa[1] unified Japan. Four Go schools were formed and a professional system was set up subsequently. Every year the Castle Games were held until the 19th century due to the Meiji Restoration. During that time, Go experienced a period of stagnation because the colleges lost their funding [22, 29]. In 1920, the Japanese Go Association was formed and newspapers began to sponsor tournaments [83]. The professional system was established in the 1950s in Korea as well as 1978 in China. Go is more popular in Korea than anywhere else in the World; more than five percent of Koreans play it regularly [32,83]. Nowadays, the manga “Hikaru no Go” boosted the popularity of Go overseas [83]. Also, it is possible that AlphaGo [84] by Google Deepmind promoted the awareness for Go around the world, as it was mentioned in television and newspapers.

**Figure 1:** A Tang lady playing Go on a $17 \times 17$ board. Source: [80]

[1] Tokugawa Ieyasu ruled Japan in the beginning of the 17th century.

### 2.1.2 Rules of the Game

Go has several different rule sets, for example AGA (American Go Association) Rules, Ing Rules or Chinese Rules [73]. The rule sets only differ in a few aspects of the game. Therefore, we will explain the Japanese Rules [67] because they are most widely used.

Go is a two player game with a small rule set. The Go board is a grid of $19 \times 19$ intersections. Beginners often play on smaller boards with only $9 \times 9$ intersections. $13 \times 13$ is also considered a possible size[2]. The game starts with an empty board. One player plays with black stones and the opponent with white stones. They will be referred to as Black and White.

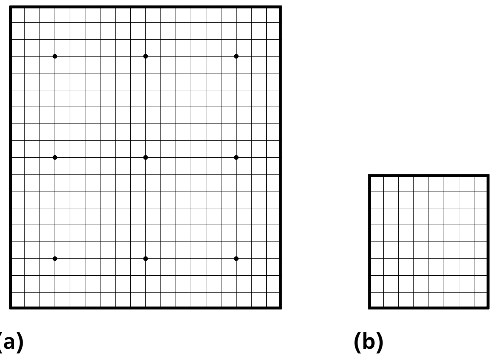


**Figure 2:** The board in (a) has $19 \times 19$ intersections and the board in (b) has $9 \times 9$ intersections.

Black begins the game by placing a black stone on one of the 361 intersections[3]. Also, there can only be one stone on each intersection at the same time. Both players alternate turns of which one turn consists of either placing a stone on the board or passing [29]. It is not allowed to move stones after they have been placed. A stone can only leave the board again, if it gets captured by the opponent.

#### Goal and End of the Game

In Japanese Rules a game is finished after two subsequent passes [29, 67]. After that, the status of all groups is discussed. The status of a group can either be *alive*, *dead* or *unsettled* [29]. A group is a set of stones of one color working together on the board. This is an abstract concept and can not be defined precisely. Both players try to enclose as much territory as possible with their groups while keeping them *alive*. A group is *dead* if a player can not hinder the opponent from capturing it [1]. If capturing is impossible, a group is considered *unconditionally alive*. The concept of capturing will be explained later on. When the game is finished, territory is counted and the player leading in territory wins the game. This is why each player strives for the maximum of territory on the board. Vacant intersections enclosed by *alive* groups of one color represent territory. [29] One enclosed intersection equals one point. *Dead* stones on enclosed intersections count as extra points for the player who enclosed them. Captured stones also count as points for the player who captured them. Figure 3 shows an example of a finished $9 \times 9$ game with marked territory.

[2] Essentially it is also possible to play Go on smaller or much larger boards. KGS Go Server [51] for example allows the Player to create a new game with $38 \times 38$ intersections at maximum. The sizes mentioned are the most commonly used ones with $19 \times 19$ being the standard size.

[3] Stones are placed on intersections and not on the rectangles.

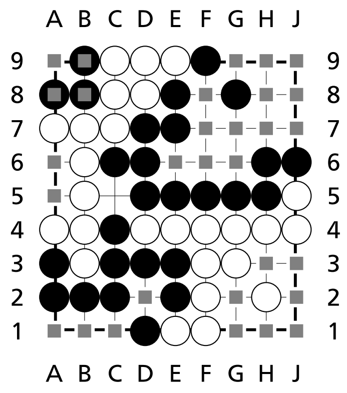


**Figure 3:** An example of a finished 9 × 9 game. A territorial point (indicated by a square) always belongs to the player that completely surrounds it. One territorial point equals one point in the final score. The coordinates A1, B1, and C1 are Black's points, because they are surrounded by black stones. The black group on the coordinates B9, B8, and A8 can be captured by White with a move on A9. Therefore, the group is considered *dead*. Each of these dead stones equals one point for White in the final score. In contrast to that, the intersection at C5 is not marked, because it belongs to neither Black nor White and therefore does not contribute to the final score. Therefore, without counting captured stones or Komi into the final score, Black leads with a single point.

**Liberties and the capturing of stones:** A stone is caught by the opponent if it has no more liberties. The liberties of a stone are its unoccupied adjacent intersections. Adjacent intersections are connected by a line as shown in figure 4.

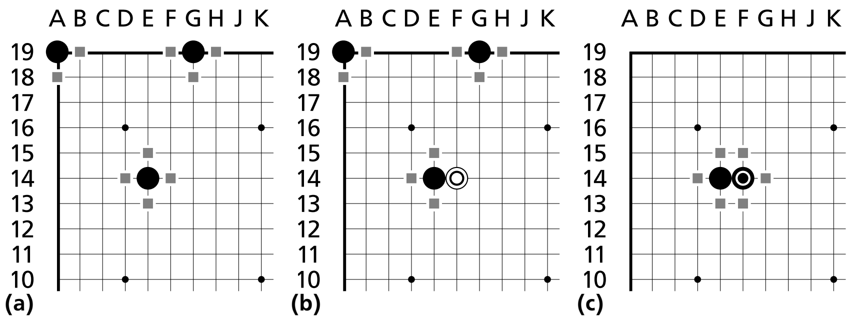


**Figure 4:** Example demonstrating the idea of liberties. The last played move is marked with a circle. The intersections (indicated by a square) around the stone in (a) are its liberties. Adjacent stones occupy the liberties of each other as shown in (b). Two adjacent stones of the same color share their remaining liberties as shown in (c) [26].

Liberties are an essential attribute of a stone. Figure 4 shows that a single isolated stone has two liberties on one of the four corner intersections, three liberties on every other border intersection, and four liberties on the remaining intersections. If another stone is placed beside the stone, both stones occupy a liberty from each other and two stones of the same color share their liberties [26]. This can also be seen in figure 4. If all liberties of a stone (or multiple connected stones) are occupied by the opponent's stones, the stone is captured by the opponent and taken from the board as a prisoner. A captured stone does not take part in the game anymore. It counts as a point for the player who captured it [1]. Figure 5 shows how many stones are needed to capture a single stone on the board. Compared to that, figure 5 also shows that the opponent needs more stones to catch two neighboring stones of the same color because they have two more liberties than an isolated stone.

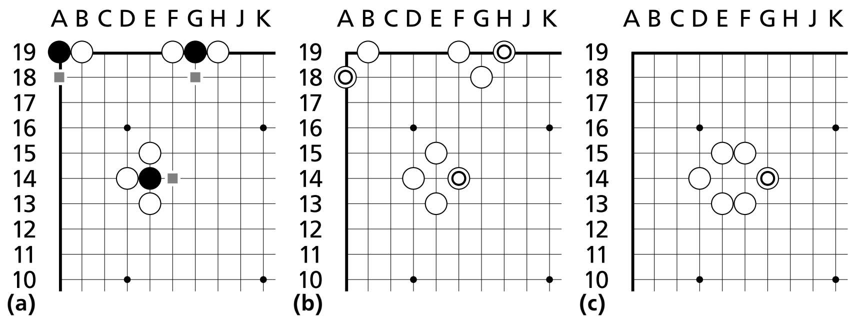


**Figure 5:** This figure shows the idea of capturing. In (a), White took the liberties of Black in figure 4. The last liberty is indicated by a square and the last played move is marked with a circle. If a stone has zero liberties, it is captured and taken from the board as shown in (b). White needs more stones to capture the two black stones in figure 4 as shown in figure (c) [29].

## Suicide

In Japanese rules, it is illegal to place a stone on an intersection, if it would result in zero liberties for the stone. This concept is called *suicide*. The only exception to this rule is, when the placed stone would occupy the last liberty of an opponent's group [1, 26, 67]. The left side of figure 6 shows an example situation where Black is not allowed to play at E14 because the placed stone would have zero liberties after being placed. The right side of figure 6 shows an example situation where it is legal for White to play at D14 because White would capture some stones in the process.

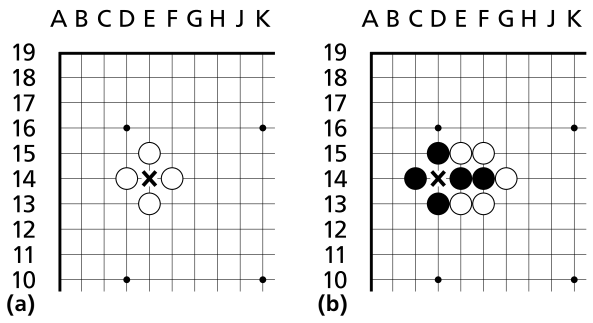


**Figure 6:** It is illegal for Black to play at E14 in (a). But it is legal for Black to play D14 in (b) because it involves capturing E14 and F14 which results in a liberty for the stone in question. This is not considered suicide. [1, 26]

## Ko

The previously stated rules of Go make infinite cycles of game states possible. Such a situation can be seen in figure 7. The game state on the left $s1$ and the game state on the right $s2$ could alternate indefinitely [1, 29]. This situation is called Ko. The situation in figure 7 is called a *direct Ko*, because the cycle does only involve two game states. Every rule set prohibits the infinite cycle of a *direct Ko* [73]. Therefore, a player is not allowed to make a move in state $s2$ that results in a direct re-occurrence of state $s1$. Ko situations can also be larger cycles that involve more than two states. It depends on the rule set how these situations are handled. In Japanese rules, larger cycles like the *triple Ko* can cause the game to be declared void [9, 67].

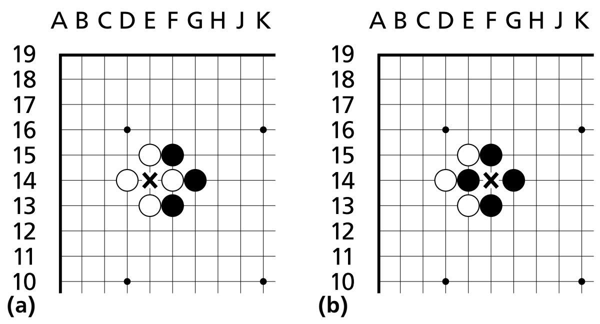


**Figure 7:** This situation is called a *direct Ko*. If Black captures F14 by placing a stone on E14 in (a), White cannot capture back immediately in (b), but has to play a *Ko threat* first. A *Ko threat* is a move elsewhere on the board that the opponent might want to answer rather than ending the Ko.

**Komi**

Black has an advantage by starting the game and placing the first stone on the board. Therefore, White gets compensation in the form of points added to the final score. This compensation is called Komi in Japanese Rules. It is still unclear how large the Komi should be to ensure a fair game for both players. Currently, a Komi of around 6.5 points is considered fair in Japan [70]. The 0.5 points are added to exclude draws.

## 2.2 Rating and Ranks in Go

The rating of a player is a single number that translates directly to a Go rank. The Go rank is a label that indicates the playing strength of a player [71]. A player gains and loses rating points by winning and losing tournament games respectively. The size of gain and loss is dependent on the rating of the opponent. The ranks in Go range from 30 Kyu amateur (30k) to 9 Dan amateur (9d) [71]. Kyu are the student ranks and a lower number corresponds to a higher rank. For example, a 10k player is stronger than a 20k player. Dan are the master ranks where a higher number corresponds to a higher rank. A 1d is stronger than a 1k but weaker than a 2d. Professional Go players range from 1 Dan professional (1d) to 9 Dan professional (9d). Professional players are considered stronger than their amateur counterparts [29]. The European Go Federation [40] has a rating system similar to the ELO system in Chess [41]. It is used for tournaments all over Europe. The difference from one rank to the next are 100 points. [41][4]. The meaning of ranks is not universally precise around the world [72], which means a specific rank does not imply the same playing strength all over the world. The playing strength of two people from different countries with the same rating can vary strongly.

### 2.2.1 EGF Rating Formula

The European Go Federation calculates the rating for each player [43] that plays competitively at European tournaments. The results are stored in the European Go Database [39]. The rating formula is derived from the ELO system used in Chess [41]. It was adopted by the Czech Go Association in 1998. The winning expectancy of the weaker player is

$$S_E(A) = \frac{1}{e^{\frac{D}{a}} + 1} - \frac{\varepsilon}{2}$$

where $D = R_B - R_A$ is the difference in rating and the amount of the constant $a$ determines the influence of $D$. The winning probability of the stronger player is $S_E(B)$ with $S_E(B) = 1 - S_E(A) - \varepsilon$ which is the converse probability of $S_E(A)$ if $\varepsilon = 0$. The variable $\varepsilon > 0$ is a correcting value to balance out deflation. At the moment, the European Go Database uses $\varepsilon = 0.016$ as a correcting value. After an even game the difference in rating is computed with

$$R_{new} - R_{old} = con \cdot (S_A - S_E(D))$$

where $S_A$ is the achieved result and the factor con is the magnitude of the change. The factor $con$ is a multiplier that is anti-proportional to rating. This means that a stronger player will gain less points, when winning against even opponents than a weaker player will, when facing a player of similar playing strength.

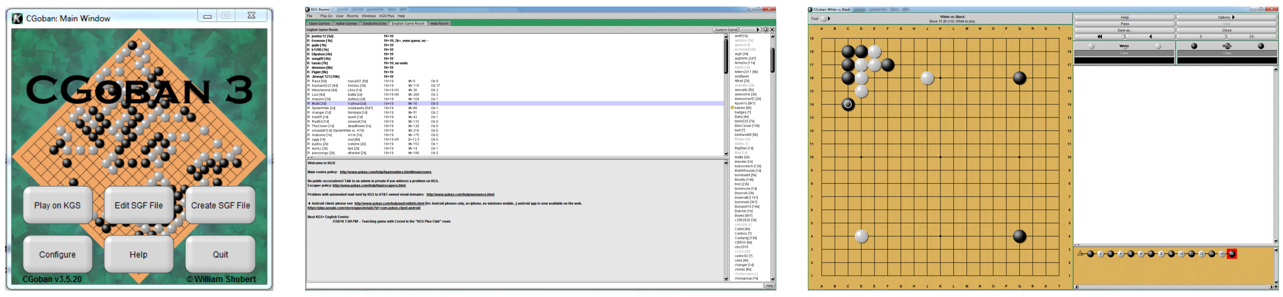
[4] A beginner starts at 100 points (20k), an average 19k has 200 points, and an average 18k has 300 points and so on.

### 2.2.2 Online Servers

There are several online servers where players can play Go with other humans or against Go programs. The focus will be on introducing KGS [66], but there are several other popular online servers like IGS, OGS, TygemBaduk, and WBaduk [57, 58, 75, 76].

**KGS**

KGS, former known as Kiseido Go Server [51], is a popular online server, with more than 1500 people logged in at any time [68]. Besides playing, users can also spectate or discuss other games or give live demonstrations. It is also possible to let Go programs play on KGS via kgsGTP. KgsGTP [69] uses GTP (Go Text Protocol) [46] for communication between KGS and the Go program. KGS can also be used as a simple SGF[5] editor which can be seen on the right in figure



**(a)** **(b)** **(c)**

**Figure 8:** Screenshots of KGS displaying various functions. (a) shows the login screen with various functions. (b) shows a chat room and users that are logged in. (c) shows the ingame screen which can be used to play but also to discuss already played games.

8. KGS game records are often used for creating pattern libraries (Pachi [3]) or as training data (AlphaGo [84]) for Go programs. Furthermore, KGS is often used to test the playing strength of a Go program against human players.

## 2.3 Different Approaches to Computer Go

Go is known as a perfect information zero sum game [74]. It has a very high branching factor and is therefore a much harder challenge for artificial intelligence than for example Chess [29]. Different approaches to computer Go that were used from the beginning until now will be introduced. Computer Go started with pattern recognition, was followed by Monte Carlo Tree Search methods and currently neural networks are on the rise. In 2016, AlphaGo by Google Deepmind, which uses neural networks, was the first program ever beating a professional high Dan player on the $19 \times 19$ board without handicap [84].

### 2.3.1 Pattern Recognition

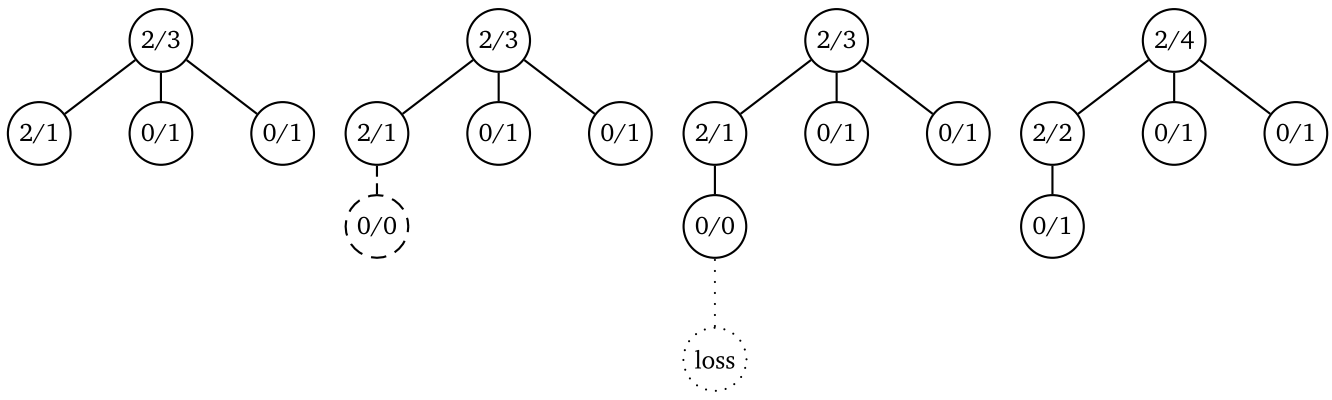
```
6 ..XOO........................
       .
   . . . . .
   . . . . .
 . . O _ O . .
   . . X . .
   . . . . .
       .
```

**Figure 9:** This is an example for the visual representation of a pattern in Go. This specific pattern is used by Pachi [3]. The move to play is represented by the underscore right in the middle of the pattern. White stones are represented by "O"s while black stones are represented by "X"s. The points represent free intersections. The pattern is shown from the viewpoint of the White player. The upper line shows how the actual pattern is stored. The number encodes the coordinates of the subsequent characters. The "6" in this example represents the following sequence of coordinates: (0,0), (0,1), (0,-1), (1,0), (-1,0), (1,1), (-1,1), (1,-1), (-1,-1), (0,2), (0,-2), (2,0), (-2,0), (1,2), (-1,2), (1,-2), (-1,-2), (2,1), (-2,1), (2,-1), (-2,-1), (0,3), (0,-3), (2,2), (-2,2), (2,-2), (-2,-2), (3,0), (-3,0). Source: [2].

[5] Smart Game Format, used for documenting and saving games. More information at http://www.red-bean.com/sgf/sgf4.html

The first Go program ALGOL (1969) [92] played on a 9 × 9 board and had the level of skill as a human player that just learned the rules. It featured a heuristic of visual organization which organized the board into spheres of influence. This was used to distinguish black and white groups. It also used a collection of pattern templates. A pattern template consisted of a small specification of the board situation. For example, a pattern could describe a situation where a black stone with only one liberty could be connected to an outside group. The suggested move would be to connect [92]. A representation of a modern pattern is seen in figure 9. Some templates for ALGOL even used tree search to look up to 100 moves ahead [92]. ALGOL used around 65 patterns without search tree and 20 with tree search [92]. In comparison to that, Pachi [3] can use[6] a pattern library with over 3 000 000 patterns that were generated from KGS games [2]. All pattern templates of ALGOL where applied and all necessary searches were executed by the program before ALGOL decided on the move to play. Overall, the program took a few seconds per move [92].

### 2.3.2 Monte Carlo Tree Search



**(a)** **(b)** **(c)** **(d)**

**Figure 10:** An example of a Monte Carlo Tree Search iteration. From left to right the distinctive steps of *Selection*(a), *Expansion*(b), *Simulation*(c), and *Backpropagation*(d) can be seen. The nodes are all marked with $a/b$ where b is the number of simulations following the position $s$ represented by the node. $a$ is the fraction of $b$ where the *Simulation* resulted in a win. In (a), the node with the best win-loss ratio is selected which is 2/1. In (b), the node is expanded, whereas in (c) the new node ist simulated. After that in (d), the number of wins $a$ and the number of simulation $b$ is updated for the whole path back to the root. In this particular example, the MCTS algorithm favors the child with the best win-loss ratio. This depends on the chosen *tree policy*.

Monte Carlo Tree Search (short MCTS) is an algorithm which iteratively builds a search tree $T$ that consists of state action pairs [10]. A state $s$ represents the current position on the Go board and an action $a$ represents a legal move in state $s$. There are four distinctive steps in each iteration of MCTS: *Selection*, *Expansion*, *Simulation* and *Backpropagation* [10]. In the *Selection* step the algorithm starts at the root and descends through child nodes until it reaches a leaf node that does not represent a terminal state $s_{terminal}$ [10,20]. How the algorithm chooses the child nodes for descent, depends on the *tree policy* used [10]. For example, the algorithm could choose the node with the best win-loss ratio [10]. This node is then expanded with available actions. How this step is carried out, also depends heavily on the chosen *tree policy* [10]. In the *Simulation* step, the algorithm plays against itself following the rules of a certain default policy[7] starting from a newly expanded leaf node until the *Simulation* is finished and a reward $z_i$ is computed [10]. The reward $z_i$ is the outcome of the ith *Simulation*. The action-value-function $Q(s,a)$ calculates the value of a state action pair which approximates the worth of an action in a particular board situation [20]. The function $Q(s,a)$ varies depending on the chosen *tree policy*[8] [10]. In the *Backpropagation* step the algorithm takes the reward $z_i$ and updates $Q(s,a)$ for all nodes in the path from the current node to the root. It also updates $n(s,a)$, the number of simulations in a specific state, starting with action $a$. The number of all simulations starting from s is defined as $n(s) = \sum_a n(s,a)$ [20]. An example for the four steps of MCTS can be seen in figure 10. MCTS repeats its steps until a threshold is reached. After the threshold, which can be either a time, iteration, or memory constraint, the algorithm stops its computation [10]. Then the action $a$, corresponding to the maximum $Q(s,a)$, is chosen [20].

