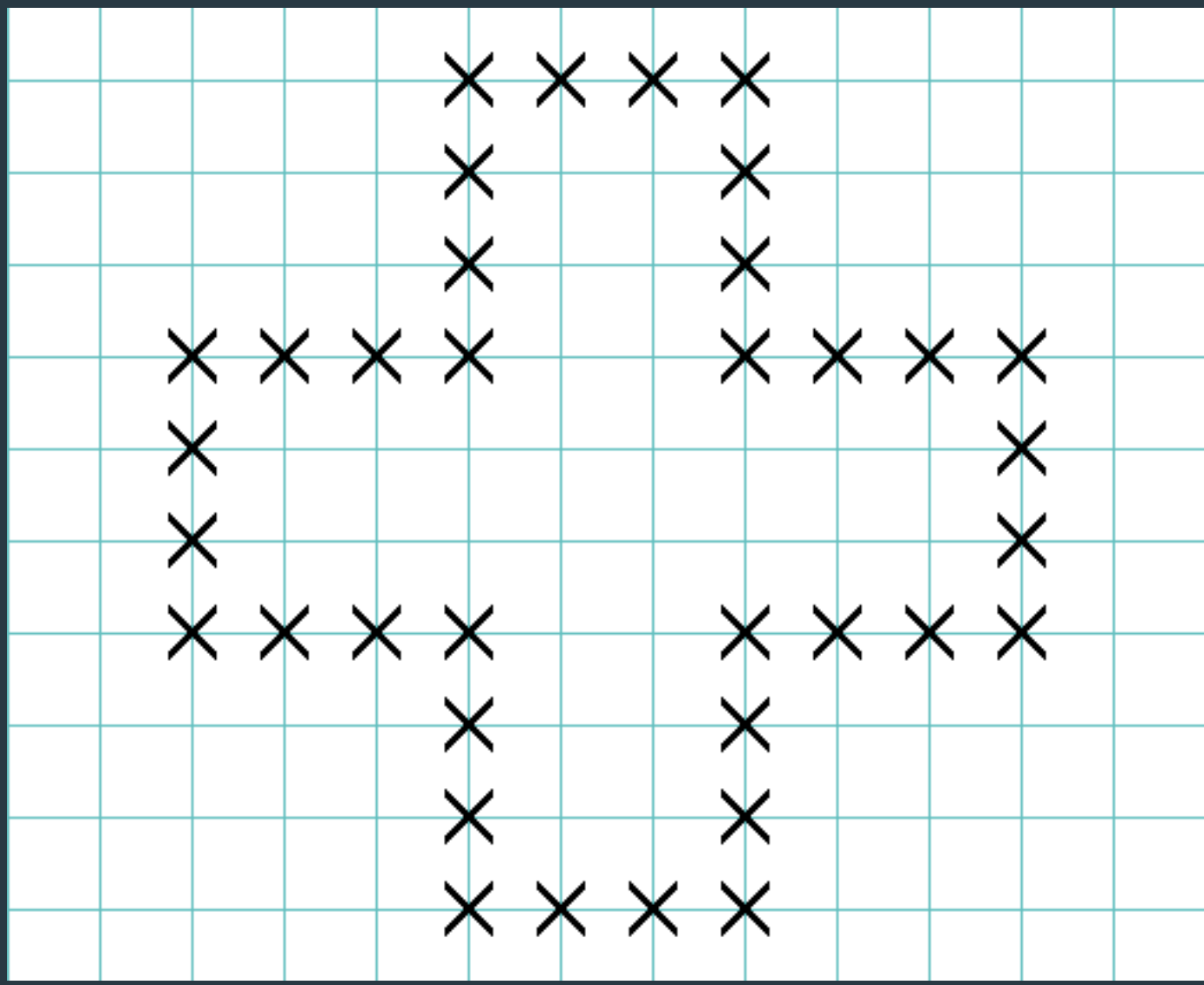
19/12/2023

# Morpion Solitaire Development Report

A Java project by **NEVEU Pierre & DOUBABI Mustapha**



# Contents

# Point class

**Key concepts:** The Point class serves as a fundamental representation of 2D coordinates, offering essential mechanism for managing and comparing points in the game board. The key point of this class is the hashing of coordinates for the hash code of its instances. Therefore, we ensure the unicity of a couple of coordinates (x, y) in a Set<Point>.

