2020/21

Gourds Implementation

Abstract

The Gourds Implementation is a project that attempts to implement a new sliding block puzzle named gourds designed by researchers in Japan and the Netherlands [1]. Similar puzzles have 15-puzzle and Rush Hour. By the implementation, a playable graphical Gourds application will help researchers have a clearer understanding of this puzzle and the algorithms for solving this puzzle.

Contents

1. Introduction

Gourds is a new sliding-block puzzle designed by researchers in Japan and the Netherlands. An O(n^3) and an O(n^2) algorithms were proposed to solve this puzzle in some specific cases.

This report focuses on the design, implementation, testing and evaluation of this sliding-block puzzle game, including the data structure, the graphic user interface, animation, with the auto-solving algorithms proposed by the game designer.

2. Aims & Objectives

2.1. Aims

2.1.1. Building up a graphical and playable Gourds application.

2.1.2. A hints function can auto solve the puzzle.

2.2. Objectives

2.2.1. Understand and use code to implement two algorithms described in the paper.

2.2.2. Implement the game board and stored with a proper data structure.

2.2.3. Verify valid move of the gourds.

2.2.4. Implement the code of finding Hamiltonian Cycles in triangular grids

2.2.5. Implement the two algorithms as hints to players (on a proper board).

2.2.6. Try to propose a better (no means best but good enough) algorithm to solve the puzzle.

2.2.7. Develop this game based on PyGame.

3. Literature Review

The article <Gourds: A sliding-block puzzle with turning> [1] introduced a puzzle with some solving algorithms for some specific boards.

Like 15-puzzle, Gourds have a board and some blocks could slide on it. However, there are some special designs that slightly increases the implementing complexity of this puzzle game.

To have a general understanding of how it works, there are some basic ideas that should be clarified first.

3.1. Hamiltonian Cycle

A Hamiltonian Cycle, or Hamiltonian Circuit is a closed loop visiting each node exactly only once on a graph. [2]

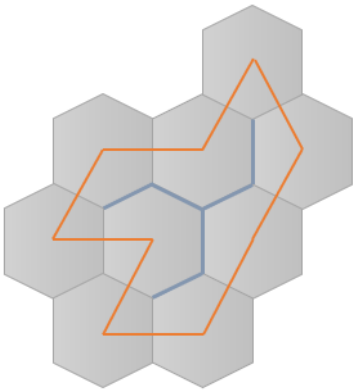


Figure 1: A Hamiltonian Cycle (Orange) on a Hexagonal Grids Board (grey)

In this game, the Hamiltonian Cycle go through all the centre of cells on the board (Figure 1), which is using in auto-solving algorithms.

However, the method to find a Hamiltonian Cycle on hexagonal grids boards have not mentioned in <Gourds> the article. Therefore, the method to find it will be described in the Implementation Section (5).

3.2. Gourds Design and Auto-solve Algorithms

This section would give some general ideas on the Gourds differences from the other puzzle game.

3.2.1. Proper Board

A proper board is the basis of the puzzle. Its design determines whether the game is playable, so it should comply with other puzzle rules (such as the movement of gourds and the goals of the game), and it also affects the design of the auto-solved algorithm (such as Find the Hamilton circle).

In this puzzle, a proper board needs to have the following characteristics:

3.2.1.1. A proper board is a hexagonal grid board to allow because a square grid board (like 15-puzzle) cannot guarantee that every 1x2 gourd can travel over the board [1].

3.2.1.2. The board should (2n+1) cells in which (2n) covered by gourds, and (1) cell is a single uncovered cell for gourds movement.

3.2.1.3. A board with holes cannot be a complete characterization; therefore, a hole-free board is required.

3.2.1.4. Cells are 2-connected, and the board should not be the Star of David. It ensures the existence of a Hamiltonian circle for algorithms implementation [1].

3.2.2. Board Types

3.2.2.1. In **The Coloured Type** (Figure 1 Left 1) [1], cells and ends of gourds covered with various colours. The goal of this type is to match colours between gourds and cells.

3.2.2.2. In **The Numbered Type** (Figure 1 Right 1), cells and ends of gourds have numbers. The goal of this type is to match numbers**.**

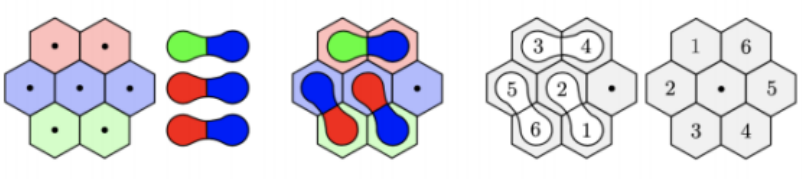


Figure 2: Coloured type and numbered type of cells and gourds. [1]

3.2.3. Gourd Movements

There are three ways to move gourds on a proper board:

- **Sliding**: a pair of gourds could be moved following a straight line in one block (half gourds) each step.
- **Turning**: a pair of gourds could perform a 120degrees tuning.
- **Pivoting**: a pair of gourds could perform a 60 degrees pivot.

  The pivot is chosen rather than a sharp turn because of a smaller room need than the latter one.

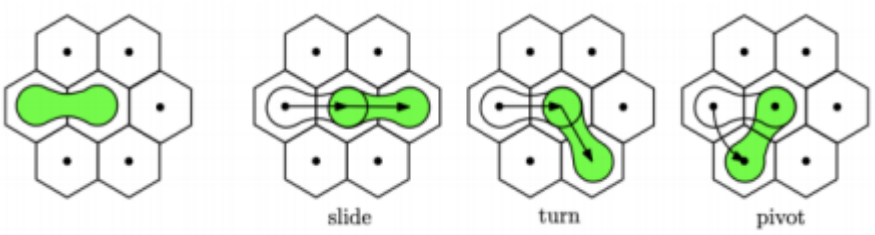


Figure 3: Three movement methods of gourds. [1]

3.2.4. Reconfiguration (Auto-Solve Algorithms)

Reconfiguration is a sequence of guards movement to reconfigure an initial placement to the target placement. There are two algorithms provided in [2] for reconfiguration. Reconfiguration is the central focusing part of this project.

3.2.4.1. The Three Phases

In the <Gourds: a sliding-block puzzle with turning> [1], the reconfiguration consisted of three main phases (Figure 4).

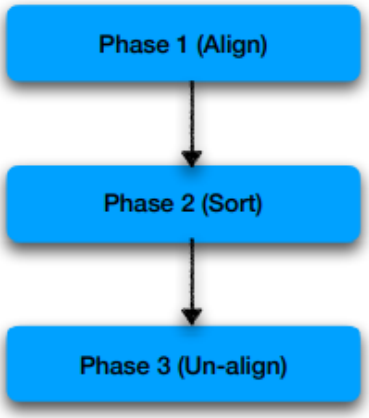


Figure 4: The three phases in reconfigurations

- **Phase One (Aligning)**

In phase one, all gourds should align with a hamiltonian cycle.

As shown in Figure 5, (b) shows a board before The Phase One; (c) and (d) shows a board after The Phase One.

- **Phase Two (Sorting)**

In phase two, gourds should be sorted in a given order. (from Figure 5 (d) to Figure 5(e))

This phase always take the longest time in reconfiguration, and it always determines the complexity of the whole reconfiguration algorithm. Therefore, it is worth wise to discuss this phase deeply.

There are two algorithms to sort these gourds on a Proper Board (described previously in section 3.1.1) proposed in this article, which would be described in the next two sections (3.1.4.2 and 3.1.4.3).

- **Phase Three (Un-aligning)**

Phase Three would un-aligning all gourds in the hamiltonian cycle to the final configuration. (from Figure 5 (e) to Figure 5(f))

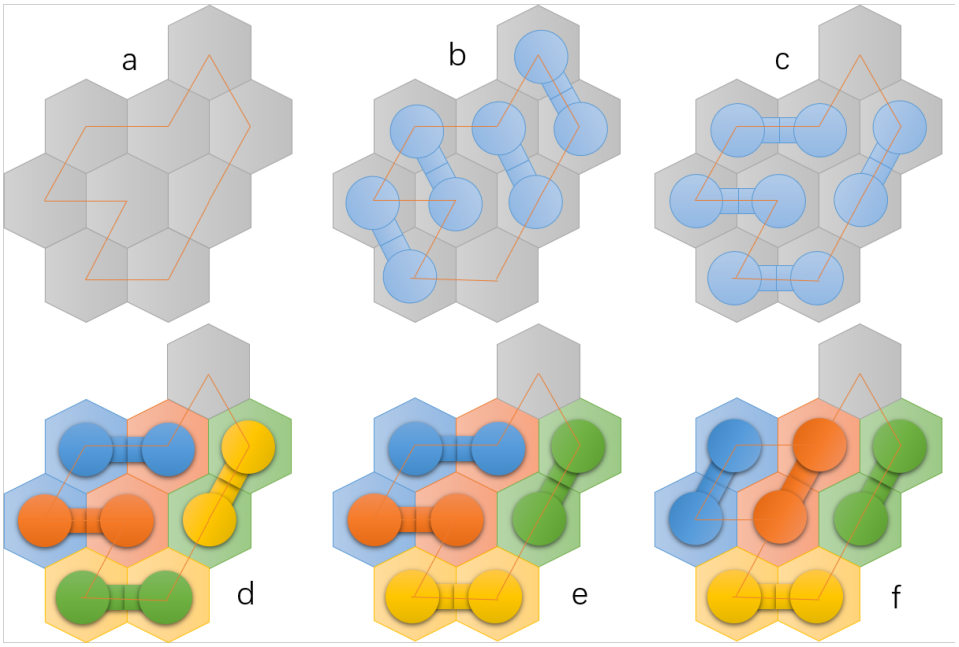


Figure 5: A general idea of reconfiguration. (a) shows a hamiltonian cycle; (b) shows a board before The Phase One; (c) and (d) shows a board after The Phase One; (e) shows a board after The Phase Two; and (f) shows a board after The Phase Three.

