Informatique
IUT Nord Franche-Comté

Report on the computers development

S2 B1 Group 9

Cédric Colin
Marvyn Levin
Baptiste Dulieux
Timothée Meyer
Hugues Estrade

10/06/2024

Table of content

1 Introduction

You'll find below a description for each of our computers with the strategies used, statistics, a detailed evaluation and explanations on how was each computer built.

2 Note about the tests

The test were conducted on a computer with the following hardware:

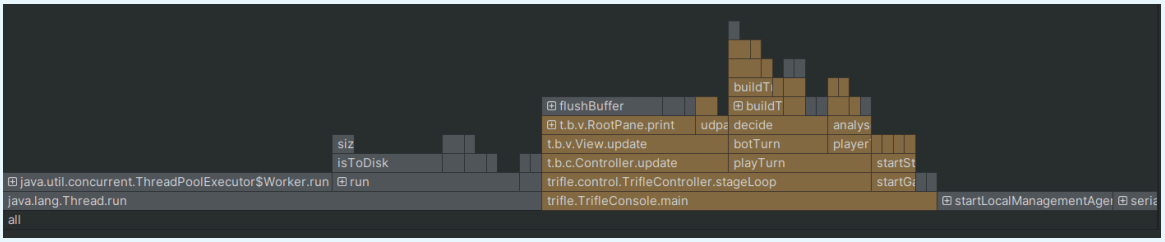
- **CPU:** Ryzen 7 7840HS; 8 cores 16 threads; 3.8Ghz to 5.1Ghz
- **Memory:** 16Gb DDR5 5600Mhz
- **Storage:** NVME 2To 7Gb Read 5Gb Write

Informations

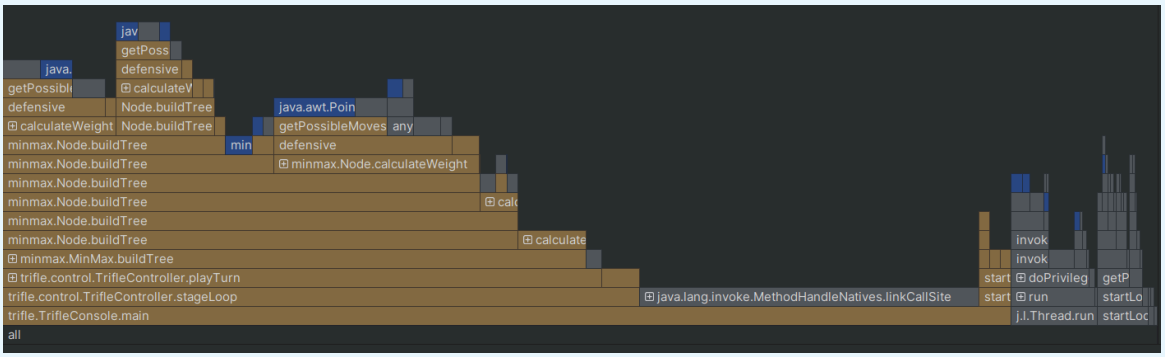
Analyzing memory usage in Java can be challenging due to limitations of the JVM itself. To address this challenge, we'll leverage the powerful capabilities of the IDEA Java Profiler. This comprehensive tool allows us to profile both memory usage and CPU performance, providing valuable insights into the resource consumption of our bot application.

All results of the profiler will be available in the final source code.

Here is an example of the profiling result for the CPU:



And the following flame graph for the memory:



3 Video

The video can be found on *youtube.com* at the following address:
https://youtu.be/2j8DYmu8ZjU

4 Mellie (Min-Max)

Mellie is a robot with a strategic mind, employing the Min-Max principle to outmaneuver her opponents.

Let's delve into how the Min-Max principle works in practice. We'll explore the implementation of the principle, examining the underlying algorithm used. Finally, we'll conclude with a performance evaluation, including the success rate of the approach.

4.1 Min-Max theory

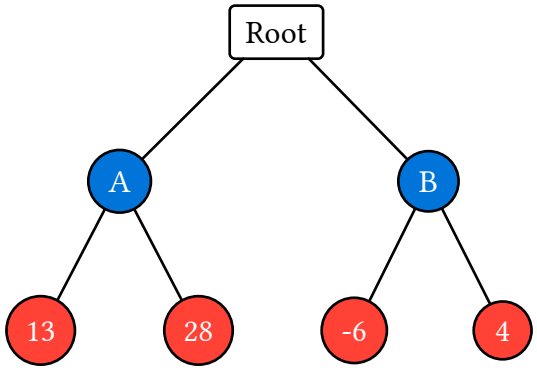
The Min-Max principle, rooted in game theory, is a powerful tool for making optimal decisions in turn-based games with two opposing players. It works by constructing a tree-like structure called a Min-Max tree. Each level represents a turn, with nodes representing possible moves by the bot and its opponent. The weights at the end of each path indicate the expected outcome (good = positive value, bad = negative value) for the bot.