---

- **Key Attributes:**
  - int **x, y**
- **Constructors:**
  - **Point(int x, int y)**
  - **Point(Point p)**: for defensive copy
- **key Methods:**
  - int **hashCode()**: returns hash(x, y)
  - boolean **equals(Object o)**: true if two points have the same coordinates x, y

# PlayedPoint extends Point

**Key concepts:** The PlayedPoint class is used to represent the point that was played and compose lines. Two type of played point are to be listed: 1) the points of the default grid: they are not involved in any line but still are considered as played poitn because they can form lines. 2) The played point: they contains multiple informations about the line that they are involved in. These informations are key to find the next round playable points.

---

- **Key Attributes:**
  - boolean **isEndOfLine:** if the point is an extrimity of a line
  - Set<Direction> **isInvolvedInDirection**: list of directions of the lines the point is involved in
- **Constructors:**
  - **Point(int x, int y**): for the default point of the grid
  - **PlayedPoint(Point p, int id)**: to convert a Point in PlayedPoint when it is played
  - **PlayedPoint(PlayedPoint p)**: for defensive copy

# Line class

### Key concepts

The **Line** class represents an aggregation of points forming a line on the game board. Thanks to the hashing of points with theit coordinates, we are able to set the hash code of a line by hashing all hash codes of the points of the line. Therefore, we ensure the unicity of lines in a Set<Line> when they have points with same coordinates (x, y).

**A line can be in two states:**

1. being a playable line and it is composed of 4 PlayedPoint and 1 Point (that is a playable point).
2. being a played line and then have 5 Played Point

---

- **Attributes:**
  - Set<Point> **points:** points that compose the line
  - Direction **direction**: Indicates the direction of the line, which can be horizontal, vertical, or diagonal.
- **Constructor:**
  - **Line(Set<Point> points, Direction direction)**
  - **Line(Line line)**: for defensive copy
- **Methods:**
  - **equals(Object o)**: True if their set of points and their direction are equal
  - **hashCode()**: Computes the hash code for the line using its points.
  - **updatePlayedPoint()**: change the Point that represent a playable point into an actual PlayedPoint
  - Set<Point> getEndsOfLine(): gives the extremities of the line

# Grid class

### Key concepts

The **Grid** class manages the game board, coordinates points, and lines. The key concept of this class is an “artificial memory” to store its points: we use a HashMap with the hash code as  keys. Therefore, we can access the grid points with their their coordinates with a reduced complexity. For example, if you want to move by (i, j) from one point (x, y), you simply have to hash (x + i, x + j).

The current world record fits in a 25x25 grid. This said, we chose to implement the grid with a non-dynamic size. This helps us to simplify the implementation of the grid.

The grid class contains all methods for searching playable points and managing the evolution of the states of the point and lines along the game

---

- **Key Attributes:**
  - **size**: Length of the square grid's sides.
  - **grid**: A map associating point hash codes with their corresponding points.
  - **lines**: A set containing all grid lines.
  - **playablePoints**: A map linking points to sets of lines, identifying playable points.
  - **minPlayablePoint** and **maxPlayablePoint**: Define the sub-grid's bottom-left and top-right corners for playable point searches.
- **Constructor:**
  - **Grid(Grid grid):** make a defensive copy by making defensive copies of sets of line, maps of points, etc
- **Key Methods:**
  - Set<Line> **findNormalLinesInDirection(Point point, Direction direction):** Detailed in the next part
  - Line **findJointLinesInDirection(Point point, Direction direction):** Detailed in the next part

# 5T AND 5D IMPLEMENTATION

## Development choices:

We have chosen to search all the playable points at each round instead of just looking at the point the player is trying to play. This will be useful later for the research algorithms.