3.2.4.2. An O(n^3)-move sorting algorithm[1] (Phase 2)

This is a sorting algorithm in phase two. The complexity was proofed in Theorem 5 of [1]. Which sort the gourds to another order but still alone with the Hamiltonian Cycle (from Figure 5d to 5e).

Depending on the leaf (substructure of a Hamiltonian Cycle), a Proper Board should have at least one of these two types of leaves [1], which can perform the Insertion Sort or the Bubble Sort separately.

3.2.4.2.1. Substructure Type One (Insertion Sort)

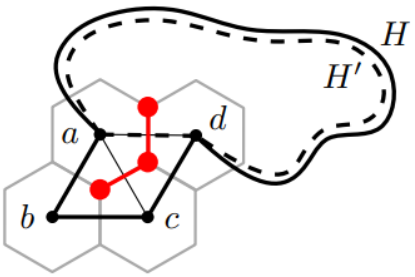


Figure 6: A leaf in type one [1]

For a hamiltonian cycle with four consecutive vertices a, b, c, and d, is a leaf (sub structure) in type one, which could perform Insertion sort.

The origin Hamiltonian Cycle denoted as H in Figure 6. By removing b and c, in the meantime, connecting a to d, there is a smaller Hamiltonian Cycle generated denoted as H'.

Here are three steps performing how to place a gourd to another place in the H cycle.

- **Store:** By moving all gourds along the H cycle, any gourd could be placed and stored on b and c.
- **Spin:** Then moving other gourds along with the H' cycle until the place to be inserted between a and d.
- **Insert:** Finally, moving all gourds along the H cycle again, the gourd stored on b and c could insert into the cycle in the H' cycle

Repeating steps above (store → spin → insert), all gourds in the H cycle could be inserted to an arbitrary place in the H cycle. Then an Insertion Sort achieved.

3.2.4.2.2. Substructure Type Two (Bubble Sort)

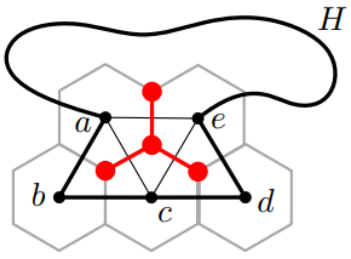


Figure 7: A leaf in type one [1]

For a hamiltonian cycle with five consecutive vertices a, b, c, d, and e, is a leaf (sub structure) in type two, which could perform Bubble sort.

This substructure is even simpler.

Here are two main steps to perform bubble sort.

- Placing two adjacent gourds and an empty cell which need a position exchange on the leaf by moving all gourds along the H Cycle.
- Moving two adjacent gourds on the leaf along a → b → c → d → e or opposite until two gourds position exchanged.

By repeating the above two steps, a Bubble Sort achieved.

3.2.4.3. A worst-case optimal O(n^2)-move sorting algorithm [1] (Phase 2)

The complexity of this algorithm was proved in Theorem 8 of [1].

This algorithm consisted of three main steps (divided and concord), which divided the board by split the Hamiltonian cycle and sorting on the sub-boards separately.

1. Split the Hamiltonian cycle;
2. Exchange gourds between two cycles;
3. Sort the gourds separately.

4. Design

4.1. Development Method

In this project, Iterative and Incremental Development was used as the development method, which is conducive to the rapid realization of basic functions in this project.

The basic board function can be implemented quickly, and the later auto-solved algorithm can be implemented in iterations as increments, which allow testing while implementing to avoid bugs in incrementation.

4.2. Environment

- Operating System: **Windows 10 / OS X 11**
- Programming Language: **Python 3.8**

  Using Python could increase the implementation speed and focusing on the algorithm itself instead of memory leak.
- IDE: **PyCharm Community 2020**

  PyCharm Community is an open source editor and free to use.
- Version Control: **GitHub Desktop**

  The GUI of GitHub Desktop is more user-friendly comparing to Git in command line.
- Libraries:

  Numpy@1.18.0

  PyGame@2.0.1

  PyGame is a cross-platform library that designed for writing 2D video games.

4.3. Directories Structure

4.3.1. Launcher and Boards Configs Container

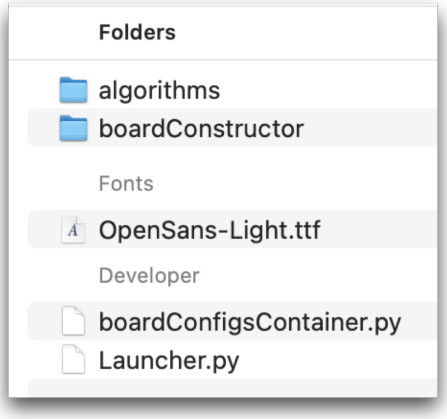


Figure 8: Root folder

The launcher file is used to launch the game, and the boardConfigsContainer file stores several boards' configurations file which will be discussed in 4.4 Data Structure Section.

4.3.2. Board Constructor

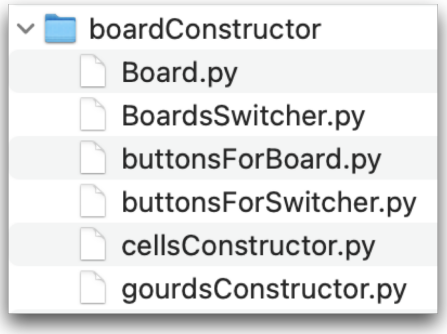


Figure 9: The ‘boardConstructor’ directory

The ‘boardConstructor’ directory contains the framework of the board, the drawer of cells, gourds, buttons and the animations.

4.3.3. Algorithms

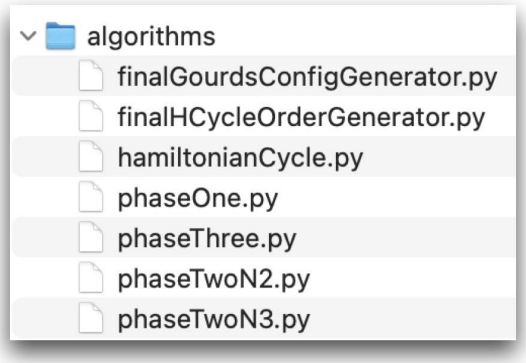


Figure 10: The ‘algorithms’ directory

The ‘algorithms’ directory mainly focusing on the auto-solve algorithms. Each steps of auto-solving were separate into a class (a .py file) to make it more concisely.

4.4. Data Structure (Board Configs Container)

4.4.1. Hexagonal Grid Board

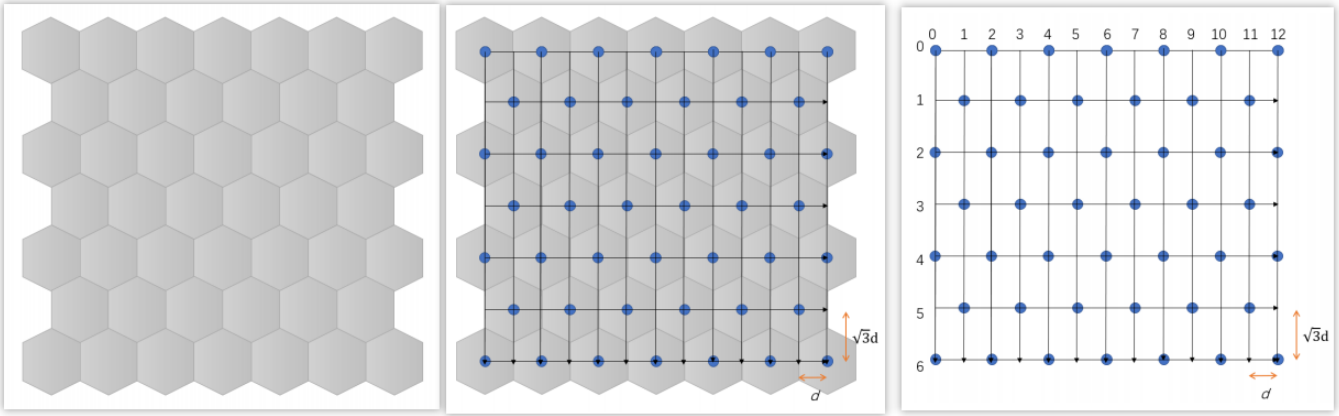


Figure 11: Board Structure

To store a hexagonal grid board (Figure 11, left), we connect each centre of hex-cells (blue dots, Figure 11, middle) using horizontal and vertical lines, which produces a 2D matrix (Figure 11, right)

The 2D matrix (Figure 11, right) has the unit height is about 1.732 ($\sqrt{3}$) times of unit width.