To make a decision, the Min-Max principle guides the bot to **explore all possible future scenarios** through the tree. For **zero-sum games** (where one player's win equals the other's loss), the bot aims to minimize its opponent's maximum score (Max-Min approach). This means the bot prioritizes moves that lead to the worst possible outcome for the opponent, ensuring the best outcome for itself. However, the exponential growth of the tree with depth makes exploring all possibilities computationally expensive.

Informations

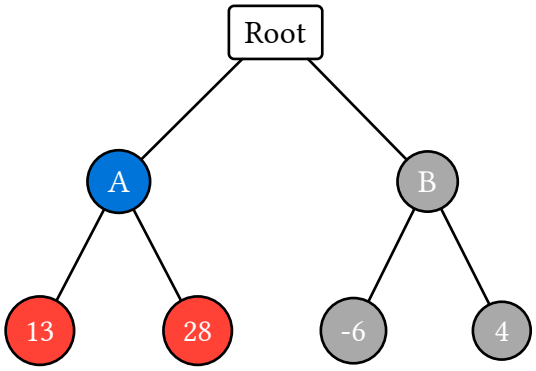
As the Min-Max principle implies that the opponent moves will also be "predicted", the bot's pawn will be **blue** and the opponent moves will be **red**.

Here is a graphical representation of the Min-Max tree:

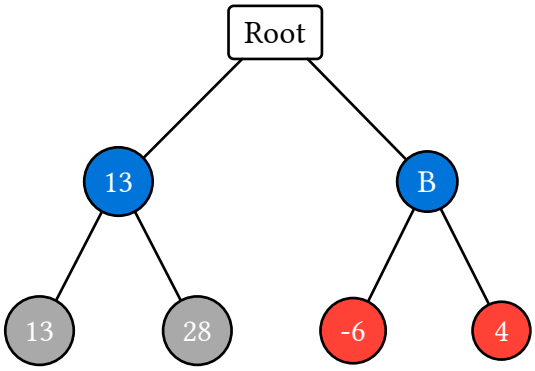


For this example, we used a small tree where there is only two possibilities and a depth of only 2.

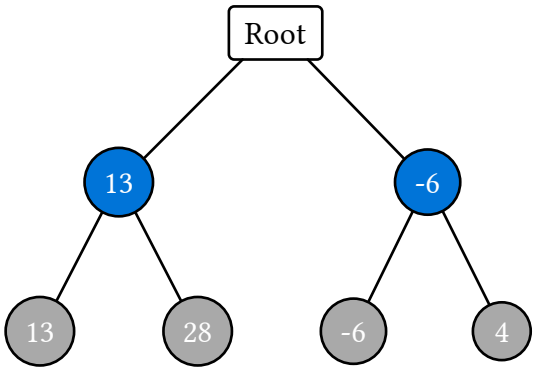
So, for this example, we'll apply the Min-Max principle and see if we should choose A or B:



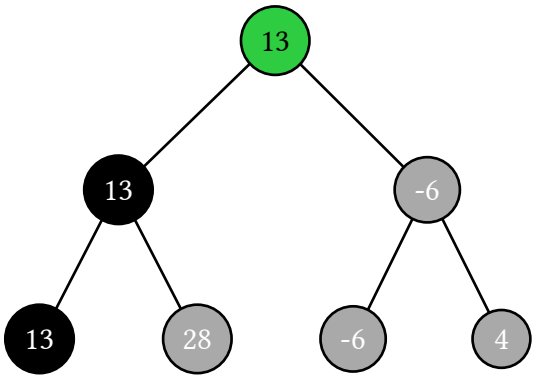
First, let's start with the branch **A**.
We want to **minimize** the cost, as it's the opponent move. Between 13 and 28, **13** is the minimum.



We can now go to **B** and apply the same as the *first step 1.*
Between –6 and 4, we want to **minimize**, so **–6** is our choice.



After the *first 1* and *second 1* steps, we want to **maximize** the cost, which is between **13** and **–6**. The max is **13**.



Finally, we can see here our tree with the path that was chosen, accordingly to the Min-Max principle. We'll want to choose the move **A**.