[6] The pattern library is not included by default and has to be downloaded separately. It can be found at: [2]

[7] The policy could, for example, determine that random moves should be played during the *Simulation*.

[8] For example, $Q(s,a)$ could simply calculate the winrate of a state action pair.

**UCT**

UCT (Upper Confidence Bound for Trees) is the UCB1 algorithm adjusted to tree search. The algorithm UCT, which we describe in this paragraph, is introduced in [21] by S.Gelly and Y.Wang. It follows the idea that it is important to have a balance between exploration and exploitation of the *search tree* $T$. Exploration is important to ensure that the favored action of MCTS has not just the highest action value locally. Actions that might seem worse at first will be considered, as their true action value can be higher than that of the favored action. Exploitation, on the other hand, means a more focused search following the action with maximum action value in the tree. To balance exploration and exploitation, UCT incorporates the exploration term $c \cdot \sqrt{\frac{\log n(s)}{n(s,a)}}$ into the action-value function. This term is big for actions that have not been simulated much. The new action-value function is

$$Q_{UCT}(s,a) = Q(s,a) + c \cdot \sqrt{\frac{\log n(s)}{n(s,a)}}$$

with $Q(s,a)$ being the standard action-value function of MCTS. The amount of $c$ decides how much exploration is made in comparison to exploitation. As in normal MCTS the algorithm always selects the action $a$ in state $s$ which maximizes $Q_{UCT}(s,a)$.In other words, the action chosen by the algorithm is $argmax_a(Q_{UCT}(s,a))$.

**Rapid Action Value**

We will describe Rapid Action Value estimation (short RAVE) as it is introduced by S.Gelly and D.Silver in [20]. RAVE uses the *all-moves-as-first* heuristic and is a faster way to estimate the value of an action. The assumption behind RAVE is that actions which often reoccur in simulations or in later stages of the search should be good no matter when they are played. Consequently, the RAVE algorithm considers actions that are further down in the tree or inside the simulation as first moves. It creates and updates a RAVE value for each state action pair every time a simulation that action $a$ was part of is finished. This way the search tree gets broader with many new actions taken into consideration as first moves. Every state action pair gets a RAVE value

$$Q_{RAVE}(s,a) = \frac{1}{n_{RAVE}(s,a)} \sum_{i=1}^{n_{RAVE}(s)} \Gamma_i(s,a) z_i$$

where

$$\Gamma_i(s,a) = \begin{cases} 1 & \text{if action a was selected somewhere in the path following state s} \\ 0 & otherwise \end{cases}$$

is an indicator function. If $\Gamma_i(s,a) = 1$ the reward $z_i$ (win or loss) of the ith simulation is incorporated into the RAVE value. $n_{RAVE}(s,a)$ is the number of simulations used to compute the value $Q_{RAVE}(s,a)$ which is a fast but heavily biased estimate of $Q(s,a)$. Since the estimate is inaccurate RAVE is often used at the beginning of MCTS but is used less with simulations. RAVE can be mixed with normal MCTS [3] as well as with UCT [20].

### 2.3.3 Neural Networks

**AlphaGo**

In 2016, Alpha Go [84], designed by Google Deepmind, defeated Lee Sedol, a strong professional Go player, in a five game match. We will describe AlphaGo in this paragraph as it was introduced in [84] by Silver et al. AlphaGo combines the former state of the art approach of Monte Carlo Tree Search with deep learning neural networks. Those networks are divided into policy networks and a value network [84]. The policy networks are used to lead the search in a particular direction and the value network evaluates a board position. For AlphaGo, a 13 Layer policy network was trained. The policy network was trained with supervised learning on 30 million KGS games. This SL policy ($P_\sigma$) had an accuracy of 57% at predicting expert moves. Next, Reinforcement Learning was used to enhance the policy intending the policy not just to be good at predicting expert moves but rather at finding good moves [84]. To improve it further, the program played against older instances of itself. They selected those older instances at random to prevent overfitting. This RL policy ($P_\rho$) was really successful as it won more than 80% against the original SL policy. It also won around 85% of games against Pachi. In comparison, the SL policy only won around 11% of games against Pachi [84]. The RL policy was used to generate 30 000 000 distinct positions using self play and used those to train a value network $V_\theta$ which evaluates a position $s$ and predicts the outcome of a game if both players use a certain policy $P$. AlphaGo was most successful when the evaluation of a position was composed out of the outcome of the value network $V_\theta$ and the reward of the *Simulation* $z_i$ forming a leaf evaluation

$$V_\alpha(s_L) = (1-\lambda)v_\theta + \lambda z_i.$$

AlphaGo combines its trained policy network $P_\sigma$ and its trained value network $V_\theta$ (incorporated in $V_\alpha$) with standard MCTS. It is important to recognize that the RL policy $P_\rho$ was only used to train the value network and that $P_\sigma$, the SL network policy, performed better against human players. $P_\sigma$ indirectly influences the search[9] to guide the MCTS search into a favorable direction. The influence of the policy on actions decays with visits to these actions to encourage exploration. The trained value network $V_\theta$ is used in the action value function $Q_\alpha$ as the function is constructed as

$$Q_\alpha(s,a) = \frac{1}{n(s,a)} \sum_{i=1}^{n} \phi(s,a,i) V(s_L^i)$$

where $\phi(s,a,i)$ is an indicator function, $\phi(s,a,i) = 1$, if the state action pair has been traversed during the ith simulation and $\phi(s,a,i) = 0$ otherwise. AlphaGo always chooses the action $a$ that was most visited during the search.

## 2.4 Composition of this Thesis

This thesis will introduce the *big move* heuristic for generating Monte Carlo Tree Search *Prior Knowledge* [19]. The *big move* heuristic approximates big moves in Fuseki, the beginning stage of a game. The heuristic assumes that it is important to play in underdeveloped areas during Fuseki. There are many possibilities at the beginning of a game [25]. This is one reason why programs often struggle with it [18]. The thesis is divided in three distinct parts. First, we will introduce the idea and setup of the *big move* heuristic in section 3. After that, we will test different eigenvector functions to optimize the time which the algorithm behind the heuristic needs to compute a result. We will introduce different functions from different languages and discuss the results in section 4. Last, we will incorporate the heuristic into Pachi, an open source Go program. We will explain roughly how the game enine of Pachi works. Then, we let the modified version Pachi* with the incorporated *big move* heuristic play against regular Pachi in various tests to see if Pachi benefits from incorporating the big move heuristic. The tests and results can be seen in section 5.

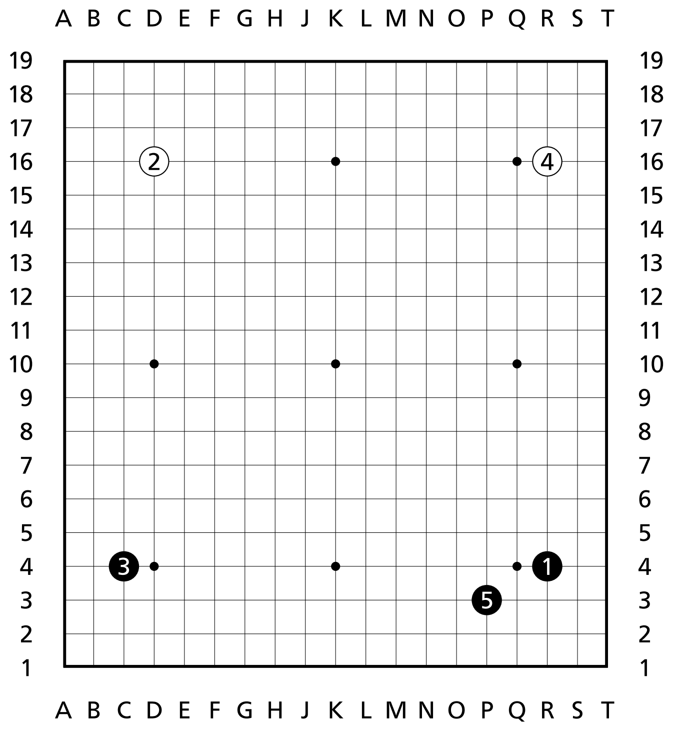
[9] AlphaGo adds a bonus to each state action value that is proportional to the probability proposed by the policy.

## 3 Introducing the Heuristic and its Underlying Algorithm

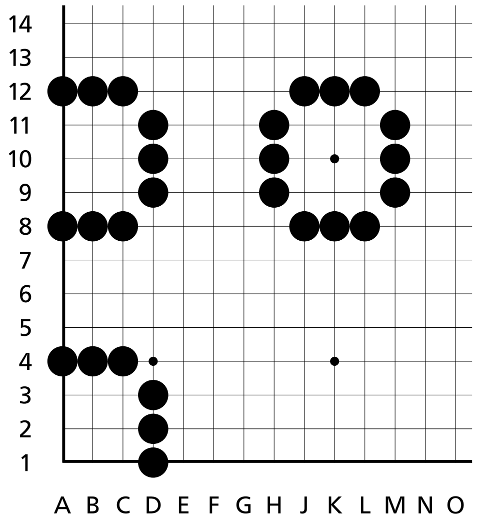
In this section, the *big move* heuristic will be presented. The heuristic was first introduced at the *Conference on Applications of Graph Spectra in Computer Science* by Josef Leydold and Manja Marz [27].
First, it has to be defined what a *big move* is. Figure 11(a) shows a typical Fuseki position. The reader might wonder why none of the players places a stone in the middle of the board. This is because it is advisable to play first in areas that can be developed into territory more easily. In the corners where both players played first a player can use two borders to build territory. On the side of the board, he can use one and in the middle he has none to build with [8, 26, 29]. Figure 11(b) illustrates this exemplarily. This is why the middle of the board is often left empty in Fuseki. However, there are special strategies in Fuseki involving intersections farther away from the border. Therefore, we will not rule out these moves completely. Another aspect is *efficiency*. Each player strives for the maximum of territory by using the minimum of stones. Players should therefore play away from the opponent's strength[10] as it is ineffective and sometimes dangerous to play near it. Placing stones close to the own strength is considered ineffective as well [25]. The assumption for the heuristic is

"Play where most space is left." [27].

This is why, in this context, a *big move* is a move played in an undeveloped area. The *big move* heuristic finds the intersection farthest away from the border and from any stones already on the board. It is important to note, that the heuristic is designed for Fuseki only. Over the course of the game, the size of undeveloped areas shrinks due to the increasing number of stones on the board.



**(a)** This is an example for the Fuseki of a game. The numbers indicate the order in which the moves were played starting with move one. First, the players play in the corners of the board because corners are considered to be areas where less stones are needed to develop alive groups. Then, the next biggest area is the side. The smallest area is the middle.



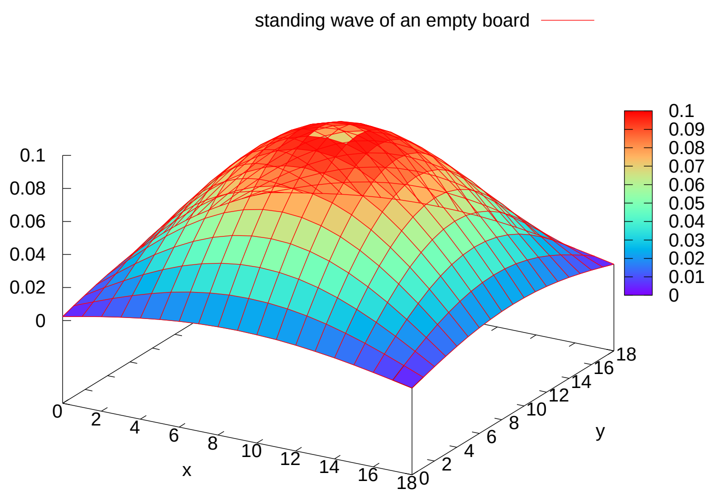
**(b)** This example illustrates how many stones a player needs to build a 9 point territory in different areas on the board. Therefore, players tend to build territories involving the border first as this is far more efficient [8].

### 3.1 Explanation of the Algorithm

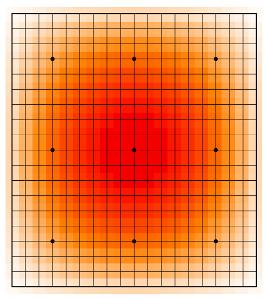
To understand the idea behind the algorithm of the heuristic the board may be pictured as a flexible grid. The lines between intersections are represented by springs or any other flexible element. Stones already placed on intersections fix those intersections into place, therefore decreasing movement in those areas. The board is clamped into a rigid border by more springs [27]. Now, the border is set in motion so that the whole board oscillates in its fundamental mode (at a

[10] Strength is an abstract concept which describes effective and secure groups of stones.

natural frequency)[11]. Mode 1 is wanted, as it is the mode with only one half wave in the vibration and therefore has a definite maximum in amplitude [23,81]. The standing wave can be seen in figure 11. The algorithm finds the intersection that has the highest amplitude as this is the move the heuristic chooses. It is important to note that the algorithm does not distinguish between black and white stones. This may or may not change in the future.



(a)



(b)

**Figure 11:** (a) demonstrates the desired standing wave of an empty board (seen in (b)). The color indicates the amplitude of an intersection. The board could also oscillate in a different mode but that would result in more half waves.

Figure 11 shows the desired mode shape of an empty board in mode 1. In this case, the algorithm chooses Tengen[12] as the move to play.

## 3.2 Mathematical Background of the Algorithm

The board position is represented by a graph $G$. The intersections are the edges $v_i$ and the lines connecting intersections are the edges $e_{i,j}$ where $i \in [1,361]$ and $j \in [1,361]$. Additional boundary vertices [4] are added as seen in figure 12.

The graph is represented by a Dirichlet matrix which belongs to the class of generalized Laplacians [4]. A matrix M is a generalized Laplacian if

1. it is symmetric
2. $M_{x,y} < 0$ whenever there is an edge between x and y
3. $M_{x,y} = 0$ if x and y are distinct and not adjacent

where x and y are vertices of M.

[11] Another word for fundamental mode is mode 1. In mode 1 there is only one half wave in the vibration. [23]
[12] The central point of the board at coordinate K10.

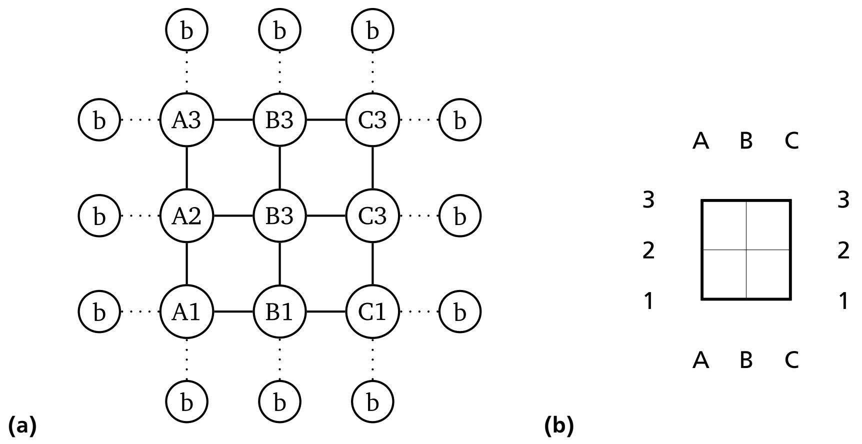


**Figure 12:** The graph in (a) represents a 3 × 3 Go board (seen in (b)). Each of the vertices (marked with a coordinate) represents an intersection on the board. Additional boundary vertices (marked with a $b$) are added. Boundary vertices are not connected to each other. The corresponding boundary edges are represented by dotted lines while the inner edges are represented by solid lines.

The Dirichlet matrix is closely related to the ordinary Laplacian matrix. The Laplacian matrix L is defined as [90]:

$$L_{i,j} := \begin{cases} deg(v_i), & \text{if } i = j \\ -1, & \text{if i adjacent to j and } i \neq j \\ 0, & \text{else} \end{cases}$$

In this context, $deg(v_i)$ is the degree of a vertex also known as the number of adjacent vertices of $v_i$. The reader might recognize the similarity to the more known Adjacency matrix. Another definition of the Laplacian matrix is the following [90]:

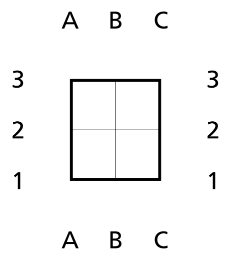
$$L = deg(v_i) \times I - A$$

where I is the Identity matrix and A is the Adjacency matrix.

The Laplacian Matrix already represents the Go board very well. It defines which intersections are connected with each other and how many neighbors an intersection has. The Dirichlet matrix for our system can be constructed by first constructing a Laplacian matrix for the graph in figure 12. The represented go board has an additional border which corresponds to the added boundary vertices. Then we delete all rows and colums that correspond to these boundary vertices [4]. The resulting $361 \times 361$ Dirichlet matrix therefore represents a graph that has no direct edges to its neighboring boundary vertices but they still contribute to the degree of its adjacent vertices. This graph can be seen in figure 12. An example for the Dirichlet matrix of a 3 × 3 Go board can be seen in figure 13. Relating to a Go board, this means that we have an imaginary zeroth line which functions as the rigid border mentioned in the beginning. The Dirichlet matrix therefore represents a graph that contains boundary vertices and edges that function as non-vibrating elements on the Go board.

$$A_{m,n} = \begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{pmatrix}$$

(a)



(b)

**Figure 13:** (a) shows how the Dirichlet matrix would look like representing the 3 × 3 Go board in (b). The matrix for the 19 × 19 Go board is 361 × 361 and would therefore be too big to display. However, the general structure is the same, except that the 361 × 361 matrix is much sparser than in (a) as it has the same amount of nonzero entries per row. In contrast to the more known Laplacian matrix, all border intersections of the represented graph have four neighbors (see figure 12) although, for example, in (b), only the point in the middle of the board has four neighbors and all the other points have three or two neighbors.

**Adding Boundary Vertices to the Graph**

Stones can be added to the board to influence the vibration. The intersections occupied by stones will oscillate less, as they are connected with a spring to a non vibrating border [27]. This happens when we connect additional boundary vertices to the graph. To connect a boundary vertex to a specific inner vertex we have to increment the diagonal entry of the matrix which corresponds to that vertex in the graph, by one. Later, it will be demonstrated how this method can be used to influence the heuristic's outcome.

## 3.3 The Purpose of Eigenvector Computation

An eigenvalue is any scalar $\lambda$ for which the equation

$$A \times v = \lambda \times v$$

is true. [85] $A$ is a matrix and $v$ is the corresponding eigenvector of $\lambda$. An eigenvector is a special kind of vector that never changes its direction. The eigenvalue is its scaling value. Thus, the equation above says that $A$ scales $v$ the same way as $\lambda$ scales $v$. Each eigenvalue $\lambda$ has a corresponding eigenvector $v$. This is relevant in the domain of vibration analysis. When analyzing an oscillating system, its eigenvalues represent the frequencies in which the system can vibrate. The eigenvectors, on the other hand, represent the different mode shapes. Together they form a natural mode of vibration [23, 81]. In this case the system represented by the Dirichlet matrix oscillates in its fundamental mode [23] and the eigenvector $v_s$ representing its mode shape is chosen. This is the eigenvector corresponding to the least dominant eigenvalue of the system [86]. Every entry of the eigenvector $v_s$ corresponds to an intersection on the board. So the mode shape of figure 11 is exactly the least dominant eigenvector $v_s$ of a Dirichlet matrix that represents an empty board. As the eigenvector $v_s$ represents the first mode shape, it will either be completely positive or completely negative. This can be proven by applying the theorem of Perron-Frobenius [4] to the matrix.

**Theorem 1** (Perron-Frobenius). *If $A \in \mathbb{R}^{n\times n}$ is a nonnegative, irreducible and symmetric matrix, then the spectral radius*[13] *is a simple eigenvalue $\lambda_{pf}$ of A. Its corresponding eigenvector $v_{pf}$ has no zero entries and all entries have the same sign.*

$v_{pf}$ is called the perron vector of A [4]. The theorem can be applied to our matrix A although A is not nonnegative because A can be transformed to a matrix B which is nonnegative but has the same spectrum.

[13] dominant eigenvalue

*Proof.* Matrix $B$ is defined as as

$$B = -(A - m \times I)$$

where I is the identity matrix and m is the largest entry on the main diagonal. Because A has positive diagonal entries and only zeros elsewhere, B is nonnegative. The graph that B represents is also irreducible. Therefore, the theorem of Perron-Frobenius can be applied to B. Be $\lambda_{pf}$ the dominant eigenvalue of B and $v_{pf}$ its corresponding perron vector. Then, one has

$$Bv_{pf} = \lambda_{pf} v_{pf} = -(A - mI)v_{pf}$$

$$-(A - mI)v_{pf} = -Av_{pf} + mv_{pf}$$

and therefore

$$-\lambda_{pf} v_{pf} + mv_{pf} = Av_{pf} = (-\lambda_{pf} + m)v_{pf}.$$

This shows that A has an eigenvalue $(-\lambda_{pf} + m)$ that has the same correspondent eigenvector $v_{pf}$ that is the perron vector of B. Because of the form $(-\lambda_{pf} + m)$ of the eigenvalues of B, the most dominant eigenvalue of B becomes the least dominant eigenvalue of A. This is why the eigenvector corresponding to the least dominant eigenvector of A is in fact a perron vector and therefore has either completely positive or completely negative entries (depending on the eigenvalue). □

Now, after obtaining the eigenvector $v_s$ that represents the mode shape of the oscillating system in its fundamental mode, the first index $i$ that corresponds to $max|e_i|$, where $e_i$ is an entry of $v_s$, is chosen.