We have decided to simplify the research of playable points: we are testing the playablility of each point of the 25x25 grid. Furthermore, we are repeating this process everytimes the player plays to update the playable points. Indeed, as we are using hashing to navigate through the grid, we are considering that the complexity of this iteration will not be an issue for the quantity of calculus we have to do.

The 5T implementation is simply the 5D imlementation with one more method that will add the possible joint lines.

## Pseudo-code

```
def updatePlayablePoints():
    for p in grid points do
        for d in directions do
            if mode == T then
                jointLine = findJointLines(p, d)
                update playable points
            disjointLines= findDisjointLines(p, d)
            update playable points
        end for
    end for
```

# D-MODE IMPLEMENTATION

Let's assume we are playing a version with lines of size L. We evaluates the playability of a point p by finding the possible lines as followed:

## Pseudo-code

```
def findDisjointointLines(direction):
    lines <- Set
    currentLine <- Line
    for n in neighbours from left to right of direction do
        if neighbour is played and is not in a line of same direction then
            currentLine .add(n)
            if size of currentLine == L - 1 then
                currentLine .add(p)
                lines.add(currentLine )
                currentLine <- Empty
            end if
        else
            currentLine <- Empty
    end for
```

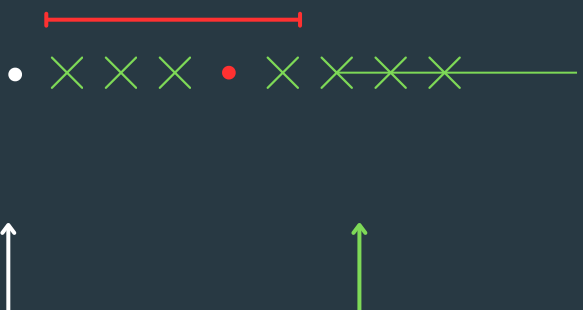
## Illustration

### Example 1

Possible line detected

Played point in a line of another direction

### Example 2



Point not played

Played line in the same direction

# T-MODE IMPLEMENTATION

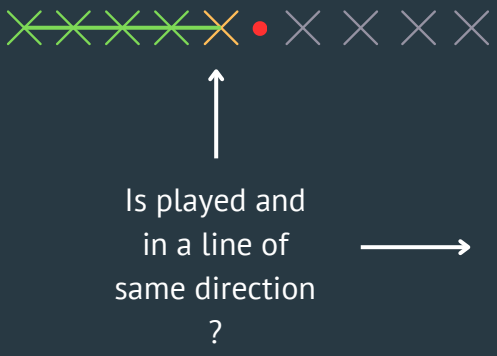
Let's assume we are playing a version with lines of size L. We evaluates the playability of a point p by finding the possible lines as followed:

## Pseudo-code

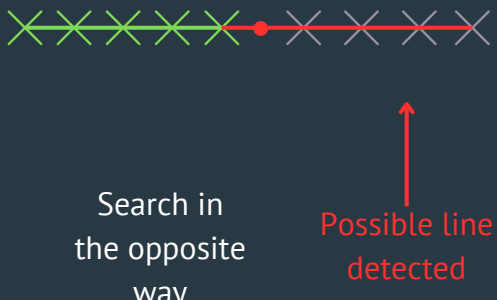
```
def findJointLines(direction):
    line <- empty Line
    for n in direct neighbour do
        if n is played and is end of a line in same direction do
            line.add(n)
            for m in neigbours of opposite direction do
                if m is played and is not in a line of same direction do
                    line.add(m)
                    if ( size of line== L - 1 ) then
                        line.add(p)
                        return line
            end for
        return null
    end for
```

## Illustration

### Step 1




### Step 2



# Random Algorithm

## Concept

The Random Algorithm consists in playing random moves until the game is over. As we have implemented our game by searching every playable points at every round, we just have to chose randomly among the playable points of the grid.