4.1.1 Why this principle

The Min-Max principle is a well-suited choice for Kamisado. Since Kamisado is a deterministic, perfect-information, zero-sum game, the Min-Max algorithm can effectively evaluate all possible moves and their outcomes on the 8x8 board. This allows the bot to strategically choose moves that minimize the opponent's progress while maximizing its own chance of reaching the opponent's side first.

4.1.2 The Strategy

As discussed earlier, evaluating the end state (cost) is crucial for the bot's decision-making. In our implementation, the cost considers factors like the bot's pawn proximity to the opponent's side (penalty) and occupying the center rows (4 and 5, bonus). This incentivizes the bot to control the central area for better maneuverability.

We employ a double-pronged strategy of attack and defense, governed by a 'defensiveness coefficient.' This coefficient is a value between 0 and 1, where 0 represents purely offensive play and 1 represents purely defensive play. By adjusting this coefficient, the bot can prioritize either attacking the opponent or shoring up its own defenses.

For instance, if the bot encounters an aggressive opponent who moves his pawns closer, the bot can increase the defensiveness coefficient to prioritize defense and avoid getting blocked. Conversely, against a more cautious opponent, the bot can decrease the coefficient to take a more offensive approach. This dynamic strategy can be challenging for human players to predict, as the bot can adapt its moves based on the opponent's behavior.

4.1.2.1 The Defense

Our instinct for defense might be to simply move a pawn to block as many of the opponent's pawns as possible. However, a more strategic approach is needed. Here's how the defense aspect of the cost calculation works:

- **Blocking Opponent Pawns:** We add **25** to the cost if the bot's move can block an opponent's pawn, restricting its movement. This incentivizes the bot to limit the opponent's options.
- **Prioritizing Defense:** If the opponent's move can **potentially** win the game (based on the game state), we prioritize defense by adding a higher penalty (**50**) to the cost. This ensures the bot reacts decisively to critical situations and **avoids losing the game**.
- **Preventing Self-Hinderance:** To avoid blocking its own pawns and limiting future options, the cost is divided by the number of the bot's pawns that would be blocked by the move. This discourages moves that restrict the bot's own mobility.
- **Capping and Coefficient:** The final defense score is capped (between −100 and 100, as said before) to ensure it stays within a manageable range. The defense coefficient, between 0 and 1, is then applied to this final defense score. A higher coefficient increases the overall weight of defense in the final cost calculation. For example, with a coefficient of 0.8 (leaning defensive), a final defense score of −40 would contribute −32 ($-40 * 0.8$) to the overall cost. This allows the bot to dynamically adjust its defensive posture based on the situation.

4.1.2.2 The Attack

The attack strategy aims to find opportunities to win or gain a positional advantage. Here's how the attack aspect of the cost calculation works:

- **Prioritizing Wins:** If a move leads to an **immediate win** for the bot, the cost is set to the maximum value (100), reflecting the **most favorable outcome**.
- **Center Control Bonus:** Moves made in rows 4 and 5, which grant better control of the board, receive a small bonus (e.g., +5) added to the cost, only if the move can't win.
- **Attack Score Calculation:** An attack score is calculated between 0 and 100 based on factors like the bot's pawn proximity to the opponent's side (closer = higher score). This score represents the potential for an attack.
- **Final Cost:** To incorporate both defense and attack, we subtract the attack score (0-100) from the maximum cost (100). A higher defense coefficient (x) will reduce the impact of the attack score by multiplying by (1-x). This effectively balances prioritizing defense when needed while still considering opportunities for offense.

For example, with a maximum cost of 100 and a defense coefficient of 0.5 (balanced approach), an attack score of 70 would result in a final cost of 30 ($100 - 70 * (1 - 0.5)$).