Then digits could be used to represent an index for a centre of cell (blue dot in Figure 11) and use ‘0’ to fill the gap between blue dots (centre of cells). As an example shown in Figure 12 left, a 2D array is defined to represent a board.

```
board = numpy.array([
# x  0, 1, 2, 3, 4,
    [0, 4, 0, 4, 0],   # 0
    [2, 0, 2, 0, 2],   # 1
    [0, 6, 0, 6, 0],   # 2
])
```

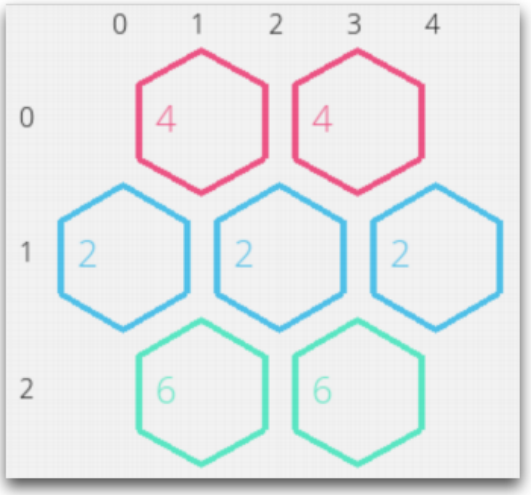


Figure 12: An example of a board with its data stored in computer

Figure 12 shows an example of each digit corresponding to a cell in a hexagonal grid board.

In this example, the digit ‘4’ represent a red hex-cell centre, ‘2’ represent a blue one, ‘6’ represent a green one, and ‘0’ means it is not a centre of hex-cell.

4.4.2. Gourds List

```
gourdsList = numpy.array([
    # x, y, x, y, i1, i2
    [1, 0, 3, 0, 6, 2],
    [0, 1, 1, 2, 2, 4],
    [2, 1, 3, 2, 4, 2],
])
```

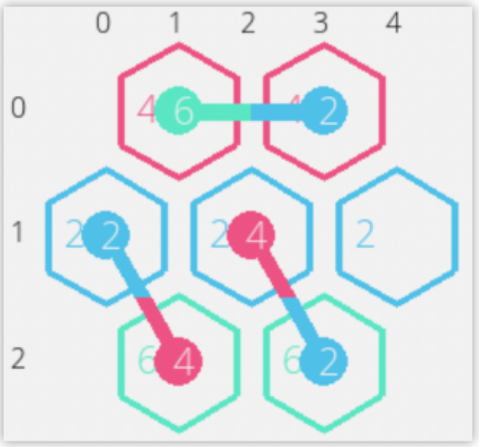


Figure 13: Gourds List for a board

Gourds List provides a board an initial configuration.

As an example in Figure 13, each line in this 2D array is a gourd.

The first line [1, 0, 3, 0, 6, 2] represent the gourd in green(6) and blue(2).

The initial two digit [1, 0] is the location of the first part, and the next two digit [3, 0] is the location of the second part of a gourd. The last two digits [6, 2] represent the index / colour of each part (in this case, 6 is green and 2 is blue).

4.4.3. Colour Library

```
coloursLibrary = {
    'backGround': (242, 242, 242),
    'black': (0, 0, 0),
    'white': (255, 255, 255),
    'hamiltonianCycle': (39, 98, 184),
    'button':(80, 193, 233),
    1: (190, 127, 73),
    2: (80, 193, 233),
    3: (122, 87, 209),
    4: (237, 84, 133),
    5: (255, 232, 105),
    6: (91, 231, 196),
}
```

Figure 14: Colour Library (dictionary) defines the colours using in the GUI

Using the Colour library could give each board a special GUI colours. For example, the correspondence between digit and colour (‘4’ means red) in the previous section (4.4.2) could be easily changed in this library.

For a coloured board, each digit should have a different colour.

For a numbered board, all digits can use only one colour.

4.5. Graphic User Interface

4.5.1. Cells and Gourds Design

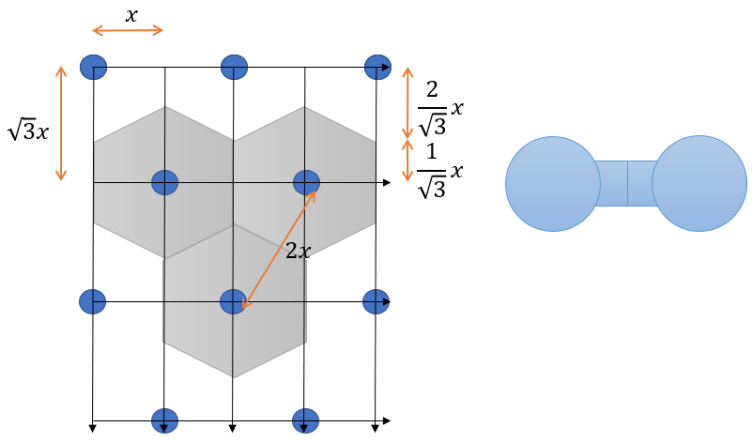


Figure 15: Cells and a Gourd sketch

Figure 15 shows a sketch in drawing cells and gourds, which helps in building the drawer’s for the cells and gourds.

4.5.2. Board Switcher

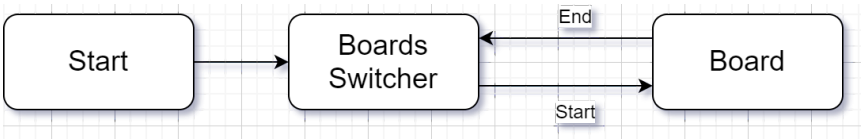


Figure 16: Program Main Loop

By running the program, the Boards Switcher will pop up (Figure 17 and Figure 18).

The boards would show on the switcher.

The two buttons on the top right can be clicked to switch to another board in the boardConfigsContainer.py.

In the Boards Switcher, the axis and the index of cells would shows automatically.

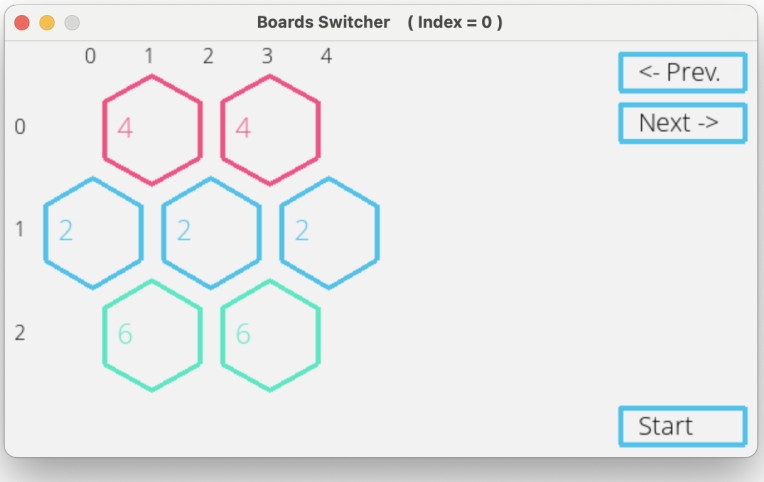


Figure 17: Boards Switcher showing a coloured board

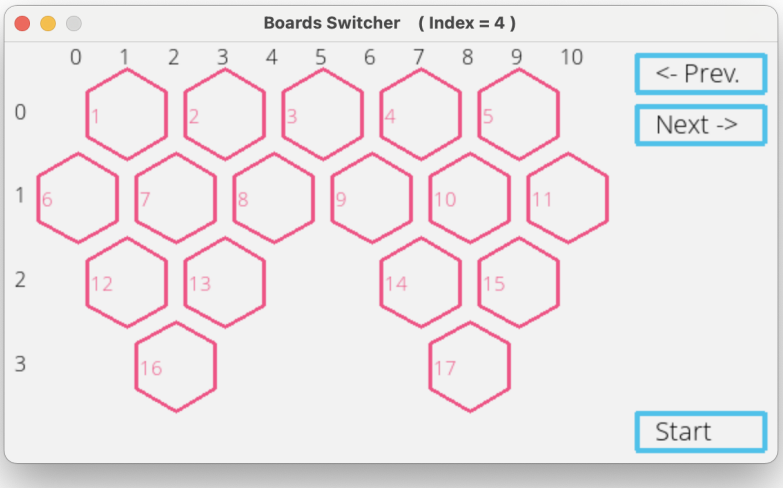


Figure 18: Boards Switcher showing a numbered board

By clicking the down right “Start” button, the Board would pop up (Figure 19).