In order to give more flexibility of the usage of this algorithm, we implemented it so it can start from any state of a party. In practise, our implementation of a Grid already represent the state of a party. Hence, we juste have to make a defensive copy of the grid we want so we don't affect the original one and we can launch multiple time the algorithm to calculate statistics.

## Pseudo-code

```
def Grid randomAlgorithm(grid):
    while list of playable points not empty do
        playRandom(point)
        update grid
    end while
    return grid
```

We will use multi threading to get better performances while training the algorithm in order to get enough data to calculate statistics on this algorithm. We chose to divide the process in 4 threads.

```
def trainRandomAlgorithm(grid, int times):
    int timesPerThread = round(times/4)
    for each Thread do
        for i in range(timesPerThread) do
            Grid result = randomAlgorithm(grid)
            save result
        end for
    end for
```

# NMCS Algorithm

(Nested Monte-Carlo Search)

## Concept

The NMCS algorithm consists in the exploration of all rounds of a game (represented by a state in a tree search) and memorizing the best move of all levels. The best move of a level is the best result from random parties from the states of this level. After memorize the best move, we play it and repeat the process until we get to a state with no more playble point. In Morpion Solitaire, the complexity of this algorithm would be too high as there is more than 20 possible moves in most of the rounds. Therefore, we have decided to adapt the NMCS algorithm to our case.

To implement this algorithm, we will inspire from the LAMSADE laboratory scientific paper about the NMCS algorithm (see the link below) in order to compare our results.

LAMSADE laboratory, Paris Dauphine University: NMCS paper

## Theorical pseudo-code

```
def NMCS(grid, level):
    if level == 0 then
        return randomAlgorithm(grid)
    bestGrid = null
    while grid still have playable points do
        for p in playable points of grid do
            nextMoveGrid = play(grid, p)
            NMCS(nextMoveGrid, level - 1)
            update bestGrid
        end for
    end while
```

# NMCS Algorithm

(Nested Monte-Carlo Search)

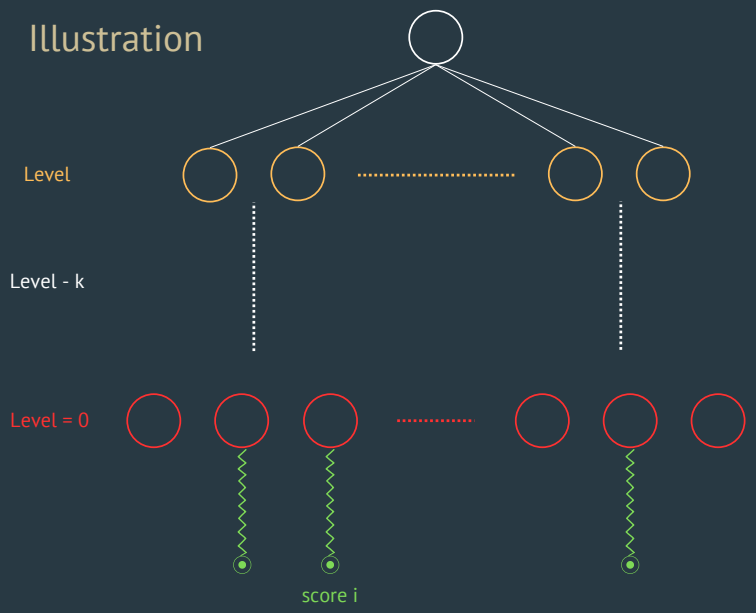
## Adaptation

First, we have set a time limit to ensure to have a fast result. Then, we have decided to limit the NMCS at one search at the chosen depth level (i.e we have deleted the red part in the above theorical pseudo code). Therefore, we can handle a relatively fast algorithm and reduce the complexity. Due to the limited time, the algorithm may not search every states of the given level, hence we shuffle the list of the current level states so we don't limit to the left path.