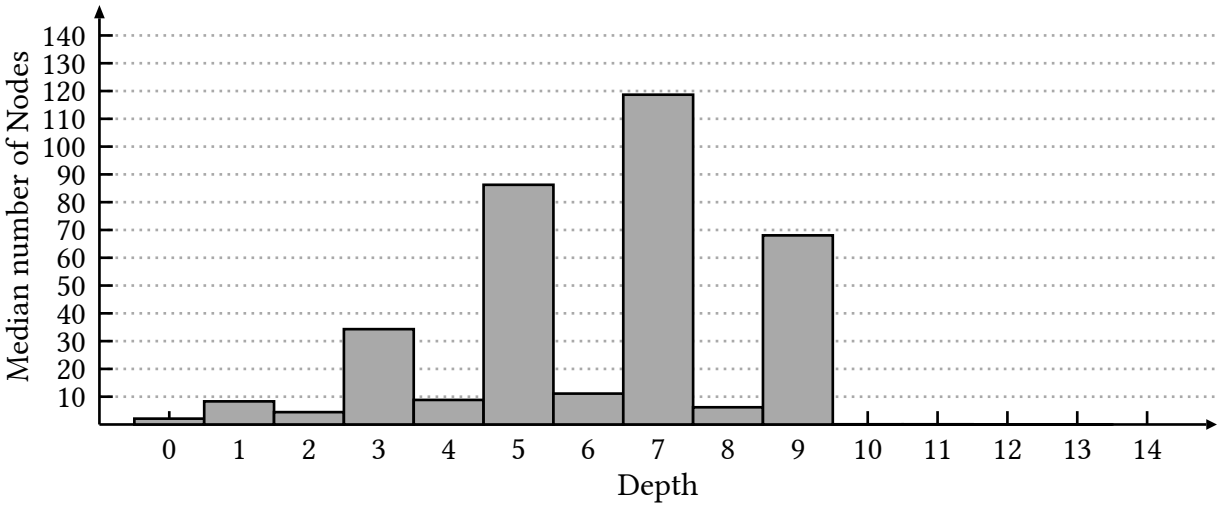
4.2 Evaluation

To benchmark the performance of the bot, we implemented an automated testing process using a *bash script 1* located at `resources/run_game.sh` in the source code. This script pits the bot against itself. The evaluation is based on a dataset of 740 299 samples.

4.2.1 Layouts

By "layouts," we refer to specific game board configurations with a defined depth (number of moves made). During benchmarking, we aimed to evaluate the bot's behavior under increasing complexity. To achieve this, we set a maximum depth of **50** and measured the number of nodes explored by the Min-Max algorithm for each layout.

The following bar chart illustrates the median number of nodes explored per layout depth.



Graph 1: Median number of nodes per layout for 740 299 samples

Depth	Median number of nodes
0	2.1051777727647885
1	8.322184684836802
2	4.447326012867774
3	34.30567919178602
4	8.835956822851307
5	86.25396495199912
6	11.118887098321084
7	118.68160297393351
8	6.195087390365244
9	68.06025268168672
10	0.0744807165753297
11	0.0744807165753297
12	0.013674204611920319
13	0.013674204611920319

The provided *Layout table 1* and *bar chart 5* demonstrate an exponential growth in the number of nodes (moves) explored by the Min-Max algorithm for each layout depth (number of moves made). However, this growth seems to significantly decrease after a depth of 7, becoming negligible at depth 10.

Furthermore, the data reveals a consistent trend where the opponent (another instance of the bot) has a significantly higher number of possible moves compared to the playing bot at each layout depth (excluding layouts with less than 1 move). This could be due to factors like the bot prioritizing specific strategies that limit its own move options.

Finally, the evaluation based on 740 299 samples indicates an average of 348 moves generated during game-play.

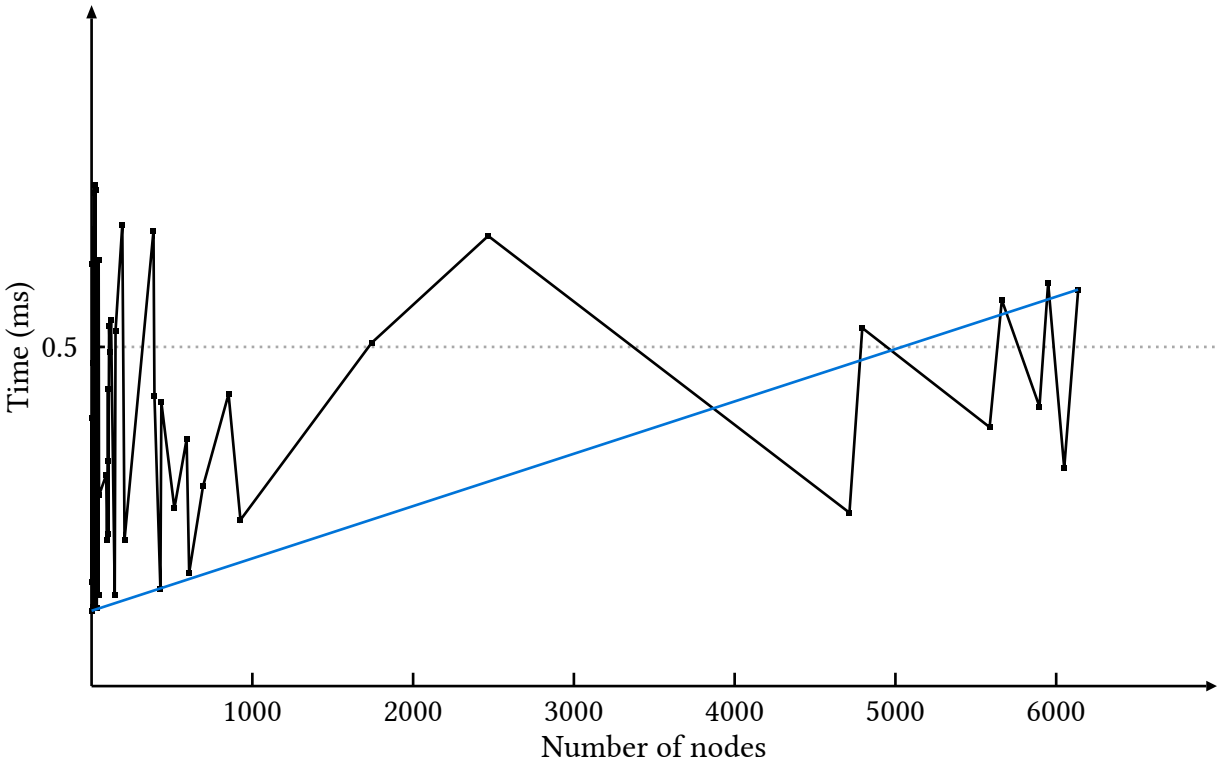
Considering the maximum depth is set to 50, this suggests that when playing against itself, the bot can typically identify winning moves within 10 moves of any possible starting position. This implies a high degree of efficiency in the Min-Max algorithm for this specific scenario (bot vs bot).