4.5.3. Board

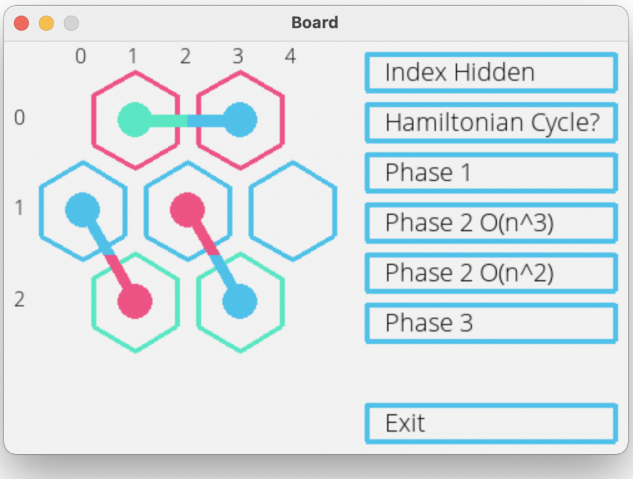


Figure 19: A coloured board

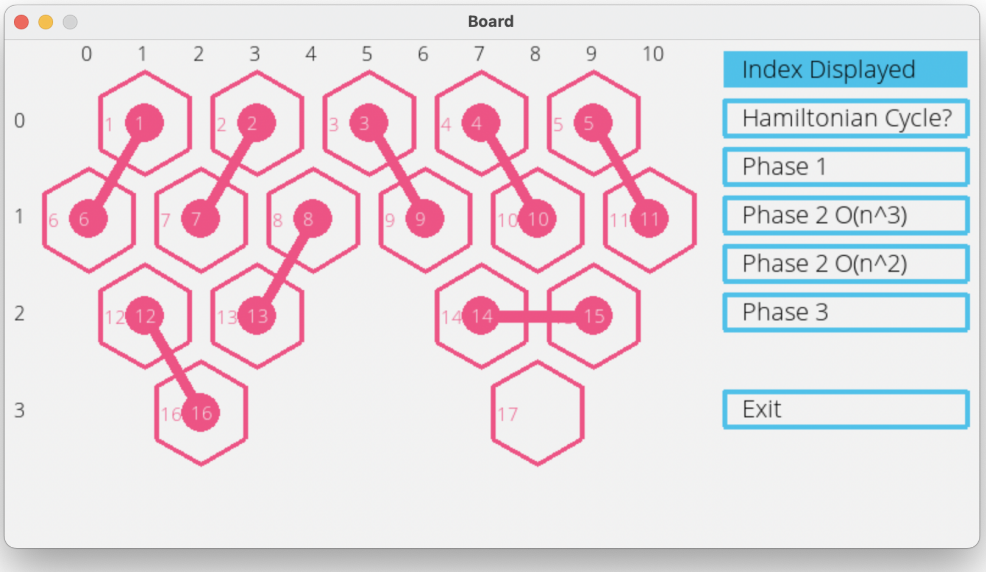


Figure 20: A numbered board

This program could adjust the window size automatically to fit the size of the board.

By clicking the down right “Exit” button to exit the game and back to the Boards Switcher.

4.5.3.1. Index for each cell

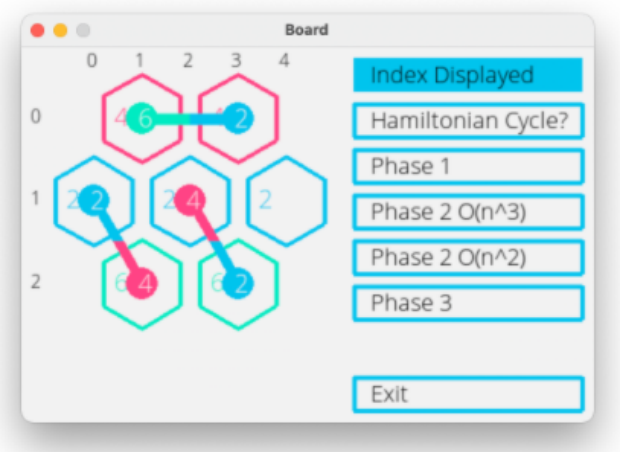


Figure 21: Index shows on cells and gourds

For a coloured board, the Index would be hidden automatically. However, it can be displayed by clicking the first button (Figure 21).

4.5.3.2. Hamiltonian Cycle

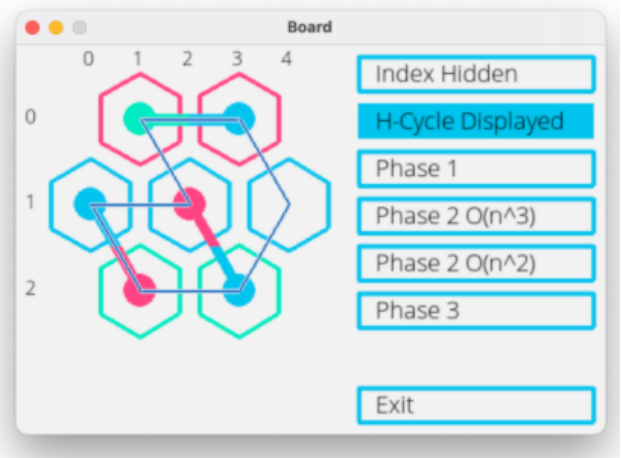


Figure 22: Hamiltonian Cycle

The Hamiltonian Cycle can be generated and displayed on the board by clicking the second button. While running Phase 1 and Phase 2, Hamiltonian Cycle will appear automatically.

4.5.4. Playable

4.5.4.1. Gourd Movements

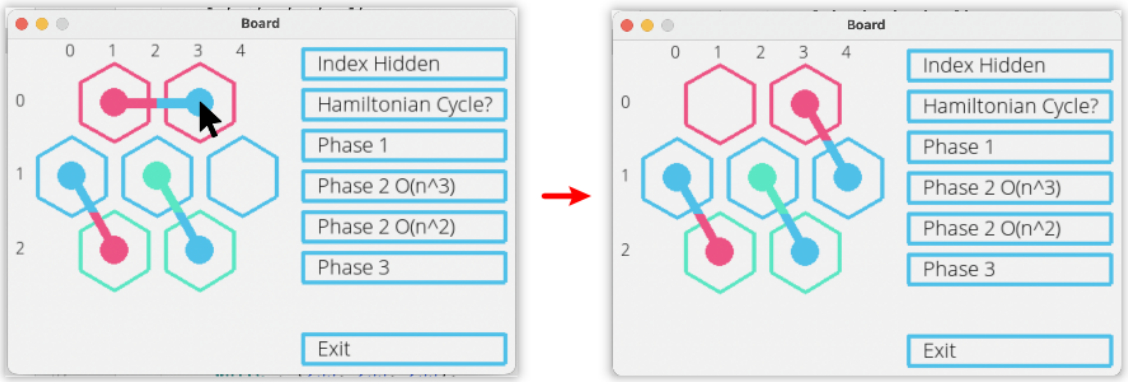


Figure 23: Click to move a gourd (turn)

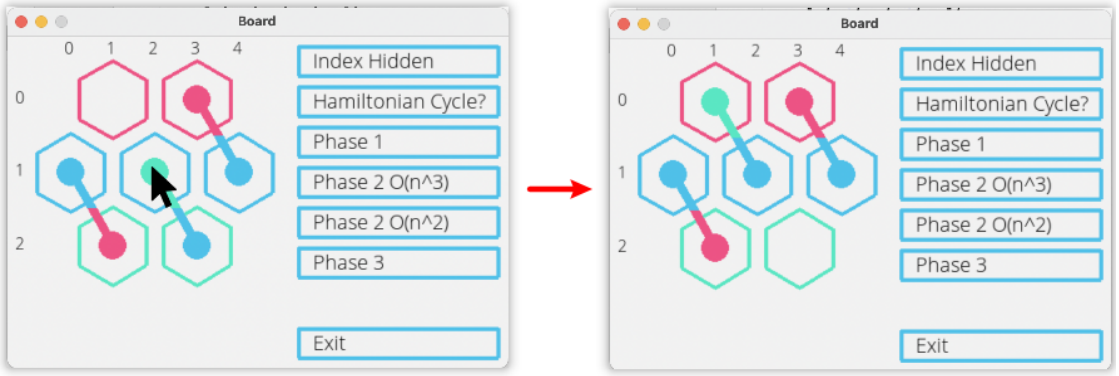


Figure 24: Click to move a gourd (slide)

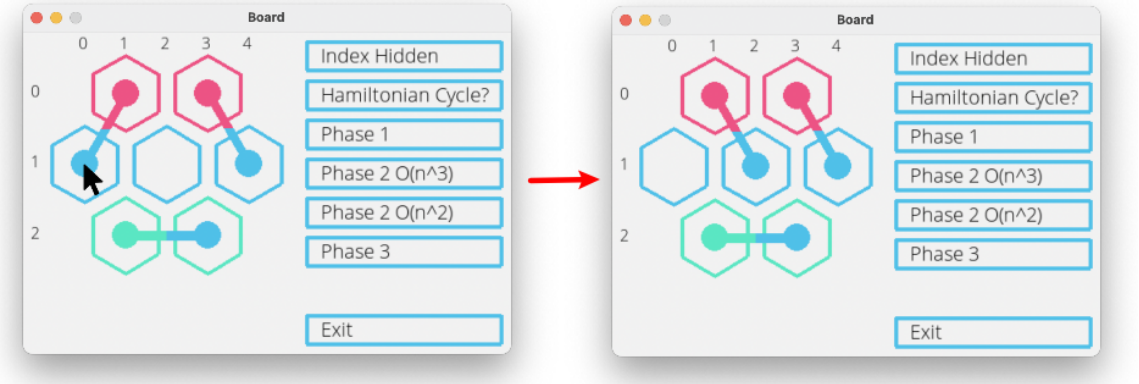


Figure 25: Click to move a gourd (pivot)

By clicking a part of a gourd next to an empty cell, this program would move the clicked part to the empty cell, and the other part of would move automatically if needed.

Figure 23 shows a 120 degrees turn, Figure 24 shows a slide, and Figure 25 shows a 60 degrees pivot.

4.5.4.2. Three Auto-solve Phases

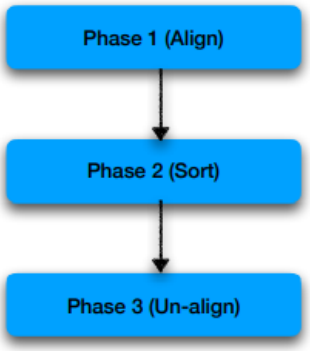


Figure 26: The three phases in reconfigurations

The three phases in reconfigurations (auto-solving) the puzzle was described in the previous sections (3.2.4).