## Actually implemented pseudo-code

```
def NMCS(grid, level):
    if level == 0 then
        return randomAlgorithm(grid)
    bestGrid = null
    shuffle(playable points)
    for p in playable points of grid do
        if time limit is not reached then
            nextMoveGrid = play(grid, p)
            NMCS(nextMoveGrid, level - 1)
            update bestGrid
    end for
    return bestGrid
```

Illustration

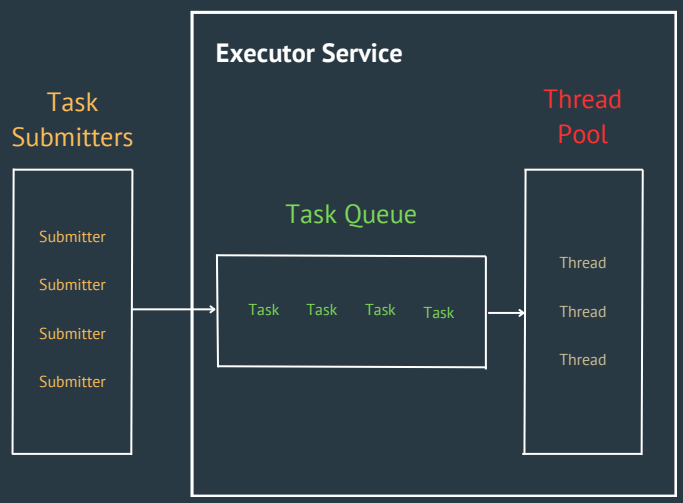



*Level 0 search is equivalent to a random algorithm

## Multi threading pseudo-code

In order to get a faster result, we are creating a thread pool of size 3

```
def NMCS(grid, level):
    if level == 0 then
        return randomAlgorithm(grid)
    bestGrid = null
    create threadPool(n_threads = 3)
    for p in playable points of grid do
        nextMoveGrid = play(grid, p)
        task = NMCS(nextMoveGrid, level - 1)
        submit task
        update bestGrid
    end for
```



# DataManager

## Concept

In ordrer to evaluate the efficiency of our algorithms and compare them to each other with distribution curves, we have developped the DataManager class. This class with essentially statics attributes and methods allows us to:

- Save the data to a csv file
- Get the data from the csv file
- Calculate statistics so that we can compare with distribution curves

---

- **Key Attributes:**
  - static int **currRunningAlgoId:** the id of the current algo running (0, 1, 2, or 3). It has to be static and external to the algorithms classes because NMCS uses the random algorithm and it would cause conflicts.
  - String **path**:  path to the csv file
- **Methods:**
  - static synchronized **insertData(int algoId, String mode, int score**): has to be syncrhonized so the different thread do not conflict with each other
  - **getData(int algoIdp, String mode)**: returns the list of the scores for the given algo and mode
  - Map<String, Double> **calculateStatistics(int algoId, String mode):** gives the necessary statistics for the gaussian curve (mean, standar deviation)

## Csv file illustration

| | A |
|---|---|
| 99 | 0,"5T","50" |
| 100 | 0,"5T","50" |
| 101 | 0,"5T","50" |
| 102 | 1,"5T","62" |
| 103 | 1,"5T","63" |
| 104 | 1,"5T","60" |
| 105 | 1,"5T","60" |
| 106 | 2,"5T","69" |
| 107 | 1,"5T","60" |
| 108 | 2,"5T","65" |
| 109 | 3,"5T","68" |
| 110 | 1,"5D","62" |
| 111 | 2,"5D","65" |
| 112 | 3,"5D","67" |