4.2.2 Building the Min-Max tree

There are two approaches to implementing the Min-Max algorithm: directly searching for the best move or building a Min-Max tree and then searching within the tree. We opted for the tree-building approach for several reasons:

- **Simplicity:** Building the tree is a more straightforward approach compared to directly searching for the best move.
- **Statistical Analysis:** The tree structure allows us to gather valuable statistics about the search space, which can be helpful for further optimizations.
- **Adaptability:** This approach is more adaptable to incorporating other algorithms, such as Alpha-Beta pruning, for improving efficiency.

On average, the time required to build the tree is of 4.252 418ms, as you can see below:



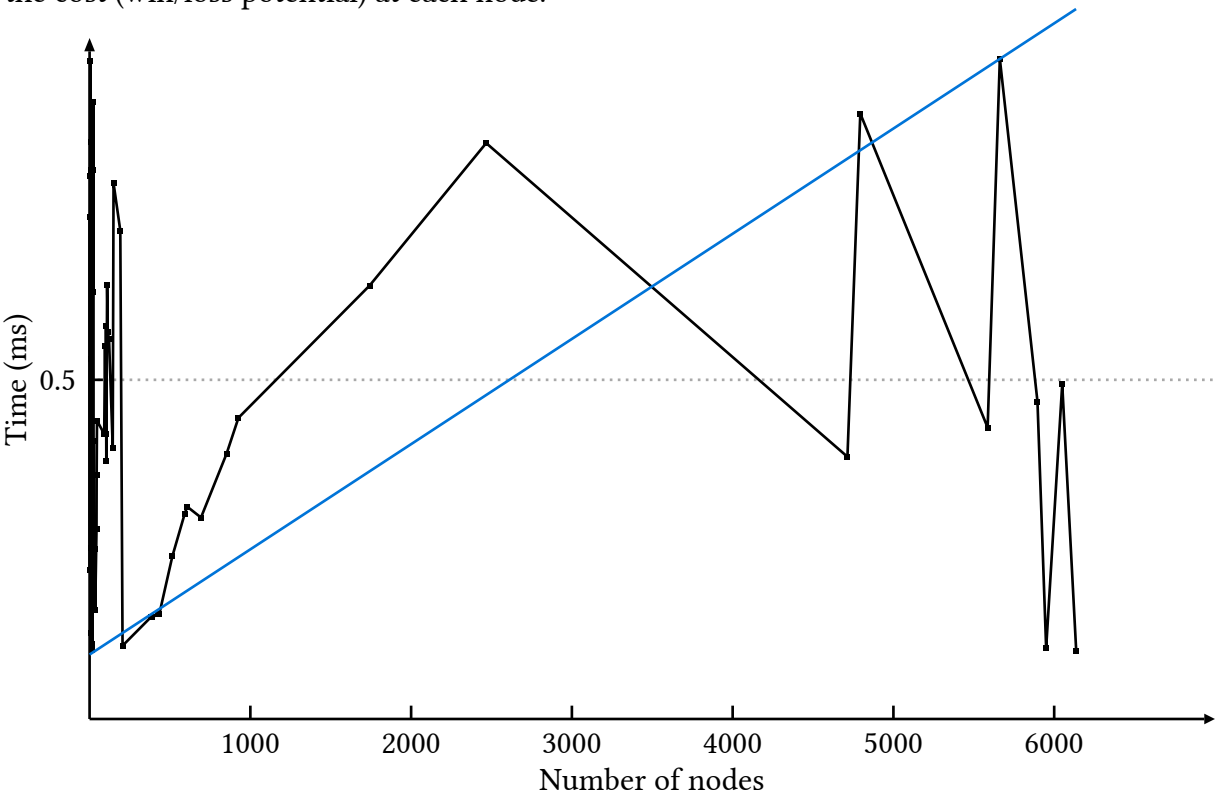
Graph 2: Min-Max graph of the average time taken by the tree building task based on the number of nodes