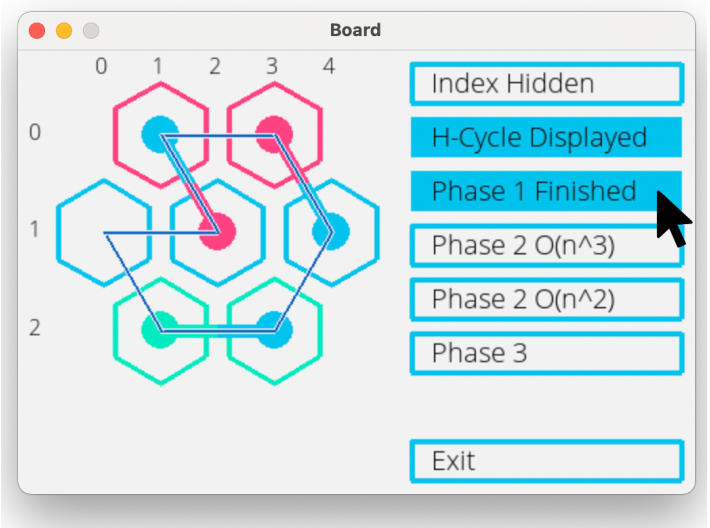


Figure 27: Phase One

After pushing the “Phase 1” button, the Hamiltonian Cycle would be displayed on the board automatically to clearly display how all the grounds are automatically aligned with the Hamiltonian Cycle. (Figure 27)

However, before “Phase 1” is finished, the “Phase 2” and “Phase 3” buttons can not be pushed.

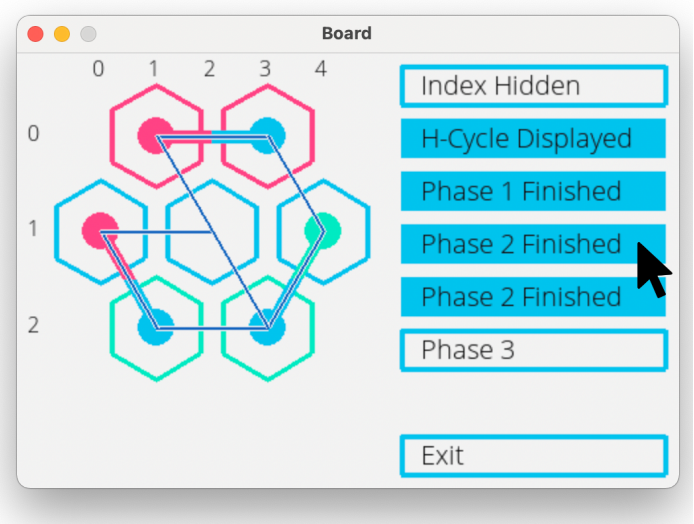


Figure 28: Phase Two

Two “Phase 2” buttons can be pushed when “Phase 1” is finished.

The “Phase 2 O(n^3)” button would run the phase 2 in a worse-case O(n^3) complexity in sorting the gourds. And the “Phase 2 O(n^2)” button runs the worse-case O(n^2) sorting algorithm. (Figure 28)

Either one of the two algorithms mentioned above was finished, the two “Phase 2” buttons would show “Finished” to avoid “Phase 2” buttons being pushed again. (Figure 28)

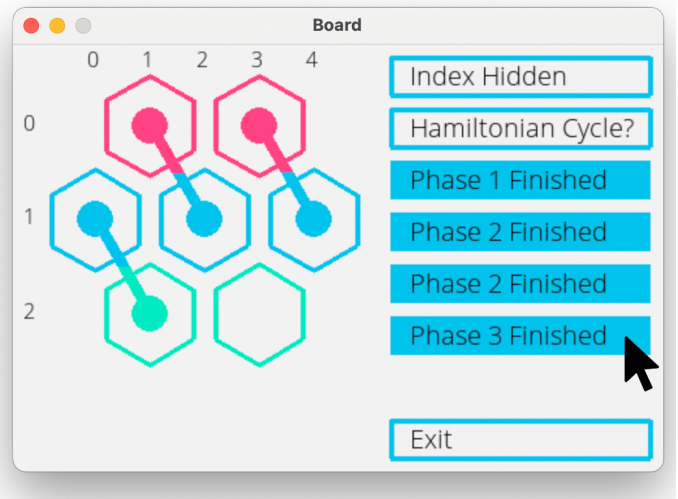


Figure 29 : Phase Three

The “Phase 3” button could be pushed after the “Phase 1” and “Phase 2” finished. Then, the Hamiltonian Cycle will be hidden again to show the reconfiguration results clearly.

Whenever the gourds was moved manually (not in the auto-solve algorithm), four buttons for “Phase 1”, “Phase 2”, and “Phase 3” would pop up automatically.
To reconfigure the puzzle again, buttons should be pushed following the “Phase 1”, “Phase 2”, and “Phase 3” sequences again.

4.5.4.3. Difference between Implementation and original paper

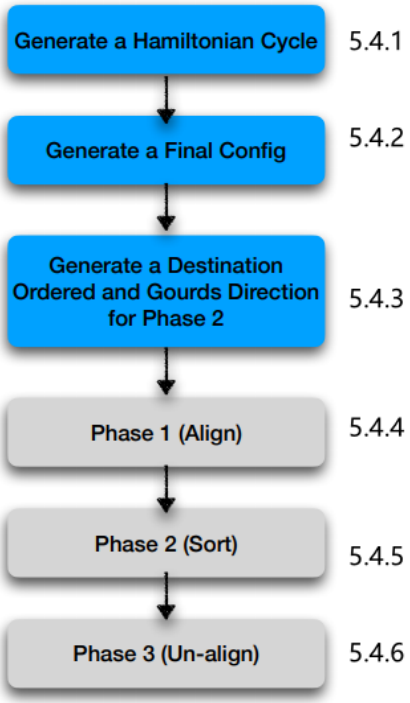


Figure 30: Three steps before the Three Phases

Before the Three Phases (Aligning → Sorting → Un-aligning), there are three necessary data should be generated which are Hamiltonian Cycle, the Final Configuration, and the Gourds order and directions of gourds in Phase 2 Sorting and un-aligning steps in Phase 3 (Figure 30, the three blue boxes).

- **Generate a Hamiltonian Cycle**: A Hamiltonian Cycle could be generated manually by clicking the second button.

Figure 31: Three acceptable Final Configs

- **Generate a Final Configuration**: In a coloured board, it might have several acceptable Final Configurations (Figure 31).

  However, before Phase 2 Sorting, there must be specific a Final Config should be generated.
- **Generate data for Phase 2 and Phase 3:** in the <Gourds> paper, it proofs that Phase 3's complexity mathematically, however, it is hard to program Phase 3. Thus, in implementation, there are some adjustments from the in original <Gourds> paper.

  The directions of gourds are ensured in the Phase 2 Sorting, and Phase 3 Un-aligning is only a reverse of Phase 1 Aligning.

  In this step, it would generate the Destination Order and Directions of Gourds for Phase 2 Sorting, and the reverse aligning steps for Phase 3 Un-aligning.

  It won't change the whole complexity of the algorithm.

  Here is a table (Table 1) shows the differences between implementation and original <Gourds> paper.

Table 1: Auto-solving process comparison between implementation and original paper

Steps in Implementation	**Description**	**Steps in <Gourds> paper**	**Description**
Generating a Hamiltonian Cycle	Hamiltonian Cycle is used in Phase 1 Aligning and Phase 2 Sorting	-	
Generating a Final Configuration	Final Configuration / Placement	-	
Generating data for Phase 2 and Phase 3	Destination gourds order and directions for Phase 2 Sorting and steps for Phase 3 Un-aligning	-	
Phase 1 Aligning	Aligning all gourds to a Hamiltonian Cycle	Phase 1 Aligning	Aligning all gourds to a Hamiltonian Cycle
Phase 2 Sorting	Sorting the Gourds to the destination order along the Hamiltonian Cycle. In the meantime, change the gourds' direction if it is opposite.	Phase 2 Sorting	Sorting the Gourds to the destination order along the Hamiltonian Cycle.
Phase 3 Un-aligning	Un-aligning gourds to reach the Final Config.	Phase 3 Un-aligning	Un-aligning gourds to reach the Final Config. In the meantime, change the gourds' direction if it is opposite.

5. Implementation

5.1. Launcher

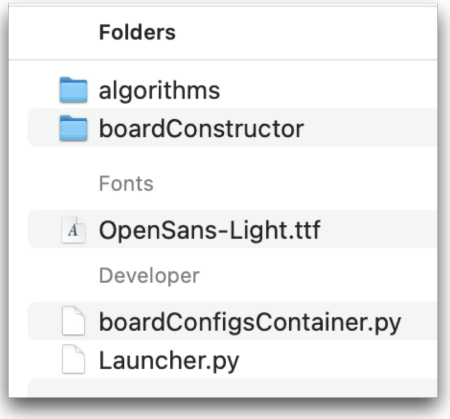


Figure 32: Root folder

The OpenSans-Light.ttf is an open source font file which is using in this program.

Launcher.py

```
1  from boardConstructor.BoardsSwitcher import boardsSwitcher
2
3  def main():
4      # run a board switcher
5      global myBoardSwitcher
6      myBoardSwitcher = boardsSwitcher()
7      myBoardSwitcher.main()
8
9  if __name__ == '__main__':
10     main()
11
```

Figure 33: Launcher.py file

The Launcher.py file is quite simple, and it just used to launch the boards switcher.

By running this Launcher.py file, the game can be easily launch.