Random Algorithm (NMCS at depth level 0)
Mode: 5T, Score: 50

NMCS at depth level 1
Mode: 5T, Score: 63

NMCS at depth level 2
Mode: 5T, Score: 69

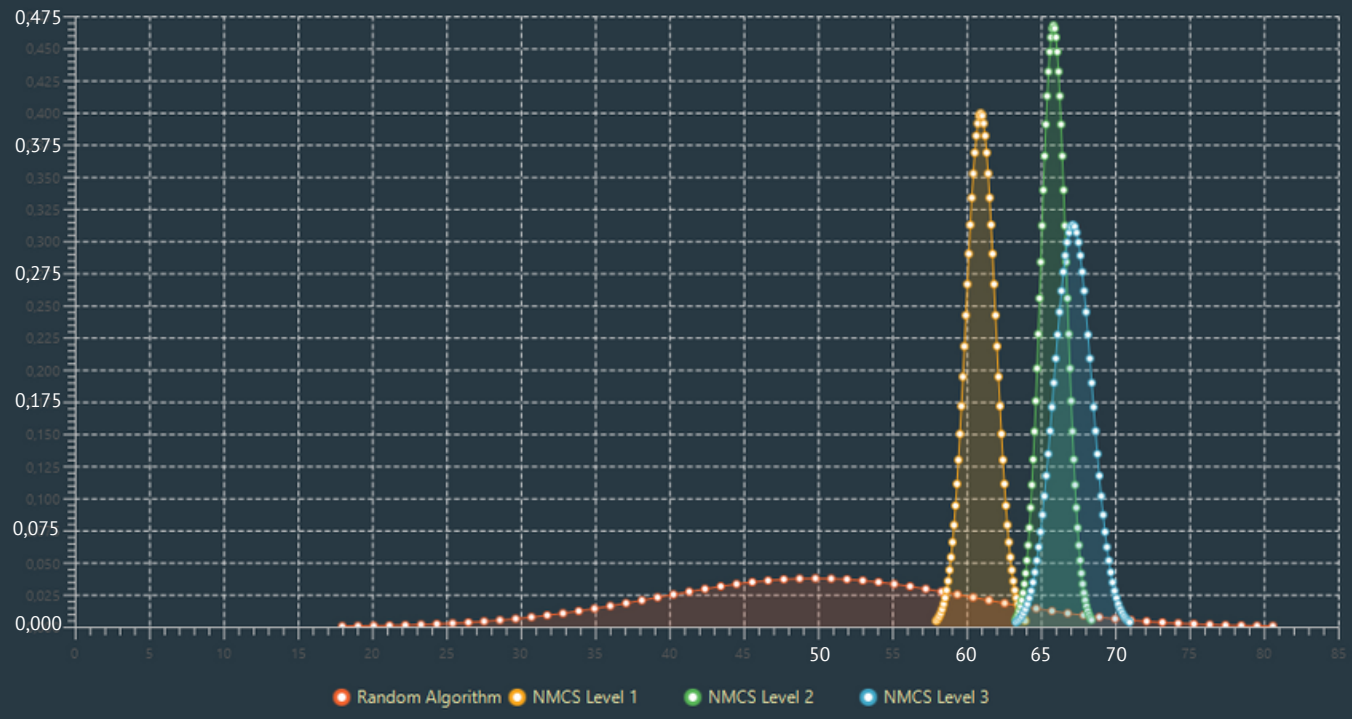
NMCS at depth level 3
Mode: 5T, Score: 68