The provided graph (*Graph of the building tree time 6*) depicts the relationship between the time taken to build the tree and the layout depth (number of moves made). While the graph may not be visually appealing, it suggests an upward trend in build time as the depth increases.

Interestingly, the graph also indicates that a significant portion of the tree exploration time is concentrated within the first 30 entries. This means that for many game scenarios, the Min-Max tree exploration might not be as time-consuming as initially perceived.

4.2.3 Selecting the best move

Once the Min-Max tree has been built (as discussed in Section 3.2.2), the next crucial step involves identifying the optimal path (sequence of moves) leading to the most favorable outcome for the bot. This process leverages the minimax algorithm to traverse the tree and evaluate the cost (win/loss potential) at each node.



Graph 3: Min-Max graph of the average time taken by the path finding task based on the number of nodes

The provided information indicates a median search time of 0.64661 milliseconds, which suggests an efficient implementation of the minimax algorithm.

The provided graph (*Min-Max graph of the time taken by the tree building task 7*) depicts the relationship between the number of nodes explored in the Min-Max tree and the time taken by the algorithm to identify the best move. While the dataset is extensive, containing approximately 740 299 samples, there appears to be a significant amount of noise present in the data.

Despite the noise, a general trend can be observed through the blue curve overlaid on the graph. This trend suggests a linear relationship between the number of nodes and the time taken. This implies that the time complexity of finding the best move grows proportionally with the number of nodes explored in the Min-Max tree.

4.2.4 Memory and CPU usage

This section analyzes memory usage and CPU performance for several game simulations. The profiling data is stored in `.jfr` files located in the `resources/profiling` directory.

4.2.4.1 Game no.1

Profiling file: `resources/profiling/game1.jfr`

Observations:

Building the Min-Max tree consumed approximately **37.47 Mb** of memory during this game, with a total execution time of **50ms**. However, the dominant factor in memory usage appears to be the calculation of cost weights (win/loss potential) for each node, which utilized **32.23 Mb** (**86%** of the total). Interestingly, building the tree structure itself without the cost weights only required **5.23 Mb**.

A closer look at the cost weight calculations reveals that the defensive section, responsible for evaluating potential opponent moves, consumed a substantial portion (**90%** or **29.09 Mb**) of the memory used for this process. This suggests that the complexity of evaluating all possible opponent moves might be a significant contributor to memory usage.

4.2.4.2 Game no.2

Profiling file: `resources/profiling/game2.jfr`

Observations:

In this game, memory usage reached **83.25 Mb** for building the Min-Max tree, with CPU time remaining under **110ms**. Notably, calculating the cost weights consumed a significant amount of memory (**69.7 Mb** or **83%** of the total). Similar to Game 1, the defensive section dominated the cost calculation, utilizing **95%** (**66.15 Mb**) of the memory dedicated to this process.

4.2.4.3 Game no.3

Profiling file: `resources/profiling/game3.jfr`

Observations:

This game exhibited a similar trend to the previous games. While CPU time remained low (only **70ms** for building the tree), memory usage was high (**84.32 Mb**). As in the other games, cost weight calculation was the primary memory consumer (**75.83 Mb** or **89%** of the total). Analyzing the cost calculation further reveals that the defensive section again took the majority of the memory (**97%** or **73.73 Mb**), with the aggressive section utilizing only a small portion (**3%** or **2.1 Mb**).

4.2.4.4 Observations of the 3 games

- **Dominant Memory Consumption:** Across all three games (`game1.jfr`, `game2.jfr`, `game3.jfr`), building the Min-Max tree consumes a significant amount of memory. The memory usage primarily stems from calculating the cost (win/loss potential) for each node in the tree. This cost calculation accounts for both defensive and aggressive strategies.
- **Breakdown of Cost Calculation:** The defensive section, which evaluates potential opponent moves and their impact, consistently consumes **the majority** of memory used for cost calculation (around **90%** in all games). The aggressive section, focusing on the bot's offensive opportunities, utilizes a **significantly smaller** portion of memory (around **10%**).