5.2. Boards Configs Container

boardConfigsContainer.py

```
1   import numpy
2   class boardsContainer(object):
3
4       container = []
5
6       def __init__(self):...
18
19  class boardConfig1(object):...
47
48  class boardConfig2(object):...
78
79  class boardConfig3(object):...
109
110 class boardConfig4(object):...
150
151 class boardConfig5(object):...
201
202 class boardConfig6(object):...
252
253 class boardConfig7(object):...
286
287 class boardConfig8(object):...
316
317 class boardConfig9(object):...
350
351 class boardConfig10(object):...
387
```

```
19  class boardConfig1(object):
20      board = numpy.array([
21          # x  0, 1, 2, 3, 4, 5, 6, 7, 8
22          [0, 4, 0, 4, 0],  # 0
23          [2, 0, 2, 0, 2],  # 1
24          [0, 6, 0, 6, 0],  # 2
25      ])
26
27      gourdsList = numpy.array([
28          # x, y, x, y, i1, i2
29          [1, 0, 3, 0, 6, 2],
30          [0, 1, 1, 2, 2, 4],
31          [2, 1, 3, 2, 4, 2],
32      ])
33
34      coloursLibrary = {
35          'backGround': (242, 242, 242),
36          'black': (0, 0, 0),
37          'white': (255, 255, 255),
38          'hamiltonianCycle': (39, 98, 184),
39          'button':(80, 193, 233) ,
40          1: (190, 127, 73),
41          2: (80, 193, 233),
42          3: (122, 87, 209),
43          4: (237, 84, 133),
44          5: (255, 232, 105),
45          6: (91, 231, 196),
46      }
```

Figure 34: BoardConfigsContainer.py (left) and a boardConfig. class (right)

The boardConfigsContainer file stores several boards' configurations classes (Figure 34 left).

In each boardConfig class (Figure 34 right), there has a 2D “board” array which storing the board, a “gourdsList” which storing the initial gourd positions, and a “coloursLibrary” which determine the UI, cells, gourds, buttons colours for a board.

5.3. Board Constructor

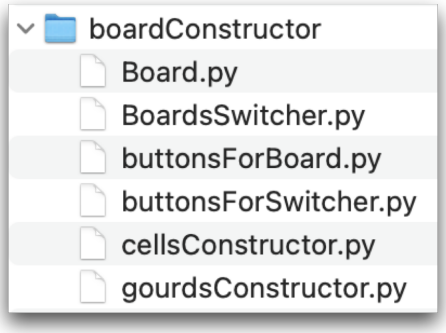


Figure 35: Board Constructor Directory

The Board Constructor directory containing the files that construct Board, Boards Switcher, buttons, hex-cells, and gourds.

5.3.1. Board and Boards Switcher

BoardsSwitcher.py

```
1   import ...
7
8   class boardsSwitcher(object):
9       def __init__(self):...
11
12      # record the clicked positions
13      # and call related functions
14      def mouseClicked(self, pos):...
23
24      # refresh the whole screen
25      def redrawTheScreen(self):...
33
34      # switch to another board
35      def boardSwitcher(self):...
47
48      # load a new board
49      def boardLoader(self):...
56
57      # main loop of this switcher
58      def mainLoop(self):...
```

Board.py

```
1   import ...
18
19  class board(object):
20      def __init__(self):...
22
23      # the main loop of the board
24      def mainLoop(self):...
46
47      # record the clicked positions
48      # and call related fuctions
49      def mouseClicked(self, pos):...
67
68      # refresh the whole screen
69      def redrawTheScreen(self):...
84
85      # set the window parameters
86      # and initialization
87      def initializer(self, indexOfBoard):...
170
171
```

Figure 36: BoardsSwitcher.py (left) and Board.py (right)

BoardsSwitcher.py and Board.py have similar structure. They both have a Main Loop, a Mouse Monitor, and a Screen Refresher.

The different is, the BoardsSwitcher.py has a Board Loader which can load a board,

but the Board.py has an initializer which can set several parameters such as the size of the window, the size of buttons, the size of board and gourds, the caption of window, the width of a hex-cell, etc.

5.3.2. Buttons Constructor

buttonsForBoard.py

```
import pygame

class buttons(object):

    def __init__(self, screen, coloursLibrary, sizeOfTheWindow, buttonSize):...

    # paint each button
    def buttonPainter(self, btnIndex, stringDefault, stringFinished, stringWorking):...

    # arrange buttons locations and labels
    def buttonConstructor(self):...

    # identify which button was clicked can change it's parameter
    def buttonsMonitor(self, pos):...
```

Figure 37: Buttons Constructor

Figure 37 shows a ButtonsForBoard.py. Which has an initializer, a Button Monitor which identify which button was clicked and change its parameters, a Button Constructor which sets buttons locations and labels, a Button Painter which paints each button.

5.3.3. Cells Constructor

cellsConstructor.py

```
import pygame

class cellsConstructor(object):

    def __init__(self, screen, board, coloursLibrary, offset, widthOfHexCell):...

    # paint each hex-cell
    def cellPainter(self, x, y):...

    # Cell painter's coordinator.
    def cellsAndAxisConstructor(self, isDisplayIndex):...
```

Figure 38: Cells Constructor.py

Cells Constructor also has a cell painter for each hexagonal cell painting. The Cells and Axis Constructor read data from the Board Configs Container and arrange hexagonal cells, in the meantime, it generates two axes on the board. (Figure 38)

5.3.4. Gourds Constructor

gourdsConstructor.py

```
1   import pygame
2
3   class gourdsConstructor(object):
4
5       def __init__(self, screen, board, gourdsList, coloursLibrary, offset, widthOfHexCell, gourdSize):...
13
14      # painting each gourd
15      def gourdPainter(self, firstPart, secondPart):...
27
28      # read from the Board Configs Container and arrange gourds
29      def gourdsConstructor(self, isDisplayIndex):...
61
62      # gourds Searching By Coordinate
63      def gourdsSearchingByCoordinate(self, pos):...
74
75      # gourds Searching By Index
76      def gourdsSearchingByIndex(self, x, y):...
84
85      # empty Cell Searching Around A Gourd
86      def emptyCellSearchingAroundAGourd(self, x, y):...
111
112     # gourds Movement Animation
113     def gourdsMovementAnimation(self, before, after, indexOfGourd):...
167
168     # gourds Movement Controller
169     def gourdsMovementController(self, indexOfGourd, xGourdClicked, yGourdClicked, xCell, yCell):...
228
229     # Gourds Clicked
230     def clicksMonitor(self, pos, mode):...
250
```

Figure 39: Gourds Constructor

Table 2: Descriptions for all methods in GourdsConstructor.py

Method Name	**Description**
Gourd Painter	Painting each gourd
Gourds Constructor	Read from the Board Configs Container and arrange gourds

Gourds Searching By Coordinate	Search if there is a gourd in a given coordinate on the window
Gourds Searching By Index	Search if there is a gourd in a given coordinate defined by the matrix
Gourds Movement Animation	This method will draw per frame and refresh to make gourds likes moving smoothly
Gourds Movement Controller	Check if it is a legal movement of a gourd, move the gourd in a proper method (slide, turn, or pivot) automatically, and controller animations.
Clicks Monitor	Receive coordinates and check what gourd was clicked, and identify the click is from the user or algorithms.

Figure 39 is gourds constructor and the Table 2 is Descriptions of all methods in GourdsConstructor.py.

5.4. Auto-solve Algorithms

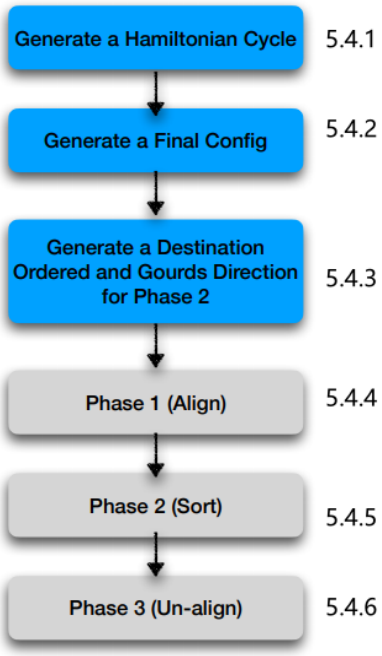


Figure 40: Three steps should be finished before Auto-solving(blue boxes)

As shown in Figure 40, there are three steps should be finished before Auto-Solving, which are Generating a Hamiltonian Cycle, Generating a Final Config, and Generating a Destination Order and Gourds Direction for phase 2.