## 3.4 Demonstration of the Algorithm

To demonstrate whether the heuristic chooses reasonable moves, a typical Fuseki board position was chosen to compare the heuristic's move to the moves from a Go database[14]. Figure 14 shows all the moves that professionals played in this situation in games that were stored in the database. The result of our algorithm without additional potential [4, 27]

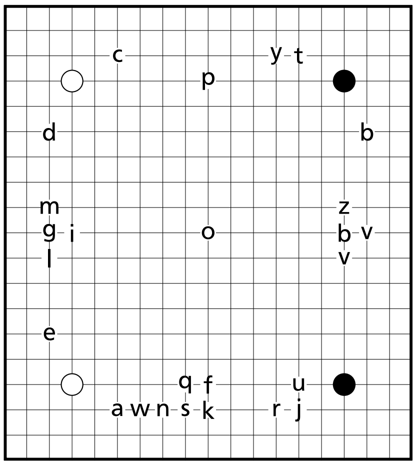


**Figure 14:** The different moves a to z that were played in this position by professionals according to the database weiqi.tools [77].

is seen in figure 15. It can be seen that the move suggested by our algorithm is indeed a move that was chosen by professionals in this particular position. It is clear why that move was chosen by the heuristic as it was the intersection farthest away from other stones. It is important to point out that the example in figure 14 is a well chosen example to show that the heuristic can predict meaningful moves. In most scenarios, however, the heuristic will suggest moves that professional players will not play. This is on one hand due to the fact that it is a move that does not regard aspects of the game like the color of stones or where to build territory first (figure 11(b)). On the other hand, in a game local fights often demand urgent moves and situations arise where a player can not afford to play elsewhere in an undeveloped area. So it is important to keep in mind that the heuristic just roughly predicts the region of interest more than the actual best move. Still, with further research, the heuristic might improve. To improve the heuristic further, potential was added to the Dirichlet matrix. First, potential was added to the diagonal entries that correspond to the seventh line on the

[14] weiqi.tools [77]

board. Next, additionally potential was added to the entries corresponding to the sixth line. The potential had the same weight as one stone per intersection. The result of the algorithm on the modified matrices is shown in Figure 16. In Fuseki, adding stones to the seventh line most likely forces a move on the fourth line while additional stones on the sixth line most likely force a move on the third line. Those two lines are really important lines in the Fuseki according to Go theory [29]. The additional stones on the board are represented by boundary vertices which are added to the graph.

The reader might wonder why a specific move was chosen over other equally big moves. This is due to the fact that at the moment the heuristic always chooses the first maximum entry of the vector. It is interesting how the heuristic changes its decision with varying potential as seen in figure 17.

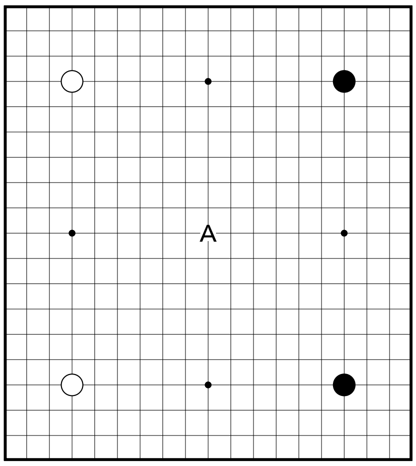
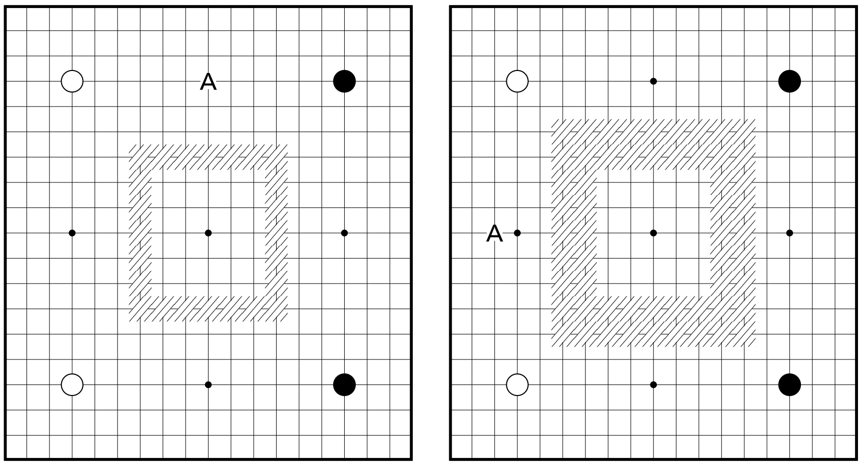


**Figure 15:** Suggested move $A$ of the heuristic on a board without added potential.



**(a)** **(b)**

**Figure 16:** Suggested move $A$ of the heuristic when adding stones on the seventh line (a) versus adding stones on the sixth and seventh line (b). The intersections with added potential are marked.

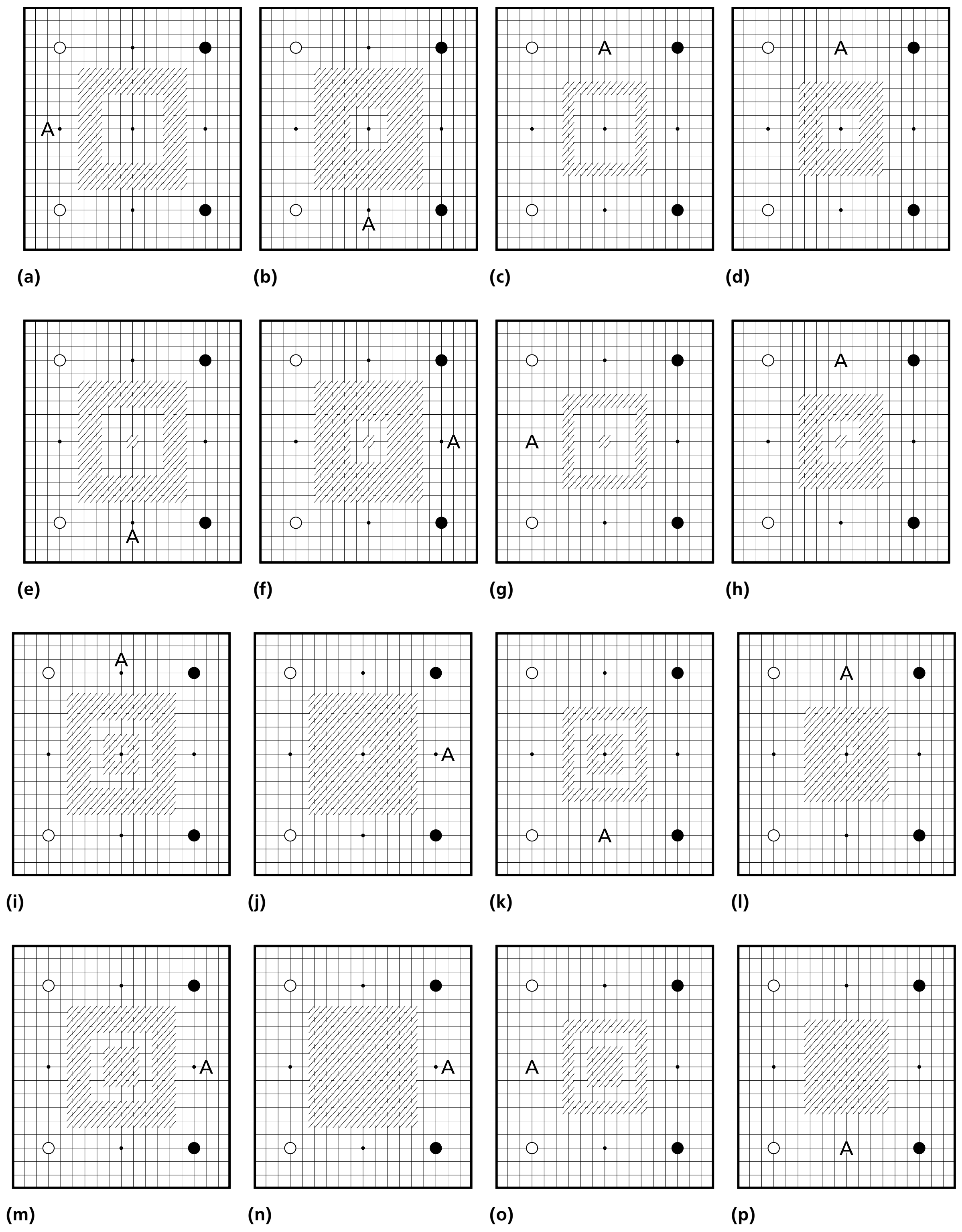
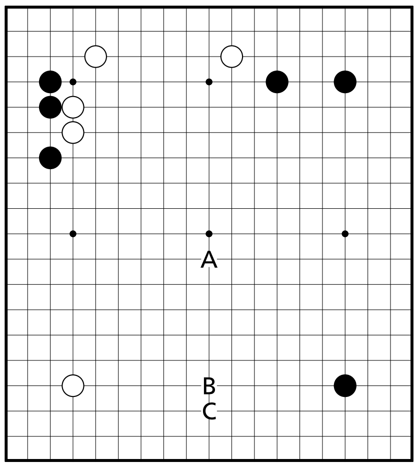


**Figure 17:** This figure shows an example of how different circles of stones influence the result of the heuristic. The intersections with added potential are marked. The heuristic always chooses the smallest index $i$ corresponding to $max|e_i|$, where $e_i$ is an entry of the eigenvector $v_s$

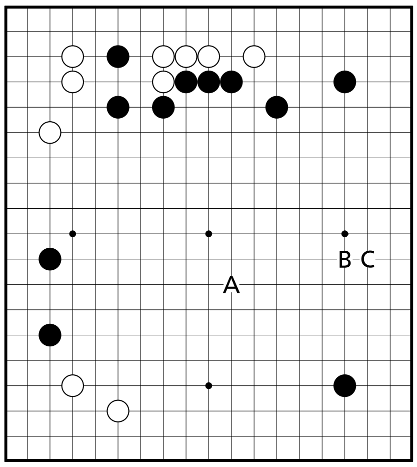
**Further Examples**

We took several positions[15] to show exemplarily how the current algorithm behaves.

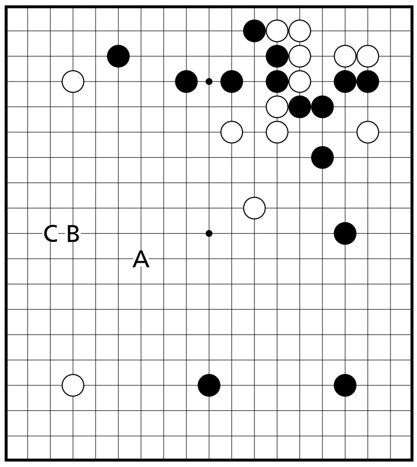
[15] Source:"A Dictionary Of Modern Fuseki, The Korean Style" [28]



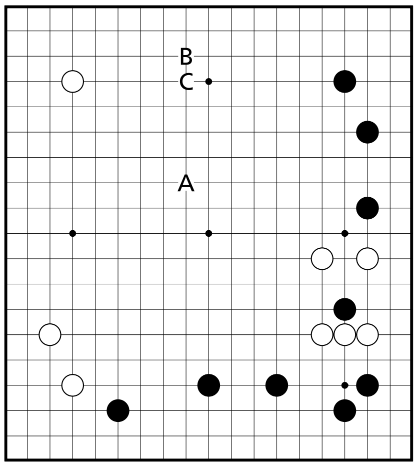
(a)



(b)



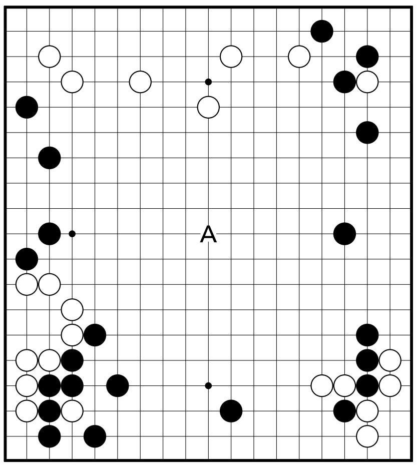
(c)



(d)

**Figure 18:** This figure shows several Fuseki positions. The move marked with $A$ is the move suggested by the heuristic if no additional potential is used. $B$ is the move the heuristic proposes if potential is added on the seventh line. $C$ is the move suggested by the heuristic if potential is added to the sixth and seventh line.

Figure 18 shows different typical Fuseki positions. The behaviour of the heuristic is very similar in all examples. Figure 19 shows a position at the end of Fuseki. Because the third and fourth line of the board are relatively occupied the middle of the board becomes big again. Therefore the middle consisting of the eighth to tenth line of the board oscillates the most, even if potential is added to the sixth or seventh line. It is unclear if this behavior is wanted or not, but it should be taken into consideration when adjusting the potential further.



(a)

**Figure 19:** This figure shows a board position at the end of Fuseki. $A$ is the move that was chosen by the heuristic on the matrix without added potential, the matrix with added potential on the seventh line, and the matrix with added potential on the sixth and seventh line.

## 4 Perfomance Comparison of Eigenvector Functions

Eigenvector calculation is a time consuming part of the *big move* heuristics algorithm because of the size of the Dirichlet matrices. In this section we compare different methods for the eigenvector calculation of a matrix. This functions/methods were chosen from different programming languages. We also test several self implementations. The time was measured using the bash time [82] command, and we use the user time for comparison. We chose the bash time command because it can be applied regardless of programming language. This makes the functions more comparable. We measured the time the program needed to calculate at least one eigenvector[16]. The model we measured the time on was

$$\text{Intel(R) Core(TM) i7 CPU L 640 @ 2.13GHz} .$$

The time was measured single-threaded and therefore only used one of the CPU cores.
We measured the time for one program call. Because the time of one calculation may vary from one computation to the other we measured the time for a different number of calls of each function. We choose to measure $i$ iterations for each function where i is either 1,5,10,50,100,500 or 1000. This makes the result more steady. We decided on three different board positions for performance testing. Our first matrix, $A$, is a Dirichlet matrix representing the graph of an empty Go board position[17]. The second matrix, $B$, is a Dirichlet matrix representing the graph of a Fuseki board position[18] where a few stones were already placed. The last Dirichlet matrix,$C$, represents the graph of an endgame position [44] where many stones have been already placed. We chose A as a minimal test example, B as an example of a typical target matrix for our algorithm and C as an extreme case with many nonzero entries on the diagonal. We remind the reader that all matrices only differ on the diagonal and are apart from it identical. We transformed each board position into a Dirichlet matrix without adding any additional potential. The Definition of a Dirichlet matrix and the concept of potential are explained in section 3. It may be of importance to take into account that all three matrices are moderately big, square, symmetric, real and sparse[19].

[16] Some functions compute all eigenvalues and corresponding eigenvectors.

[17] This is the beginning position where no player has placed a stone yet.

[18] The experienced Go player might recognize that the example looks “unnatural”. This is due to the fact that this position was constructed for test purposes only.

[19] The Dirichlet matrices we use are very sparse with having $\approx 5 \cdot 361 = 1805$ non zero entries at maximum which is around 1% of the matrix.

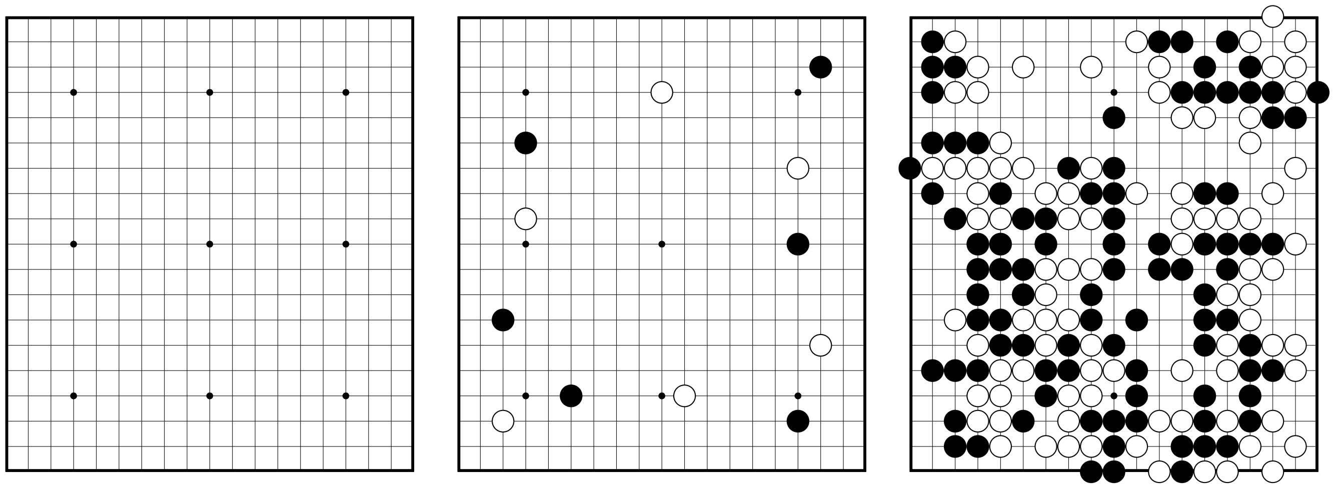


**Figure 20:** We use three different board positions for testing. An empty board is shown in (a), a Fuseki position is shown in (b), and an endgame position can be seen in (c). Source of (c): [44]

For each method, we calculated

- the arithmetic mean for one call of the program in which the method is executed exactly one time, $\bar{x}(1) = \frac{t(A,1)+t(B,1)+t(C,1)}{3}$ where $t(M,1)$ is the measured time for one iteration on matrix $M$.
- the average time for one program call of the program and one iteration of method on a particular matrix, calculated from the time for 1000 iterations, $\bar{x}(M,1000) = \frac{t(M,1000)}{1000}$ for a particular matrix $M$.
- the arithmetic mean composed out of the averages from each matrix $\bar{x}(M,1000)$, $\bar{x}(1000) = \frac{\bar{x}(A,1000)+\bar{x}(B,1000)+\bar{x}(C,1000)}{3}$.
- the standard deviation $\sigma(1000) = \sqrt{(\frac{(\bar{x}(A,1000)-\bar{x}(1000))^2+(\bar{x}(B,1000)-\bar{x}(1000))^2+(\bar{x}(C,1000)-\bar{x}(1000))^2}{3}}$ and the coefficient of variant $cv = \frac{\sigma(1000)}{\bar{x}(1000)}$.
- the approximated average time used for other tasks of the program $\bar{\epsilon} = \bar{x}(1) - \bar{x}(1000)$.

We calculate the arithmetic mean because we feel that one value is easier to compare and we think that the arithmetic mean represents the overall performance well. We calculate the coefficient of variance as it can be used to compare the robustness[20] of each method. Lastly, we compare the approximated average time for other tasks of the program, which is a rough estimate on how many percent of the program were used for other tasks like file loading. Next, we will show the results of our tests. Later, we will discuss which function performed best. First, we will test direct methods as they are the most commonly available methods.

## 4.1 Direct Methods

First we tested direct methods. By direct method we mean a method that directly computes all eigenvalues and eigenvectors.

### 4.1.1 Performance of *eigen()*

The original program which implemented the heuristic was written in R and used *eigen()*, a function that uses the following routines from LAPACK[21]: *DSYEVR* [53], *DGEEV* [52], *ZHEEV* [55], and *ZGEEV* [54] [59]. These fortran routines use eigendecomposition [89] to compute eigenvectors for real symmertic, real $n \times n$ nonsymmetric, complex hermitian or $n \times n$ complex nonsymmetric matrices [59]. We set the parameter *symmetric* to TRUE, so that *eigen()* only uses the lower triangle of the matrix for computation [59].

Figure 21 shows how *eigen()* performed on the three matrices A, B, C. The measured times for a single call of the program and one iteration of *eigen()* on each matrix are

[20] By robustness we mean the ability of a method to compute equally fast on different matrices.

[21] The Linear Algebra Package (LAPACK) is a software library originally written in fortran77. For more information visit: [47]

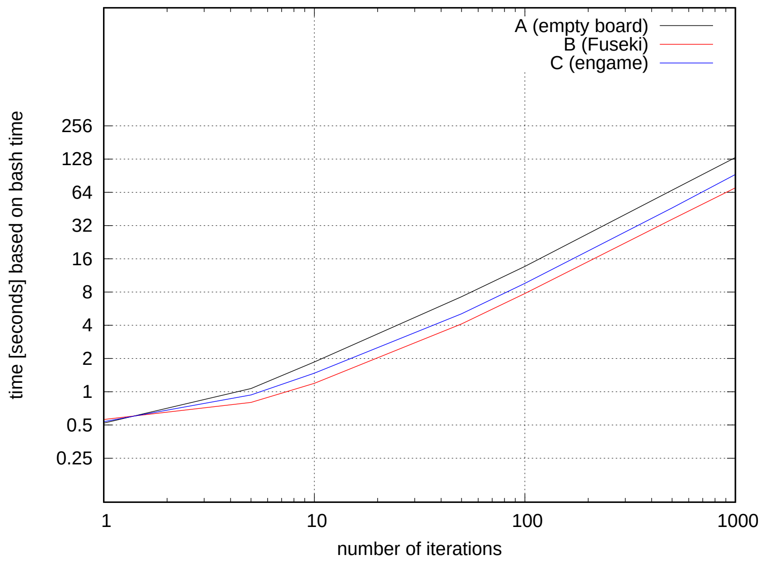


**Figure 21:** The performance of *eigen()* on the matrices A, B, and C.

$t_R(A,1) = 0.52$ seconds, $t_R(B,1) = 0.56$ seconds, and $t_R(C,1) = 0.536$ seconds. The corresponding arithmetic mean of one call is $\bar{x}_R(1) = 0.538667 \approx 0.539$ seconds. Furthermore, we present the times for a thousand iterations and their corresponding averages for one iteration

$$t_R(A,1000) = 132.34 \text{ seconds with } \bar{x}_R(A,1000) = 0.13234 \text{ seconds,}$$

$$t_R(B,1000) = 70.38 \text{ seconds with } \bar{x}_R(B,1000) = 0.07038 \text{ seconds,}$$

$$t_R(C,1000) = 92.832 \text{ seconds with } \bar{x}_R(C,1000) = 0.092832 \text{ seconds.}$$

With these averages we calculated an total average of $\bar{x}_R(1000) = 0.098517 \approx 0.099$ seconds. The correspondent standard deviation is $\sigma_R(1000) = 0.025613$ seconds which amounts to a coefficient of variance of $cv_R(1000) = \frac{\sigma_R(1000)}{\bar{x}_R(1000)=0.25998}$. Lastly, we computed $\bar{\epsilon}_R = \bar{x}_R(1) - \bar{x}_R(1000) = 0.44015$ seconds which is an approximation of the average time the program needed for other tasks. This time amounts to roughly 82% of the program. The reader might see that the measured times for a thousand iterations vary heavily depending on the matrix. This can also be seen when looking at the coefficient of variance $cv_R(1000)$. $\bar{x}_R(1000)$ shows that *eigen()* roughly took the tenth of a second for one computation. We will use this result to compare it to the other programs in this section. The 82% $\bar{\epsilon}$ takes up of the complete time indicates that a a great part of the programs time was composed of other tasks like file loading and matrix preparation. But ensuring this would need further investigation.