- **High Memory Usage in Defensive Calculations:** Within the defensive section, the most memory-intensive part appears to be calculating all possible moves the opponent can make with each pawn. This suggests that the number of possible opponent moves and the associated calculations might be a bottleneck for memory usage.
- **CPU Time:** While the provided data doesn't show extremely high CPU times (ranging from 50ms to 110ms for building the tree), it's worth keeping an eye on CPU usage in future analyses, especially if the game complexity increases.

Here is the code in problem:

```
for (Pawn opponentPawn: boardStatus.getPawns((playerID + 1) % 2)) {
    List<Point> possibleMoves = boardStatus.getPossibleMoves(
            (playerID + 1) % 2,
            opponentPawn.getCoords()
    ); // the bottleneck occurs for this precise call!

    boolean canWin = possibleMoves.stream()
            .anyMatch(m -> playerID == 0 ? m.x == 0 : m.x == 7);

    for (Point possibleMove: possibleMoves) {
        if (possibleMove == move)
            defensiveWeight += canWin ? 50 : 25;
    }
}
```

This problem will be addressed in the section 3.3 (*Opportunities and improvements 4.3*).

4.2.5 Efficiency

Information

The efficiency of the bot was assessed through a series of games against human players. This approach provides valuable insights into the bot's performance under real-world conditions and how it adapts to different playing styles.

All games was done in a single round each.

4.2.5.1 Observations

Based on extensive testing, the current iteration of the Kamisado bot exhibits a distinct playing style:

- **Strong Defensive Strategy:** The bot prioritizes building a strong defensive position, effectively blocking your pawns and limiting your movement options.
- **Tactical Advantage Creation:** The bot demonstrates tactical skill in forcing you to make moves that open up weaknesses in your own defense. This could involve strategically placing its pawns to create situations where your pawns are forced to move and compromise your defensive setup.

4.2.5.1.1 Potential Imbalance

While these defensive and tactical capabilities are valuable, the bot currently seems to prioritize them over immediate wins. This suggests a potential imbalance in its decision-making process, where achieving a tactical advantage takes precedence over securing a direct victory when the opportunity arises.

4.2.5.1.2 Moving Forward: Balancing Strategies

To address this potential imbalance, we can explore the following:

- **Rebalancing Win Evaluation:** Refining the evaluation function used by the bot to assign higher weights to moves that lead directly to victory could encourage it to capitalize on winning opportunities more readily.
- **Incorporating Offensive Strategies:** Developing or integrating additional offensive strategies that focus on directly achieving victory could broaden the bot's skill set.

By making these adjustments, we aim to create a more well-rounded bot that excels not only in defense and tactics but also in actively pursuing and securing wins.

4.2.6 Limitations and Further Analysis

It's important to acknowledge that the build time and finding path complexity observed here might not be solely due to the Min-Max algorithm itself, processes running alongside the game can influence the measurements. Additionally, the current dataset size (740 299 samples) might not be sufficient to capture the full picture of the time complexity trend without noise, especially for deeper layouts.

Furthermore, the presence of a significant spike in the build time for some layouts in the middle of the graph (*Min-Max graph of the time taken by the tree building task 6* and *Min-Max graph of the time taken by the tree building task 7*) could be indicative of outliers or noise in the data.

To obtain a more robust understanding of the time complexity, running a larger number of games (e.g., **100,000** games) would be beneficial. This would provide a dataset with over **2,500,000** samples, leading to a more statistically significant analysis of the build time trend.

4.3 Opportunities of improvements

If we like the current implementation, there is also a lot of room for improvements and new functionalities, here is a non-exhaustive list of things we want to implement:

- **Dynamic defense coefficient:** We didn't got enough time to implement the algorithm which can adapt the defense coefficient, it is currently a *random method* which rest the same for the whole game.
- **Less memory print:** If we had the memory efficiency in mind, this is not enough. We clone some objects and may calculate more than one time certain things such as the allowed moves. We'll rewrite some methods to directly use these results without wasting a precious CPU time.
- **Use older results from the algorithm:** In the Min-Max, we always generated the result which will be used after. We can use the tree generated to use less resources, as it'll require to only re-calculate the costs and not to generate all nodes/determine possible moves.
- **Heuristic Evaluation Functions:** Developing more sophisticated heuristic evaluation functions can improve the algorithm's ability to prioritize promising paths early in the search, potentially reducing the overall exploration time.
- **Bottleneck:** Analysis of resource usage (reference: *Section 4.2.4*) reveals a significant bottleneck in the defensive cost calculation within the Min-Max tree. This calculation, which evaluates potential opponent moves and their impact on the game state, consumes a substantial portion of memory and may contribute to slow performance.