5.4.1. Hamiltonian Cycle Generator

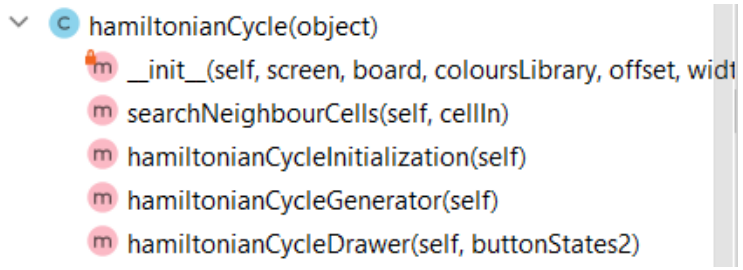


Figure 41: Methods in Hamiltonian Cycle.py

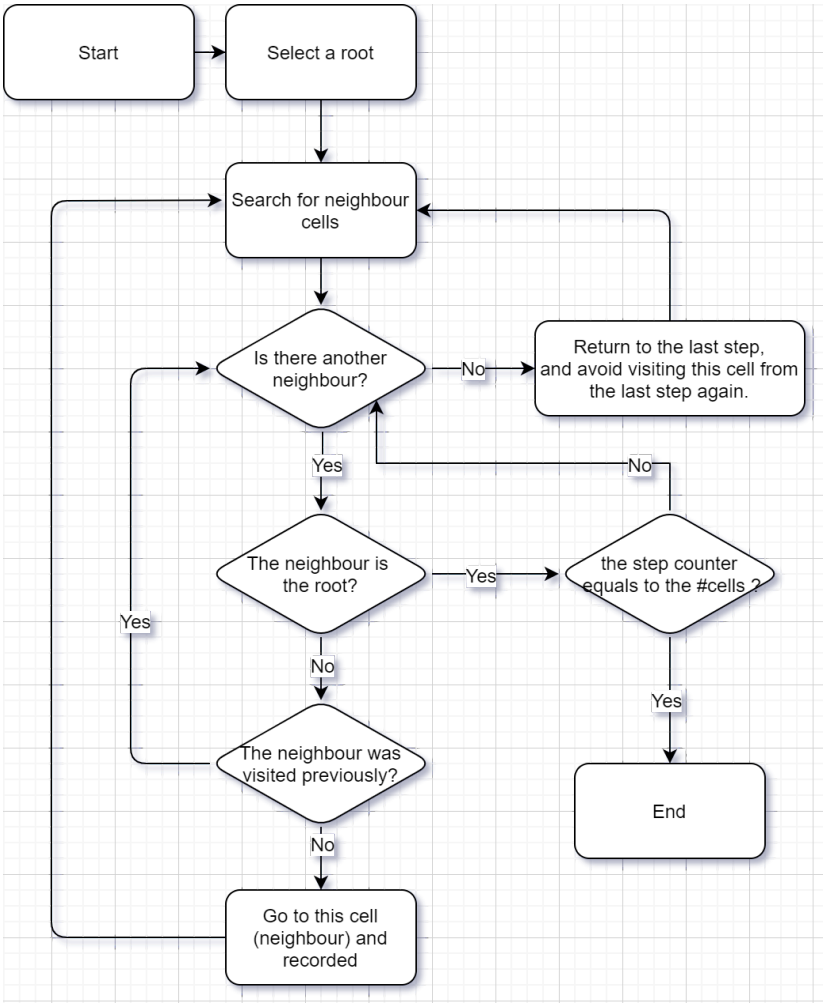


Figure 42: The Flowchart for Hamiltonian Cycle Generator

Figure 42 shows a Hamiltonian Cycle Generator with DFS (Deep First Searching) design.

```
# DFS
while (not completeFlag):

    # search for next step
    neighbourList = self.searchNeighbourCells(thisCell)
    availableNextCellList = []

    # filter, choose one available next step
    for neighbour in neighbourList:...

    # go to next step
    if not completeFlag:...
```

Figure 43: The DFS realization in the hamiltonianCycleGenerator method

Figure 41 shows the methods in HamiltonianCycle.py and Figure 43 shows the DFS realization in the codes (for conciseness, the details are collapsed as '...' and the methods details are not shown in the Figure 43).

5.4.2. Final Configuration Generator

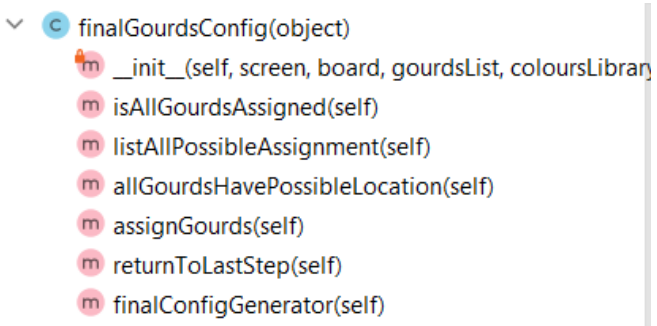


Figure 44: Methods in FinalGourdsConfigGenerator.py

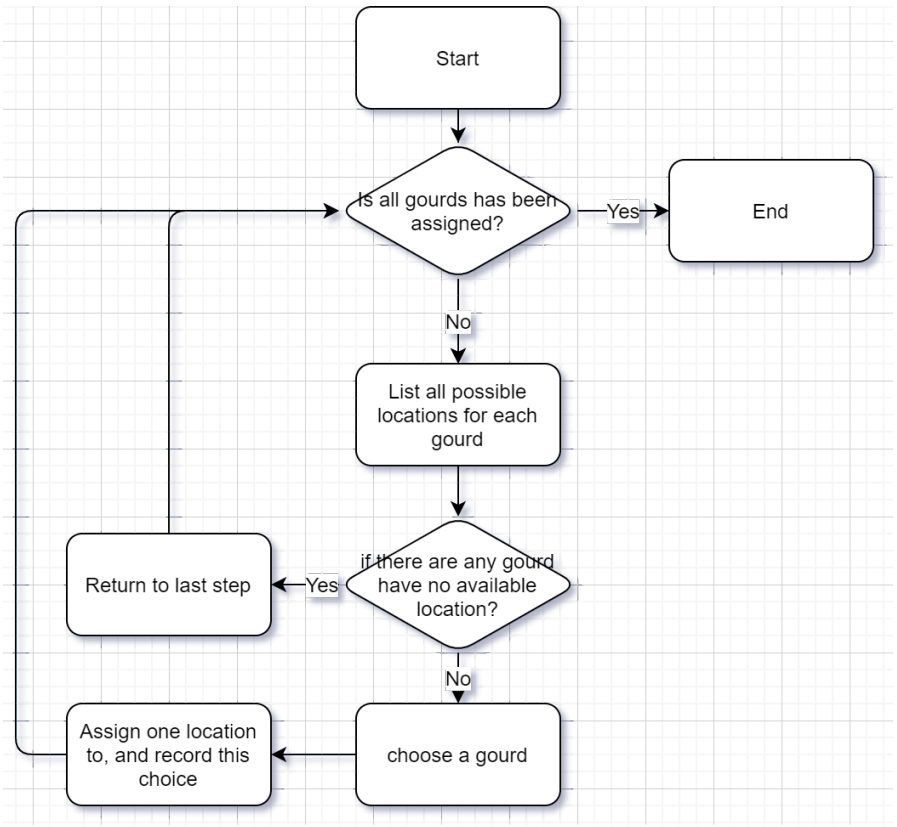


Figure 45: The Flowchart for Final Configuration Generator

The Final Configuration Generator is also a DFS-like algorithm (Figure 45).

```
# DFS-like
while(not self.finishedFlag):

    # all gourds have been assigned?
    if self.isAllGourdsAssigned():
        self.finishedFlag = True
        print("The final configuration: ", self.gourdsAssignedDict)
        return True

    # list all possible location
    self.listAllPossibleAssignment()

    # If there are gourds have no possible location?
    if self.allGourdsHavePossibleLocation():
        # go to the next step
        self.assignGourds()


    else:
        # return to the last step
        self.returnToLastStep()
```

Figure 46: The DFS realization in the hamiltonianCycleGenerator method

Figure 44 shows the methods in *hamiltonianCycleGenerator*.py and Figure 46 shows the DFS-like algorithm realization in the codes.

5.4.3. Relevant Data (for Phase 2 and 3) Generator

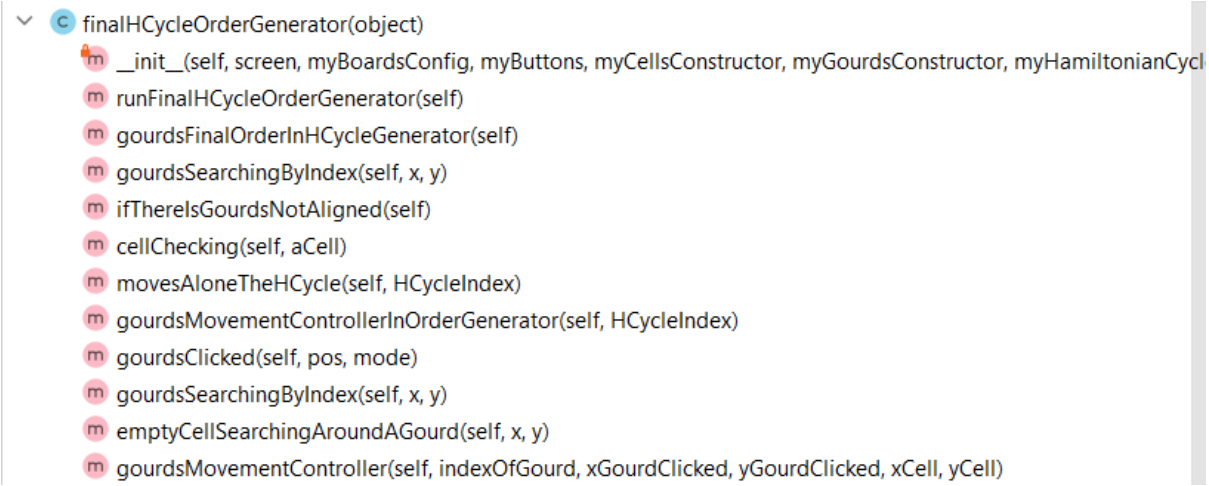


Figure 47: Methods in FinalHCycleOrderGenerator.py

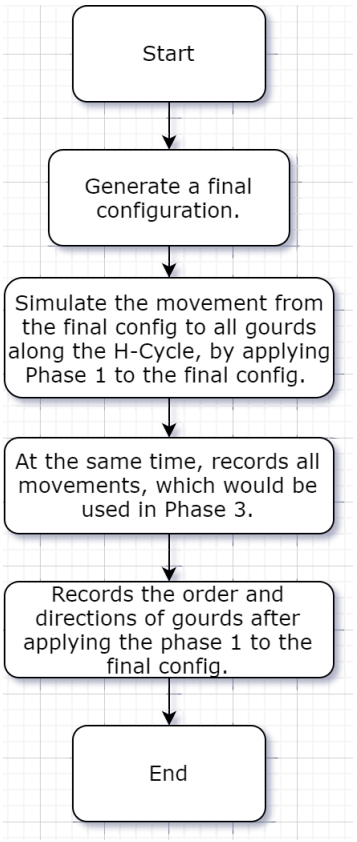


Figure 48: The Flow chart for FinalHCycleOrderGenerator.py

In this class, destination gourds order and directions for Phase 2 Sorting and steps for Phase 3 Un-aligning could be generated.