### 4.1.2 Performance of the *EigenSolver*

The next program we tested was written in C++ and used the *EigenSolver* of the Eigen Library [35]. The implementation of the EigenSolver is adapted from JAMA[22] [35]. The code from JAMA [15] is initially based on EISPACK [79], a collection of Fortran subroutines [35]. The *EigenSolver* uses Eigendecomposition [89] to achieve the eigenvalues and vectors. We measured $t_{cpp}(A,1) = 0.42$ seconds, $t_{cpp}(B,1) = 0.54$ seconds, and $t_{cpp}(C,1) = 0.56$ seconds for one program call and one iteration of the EigenSolver, as well as their corresponding arithmetic mean of $\bar{x}_{cpp}(1) = 0.506667 \approx 0.507$ seconds. This result of $\bar{x}_{cpp}(1)$ is similar to the result in R but not informative on its own.
The measured times for 1000 iterations and their corresponding averages for one iteration are

$$t_{cpp}(A,1000) = 406.79 \text{ seconds with } \bar{x}_{cpp}(A,1000) = 0.40679 \text{ seconds,}$$

$$t_{cpp}(B,1000) = 506.78 \text{ seconds with } \bar{x}_{cpp}(B,1000) = 0.50678 \text{ seconds,}$$

$$t_{cpp}(C,1000) = 408.12 \text{ seconds with } \bar{x}_{cpp}(C,1000) = 0.40812 \text{ seconds.}$$

[22] A basic linear algebra package for the Java programming language

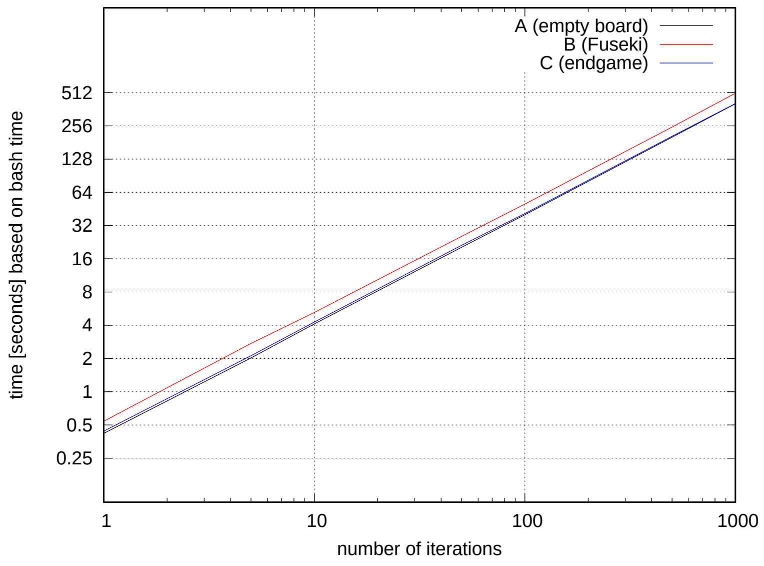


**Figure 22:** Performance of the EigenSolver on the test matrices A, B and C.

The arithmetic mean composed of all averages is $\bar{x}_{cpp}(1000) = 0.440563 \approx 0.441$ seconds with a standard deviation of $\sigma_{cpp}(1000) = 0.046825 \approx 0.047$ seconds. The coefficient of variation is therefore $cv = \frac{\sigma_{cpp}(1000)}{\bar{x}_{cpp}(1000)} = 0.106285$. The program therefore seems more steady than eigen(R) in R. The approximated average time for other tasks amounts to $\bar{\epsilon}_{cpp} = \bar{x}_{cpp}(1) - \bar{x}_{cpp}(1000) = 0.066104 \approx 0.066$ seconds. If we divide $\bar{\epsilon}$ by $\bar{x}(1)$ we get an amount of around 13%. When comparing the average mean of the *EigenSolver* ($\bar{x}_{cpp}(1000)$) with the arithmetic mean of *eigen()* ($\bar{x}_R(1000)$) in section 4.1.1, it is clearly visible that the *EigenSolver* performed worse as it is more than the quadruple of the time of *eigen()*. The amount of $\bar{\epsilon}$ indicates that other tasks absorbed less time in the C++ implementation than in the R implementation. But further investigation would be necessary to be sure.

### 4.1.3 Performance of *gsl_eigen_symmv()*

The next program was written in C, using *gsl_eigen_symmv()*, a function of the Gnu Scientific Library that computes all eigenvalues and eigenvectors of a real symmetric matrix [45]. It is not stated which method the function uses.

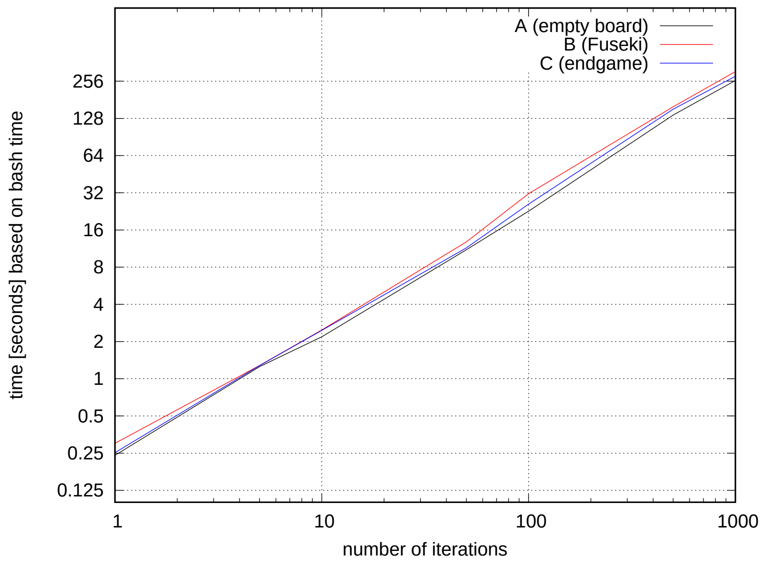


**Figure 23:** Performance of *gsl_eigen_symmv()* on the matrices A, B and C shown in figure 20

We measured the time for one program call and one iteration of *gsl_eigen_symmv()*. The times for each matrix are $t_{gsl}(A,1) = 0.252$ seconds, $t_{gsl}(B,1) = 0.300$ seconds, and $t_{gsl}(C,1) = 0.240$ seconds. The arithmetic mean of those times is $\bar{x}_{gsl}(1) = 0.264$ seconds. For one program call and 1000 iterations on the other hand we measured the following times and calculated the average times for one iteration

$$t_{gsl}(A, 1000) = 280.736 \text{ seconds with } \bar{x}_{gsl}(A, 1000) = 0.281736 \text{ seconds,}$$

$$t_{gsl}(B, 1000) = 306.116 \text{ seconds with } \bar{x}_{gsl}(B, 1000) = 0.306116 \text{ seconds, and}$$

$$t_{gsl}(C, 1000) = 258.824 \text{ seconds with } \bar{x}_{gsl}(C, 1000) = 0.258824 \text{ seconds.}$$

We calculated the arithmetic mean of these average times $\bar{x}_{gsl}(1000) = 0.281892 \approx 0.282$ seconds. We also computed the standard deviation $\sigma_{gsl}(1000) = 0.019324 \approx 0.020$ seconds, together they form the coefficient of variance $cv_{gsl}(1000) = \frac{\sigma_{gsl}(1000)}{\bar{x}_{gsl}(1000)} = 0.068552$. The approximated average time for other tasks $\bar{\epsilon}_{gsl} = \bar{x}_{gsl}(1) - \bar{x}_{gsl}(1000) =$ -0.017892 seconds is negative, meaning that $\bar{x}_{gsl}(1) < \bar{x}_{gsl}(1000)$. We assume this is because of the repeated reallocation of workspace for this method. $\bar{\epsilon}_{gsl}$ can therefore not be considered meaningful.

### 4.1.4 Performace of SciPy and NumPy Routines

**Performance of *numpy.linalg.eig()***

The next program was written in Python and used *numpy.linalg.eig()*, a function of the NumPy package [56]. It uses the LAPACK [47] routine _geev [48] which computes the eigenvalues and optionally corresponding eigenvectors for a $n \times n$ non symmetric matrix [61].

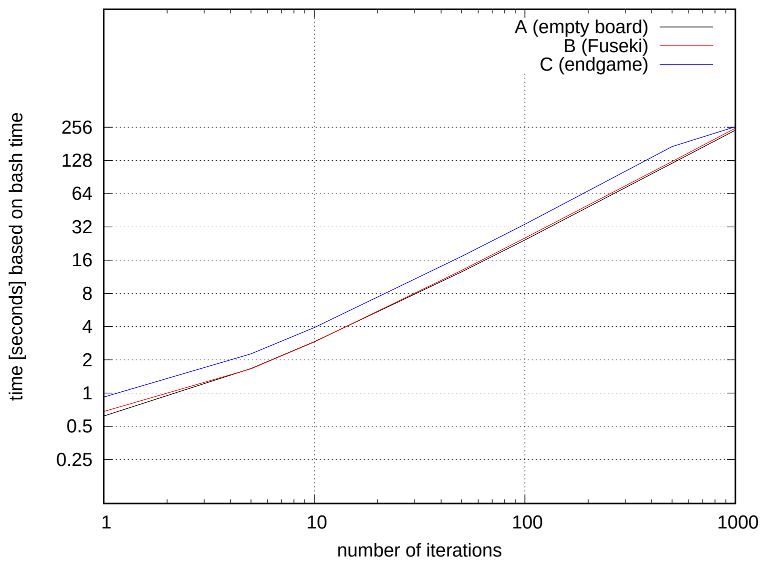


**Figure 24:** Performance of *numpy.linalg.eig()* on the matrices A, B, and C proposed in figure 20

Analogously to the last programs introduced we present the following results. We measured $t_{neig}(A, 1) =$ 0.62 seconds, $t_{neig}(B, 1) =$ 0.68 seconds, and $t_{neig}(C, 1) =$ 0.920 seconds for one call of the program in which *numpy.linalg.eig()* was executed once. The arithmetic mean of this data is $\bar{x}_{neig}(1) =$ 0.74 seconds. We also measured the times for one call of the program with a thousand executions of *numpy.linalg.eig()* and calculated the average time for one execution of *numpy.linalg.eig()* with it. The results are

$$t_{neig}(A, 1000) = 240.75 \text{ seconds with } \bar{x}_{neig}(A, 1000) = 0.24075 \text{ seconds,}$$

$$t_{neig}(B, 1000) = 250.77 \text{ seconds with } \bar{x}_{neig}(B, 1000) = 0.25077 \text{ seconds,}$$

$$t_{neig}(C, 1000) = 259.120 \text{ seconds with } \bar{x}_{neig}(C, 1000) = 0.259120 \text{ seconds.}$$

The arithmetic mean of these averages is $\bar{x}_{neig}(1000) = 0.250213 \approx 0.25$ seconds with a standard deviation of $\sigma_{neig}(1000) = 0.007510$ seconds. The coefficient of variation is $cv_{neig}(1000) = \frac{\sigma_{neig}(1000)}{\bar{x}_{neig}(1000)} = 0.030014$, which is not particularly high. In figure 24 the reader can see that the deviation is at its lowest at a thousand iterations. So the deviation might be a bit higher overall. The average time taken for other tasks of the program is approximately $\bar{\epsilon}_{neig} = \bar{x}_{neig}(1) - \bar{x}_{neig}(1000) = 0.489787$ seconds which amounts to roughly 66% of the program.

**Performance of *numpy.linalg.eigh()***

The next program uses *numpy.linalg.eighh()*, another function of the NumPy package [56]. In contrast to to *numpy.linalg.eig()*, *numpy.linalg.eigh()* works best on a Hermitian or symmetric matrix [62]. It uses the LAPACK [47] routines *_SYEVD* [50] and *_HEEVD* [49], which compute the eigenvalues of a real symmetric matrix and complex Hermitian matrices [62].

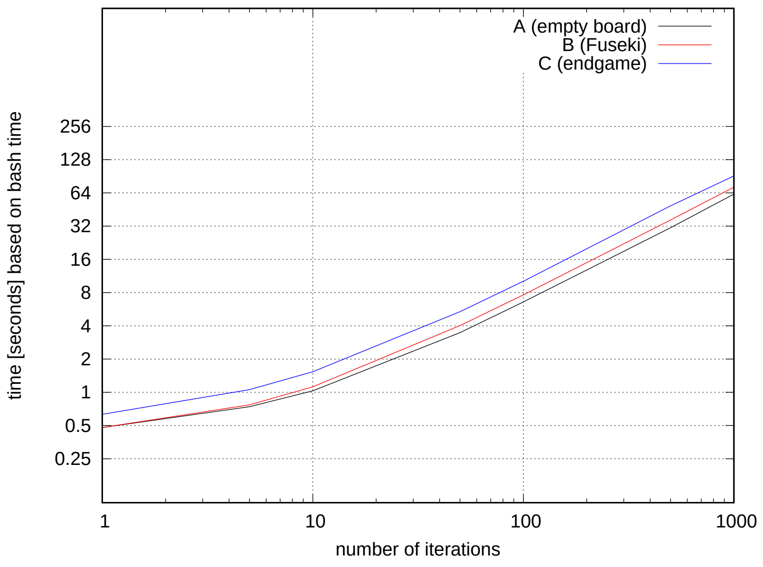


**Figure 25:** Performance of *numpy.linalg.eigh()* on the matrices A, B and C proposed in figure 20

The measured times for one call of the progam and one iteration of *numpy.linalg.eigh()* are $t_{neigh}(A,1) = 0.48$ seconds, $t_{neigh}(B,1) = 0.50$ seconds, and$t_{neigh}(C,1) = 0.632$ seconds. The arithmetic mean of these results is $\bar{x}_{neigh}(1) = 0.537333 \approx 0.537$ seconds. The measured times for one call of the program and thousand iterations of *numpy.linalg.eigh()* are

$$t_{neigh}(A,1000) = 62.38 \text{ seconds with } \bar{x}_{neigh}(A,1000) = 0.06238 \text{ seconds,}$$

$$t_{neigh}(B,1000) = 59.59 \text{ seconds with } \bar{x}_{neigh}(B,1000) = 0.05959 \text{ seconds,}$$

$$t_{neigh}(C,1000) = 91.120 \text{ seconds with } \bar{x}_{neigh}(C,1000) = 0.091120 \text{ seconds.}$$

where $\bar{x}_{neigh}(M,1000)$ is the computed average for one execution of *numpy.linalg.eigh()* on matrix $M$. The arithmetic mean of these averages is $\bar{x}_{neigh}(1000) = 0.071030 \approx 0.071$ seconds and the standard deviation is $\sigma_{neigh}(1000) = 0.014251$ seconds which leads to a coefficient of variance of $cv_{neigh}(1000) = \frac{\sigma_{neigh}(1000)}{\bar{x}_{neigh}(1000)} = 0.200639$. Additionally, we computed the program's approximated average time for other tasks $\bar{\epsilon}_{neigh} = \bar{x}_{neigh}(1) - \bar{x}_{neigh}(1000) = 0.466303$ seconds which translates to roughly 87% of the program. Compared to the previous programs, *numpy.linalg.eigh()* performed the best, even slightly better than eigen() in R. It performed a lot better than *numpy.linalg.eig()* which could be due to the fact that *numpy.linalg.eigh()* works best on symmetric matrices while *numpy.linalg.eig()* works best on nonsymmetric matrices [61, 62].

**Performance of *scipy.linalg.eig()***

This program used *scipy.linalg.eig()*, which has major similarities with *numpy.linalg.eig()* [61]. It uses the same LAPACK [47] routine in the background but can be configured more by offering a variety of parameters [63]. The measured times for one program call and one iteration of *scipy.linalg.eig()* are $t_{seig}(A,1) = 0.70$ seconds,$t_{seig}(B,1) = 0.70$ seconds, and $t_{seig}(C,1) = 0.73$ seconds. The arithmetic mean of these results is

$$\bar{x}_{seig}(1) = 0.71 \text{ seconds.}$$

The times for one program call and thousand iterations of *scipy.linalg.eig()* for each matrix and their calculated average times for one iteration of the function are

$$t_{seig}(A,1000) = 239.80 \text{ seconds with } \bar{x}_{seig}(A,1000) = 0.23980 \text{ seconds,}$$

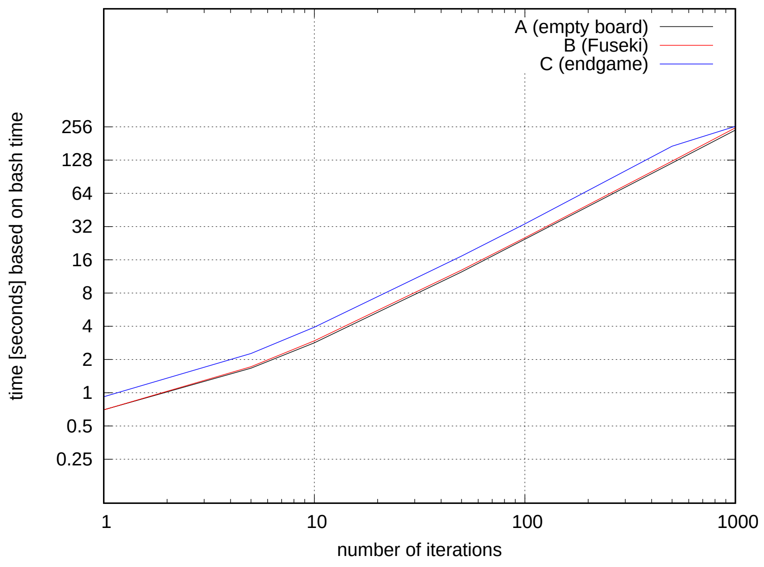


**Figure 26:** Performance of *scipy.linalg.eig()* on the matrices A, B, and C proposed in figure 20

$$t_{seig}(B,1000) = 250.92 \text{ seconds with } \bar{x}_{seig}(B,1000) = 0.25092 \text{ seconds,}$$

$$t_{seig}(C,1000) = 260.42 \text{ seconds with } \bar{x}_{seig}(C,1000) = 0.26042 \text{ seconds.}$$

The arithmetic mean of the calculated averages is $\bar{x}_{seig}(1000) = 0.25038$ seconds and the corresponding standard deviation is $\sigma_{seig}(1000) = 0.008427$ seconds which leads to a coefficient of variance of $cv = \frac{\sigma_{seig}(1000)}{\bar{x}_{seig}(1000)} = 0.033656$. The approximated average time for other tasks is $\bar{\epsilon}_{seig} = \bar{x}_{seig}(1) - \bar{x}_{seig}(1000) = 0.45962$ seconds which translates to approximately 65% of the program. Because the used LAPACK routine is the same as in *numpy.linalg.eig()*, which is not very suited for our problem, the results in figure 26 are similar to those in figure 24. This method proved not suitable for our problem.

**Performance of *scipy.linalg.eigh()***

This program uses *scipy.linalg.eigh()* and the documentation does not state if and which Fortran methods are used [64].

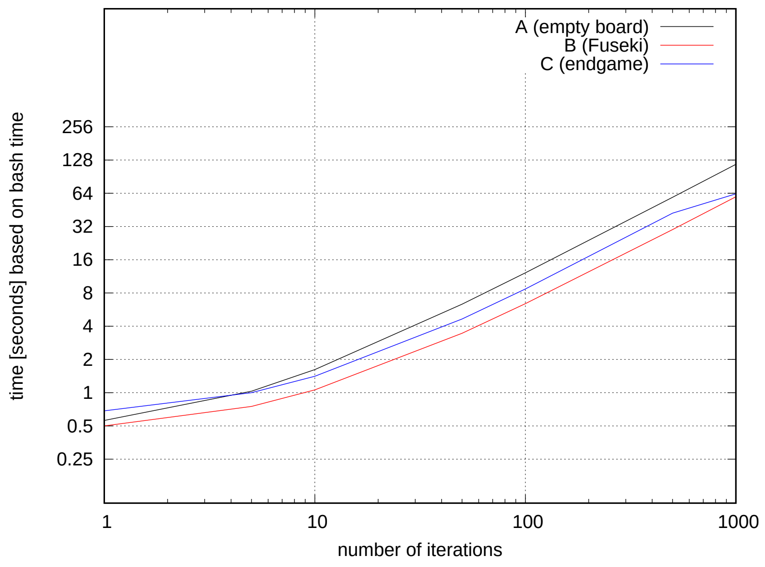


**Figure 27:** Performance of *scipy.linalg.eigh()* on the matrices A, B and C proposed in figure 20.

We measured the following times for one call of the program in which *scipy.linalg.eigh()* was executed once, $t_{seigh}(A,1) = 0.56$ seconds,$t_{seigh}(B,1) = 0.50$ seconds, and $t_{seigh}(C,1) = 0.684$ seconds. The corresponding arithmetic mean is $\bar{x}_{seigh}(1) = 0.581333 \approx 0.581$ seconds. For thousand iterations of scipy.linalgeigh() we measured the following times and calculated their average times for one iteration,

$$t_{seigh}(A, 1000) = 117.22 \text{ seconds with } \bar{x}_{seigh}(A, 1000) = 0.11722 \text{ seconds,}$$

$$t_{seigh}(B, 1000) = 59.59 \text{ seconds with } \bar{x}_{seigh}(B, 1000) = 0.05959 \text{ seconds, and}$$

$$t_{seigh}(C, 1000) = 63.232 \text{ seconds with } \bar{x}_{seigh}(C, 1000) = 0.063232 \text{ seconds.}$$

The arithmetic mean of these averages is $\bar{x}_{seigh}(1000) = 0.080014$ seconds and the corresponding standard deviation is $\sigma_{seigh}(1000) = 0.026351$ seconds. The coefficient of variance is $cv = \frac{\sigma_{seigh}(1000)=}{\bar{x}_{seigh}(1000)} = 0.319325$. The approximated average time for other tasks in the program is $\bar{\epsilon}_{seigh} = \bar{x}_{seigh}(1) - \bar{x}_{seigh}(1000) = 0.501323$ seconds. This translates $\approx 86\%$ of the program. Overall, *scipy.linalg.eigh()* behaved similar to numpy.linalg.eigh() and both performed slightly better than *eigen()* introduced in section 4.1.1.