NMCS at depth level 3
Mode: 5D, Score: 67

# Comparison of the algorithms

## Distribution curves

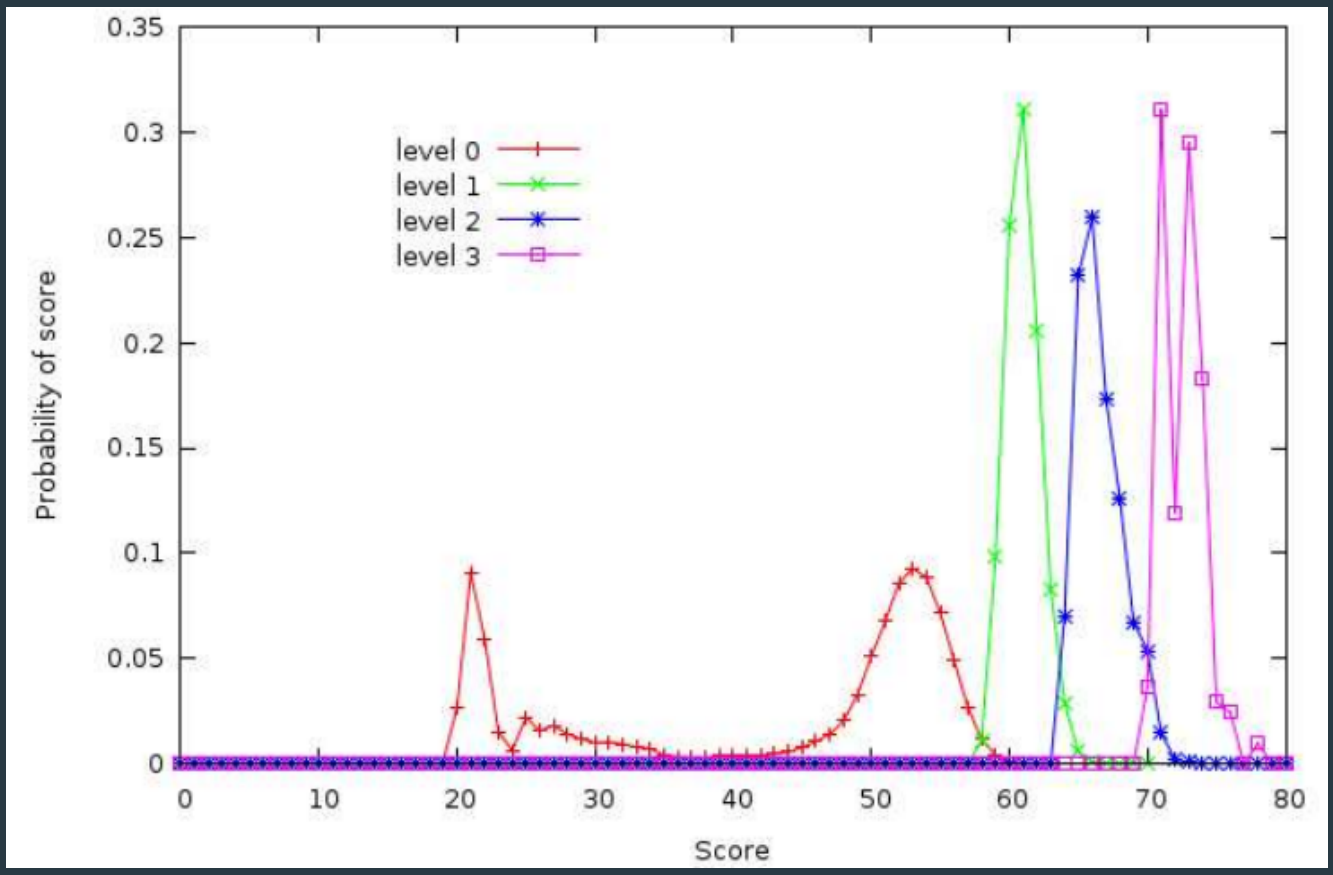
- Our results for 5T mode



# Comparison of the algorithms

## Distribution curves

- Lamsade laboratory results for 5T mode



# Comparison of the algorithms

We can notice that the results of the simplified NMCS algorithm do not diverge much from the results in the LAMSADE paper. Indeed, we can see a better improvment of the scores between level 0 and 1, 1 and 2 than between 2 and 3. The overall best scores oscilliate between 65 and 70 where the scores of the LAMSADE paper are between 70 and 80.

As expected, the random algorithm has a high standard deviation due to its random nature and we can notice the NMCS algorithms have a very low standard deviation. Furthermore, the scores did not depreciate with the time limit. The score doesn't seem to improve much after level 2. For example, we could change the decision making with weights using method like Upper Confident Bound (UCB) to enhance the scores.

# Distribution of the tasks

**Doubabi Mustapha**

- Front development: graphic interface
- Scoreboard
- GameManager
- Cross tests

**Neveu Pierre**

- Back development: different modes implementation
- Algorithms
- DataManager
- Doc redaction
- Cross tests