5 BotEurdeku (simple arithmetic bot)

BotEurdeku is a simple bot designed to be accessible for beginner.

5.1 Principe

The principe of BotEurdeku is very simple. The main objective of the bot is to create as most winning condition as possible. In the same time create the lowest defeat condition as possible.

5.1.1 Winning and losing condition

A winning condition is a pawn that can win the game in one move. If the opponent play a pawn in a place with the same color that your winning condition pawn, the bot will move his pawn to the winning location. A defeat condition is the strict opposite. If the enemy have a winning condition on you, then this move will be a defeat condition for you.

5.2 What the algorithm do

First we update all the winning and the losing conditions. We also update losing colors, so location that have the same color that one of the pawn in your winning condition.

Then we check all the possible moves available to us. If in our moves available we see a winning condition, we play it and win the game. Otherwise we sort all the possible moves by the proximity with the top, this way the opponent have less chance to block our pawn. We also make sure that we don't play on a defeat color, that leads to a possible defeat move.

5.3 Complexity

This program is short in size (156 LOC), and have a complexity of O (n log n) where n is the number of possible moves. This is due to the fact that the most complex operation is the sorting of the list of possible moves. This complexity is very low as there is at the most 14 possible moves for a pawn and we doesn't look deeper in the game.

5.4 Efficiency

This program is a quite simple bot. He have a very offensive strategy and only look at the current round. This leads to a lack of diversity in the plays. The opponent can capitalize on this weakness to win the game.

5.5 Improvement

To upgrade this algorithm we can :

- Look further in the moves (see possibilities that create our move for the opponent for instance)
- Block losing condition in order to adopt more defensive manner.
- Look for plays that will create victory condition in the next move

6 Annex

6.1 Scripts

```
#!/bin/bash

classes_path="$1"
echo "The game files are stored in ${classes_path}"

read -p "How many games do you want to run? " -r number_of_games_to_run

folder_id=$(ls -l | grep -cE0 "run_[0-9]+")
((folder_id++))

mkdir -p "run_${folder_id}"
mkdir -p "run_${folder_id}/games"

output=-1

run_game(){
    WAIT_BEFORE_END=0 \
      java -cp "${classes_path}" \
      trifle.TrifleConsole 2 \
      --output-moves "run_${folder_id}/games/game_$1.in"

    output="$?"
}

game_count=0

while [ "$game_count" -lt "$number_of_games_to_run" ]
do
  echo "Game no.${game_count}"
  run_game "${game_count}"

  if [ "${output}" != 0 ]; then
    echo "Error: Game execution failed!"
    echo "Failed with code ${output}"
    exit 1;
    break
  fi
  ((game_count++))
done

echo "$game_count games have been run"
```

bash script 1: Bash script used to run a number of games

6.2 Move files

```
C1C2
E8E7
G1C5
B8B7
H1C6
A8A7
C2C3
H8H7
F1C4
G8G7
A1A2
C8C7
E1E2
G7G6
D1A4
E7E6
C4D5
A7A6
B1E4
A6A5
E2C4
G6G5
C3E5
H7H6
C5A7
F8F7
A4B5
C7D6
E5F6
H6H5
B5B6
D8D7
E4E5
H5H4
C4C5
B7A6
A6B5
B5B4
F6E7
B4B3
B6C7
C7D8
```

game1.in

```
C1C2
E8E7
G1C5
B8B7
H1C6
A8A7
C2C3
H8H7
F1C4
G8G7
A1A2
C8C7
E1E2
G7G6
D1A4
E7E6
C4D5
A7A6
B1E4
A6A5
E2C4
G6G5
C3E5
H7H6
C5A7
F8F7
A4B5
C7D6
E5F6
H6H5
B5B6
D8D7
E4E5
H5H4
C4C5
B7A6
A6B5
B5B4
F6E7
B4B3
B6C7
C7D8
```

game2.in

```
e1h4
D8D7
b1b6
D7D6
c1c3
H8H7
f1f7
E8E7
g1g4
C8C7
h4f6
H7H6
g4g7
H6H5
d1a4
E7E6
f7e8
```

game3.in

```
f1f5
G8G7
a1a4
E8E7
g1g6
E7E6
f5f6
H8H7
f6f7
E6E5
a4e8
```

game4.in