### 4.1.5 Overview and Discussion of Direct Methods

Now, we will discuss the results of all programs listed. We will compare all functions and methods in three different categories. First, we will compare the arithmetic mean $\bar{x}(1000)$ of each program as an indicator of its overall performance. Second, we compare the results for the coefficient of variance $cv(1000)$, which gives information about the robustness of a function or method to changes of the matrices. Last, we compare $\bar{\epsilon}$, the approximated average time for other tasks of the program. This is a rough estimate and should be handled with caution. But it gives a vague idea which languages handled the tasks such as file loading better.

**Comparison of the Arithmetic Mean**

The arithmetic mean of all methods from best to worst is:

1. *numpy.linalg.eigh()* with $\bar{x}_{neigh}(1000) \approx 0.071$ seconds
2. *scipy.linalg.eigh()* with $\bar{x}_{seigh}(1000) \approx 0.080$ seconds
3. *eigen()* with $\bar{x}_R(1000) \approx 0.098$ seconds
4. *gsl_eigen_symmv()* with $\bar{x}_{gsl}(1000) \approx 0.282$ seconds
5. *numpy.linalg.eig()* with $\bar{x}_{neig}(1000) \approx 0.25$ seconds
6. *scipy.linalg.eig()* with $\bar{x}_{seig}(1000) \approx 0.25$ seconds
7. *EigenSolver* with $\bar{x}_{cpp}(1000) \approx 0.441$ seconds

There were not many functions that performed better than *eigen()*, the first first function introduced in 4.1.1. The only functions performing slightly better were *numpy.linalg.eigh()* and *scipy.linalg.eigh()* which were presented in section 4.1.4. Both functions used routines that worked better on symmetrical matrices [62, 64] and therefore did perform much better than their nonsymmetrical counterparts *numpy.linalg.eig()* and *scipy.linalg.eig()* [61, 63].

**Coefficient of Variance**

To see how robust a function worked on different matrices, we calculated $cv(1000)$ which is a percentage showing how much the the performance deviated from the arithmetic mean $\bar{x}(1000)$. A lower value of $cv(1000)$ indicates a higher robustness. In this category from best to worst are:

1. *numpy.linalg.eig()* with $cv_{neig}(1000) \approx 0.03$
2. *scipy.linalg.eigh()* with $cv_{seig}(1000) \approx 0.03$
3. *gsl_eigen_symmv()* with $cv_{gsl}(1000) \approx 0.07$
4. *EigenSolver* with $cv_{cpp}(1000) \approx 0.11$
5. *numpy.linalg.eigh()* with $cv_{neigh}(1000) \approx 0.20$
6. *eigen()* with $cv_R(1000) \approx 0.26$
7. *scipy.linalg.eigh()* with $cv_{seigh}(1000) \approx 0.33$

It is interesting to see that the programs which performed worse when comparing the arithmetic mean, seem much more robust to matrix changes. The functions *numpy.linalg.eigh()* and *scipy.linalg.eigh()* appear least robust when compared to the other methods although they were the fastest overall.

**Approximated Average Time for Other Tasks**

When comparing the approximated average time for other tasks $\bar{\epsilon}$ some programs were faster than others. We exclude the function *gsl_eigen_symmv()* from this category as the $\bar{\epsilon}_{gsl}$ had no meaning being a negative value. The *EigenSolver* performed best in this category with $\bar{\epsilon}_{cpp} = 0.066104$ which translates to around 13% of the program. The other function were significantly worse, with 82% in *eigen()*, 66% in *numpy.linalg.eig()*, 65% in *scipy.linalg.eig()*, 87% in *numpy.linalg.eigh()*, and 86% in *scipy.linalg.eigh()*. Our assumption is that programming languages like C and C++ might be faster at handling tasks like file loading than Python or R. But this can not be considered more than an assumption because more investigation would be needed.

**Conclusion on Direct Methods**

All in all, the times we measured using direct functions or methods were not fast enough for our needs. Therefore, we will discuss more methods we tried in the next section. Those methods approximate certain eigenvalues and eigenvectors iteratively.

## 4.2 Iterative Methods

We switched to iterative methods because we assumed that approximating the eigenvector directly could potentially be much faster.

### 4.2.1 Performance of the Arnoldi and Lanczos Method

Both subsequent functions are based on ARPACK [31] routines which is a Fortran package that uses the Arnoldi Method [91] and its simplification for symmetric matrices, the Lanczos Method [6]. Both methods are designed for sparse matrices.

**Performance of *scipy.sparse.linalg.eigs()***

Scipy.sparse.linalg.eigs() works on square matrices. It uses the implicitly restarted Arnoldi Method [91] of ARPACK [31] which is designed for sparse matrices.

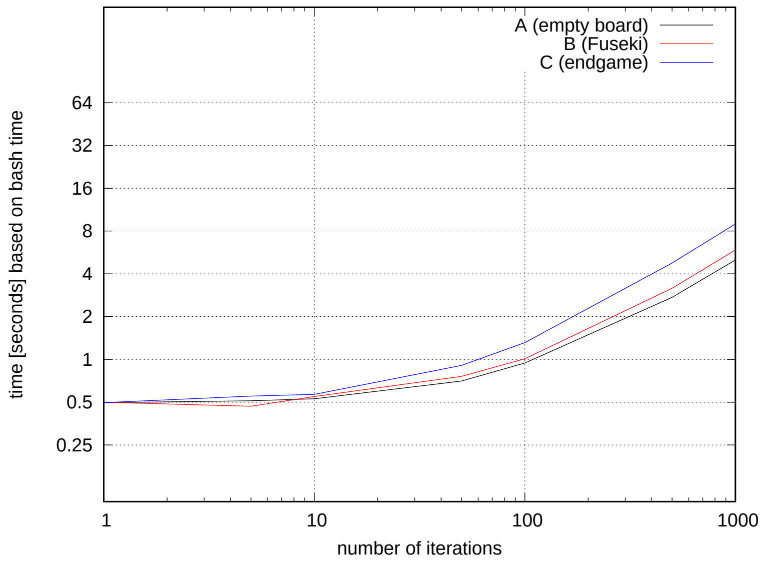


**Figure 28:** This diagram shows the performance of *scipy.sparse.linalg.eigs()* on the matrices A, B, and C shown in figure 20

One call of the program with one iteration of *scipy.sparse.linalg.eigs()* took $t_{speigs}(A,1) = 0.496$ seconds on matrix A, $t_{speigs}(B,1) =$ 0.500 seconds on matrix B, and $t_{speigs}(C,1) = 0.496$ seconds on matrix C. The average of those three results is $\bar{x}_{speigs}(1) = 0.497333$ seconds. We also present the times for 1000 iterations of *scipy.sparse.linalg.eigs()* on each matrix and the calculated averages for 1 iteration

$$t_{speigs}(A,1000) = 5.004 \text{ seconds with } \bar{x}_{speigs}(A,1000) = 0.005004 \text{ seconds},$$

$$t_{speigs}(B,1000) = 5.884 \text{ seconds with } \bar{x}_{speigs}(B,1000) = 0.005884 \text{ seconds},$$

$$t_{speigs}(C,1000) = 8.976 \text{ seconds with } \bar{x}_{speigs}(C,1000) = 0.008976 \text{ seconds.}$$

The total average composed off these averages is $\bar{x}_{speigs}(1000) = 0.006621$ seconds and the standard deviation is $\sigma_{speigs}(1000) = 0.001703$ seconds. this results in a coefficient of variance of $cv = \frac{\sigma_{speigs}(1000)}{\bar{x}_{speigs}(1000)} = 0.257247$ Lastly, we calculated $\bar{\epsilon}_{speigs} = \bar{x}_{speigs}(1) - \bar{x}_{speigs}(1000) = 0.49712$ seconds, the approximated average time the program took for other tasks. This translates to around 99% of the program.

Figure 28 shows how well this method performed in comparison to all direct methods. It worked best on an empty matrix taking only around 5.004 seconds for 1000 iteratins. That translates to $\approx 0.005$ seconds which is ten times faster than the faster direct methods.

### Performance of *scipy.sparse.linalg.eigsh()*

In contrast to *scipy.sparse.linalg.eigs()*, *scipy.sparse.linalg.eigsh()* is also specialised on symmetric matrices [65]. It uses the Lanczos Method [6] implemented in the Fortran package Arpack [31]. The Lanczos Method is a specialization of the Implicitly Restarted Arnoldi Method for symmetric matrices. It is an effective method for the computation of a few eigenvalues and corresponding eigenvectors of large symmetric matrices [6,12].

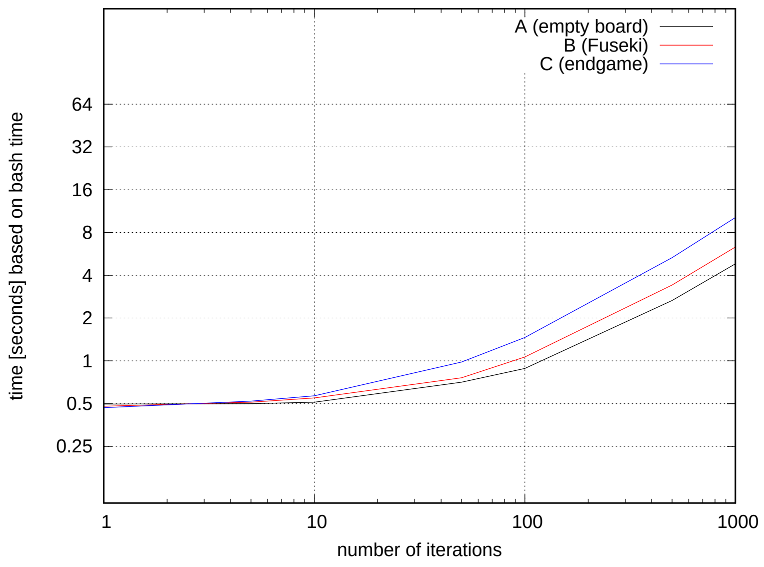


**Figure 29:** This diagram shows the performance of *scipy.sparse.linalg.eigsh()* on the matrices A, B, and C shown in figure 20

The measured times for one program call in which one iteration of *scipy.sparse.linalg.eigsh()* took place are $t_{speigsh}(A,1) = 0.496$ seconds,$t_{speigsh}(B,1) = 0.480$ seconds, and$t_{speigsh}(C,1) = 0.468$ seconds. The average time composed of these results is $\bar{x}_{speigsh}(1) = 0.481333$ seconds. The times for one call of the program with 1000 iterations and their respective averages are

$$t_{speigsh}(A,1000) = 4.812 \text{ seconds with } \bar{x}_{speigsh}(A,1000) = 0.004812 \text{ seconds,}$$

$$t_{speigsh}(B,1000) = 6.324 \text{ seconds with } \bar{x}_{speigsh}(B,1000) = 0.006324 \text{ seconds,}$$

$$t_{speigsh}(C,1000) = 10.204 \text{ seconds with } \bar{x}_{speigsh}(C,1000) = 0.010204 \text{ seconds.}$$

The calculated average of one iteration of all matrices is

$\bar{x}_{speigsh}(1000) = 0.007113$ seconds with a standard deviation of $\sigma_{speigsh}(1000) = 0.002271$ seconds. The coefficient of variance is $cv_{speigsh}(1000) = 0.31925$. The approximated average for other tasks is $\bar{\epsilon}_{speigsh} = \bar{x}_{speigsh}(1) - \bar{x}_{speigsh}(1000) = 0.47422$ seconds which translates to 99% of the program. Figure 29 shows that *scipy.sparse.linalg.eigsh()* works similarly good on our matrices than *scipy.sparse.linalg.eigs()*. Here it seems not to matter much that our matrix is symmetrical.

### Conclusion of the scipy.sparse.linalg Eigenvalue Functions

Although we were satisfied with the performance of these iterative functions, we had the problems finding a Go program which was actually written in Python. We wanted to test the heuristic with Pachi [3] which is written in C. Calling Python from C seemed inconvenient. Using the Fortran Routines of ARPACK [31] directly with C was a possibility that we contemplated but then dismissed because it would have gotten to far beyond the scope of this thesis.
So we tried to obtain a solution in C or C++ that is as fast as the Python's functions.

### 4.2.2 Performance of the Spectral Shift Power Method

We wanted to experiment with the Power Method [30] because it is a simple method for approximation of the dominant eigenvector of a matrix. At each step of the Power Method the following is calculated:

$$v = \frac{Av}{\lambda} \text{ where } \lambda = \|Av\|$$

where $v$ is a vector, $\lambda$ is a norm, and A is the matrix we want to compute the dominant eigenvalue and eigenvector of.

This calculation is repeated until a threshold is reached. The threshold can either a fixed number of iterations or a certain tolerance. The resulting scalar $\lambda$ is the approximated dominant eigenvalue $\lambda_{dominant}$ eigenvalue and $v$ its approximated corresponding eigenvector $v_{dominant}$. For our purpose we need to find the least dominant eigenvector $\lambda_s$. Thats why we have to shift the Matrix A by its dominant eigenvalue $\lambda_{dominant}$:

$$B = -A + (\lambda_{dominant} I).$$

Here, $I$ is the identity matrix. The eigenvalues of B have the shape $(-\lambda_i - \lambda_{dominant})$ which makes the least dominant eigenvalue of A the dominant eigenvalue of B. Now, we calculate the eigenvector of B using the Power method again to obtain the least dominant eigenvector $\lambda_s$ of A. This method only works on symmetric matrices [30].

#### Performance of a *Naive Self Implementation*

To further deepen our understanding for the subject we implemented a version of the Spectral Shift Power Method without using any libraries. We included this naive approach only for comparison. We measured the following times for

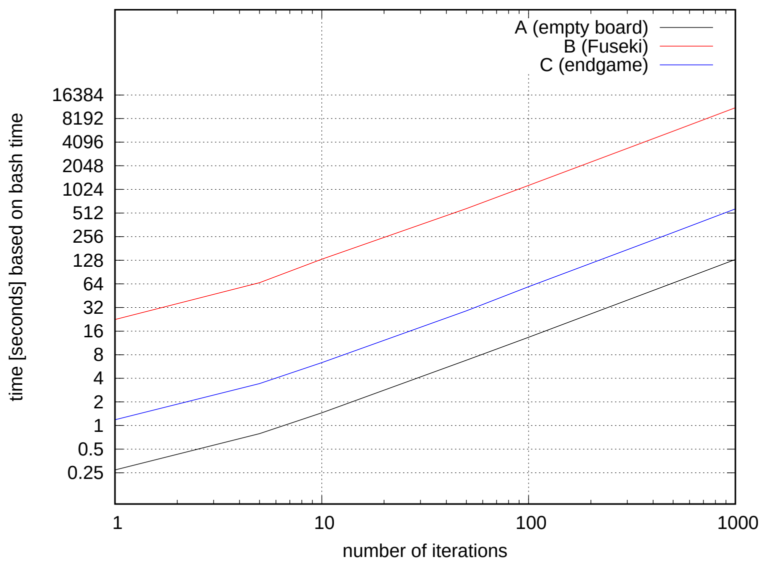


**Figure 30:** This diagram shows the performance of our naive self implementation using Spectral Shift on the matrices that represent the board positions shown in figure 20

one call of the program and one iteration: $t_{specc}(A,1) = 0.272$ seconds, $t_{specc}(B,1) = 22.44$ seconds, and $t_{specc}(C,1) = 1.184$ seconds. On the first glance it is very visible how slow this approach is compared to the others. It seems also very vulnerable to slight differences in matrices. The arithmetic mean for these values is $\bar{x}_{specc}(1) = 7.965333$ seconds. We also calculated the averages for one call on a particular matrix using the times for one call of the program with a 1000 iterations. The measured times and calculated averages are

$$t_{specc}(A,1000) = 131.092 \text{ seconds with } \bar{x}_{specc}(A,1000) = 0.131092 \text{ seconds,}$$

$$t_{specc}(B,1000) = 11273.832 \text{ seconds with } \bar{x}_{specc}(B,1000) = 11.273832 \text{ seconds, and}$$

$$t_{specc}(C,1000) = 577.404 \text{ seconds with } \bar{x}_{specc}(C,1000) = 0.577407 \text{ seconds.}$$

The calculated arithmetic mean of these averages is $\bar{x}_{specc}(1000) = 3.994$ seconds and the standard deviation is $\sigma_{specc}(1000) = 5.150956$ seconds, which leads to a coefficient of variant of $cv_{specc}(1000) = \frac{\sigma_{specc}(1000)}{\bar{x}_{specc}(1000)} = 1.289674$. This an unusual high coefficient which emphasizes the programs vulnerability towards changes on the matrix diagonal. The average time for other tasks of the program is approximately $\bar{\epsilon}_{specc} = \bar{x}_{specc}(1) - \bar{x}_{specc}(1000) = 3.97133$ seconds which translates to around 50%. Figure 30 and the results show that the program using the naive self implementation is much slower than the other programs so far. We assume this is because of the time consuming matrix vector product.

**Performance of *Spectral Shift using Eigen***

In Contrast to the *Naive Self Implementation* we tried another approach of the Spectral Shift Power Method using matrix routines from the Eigen Library [36].

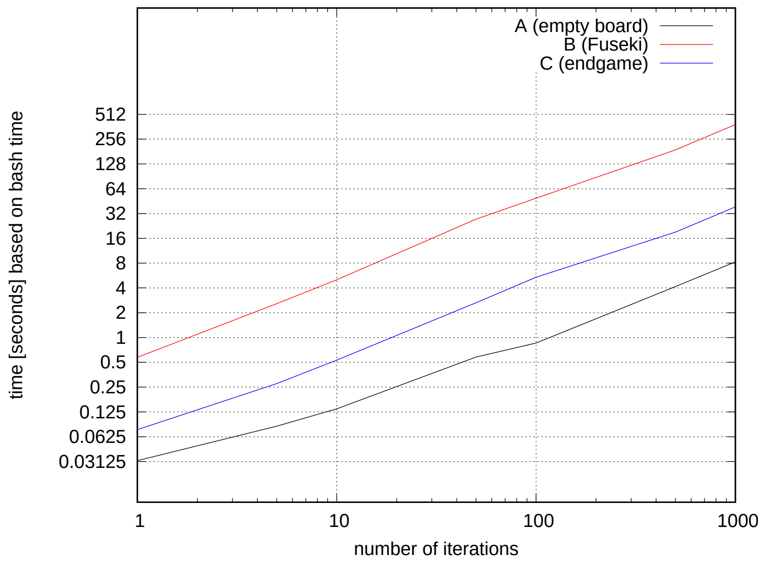


**Figure 31:** This diagram shows the performance of *Spectral Shift using Eigen* on the matrices that represent the board positions shown in figure 20.

The measured times for one call of the program in which one iteration of the algorithm took place are $t_{speccpp}(A,1) = 0.032$ seconds, $t_{speccpp}(B,1) = 0.576$ seconds, and$t_{speccpp}(C,1) = 0.076$ seconds. The arithmetic mean of the measured times is $\bar{x}_{speccpp}(1) = 0.228$ seconds. The measured times for a thousand iterations of the algorithm and their corresponding averages for one iteration on a specific matrix are

$$t_{speccpp}(A,1000) = 8.276 \text{ seconds with } \bar{x}_{speccpp}(A,1000) = 0.008276 \text{ seconds,}$$

$$t_{speccpp}(B,1000) = 384.872 \text{ seconds with } \bar{x}_{speccpp}(B,1000) = 0.384872 \text{ seconds, and}$$

$$t_{speccpp}(C,1000) = 38.552 \text{ seconds with } \bar{x}_{speccpp}(C,1000) = 0.038552 \text{ seconds.}$$

The arithmetic mean of those averages is $\bar{x}_{speccpp}(1000) = 0.1439$ seconds and together with the standard deviation $\sigma_{speccpp}(1000) = 0.170841$ seconds, they form the coefficient of variance $cv_{speccpp}(1000) = \frac{\sigma_{speccpp}(1000)}{\bar{x}_{speccpp}(1000)} = 1.187218$. The average time for other tasks of the program was approximately $\bar{\epsilon}_{speccpp} = \bar{x}_{speccpp}(1) - \bar{x}_{speccpp}(1000) = 0.0841$ seconds which is about 37%. Like with the *Naive Self Implementation* the robustness of the algorithm when treating slight changes in the matrix is really bad. The program was very fast on matrix A but really bad on matrix B. The Spectral Shift Method seems rather unstable, so we left it at that and tried another method.

### 4.2.3 Performance of the Inverse Power Method

After observing the instability of the Spectral Shift Method, we decided to use the Inverse Power iteration to find the least dominant eigenvector. We still use the Eigen Library for this test. The Inverse Power Method [7,30] is similar to the Power Method.

$$v = \frac{(A-\mu I)^{-1} \times v}{\lambda} \text{ where } \lambda = ||A * v||$$

So instead of applying the Power Iteration directly to $A$, in each iteration it is applied to the $(A-\mu I)^{-1}$. The Inverse Power Iteration finds eigenvalues near $\mu$. Because we want the least dominant eigenvector it is sufficient to choose $\mu = 0$ and calculate $A^{-1}$ once. This makes the algorithm much faster. So first, we have to choose a method to invert the matrix. Next, we apply Power Iteration to the inverse of A. We use matrix decomposition to solve

$$A \times x = i$$

for each column $i$ of the Identity matrix. The resulting vectors $x_i$ are the columns of $A^{-1}$. So we use matrix decomposition to acquire the inverse of A. So next, we will discuss the results of Inverse Power Method using different matrix decompositions to achieve $A^{-}1$.

### Inverse Power Method with LU Factorization

The SparseLU class uses supernodal LU factorization for general matrices [38]. It uses the techniques from sequential SuperLU package [16]. We will give a short description of a standard LU factorization. The matrix A is factored into a lower triangular matrix L and an upper triangular matrix U [88].

$$A = LU$$

The linear equation $Ax = b$ is solved by solving the two linear systems $Lz_2 = z_1$ and $Ux = z_2$ SparseLU uses routines from BLAS [78].

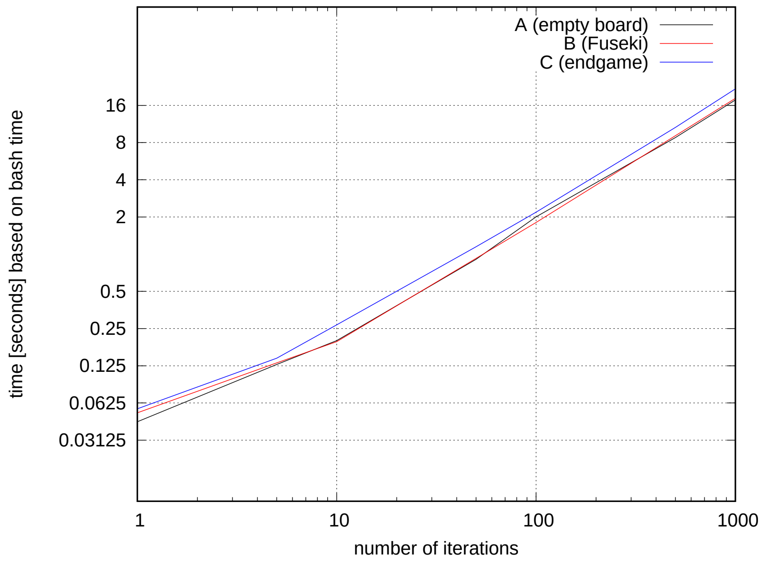


**Figure 32:** This diagram shows the performance of our implementation using the SparseLU solver (for preparing the matrix for inverting) on the matrices that represent the board positions shown in figure 20.

The measured times for one call of the program which uses one itertation of inverse Power Method are $t_{LU}(A, 1) = 0.044$ seconds,$t_{LU}(B, 1) = 0.052$ seconds, and$t_{LU}(C, 1) = 0.056$ seconds. The arithmetic mean for these times is $\bar{x}_{LU}(1) = 0.050667$ seconds. The measured times for thousand iterations of the Inverse Power Method and there correspondent averages for one iteration are

$$t_{LU}(A, 1000) = 17.704 \text{ seconds with } \bar{x}_{LU}(A, 1000) = 0.017704 \text{ seconds,}$$

$$t_{LU}(B, 1000) = 18.224 \text{ seconds with } \bar{x}_{LU}(B, 1000) = 0.018224 \text{ seconds, and}$$

$$t_{LU}(C, 1000) = 21.792 \text{ seconds with } \bar{x}_{LU}(C, 1000) = 0.021792 \text{ seconds.}$$

The arithmetic mean of these averages is $\bar{x}_{LU}(1000) = 0.019240$ seconds. Together with the standard deviation $\sigma_{LU}(1000) = 0.001817$ seconds, this leads to a coefficient of variant of $cv_{LU}(1000) = \frac{\sigma_{LU}(1000)}{\bar{x}_{LU}(1000)} = 0.094438$. The average time for other tasks is approximately $\bar{\epsilon}_{LU} = \bar{x}_{LU}(1) - \bar{x}_{LU}(1000) = 0.031427$ seconds which translates to roughly 62% of the program. The program using the SparseLU solver for inverting was not as fast as scipy.sparse.linalg.eigs() and scipy.sparse.linalg.eigsh() but relatively fast as well. The coefficient of variance $cv_{LU}(1000)$ tells us that the program is robust towards slight differences of the matrices.

Figure 33 shows how much time the Power Method took for approximating the eigenvector. It is remarkable how fast the Power Method works on $A^{-1}$. It only takes a fraction of the time that scipy.sparse.linalg.eigs() and scipy.sparse.linalg.eigsh() needed to calculate the eigenvector. So a huge part of the actual program consists of the factorization of the matrix and the computing of the inverse.

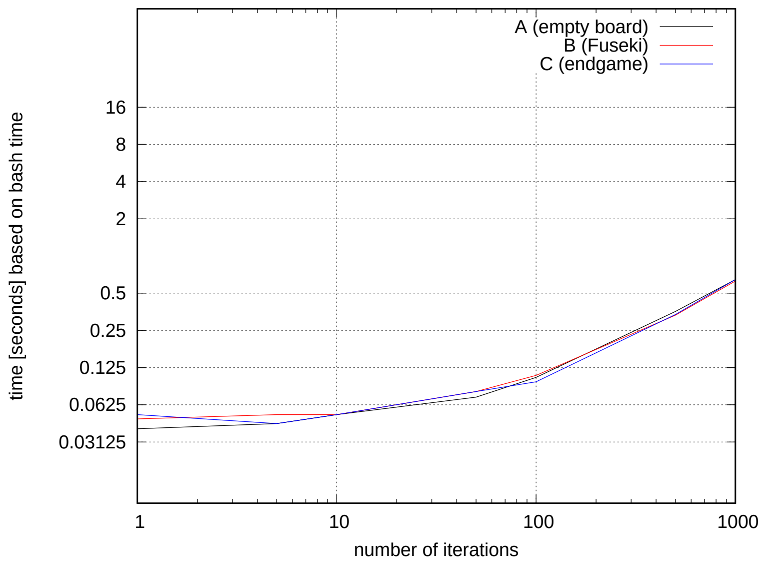


**Figure 33:** This figure shows the time the Power Method took after the $LU$ decomposition on the matrix to calculate the eigenvector.

### Inverse Power Method with Cholesky Factorization

The *SimplicialLDLT* solver is a solver build into Eigen that is recommended for very sparse but not to large problems [37]. The solver uses the $LDL^T$ Cholesky Factorization. The matrix A is decomposed into

$$A = LDL^T$$

and solved in three steps

$$L^T x = y$$

$$Dy = z$$

$$Lz = b$$

[13] .

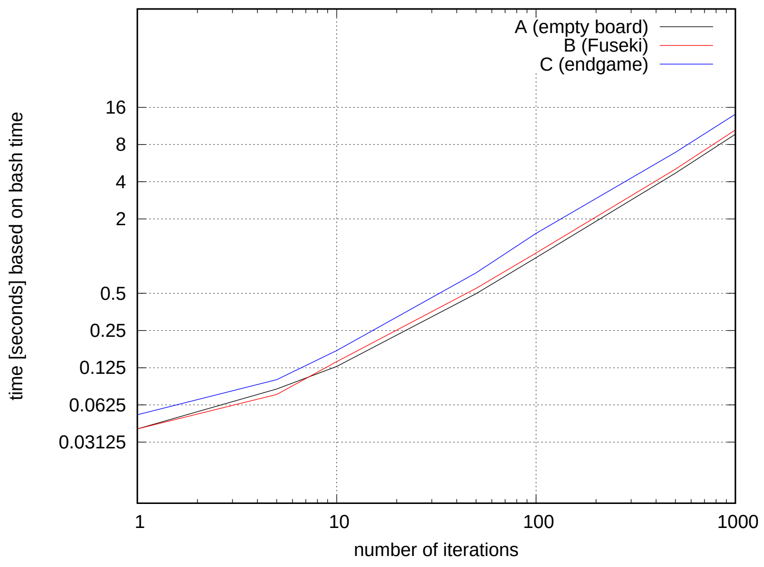


**Figure 34:** This diagram shows the performance of our implementation using the *SimplicialLDLT* solver (for preparing the matrix for inverting) on the matrices that reoresent the board positions shown in figure 20.

First, the times for one call of the program and one iteration: $t_{chol}(A,1) = 0.040$ seconds,$t_{chol}(B,1) = 0.040$ seconds, and$t_{chol}(C,1) = 0.052$ seconds. The corresponding arithmetic mean is $\bar{x}_{chol}(1) = 0.044$ seconds. Again, we calculate the times for a thousand iteration and calculate the average for one iteration for a specific matrix:

$$t_{chol}(A,1000) = 9.708 \text{ seconds with } \bar{x}_{chol}(A,1000) = 0.009708 \text{ seconds,}$$

$$t_{chol}(B,1000) = 10.512 \text{ seconds with } \bar{x}_{chol}(B,1000) = 0.010512 \text{ seconds, and}$$

$$t_{chol}(C,1000) = 14.072 \text{ seconds with } \bar{x}_{chol}(C,1000) = 0.014072 \text{ seconds.}$$

The arithmetic mean of the averages for one iteration is $\bar{x}_{chol}(1000) = 0.011431$ seconds where $\sigma_{chol}(1000) = 0.001896$ seconds is the standard deviation and $cv_{chol}(1000) = \frac{\sigma_{chol}(1000)}{\bar{x}_{chol}(1000)} = 0.165898$ is the coefficient of variance. The average time for other tasks of the program is approximated to $\bar{\epsilon}_{chol} = \bar{x}_{chol}(1) - \bar{x}_{chol}(1000) = 0.032569$ seconds which seems to be roughly 74% of the program.Overall, SimplicialLLT performed a bit better than SparseLU but still worse than scipy.sparse.linalg.eigs() and scipy.sparse.linalg.eigsh(). Although, it seems to be slighty less robust than SparseLU when observing the coefficient of variance. Figure 35 shows how much time the Power Method took for approximating

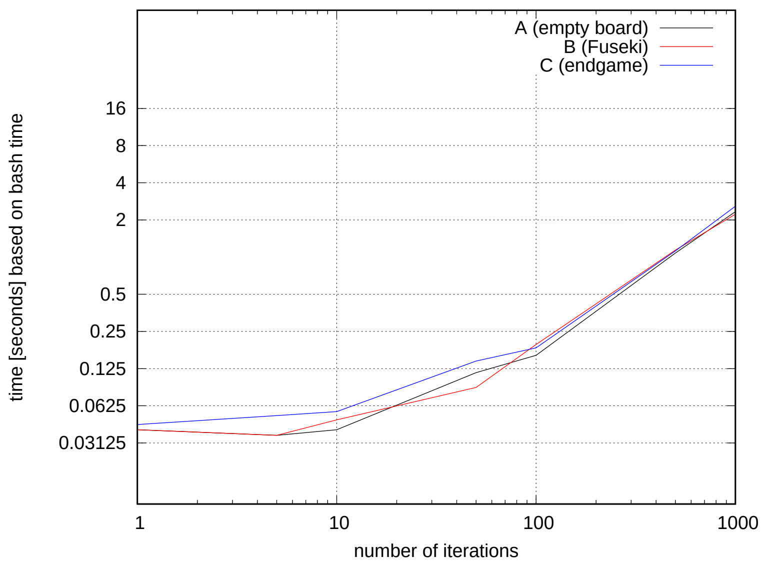


**Figure 35:** This figure shows the time the Power Method took after the $LDL^T$ decomposition on the matrix to calculate the eigenvector.

the eigenvector after the inverse of the matrix was computed. The Power Method converges not nearly as fast as it does when using the $LU$ decomposition but it still works faster than scipy.sparse.linalg.eigs() and scipy.sparse.linalg.eigsh(). We conclude that it is important which method for factorization is used.

### Inverse Power Method with Conjugate Gradient Descent

We also tested how fast Conjugate Gradient Descent [87] would invert the matrix and prepare it for the Inverse Power Method. We tested *ConjugateGradient*, a solver from Eigen, which is recommended for large symmetric problems [36].

We measured the following times for one call of the program and one iteration of the *ConjugateGradient* solver: $t_{cg}(A,1) = 0.392$ seconds,$t_{cg}(B,1) = 0.432$ seconds, and$t_{cg}(C,1) = 0.208$ seconds. The arithmetic mean of these values is $\bar{x}_{cg}(1) = 0.344$ seconds.

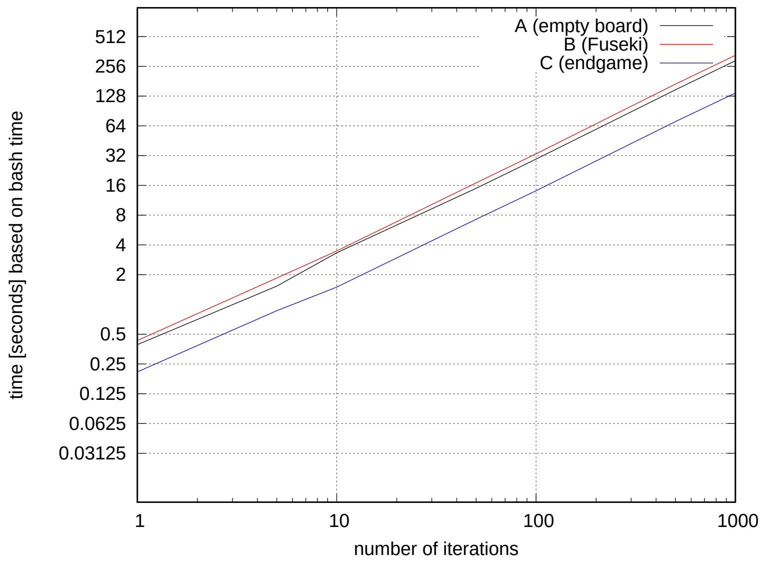
We also calculate the average for one call of each matrix using the measured times of a 1000 iterations for each matrix. The results are

$$t_{cg}(A,1000) = 291.596 \text{ seconds with } \bar{x}_{cg}(A,1000) = 0.291596 \text{ seconds,}$$

$$t_{cg}(B,1000) = 331.712 \text{ seconds with } \bar{x}_{cg}(B,1000) = 0.331712 \text{ seconds, and}$$

$$t_{cg}(C,1000) = 137.468 \text{ seconds with } \bar{x}_{cg}(C,1000) = 0.137468 \text{ seconds.}$$

The average mean composed of this averages on the other hand is $\bar{x}_{cg}(1000) = 0.253592$ seconds and the corresponding standard deviation is $\sigma_{cg}(1000) = 0.083729$ seconds with a coefficient of variance of $cv_{cg}(1000) = \frac{\sigma_{cg}(1000)}{\bar{x}_{cg}(1000)} = 0.330174$. The approximated average time for other tasks is $\bar{\epsilon}_{cg} = \bar{x}_{cg}(1) - \bar{x}_{cg}(1000) = 0.090408$ seconds which translates to roughly 26% of the program.



**Performance of the Iterative Method with *BiCGSTAB***

We also tried the Biconjugate Gradient Stabilized Method [5], a variant of the Conjugate Gradient Descent [87] using the *BiCGSTAB* solver of the Eigen Library [34].

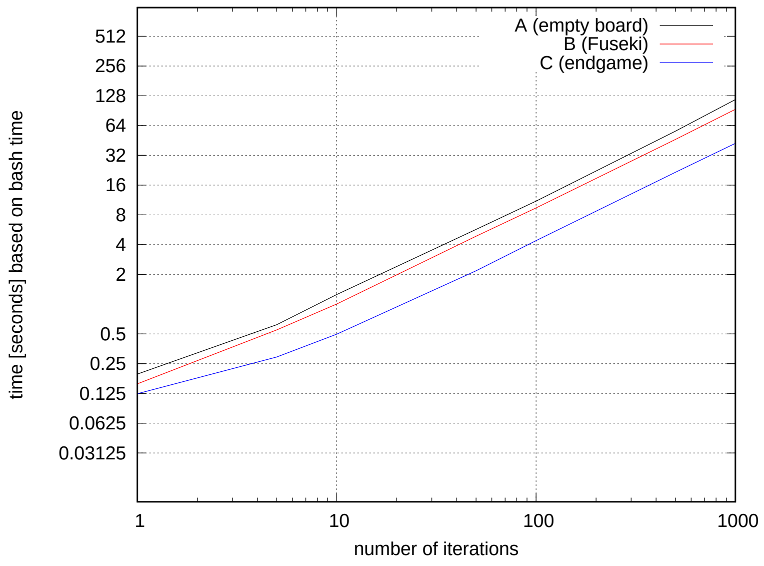


**Figure 36**

We measured $t_{bcg}(A,1) = 0.196$ seconds,$t_{bcg}(B,1) = 0.156$ seconds, and$t_{bcg}(C,1) = 0.124$ seconds for one call of the program and one iteration of the *BiCGSTAB* solver. The corresponding arithmetic mean is $\bar{x}_{bcg}(1) = 0.158667$ seconds. The measured times for 1000 iterations on each matrix and their corresponding averages for one iteration are

$$t_{bcg}(A,1000) = 117.448 \text{ seconds with } \bar{x}_{bcg}(A,1000) = 0.117448 textseconds,$$

$$t_{bcg}(B,1000) = 93.468 \text{ seconds with } \bar{x}_{bcg}(B,1000) = 0.093468 \text{ seconds, and}$$

$$t_{bcg}(C,1000) = 42.380 \text{ seconds with } \bar{x}_{bcg}(C,1000) = 0.042380 \text{ seconds.}$$

The arithmetic mean composed of these averages is $\bar{x}_{bcg}(1000) = 0.084432$ seconds and $\sigma_{bcg}(1000) = 0.031305$ seconds is the standard deviation. The coefficient of variance is $cvbcg(1000) = \frac{\sigma_{bcg}(1000)}{\bar{x}_{bcg}(1000)} = 0.370776$.
The approximated average time for other tasks is $\bar{\epsilon}_{bcg} = \bar{x}_{bcg}(1) - \bar{x}_{bcg}(1000) = 0.074235$ seconds which translates to roughly 47% of the program.

### 4.2.4 Overview and Discussion of Iterative Methods

We tried three different approaches:

- the Arnoldi/Lanczos Method using SciPy methods
- Spectral Shift Power Method
- Inverse Power Iteration using several solver from Eigen [36]

To give a general overview of the tested methods we will compare them in three categories as already done in section 4.1.

**Comparing the Arithmetic Mean**

1. *scipy.sparse.linalg.eigs()* with $\bar{x}_{speigs}(1000) \approx 0.007$ seconds
2. *scipy.sparse.linalg.eigsh()*$\bar{x}_{speigsh}(1000) \approx 0.007$ seconds
3. *SimplicialLDLT* with $\bar{x}_{chol} \approx 0.011$ seconds
4. *SparseLU* with $\bar{x}_{LU} \approx 0.019$ seconds
5. *BiCGSTAB* with $\bar{x}_{bcg} \approx 0.084$ seconds
6. *Spectral Shift using Eigen* with $\bar{x}_{speccpp} \approx 0.144$ seconds
7. *ConjugateGradient* with $\bar{x}_{cg} \approx 0.254$ seconds
8. *Naive Self Implementation* with $\bar{x}_{specc} \approx 3.994$ seconds

The best method, in regards to average time consumption, was scipy.sparse.linalg.eigsh(), closely followed by scipy.sparse.linalg.eigs(). Therefore, the matrix' symmetry seemingly had no big impact. The second best method was the Inverse Power Method using the SimplicialLLT solver to compute the inverse for the matrices. This method roughly took around twice as much time as the SciPy methods. SparseLU took roughly a quadruple of the SciPy's methods time. Interesting is that the Power Method converged really fast on the inverse using both factorization methods. The Power Method performed especially on the inverse computed with the SparseLU solver. So we conclude, that it seems to be of importance which method is used. The Spectral Shift Power Methods on the other hand did not perform sufficiently. The C++ implementation using the Eigen Library was not faster than some direct methods in section 4.1 and the *Naive Self Implementation* was the overall worst approach.

**Coefficient of Variance**

1. *SparseLU* with $cv_{LU}(1000) \approx 0.09$
2. *SimplicialLDLT* with $cv_{chol}(1000) \approx 0.17$
3. *scipy.sparse.linalg.eigs()* with $cv_{speigs}(1000) \approx 0.26$
4. *scipy.sparse.linalg.eigsh()* with $cv_{speigsh}(1000) \approx 0.32$
5. *ConjugateGradient* with $cv_{cg}(1000) \approx 0.33$
6. *BiCGSTAB* with $cv_{bcg}(1000) \approx 0.37$
7. *Spectral Shift using Eigen* with $cv_{speccpp}(1000) \approx 1.19$
8. *Naive Self Implementation* with $cv_{specc}(1000) \approx 1.29$

It is interesting to see that the programs who performed best in aspect of time also performed best in the aspect of robustness. This means that they handled changes in the matrices better than the slower methods. The Spectral Shift Power Methods performed by far worse and were exceptionally vulnerable to slight changes on the matrix diagonal.

**Approximated Average Time for Other Tasks**

The summary of all $\bar{\epsilon}$ ordered from smallest to largest percentage is presented here:

1. *ConjugateGradient* with $\bar{\epsilon}_{cg} \approx 26\%$
2. *Spectral Shift using Eigen* with $\bar{\epsilon}_{speccpp} \approx 37\%$
3. *BiCGSTAB* with $\bar{\epsilon}_{bcg} \approx 47\%$
4. *Naive Self Implementation* with $\bar{\epsilon}_{specc} \approx 50\%$
5. *SparseLU* with $\bar{\epsilon}_{LU} \approx 62\%$
6. *SimplicialLDLT* with $\bar{\epsilon}_{chol} \approx 74\%$
7. *scipy.sparse.linalg.eigs()* with $\bar{\epsilon}_{speigs} \approx 99\%$
8. *scipy.sparse.linalg.eigsh()* with $\bar{\epsilon}_{speigsh} \approx 99\%$

Again (like in section 4.1), the methods which performed faster overall did take up smaller amounts of the whole program. The amounts of $\bar{\epsilon}_{speigs}$ and $\bar{\epsilon}_{speigsh}$ indicate that for both SciPy methods only 1% of the whole program's time was dedicated to the eigenvector computation. It seems logical to assume, that the bottleneck of the program shifts if it computes the eigenvalues faster. We feel that this is the major significance which can be concluded from this data. On the other hand, it is likely that the percentage is also influenced by other variables, such as chosen programming language.

## 4.3 Conclusion on Performance Tests

We can conclude that iterative methods seem to be the better choice for large sparse matrices. The SciPy methods scipy.sparse.linalg.eigs() and scipy.sparse.linalg.eigsh() performed best overall. What we can not recommend using on the matrices is the Spectral Shift Power Method, as she converged slowly and was not robust as well. Inverse Power Iteration seemed like a simple but also slightly slower approach compared to the methods in Python. Also, it seems to be of importance to look for a fitting decomposition when computing the inverse. The Power method converged much faster on the $LU$ factored matrix than on the $LDL^T$ and was even faster than the functions in Python. The power method on its own worked pretty well, although more complex methods could enhance the performance further. As scipy.sparse.linalg.eigs() and scipy.sparse.linalg.eigsh() using the Arnoldi and Lanczos methods [6,91] performed best in the end, we recommend experimenting with these methods.

## 5 Testing the Heuristic in Pachi

In this section, several experiments were run to determine the effect of the *big move* heuristic incorporated as *Prior Knowledge* in Pachi's Monte Carlo Tree Search [19]. The meaning of *Prior Knowledge* will be explained in section 5.1. Pachi*, the modified Pachi, will be tested against original Pachi using several different modifications.

### 5.1 About Pachi

Pachi is one of the leading open source Go programs at the moment [3]. Pachi consists of the main loop, the game engine, the playout policy, the Go board representation and the tactical and Aux Library [3]. The focus will be on the parts of Pachi relevant for incorporating the heuristic, namingly the game engine and the playout policy[23].

**The Game Engine**

The default tree policy is called *ucb1amaf* and consists of MCTS with *RAVE* [3, 20]. The policy will be described as specified in [3] by P.Baudis and J.Gailly.

$$Q(s,a) = \frac{w(s,a)}{n(s,a)}$$

is the action value function of the standard MCTS and $n(s,a)$ is the number of simulations of action $a$ in state s. $w(s,a)$ is the number of won simulations started with action $a$ in state $s$.

$$Q_{RAVE}(s,a) = \frac{w_{RAVE}(s,a)}{n_{RAVE}(s,a)}$$

is the action value function of the RAVE component. The details on how $Q_{RAVE}(s,a)$ is computed can be found in the section 2.3.2. Analogously, $n_{RAVE}(s,a)$ is the number of simulations where action $a$ happened somewhere in the simulation that started with state $s$. The action value of *ucb1amaf* is a mixed value of $Q(s,a)$, the action value of a standard MCTS, and $Q_{RAVE}(s,a)$, the RAVE value of an action in $a$ specific state $s$. It is

$$Q_{ucb1amaf}(s,a) = (1-\beta)Q(s,a) + \beta Q_{RAVE}(s,a)$$

where

$$\beta = \frac{n_{RAVE}(s,a)}{n_{RAVE}(s,a) + n(s,a) + \frac{n_{RAVE}(s,a)\cdot n(s,a)}{equiv_{RAVE}}}$$

is the mixing parameter. The value of $\beta$ is initialized with 1, hence, RAVE is used in the beginning of a game, but is used less as $\beta$ decreases [3]. It might confuse the reader that the policy is called ucb1amaf but does not actually use UCT. This seems to have historical reasons.Each state action pair has a value $Q_{ucb1amaf}(s,a)$ and the action $a$ that corresponds to the highest $Q_{ucb1amaf}(s,a)$ is chosen [3]. The playout policy is by default called *moggy*. It produces semi-random *MCTS* playouts using $3 \times 3$ patterns and various technical checks [3].

**Adding Domain Knowledge**

Pachi uses the idea of incorporating Domain Knowledge, also known as Prior Knowledge, into MCTS that was previously used in the MoGo Go program [3, 19]. Pachi has several heuristics that approximate the value of certain moves on the board. This heuristics are used as so called *Priors* which manipulate the *Monte Carlo Tree Search*. The Priors are state-action pairs with a Prior value $Q_{prior}(s,a)$ and an *equivalent experience* $n_{prior}(s,a)$ which is similar to $n(s,a)$ but is the number of simulations the Prior would have needed to achieve a value of $Q_{prior}(s,a)$ [19]. So a Prior is an state action pair (s,a) that has an assumed value of $Q_{prior}(s,a)$ if it would have been simulated $n_{prior}$ times. Per default Prior values and their *equivalent experience* are counted to the normal MCTS value ($Q(s,a)$) and not to $Q_{RAVE}(s,a)$ in the $Q_{ucbamaf}$ formula. The *big move* heuristic also incorporates Domain knowledge about the game of Go into Pachi. Therefore, the heuristic is implemented as a Prior into Pachi. So the program implementing the heuristic will be called for every state of the search tree.

### 5.2 Testing Pachi* against Pachi

The *big move* heuristic was tested using GOGUI [14], an interface where it is possible to test Go programs via GTP [46]. We use its *gogui-twogtp*, a Go Text Protocol which lets two go programs play against each other. So we let Pachi*, the modified version of Pachi play against the original Pachi.

[23] Also known as default policy

## 5.3 First Tests

Here, we present the first tests for Pachi*. The basic settings for all tests are given in table 1. Every test, Pachi and Pachi* played hundred games against each other and alternated colors each game. We chose a Komi[24] of 6.5 points. We used the default version of Pachi, but chose to include its pattern library [2]. Furthermore, we had to enable the use of Priors. We decided to give each program a thinking time of 20 min S.D.[25]. We let the programs alternate colors after each game. We added three different Priors for the *big move* heuristic to Pachi*. Those three Priors differ in the amount of potential

| | |
|---|---|
| Komi | 6.5 |
| number of games | 100 |
| thinking time | 20 min S.D |
| patterns | yes |
| Priors | yes |
| alternation of colors | yes |
| board size | $19 \times 19$ |
| eqex | -800 |

**Table 1:** Basic Settings for Pachi and Pachi*. By eqex we mean the *equivalent experience* $n_{prior}(s, a)$. Komi are compensation points for White which are added to the final score.

added to the matrix representing the board state. The concept of potential is explained in section 3. The three Priors are

- p1= *big move* heuristic on a Dirichlet matrix without extra potential.
- p2= *big move* heuristic on a Dirichlet matrix with potential corresponding to all stones on the seventh line of the Go Board.
- p3= *big move* heuristic on a Dirichlet matrix with potential corresponding to all stones on the sixth and seventh line of the Go board.

We chose to use the same value for *equivalent experience* as the pattern Prior of Pachi which is -800. The number connotes that the eightfold of the base equivalent experience is used[26].

### 5.3.1 First Test without modifications

The first test we made was simple and without further modifications. Table 2 shows the main modifications.

| | |
|---|---|
| Prior value | 1.0 |
| eqex | -800 |
| matrix modification | p1,p2,p3 |

**Table 2:** Settings for the first test. By *prior value* we mean $Q_{prior}(s, a)$, by eqex we mean the *equivalent experience* $n_{prior}(s, a)$. The term "matrix modifications" refers to different potential added to the matrix before computation.

Pachi* won 0% of games with this configuration. Our assumption was that the heuristic still takes too much time and that Pachi simply had more time for building the search tree and simulations.

### 5.3.2 Test 2: Adding a Phantom Function to Pachi

Seeing that Pachi* performed really bad in the first test, we incorporated a phantom function into Pachi which also calculates the heuristics' algorithm but does not save the result. Therefore, it does not matter anymore how long the algorithm needs for computing the eigenvector. The main settings for this test are given in table 3.

After incorporating a phantom function into Pachi, Pachi* performed better and won around 19% of the games.

[24] explained in section 2.1.2
[25] Sudden Death. A player loses the game if his or her 20 minutes are over
[26] According to the source code of Pachi [3] or more specifically the file *prior.c*

| | |
|---|---|
| prior value | 1.0 |
| eqex | -800 |
| matrix modification | p1,p2,p3 |
| phantom function | yes |

**Table 3:** Settings for test 2. By *prior value* we mean $Q_{prior}(s,a)$, by eqex we mean the *equivalent experience* $n_{prior}(s,a)$. The matrix modifications determine the potential for the matrices. The matrix modifications determine the potential for the matrices. A phantom function is added to balance out certain time advantages of Pachi.

### 5.3.3 Test 3: Confining the Algorithm to Fuseki

The next test checks if Pachi* performs better when having a move restriction. By move restriction we mean that Pachi* only calculates the algorithm up to a specific move on the board. We chose 50 moves as that value. We choose this setting because we wanted to restrain the heuristic to Fuseki. The settings can be viewed in table 4. Pachi* won around 47% against Pachi which proved this restriction to be very effective.

| | |
|---|---|
| prior value | 1.0 |
| eqex | -800 |
| matrix modification | p1,p2,p3 |
| phantom function | yes |
| move restriction | 50 moves |

**Table 4:** Settings for test 3. By *prior value* we mean $Q_{prior}(s,a)$, by eqex we mean the *equivalent experience* $n_{prior}(s,a)$. The matrix modifications determine the potential for the matrices. The term "matrix modifications" refers to different potential which is added to the matrix before computation. The heuristic in now confined to the 50 first moves of the game.

### 5.3.4 Test 4: Without p1

While looking at the SGF [24] files we noticed that some moves of Pachi* looked very "unnatural" in a sense that too many moves were played on the sixth to tenth line on the go board. These lines are not considered to be very helpful in the Fuseki as already mentioned in section 3. We assumed that it could be due to the matrix modification p1. We decided to test the performance of Pachi* without p1 to see if Pachi* performes better without it. With the configurations in table 5 Pachi* won 48% of the games against Pachi. We decided to further test without p1, as it did not contribute to the result significantly.

| | |
|---|---|
| prior value | 1.0 |
| eqex | -800 |
| matrix modification | p2,p3 |
| phantom function | yes |
| move restriction | 50 moves |

**Table 5:** Settings for test 4. This time we test how Pachi* performs without the p1 Prior.

### 5.3.5 Conclusion on the First Tests

| | |
|---|---|
| prior value | 1.0 |
| matrix modification | p2,p3 |
| phantom function | yes |
| move restriction | 50 moves |

**Table 6:** Here, the main settings are presented that will not change in the following tests.

We decided to take the settings in table 6 to the next round of tests. We assumed that the reason why Pachi* still performed worse than Pachi could be due to the fact that we were not sure how to balance the Prior with the other Priors of Pachi. The goal was that Pachi* should use the *big move* heuristic but not to often, so that for example important patterns would still be played preferably. This is why we experimented with the *equivalent experience* next, as it is is the parameter varying in other Priors of Pachi[27].

## 5.4 Additional Tests: Varying Equivalent Experience

In this section we will test with varying *equivalent experience* to better incorporate the *big move* heuristic into Pachi's original design. We do that, because we want to have a good balance between the *big move* heuristic and the other Priors of Pachi. For example, a local move that could save a group from *dying* should be played instead of the *big move* the heuristic approximates.

### 5.4.1 Eqex Formula inspired by Pachi's Pattern Prior

We examined the source code of Pachi [3] to see how it incorporates other Priors and with which amount the *value* $Q_{prior}(s,a)$ and the *equivalent experience* $n_{prior}$ were initiated for these Priors.

The patterns of Pachi's Pattern Library [2] are incorporated as Priors into Pachi as well [3].

All moves suggested by patterns have the same *value* $Q_{prior}(s,a)$ which is set to an amount of 1.0 for the pattern Prior. The *equivalent experience* on the other hand varies slightly. The *equivalence experience* is calculated as:

$$eqex_{pattern}(x) = \sqrt{P(x)} \cdot eqex$$

where eqex, the basic *equivalent experience*, is set to a default value of $-800$[28] and $P(x)$ is the playing probability of a pattern $x$ after it occurred on the board [2].

Figure 37 shows the playing probability distribution of all patterns. It shows how often patterns were actually played

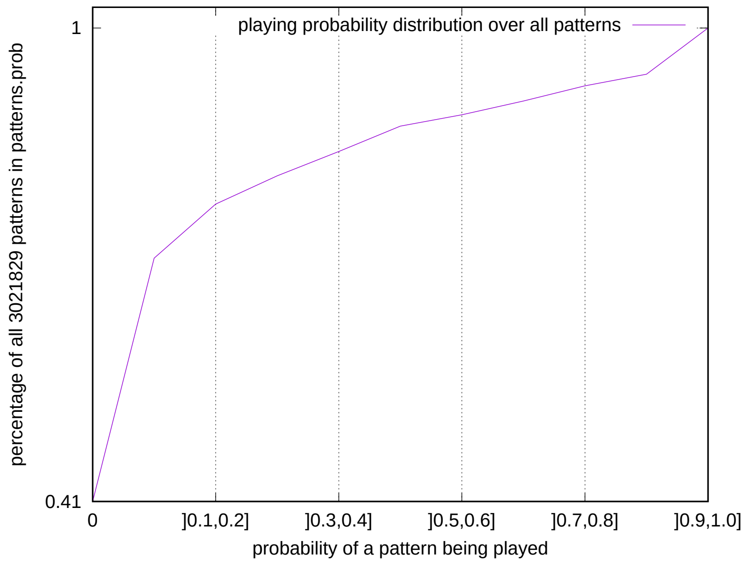


**Figure 37:** This figure depicts how many patterns (y-axis) were actually played with a certain probability (x-axis) when occurring in a game that was used for the pattern library. For example, 41% of all patterns that have occurred at some point have not been played at all in any games recorded. We extracted these values from the pattern files directly [2].

by people when encountered in games used for the library. Now, we know how the pattern Prior is set. This means we can set Pachi* accordingly. Because the formula for each pattern Prior holds that $\sqrt{(p(x))} \leq 1 \geq 0 \; \forall P(x)$, we designed a formula that behaves similarly but takes the number of moves played on the board into account, because we want to decrease the influence of our heuristic with each moves. The formula for the *equivalent experience* of the *big move* heuristic is therefore

$$eqex_{big} = \sqrt{1.0 - \frac{moves}{x \cdot 100}} \cdot eqex$$

[27] When looking at the source code of Pachi [3], one can see, that the value of a Prior is set at a fixed value. The priority of one Prior over the other seems to be depending only on the varying *equivalent experience*.

[28] as explained in section 5.3.

where moves is the number of moves already played on the board, and equex is the *equivalent experience*. The number of moves indicates how far the game has progressed and confines the heuristic to Fuseki decreasing $big\_eqex$, the *equivalent experience* for the *big move* heuristic. proportionally with the number of moves already played.

We will choose x so that the term $\sqrt{1.0 - \frac{moves}{x \cdot 100}}$ will always be between 0 and 1 just like $\sqrt{(P(x))}$. The larger x in $\sqrt{1.0 - \frac{moves}{x \cdot 100}}$, the smaller the decrease proportionally to the number of moves becomes.
Now, we did a few tests to check if decreasing the *equivalent experience* increases the win rate. The test arrangement can be seen in table 7.

| | |
|---|---|
| prior value | 1.0 |
| $eqex_{big}$ | $\sqrt{1.0 - \frac{moves}{x \cdot 100}} \cdot (eqex)$ with $x = \{1, 2, 3, 4, 5\}$ with $eqex = -800$ |
| matrix modification | p2,p3 |
| phantom function | yes |
| move restriction | 50 moves |

**Table 7:** Settings for the test which tests the new formula for the *equivalent experience* $eqex_{big}$.

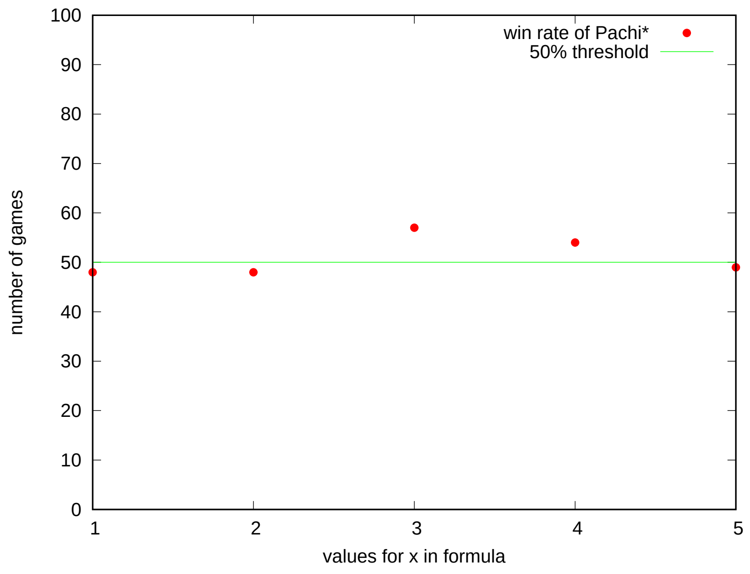


**Figure 38:** This figure depicts the win rate of Pachi* with varying values for $x$ in $\sqrt{1.0 - \frac{moves}{x \cdot 100}} \cdot (eqex)$.

Figure 38 shows how Pachi* performed. As the reader might see, the setting with $x = 3$ seemed to be the best and had a win rate of 57%. So the next tests will focus on further enhancing the formula with $x = 3$.

### 5.4.2 Enhancing the Formula

Next, different settings for $y$ in

$$eqex_{big} = \sqrt{y - \frac{moves}{3 \cdot 100}} \cdot (eqex)$$

were tested. The results can be seen in figure 39.

This time we also experimented with values for $y$ that made $\sqrt{y - \frac{moves}{3 \cdot 100}} > 1$, to see what happens. When setting $y = 3.0$, Pachi* performed really badly as seen in figure 39. Setting y to 0.9 slightly increased the win rate as it resulted in 59 of 100 games won.

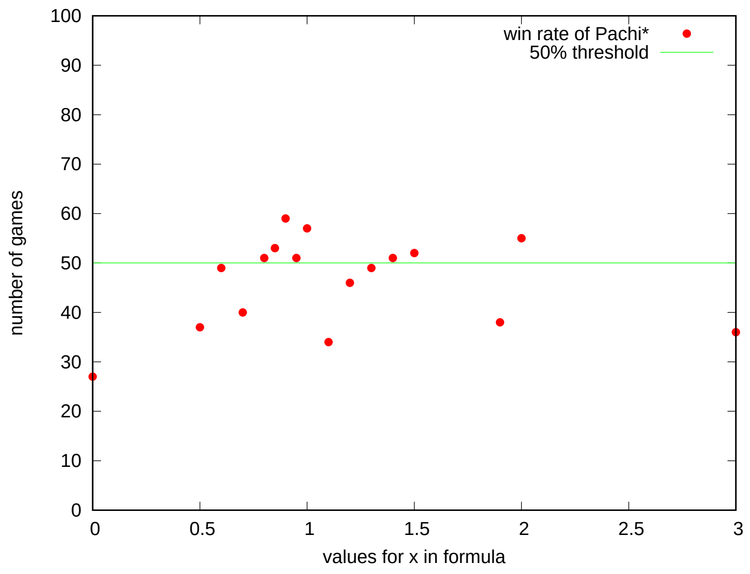


**Figure 39:** This figure shows the win rate of Pachi* with a varying *equivalent experience* of $\sqrt{y - \frac{moves}{3 \cdot 100}} \cdot (eqex)$ for different values of $y$.

## 5.5 Conclusion on Pachi

The *big move* heuristic, aside from the obvious problem that it still consumes too much time, altered Pachi so that Pachi* (with the best configuration) won around 59% of games against Pachi. We had the most success using $\sqrt{0.9 - \frac{moves}{3 \cdot 100}} \cdot (eqex)$ as the *equivalent experience* for the *big move* Prior. We incorporated a phantom function into Pachi* to compensate the time Pachi* needed for the calculation of the heuristic. Without it, Pachi* would have never performed nearly as good. Also, restricting the heuristic to a certain number of moves proved to be a good choice. Although, it is uncertain if the move restriction to 50 moves of the game was optimal. Over all, we are unsure about the results' relevance. The win rate seems rather high but further testing is crucial to see if Pachi* can perform better than Pachi on the long run. We had not enough time to test this sufficiently as the tests were very time consuming. Lowering the thinking time to raise the number of games could mitigate that problem in the future. A further improvement we suggest is optimizing the performance of the algorithm further. The faster the heuristic works the more *Simulations* of MCTS can be made. Additional Performance could also be achieved by restricting the heuristic in terms of search tree depth. For example, if the heuristic would be called only for the root node and its direct child nodes it could save a lot of time and we assume it would not hinder the end result. Also, the example in figure 19 of section 3.4 showed that experimenting with potential could be crucial or at least beneficial. Testing with different Prior values $Q_{prior}(s, a)$ could also be profitable. Last but not least, we feel that the whole potential of the eigenvector is not used sufficiently. The whole eigenvector could also be used as a rough estimate of the whole board by giving each state action pair $(s, a)$ a $Q_{prior}(s, a)$ and $n_{prior}(s, a)$ according to the amount of the corresponding eigenvector entry.

# 6 Conclusion

At the beginning of this thesis, the reader was introduced to the game of Go, some of its historical background, and its rules. Then, the reader was introduced to the *big move* heuristic and its purpose and mathematical background. After that, the performance of several eigenvector functions was tested. In the last section, the modified Go program Pachi* was tested against Pachi*. As a last point, we will list all critique and ideas that we collected from the different sections of the thesis.

**Conclusion on the Heuristic Approach**

The *big move* heuristic in itself seems to be an interesting approach, even if slightly flawed, and could potentially be used for other games than Go as well. It is unclear if the current algorithm is the easiest and fastest solution for implementing the heuristic. Further investigation could potentially simplify this approach by computing the intersection farthest away from other stones directly without using eigenvectors. Eigenvector computation still has some advantages. The algorithm could be enhanced in various ways. One idea would be to use the full potential of the eigenvector for calculation of a rough biased estimate for every legal move on the board. Every state action pair $(s, a)$ would have a value $Q_{prior}(s, a)$ that determines how far away it is from other stones and from the border. The whole board could be estimated fast and roughly just as RAVE [20] does, explained in section 2.3.2. Another question that arose while working on this thesis was how equivalent big points should be treated. If for example two or more areas are equally big as shown in figure 16 section 3, the heuristic only proposes one move to play, which translates to one particular entry of the computed eigenvector. One could change the output of the algorithm to an array containing all biggest moves to mitigate this problem. This problem would also be solved when using the proposed whole board approach. Another part of the heuristic that could need more attention is the aspect of potential. Right now, the heuristic always searches for the center of the largest area. Figure 19 in section 3.4 illustrates a board situation in which the current algorithm does not perform very well. The center of the board oscillates the most, even with potential on the sixth and seventh line, because the third and fourth line are relatively occupied. It is uncertain, if this behavior is wanted or not, as the example is already at the end of Fuseki.

**Conclusion on Performance Tests**

Several direct and iterative methods were tested. In the end, the best methods were using the Arnoldi Method [91]. It was interesting to see how well the Iterative Power Method performed on the Dirichlet matrices [4] and how important the choice of factorization is when computing the inverse of a matrix. It seems possible that the performance of eigenvector computation can be improved further. Fortran Libraries seem to be still relevant as they were used in the background of several methods. ARPACK [31] seems to be a good choice performance wise, as the Arnoldi Method [91] did perform best in the end. Therefore, it could be fruitful to write a wrapper for ARPACK in C. Furthermore, investigating more time in finding an optimal factorization for the Dirichlet matrices could turn out successful as well. Spectral Shift, on the other hand, seems impractical due to its overall weak performance. Maybe it would have performed better with a different method than the Power Method but it cannot be said for sure.

**Conclusions on Testing Pachi***

Pachi*, the modified version incorporating the *big move* heuristic, played against Pachi, an open source Go program [3]. It is possible, that the test results are not reliable due to the small test size. The tests were very time consuming so it was not possible to double check all results. Thus, the reader should take into account that the results could be partially inconclusive. Nevertheless, it is plausible to conclude that Pachi* performed at a similar level as Pachi. The best result achieved was Pachi* winning 59 out of 100 games against Pachi. If possible, it could be beneficial to increase the test size from a hundred to a thousand games to minimize fluctuation in the future. If this is too time consuming, decreasing the thinking time of the Go programs could help. All in all, this would be a faster way to estimate the strength of Pachi*. Later, the thinking time could be increased again in order to test the more promising results. It also seems very important to test future versions of Pachi* against humans. Even if Pachi* performs better than Pachi on the long run, it could be due to overfitting. Testing against different opponents seems crucial. Due to the scope of this thesis, different values $Q_{prior}(s, a)$ for the *big move* heuristic in Pachi have not been tested. Investing more time to balance the influence of the heuristic in Pachi* seems crucial. Another important question that came up was how often the heuristic should be computed. It may be beneficial to compute the heuristic only until a specific depth in the search tree is reached. After all, the heuristic's significance decreases with more moves on the board. Therefore, it is also questionable if confining the heuristic to exactly 50 moves is best. If the reader might want to implement the whole board estimation proposed earlier in this section he could use the Pattern Prior of Pachi as reference. The implementation could be similar as each state action pair would get a particular *equivalent experience* depending on the correspondent eigenvector entry.

# Appendices

# A Performance Files of eigen()

## A.1 matrix A

```
1 0.52
5 1.07
10 1.86
50 7.26
100 13.66
500 67.00
1000 132.34
```

## A.2 matrix B

```
1 0.56
5 0.80
10 1.19
50 4.11
100 7.76
500 36.48
1000 70.38
```

## A.3 matrix C

```
1 0.536
5 0.936
10 1.472
50 5.080
100 9.580
500 46.256
1000 92.832
```

# B Performance Files of gsl_eigen_symmv()

## B.1 matrix A

```
1  0.240
5  1.252
10  2.184
50  11.072
100  22.700
500  136.180
1000  258.824
```

## B.2 matrix B

```
1  0.300
5  1.280
10  2.480
50  12.816
100  31.464
500  158.684
1000  306.116
```

## B.3 matrix C

```
1  0.252
5  1.276
10  2.460
50  11.440
100  25.896
500  151.700
1000  280.736
```

# C Performance Files of Naive Self Implementation

## C.1 matrix A

```
1  0.62
5  1.67
10  2.92
50  12.53
100  24.39
500  120.76
1000  240.75
```

## C.2 matrix B

```
1  0.68
5  1.66
10  2.90
50  12.91
100  25.37
500  125.37
1000  250.77
```

## C.3 matrix C

```
1  0.920
5  2.264
10  3.920
50  17.304
100  33.828
500  171.232
1000  259.120
```

# D Performance Files of numpy.linalg.eigh()

## D.1 matrix A

```
1 0.48
5 0.74
10 1.03
50 3.47
100 6.58
500 30.92
1000 62.38
```

## D.2 matrix B

```
1 0.50
5 0.75
10 1.06
50 3.45
100 6.41
500 30.05
1000 59.59
```

## D.3 matrix C

```
1 0.632
5 1.056
10 1.536
50 5.376
100 10.124
500 48.800
1000 91.120
```

# E Performance Files of scipy.linalg.eig

## E.1 matrix A

```
1  0.70
5  1.67
10  2.83
50  12.42
100  24.57
500  120.29
1000  239.80
```

## E.2 matrix B

```
1  0.70
5  1.72
10  2.95
50  12.92
100  25.34
500  125.17
1000  250.92
```

## E.3 matrix C

```
1  0.73
5  1.76
10  3.11
50  13.60
100  26.56
500  131.40
1000  260.42
```

## F Performance Files of scipy.linalg.eigh()

### F.1 matrix A

```
1  0.56
5  1.03
10  1.62
50  6.33
100  12.18
500  58.92
1000  117.22
```

### F.2 matrix B

```
1  0.50
5  0.75
10  1.06
50  3.45
100  6.41
500  30.05
1000  59.59
```

### F.3 matrix C

```
1  0.684
5  1.000
10  1.408
50  4.648
100  8.708
500  42.252
1000  63.232
```

## G Performance Files of EigenSolver

### G.1 matrix A

```
1  0.42
5  2.02
10  4.11
50  20.32
100  40.12
500  202.02
1000  406.79
```

### G.2 matrix B

```
1  0.54
5  2.74
10  5.22
50  25.51
100  50.14
500  249.93
1000  506.78
```

### G.3 matrix C

```
1  0.44
5  2.12
10  4.27
50  21.19
100  41.14
500  205.86
1000  408.12
```

# H Performance Files of Naive Self Implementation

## H.1 matrix A

```
1  0.272
5  0.788
10  1.452
50  6.808
100  13.340
500  65.660
1000  131.092
```

## H.2 matrix B

```
1  22.44
5  66.296
10  132.208
50  581.924
100  1153.904
500  5652.776
1000  11273.832
```

## H.3 matrix C

```
1  1.184
5  3.420
10  6.344
50  29.000
100  58.924
500  288.848
1000  577.404
```

# I Performance Files of Spectral Shift using Eigen

## I.1 matrix A

```
1 0.032
5 0.084
10 0.136
50 0.580
100 0.856
500 4.156
1000 8.276
```

## I.2 matrix B

```
1 0.576
5 2.576
10 5.020
50 27.344
100 49.184
500 189.536
1000 384.872
```

## I.3 matrix C

```
1 0.076
5 0.276
10 0.532
50 2.648
100 5.384
500 19.000
1000 38.552
```

# J Performance Files of scipy.sparse.linalg.eigs

## J.1 matrix A

```
1  0.496
5  0.512
10  0.528
50  0.704
100  0.944
500  2.736
1000  5.004
5000  22.944
```

## J.2 matrix B

```
1  0.500
5  0.468
10  0.548
50  0.760
100  1.012
500  3.164
1000  5.884
5000  27.43
```

## J.3 matrix C

```
1  0.496
5  0.552
10  0.568
50  0.908
100  1.312
500  4.760
1000  8.976
5000  43.092
```

# K Performance Files of scipy.sparse.linalg.eigsh

## K.1 matrix A

```
1 0.496
5 0.500
10 0.512
50 0.708
100 0.884
500 2.656
1000 4.812
5000 22.068
```

## K.2 matrix B

```
1 0.480
5 0.512
10 0.548
50 0.760
100 1.064
500 3.416
1000 6.324
5000 29.920
```

## K.3 matrix C

```
1 0.468
5 0.520
10 0.568
50 0.980
100 1.460
500 5.324
1000 10.204
5000 49.392
```

# L Performance Files of ConjugateGradient

## L.1 matrix A

```
1  0.392
5  1.528
10  3.324
50  14.960
100  29.564
500  148.260
1000  291.596
```

## L.2 matrix B

```
1  0.432
5  1.852
10  3.472
50  16.980
100  33.456
500  169.572
1000  331.712
```

## L.3 matrix C

```
1  0.208
5  0.864
10  1.496
50  7.272
100  14.14
500  70.716
1000  137.468
```

# M Performance Files of BiCSTAB

## M.1 matrix A

```
1  0.196
5  0.620
10  1.252
50  5.708
100  11.028
500  56.172
1000  117.448
```

## M.2 matrix B

```
1  0.156
5  0.548
10  1.004
50  4.864
100  9.400
500  46.372
1000  93.468
```

## M.3 matrix C

```
1  0.124
5  0.292
10  0.496
50  2.176
100  4.384
500  21.600
1000  42.380
```

## N Performance Files of SparseLU

### N.1 matrix A

```
1  0.044
5  0.128
10  0.200
50    0.908
100  2.004
500  8.780
1000  17.704
```

### N.2 matrix B

```
1  0.052
5  0.132
10  0.196
50    0.924
100  1.804
500  9.100
1000  18.224
```

### N.3 matrix C

```
1  0.056
5  0.144
10  0.268
50    1.148
100  2.184
500  10.612
1000  21.792
```

# O Performance Files of SimplicialLDLT

## O.1 matrix A

```
1  0.040
5  0.084
10  0.128
50  0.496
100  0.972
500  4.692
1000  9.708
```

## O.2 matrix B

```
1  0.040
5  0.076
10  0.140
50    0.548
100  1.060
500  5.060
1000  10.512
```

## O.3 matrix C

```
1  0.052
5  0.100
10  0.172
50  0.732
100  1.528
500  6.916
1000  14.072
```

## References

[1] John Bates. A beginner's introduction to go. http://www.cs.umanitoba.ca/~bate/BIG/BIG.contents.html#anchor967338. Accessed: 2016-11-24.

[2] Gailly Baudis. Pachi pattern files. http://pachi.or.cz/pat/. Accessed: 2016-11-24.

[3] Petr Baudiš and Jean-loup Gailly. Pachi: State of the art open source go program. In *Advances in computer games*, pages 24–38. Springer, 2011.

[4] Türker Bıyıkoglu, Josef Leydold, and Peter F Stadler. Laplacian eigenvectors of graphs. *Lecture notes in mathematics*, 1915, 2007.

[5] Shirley; Black, Noel; Moore and Eric W. Weisstein. MathWorld biconjugate gradient stabilized method. http://mathworld.wolfram.com/BiconjugateGradientStabilizedMethod.html. Accessed: 2016-11-29.

[6] Susan Blackford. Implicitly Restarted Lanczos Method. http://www.netlib.org/utk/people/JackDongarra/etemplates/node117.html. Accessed: 2016-11-23.

[7] Susan Blackford. www.netlib.orginverse iteration. http://www.netlib.org/utk/people/JackDongarra/etemplates/node96.html. Accessed: 2016-11-25.

[8] Richard Bozulich. *THE SECOND BOOK OF GO*. ISHI PRESS INTERNATIONAL, 1400 N. Shoreline Blvd., Bldg. A-7 Mountain View, California, first edition edition, 1987.

[9] Andries E. Brouwer. Triple Ko. http://homepages.cwi.nl/~aeb/go/rules/triple_ko.html. Accessed: 2016-11-24.

[10] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[11] Xindi Cai and Donald C Wunsch II. Computer go: A grand challenge to ai. In *Challenges for Computational Intelligence*, pages 443–465. Springer, 2007.

[12] Daniela Calvetti, L Reichel, and Danny Chris Sorensen. An implicitly restarted lanczos method for large symmetric eigenvalue problems. *Electronic Transactions on Numerical Analysis*, 2(1):21, 1994.

[13] Nguyen Duc. mathforcollege.com-cholesky and ldlt decomposition. http://mathforcollege.com/nm/mws/gen/04sle/mws_gen_sle_txt_cholesky.pdf. Accessed: 2016-11-25.

[14] enz. GoGui. http://gogui.sourceforge.net/. Accessed: 2016-11-25.

[15] Joe Hicklin et al. JAMAa java matrix package. http://math.nist.gov/javanumerics/jama/. Accessed: 2016-11-28.

[16] X. Sherry Li et al. SuperLU. http://crd-legacy.lbl.gov/ xiaoye/SuperLU/. Accessed: 2016-11-25.

[17] John Fairbairn. Gobase go in ancient china. http://gobase.org/reading/history/china/. Accessed: 2016- -.

[18] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012.

[19] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[20] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[21] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

[22] Andrew Grant. *400 years of go in Japan*. Slate & Shell, 2003.

[23] Jim Hetrick. waves and modes. http://www-personal.umd.umich.edu/~jameshet/IntroLabs/IntroLabDocuments/150-12%20Waves/Waves%205.0.pdf. Accessed: 2016-11-29.

[24] Arno Hollosi. SGF user guide. http://www.red-bean.com/sgf/user_guide/. Accessed: 2016-11-28.

[25] Ishigure Ikuro. *In the Beginning - The Opening in the Game of Go*. THE ISHI PRESS, INC., Ishi Press International CPO Box 2126 1400 North Shoreline Blvd. Tokyo, Japan Building A7 Mountain View, CA 94043 USA, 1973.

[26] Kaoru Iwamoto. *Go for Beginners -*. Pantheon Books, New York, american. edition, 1976.

[27] Manja Marz Josef Leydold. Dirichlet eigenvectors go go. Conference on Applications of Graph Spectra in Computer Science, 2012.

[28] Kim Seong June. *A Dictionary of Modern Fuseki, The Korean Style*. Kiseido, 2004.

[29] Matthew Macfayden. *The Game of Go - Learning and Mastering the Most Challenging Game in the World*. Carlton Books, Limited, Italien, new edition edition, 2002.

[30] Massoud Malek. The Power Method for Eigenvalues and Eigenvectors. http://www.mcs.csueastbay.edu/~malek/Class/power.pdf. Accessed: 2016-11-25.

[31] N.A. ARPACK. http://www.caam.rice.edu/software/ARPACK/. Accessed: 2016-11-25.

[32] N.A. British Go Association a brief history of go. http://www.britgo.org/intro/history. Accessed: 2016-10-18.

[33] N.A. British Go Association frequently asked questions about go. http://www.britgo.org/press/faq.html. Accessed: 2016-11-14.

[34] N.A. Eigen bicgstab. https://eigen.tuxfamily.org/dox-devel/classEigen_1_1BiCGSTAB.html. Accessed: 2016-11-25.

[35] N.A. Eigen eigensolver. https://eigen.tuxfamily.org/dox/classEigen_1_1EigenSolver.html. Accessed: 2016-11-25.

[36] N.A. Eigenlibrary. http://eigen.tuxfamily.org/index.php?title=Main_Page. Accessed: 2016-11-25.

[37] N.A. Eigensimplicialldlt. http://eigen.tuxfamily.org/dox/classEigen_1_1SimplicialLDLT.html. Accessed: 2016-11-25.

[38] N.A. Eigensparselu. https://eigen.tuxfamily.org/dox-devel/classEigen_1_1SparseLU.html. Accessed: 2016-11-25.

[39] N.A. European Go Database. http://www.europeangodatabase.eu/EGD/. Accessed: 2016-11-14.

[40] N.A. European Go Federation. http://www.eurogofed.org/. Accessed: 2016-11-20.

[41] N.A. European Go Federation official ratings system. http://www.europeangodatabase.eu/EGD/EGF_rating_system.php. Accessed: 2016-07-20.

[42] N.A. European Go Federation pro qualification. http://www.eurogofed.org/proqualification/index_2015.html. Accessed: 2016-11-14.

[43] N.A. European Go Federation rating. http://www.eurogofed.org/rating/. Accessed: 2016-11-14.

[44] N.A. gokifu.com cho chikun (9p) vs. cho hunhyun (9p). http://gokifu.com/s/2ie7-gokifu-20160123-Cho_Chikun%289p%29-Cho_Hunhyun%289p%29.html. Accessed: 2016-11-25.

[45] N.A. GSL real symmetric matrices. https://www.gnu.org/software/gsl/manual/html_node/Real-Symmetric-Matrices.html#Real-Symmetric-Matrices. Accessed: 2016-10-17.

[46] N.A. Gtp - go text protocol. http://www.lysator.liu.se/~gunnar/gtp/. Accessed: 2016-08-31.

[47] N.A. http://www.netlib.org lapack. http://www.netlib.org/lapack/. Accessed: 2016-11-25.

[48] N.A. Intel-Developer Zone ?geev. https://software.intel.com/en-us/node/521147. Accessed: 2016-11-25.

[49] N.A. Intel Developer Zone heevd. https://software.intel.com/en-us/node/521123. Accessed: 2016-11-25.

[50] N.A. Intel-Developer Zone ?syevd. https://software.intel.com/en-us/node/521122. Accessed: 2016-11-25.

[51] N.A. KGS Go Server. http://www.gokgs.com/. Accessed: 2016-11-14.

[52] N.A. Netlib subroutine dgeev. http://www.netlib.org/lapack/lapack-3.1.1/html/dgeev.f.html. Accessed: 2016-10-17.

[53] N.A. Netlib subroutine dsyevr. http://www.netlib.org/lapack/explore-3.1.1-html/dsyevr.f.html. Accessed: 2016-10-17.

[54] N.A. Netlib subroutine zgeev. http://www.netlib.org/lapack/explore-3.1.1-html/zgeev.f.html. Accessed: 2016-10-18.

[55] N.A. Netlib subroutine zheev. http://www.netlib.org/lapack/explore-3.1.1-html/zheev.f.html. Accessed: 2016-10-17.

[56] N.A. NumPy. http://www.numpy.org/. Accessed: 2016-11-25.

[57] N.A. OGS. https://online-go.com/. Accessed: 2016-11-14.

[58] N.A. Pandanet IGS. http://pandanet-igs.com/communities/pandanet. Accessed: 2016-11-14.

[59] N.A. R manual eigen base. https://stat.ethz.ch/R-manual/R-devel/library/base/html/eigen.html. Accessed: 2016-11-25.

[60] N.A. scipy.org. https://www.scipy.org/. Accessed: 2016-11-28.

[61] N.A. scipy.orgnumpy.linalg.eig. https://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.linalg.eig.html. Accessed: 2016-11-25.

[62] N.A. scipy.orgnumpy.linalg.eigh. https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.linalg.eigh.html. Accessed: 2016-11-25.

[63] N.A. scipy.orgscipy.linalg.eig. https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.linalg.eig.html. Accessed: 2016-11-25.

[64] N.A. scipy.orgscipy.linalg.eigh. https://docs.scipy.org/doc/scipy-0.18.1/reference/generated/scipy.linalg.eigh.html. Accessed: 2016-11-25.

[65] N.A. scipy.orgscipy.sparse.linalg.eigsh. https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.sparse.linalg.eigsh.html. Accessed: 2016-11-28.

[66] N.A. Sensei's Library go servers. http://senseis.xmp.net/?GoServers#toc2. Accessed: 2016-11-14.

[67] N.A. Sensei's Library japanese rules. http://senseis.xmp.net/?JapaneseRules. Accessed: 2016-11-24.

[68] N.A. Sensei's Library kgs go server. http://senseis.xmp.net/?KGS. Accessed: 2016-08-31.

[69] N.A. Sensei's Library kgsgtp. http://senseis.xmp.net/?KgsGtp. Accessed: 2016-08-31.

[70] N.A. Sensei's Library komi. http://senseis.xmp.net/?Komi. Accessed: 2016-11-24.

[71] N.A. Sensei's Library rank. http://senseis.xmp.net/?Rank. Accessed: 2016-11-29.

[72] N.A. Sensei's Library rank - worldwide comparison. http://senseis.xmp.net/?RankWorldwideComparison. Accessed: 2016-08-31.

[73] N.A. Sensei's Library rules of go. http://senseis.xmp.net/?RulesOfGo. Accessed: 2016-10-18.

[74] N.A. Sensei's Librarygame theory. http://senseis.xmp.net/?GameTheory. Accessed: 2016-11-29.

[75] N.A. TYGEMGO. http://www.tygemgo.com/. Accessed: 2016-11-14.

[76] N.A. Wbaduk. http://www.wbaduk.com/. Accessed: 2016-11-14.

[77] N.A. weiqi.tools. `http:/weiqi.tools`. Accessed: 2016-11-26.

[78] N.A. www.netlib.org blas. `http://www.netlib.org/blas/`. Accessed: 2016-11-25.

[79] N.A. www.netlib.org eispack. `http://www.netlib.org/eispack/`. Accessed: 2016-11-28.

[80] N.A. www.yutopian.com gallery. `http://www.yutopian.com/go/gallery/tanglady.jpg`. Accessed: 2016-11-14.

[81] Phongsaen PITAKWATCHARA. Vibration of Multi-DOF System. `http://pioneer.netserv.chula.ac.th/~pphongsa/teaching/vibration/Ch4.pdf`. Accessed: 2016-11-29.

[82] Chet Ramey. Bash Reference Manual. `https://www.gnu.org/software/bash/manual/bash.pdf`. Accessed: 2016-11-25.

[83] Peter Shotwell. American Go Association a brief history of go. `http://www.usgo.org/brief-history-go`. Accessed: 2016-10-18.

[84] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[85] Gilbert Strang. Eigenvalues and Eigenvectors. `http://math.mit.edu/~gs/linearalgebra/ila0601.pdf`. Accessed: 2016-11-29.

[86] Martin Mohlenkamp Todd Young. Numerical Methods for Eigenvalues. `http://www.math.ohiou.edu/courses/math3600/lecture16.pdf`. Accessed: 2016-11-29.

[87] L. Vandenberghe. Conjugate gradient method. `http://www.seas.ucla.edu/~vandenbe/236C/lectures/cg.pdf`. Accessed: 2016-11-29.

[88] Lieven Vandenberghe. LU factorization. `https://www.seas.ucla.edu/~vandenbe/103/lectures/lu.pdf`. Accessed: 2016-11-25.

[89] Eric W. Weisstein. MathWorld eigen decomposition. `http://mathworld.wolfram.com/EigenDecomposition.html`. Accessed: 2016-11-28.

[90] Eric W. Weisstein. MathWorld laplacian matrix. `http://mathworld.wolfram.com/LaplacianMatrix.html`. Accessed: 2016-11-29.

[91] Chao Yang. The Implicitly Restarted Arnoldi Method. `http://www.caam.rice.edu/software/ARPACK/UG/node45.html`. Accessed: 2016-11-25.

[92] Albert L Zobrist. A model of visual organization for the game of go. In *Proceedings of the May 14-16, 1969, spring joint computer conference*, pages 103–112. ACM, 1969.