5.4.4. Phase One (Aligning)

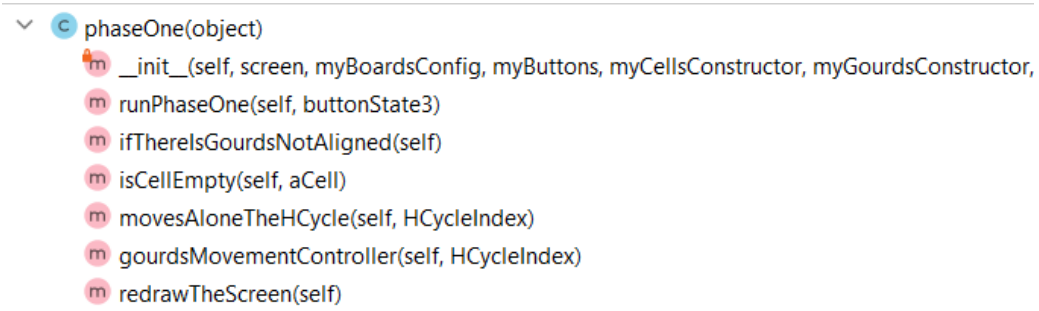


Figure 49: Methods in PhaseOne.py

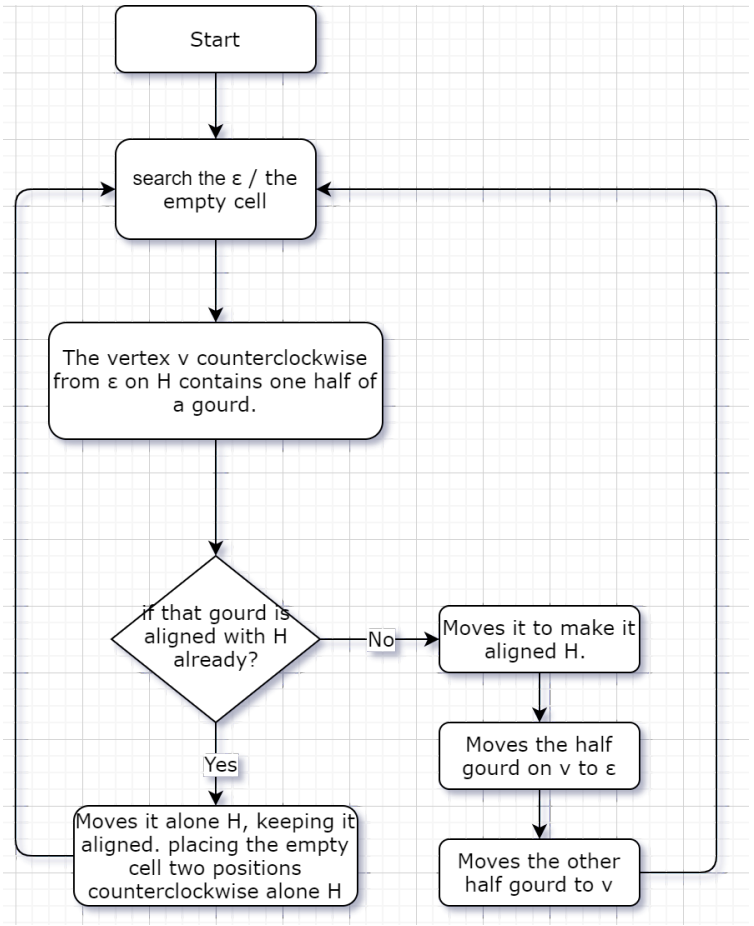


Figure 50: The Flowchart for PhaseOne.py

```
# align the gourds
while(self.ifThereIsGourdsNotAligned()):

    # Find a empty cell
    rootIndex = -1
    for aCell in self.myHamiltonianCycle.hamiltonianCycleStack:...

    # moves the gourd next to the empty cell
    self.gourdsMovementController(rootIndex)
```

Figure 51: The realization of the Figure 50 flowchart above

```
def gourdsMovementController(self, HCycleIndex):
    # check if the gourd next to the empty is align with H-Cycle
    isEmpty, indexOfGourds = self.isCellEmpty(self.HCycleAux[HCycleIndex])
    # not aline with H-Cycle
    if not isEmpty:...

    # is align with H-Cycle
    self.movesAloneTheHCycle(HCycleIndex)

    self.redrawTheScreen()
    return True
```

Figure 52: The GourdsMovementController method

Figure 51 is the realization for the Figure 50 flowchart. To have a better understanding for how to “move a gourd next to the empty cell”, the Figure 52 shows some details in GourdsMovementController method.

5.4.5. Phase Two (Sorting)

5.4.5.1. An O(n^3) Sorting Algorithm

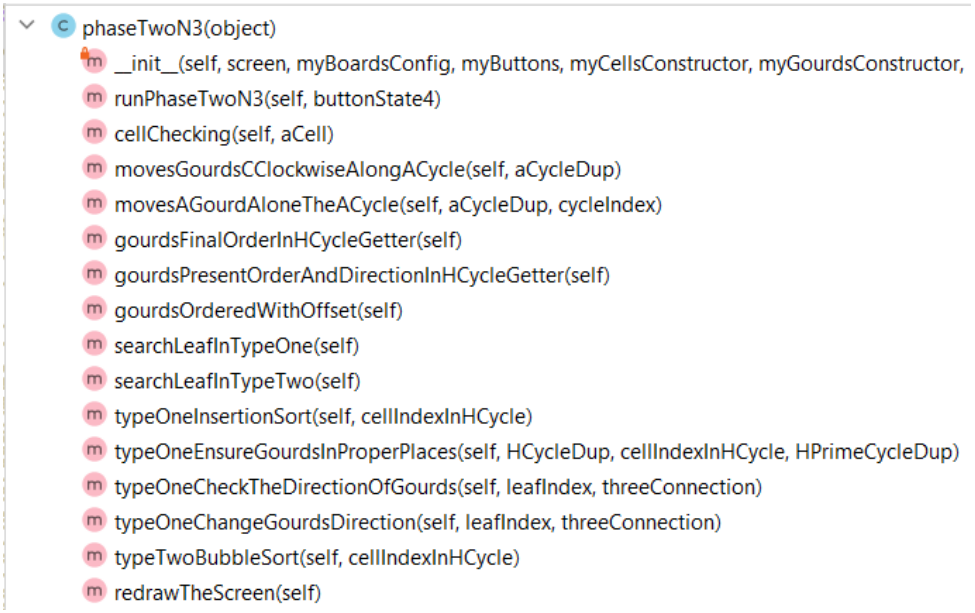
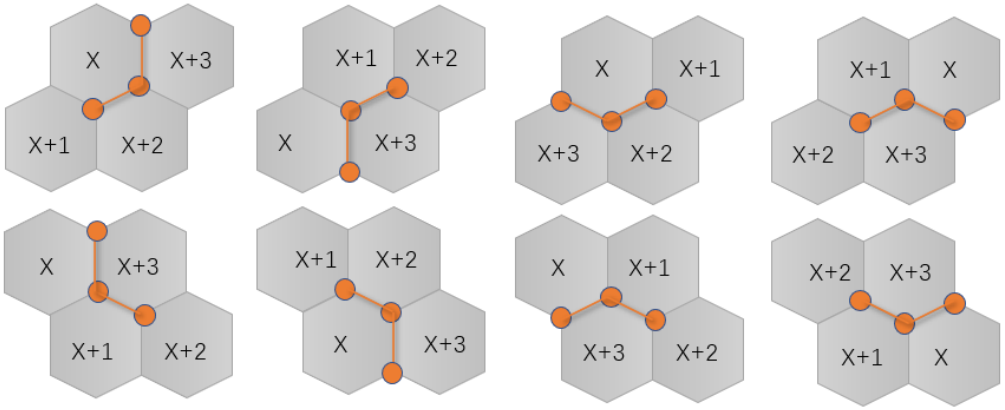
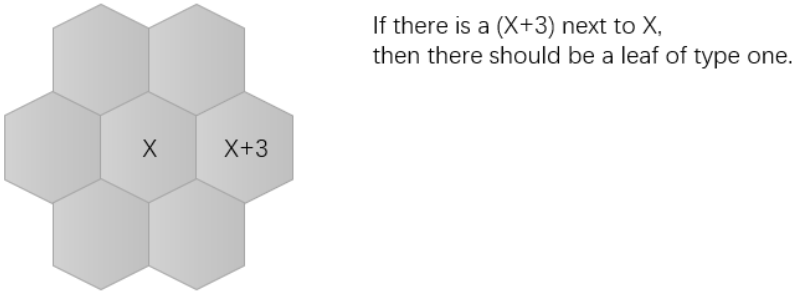


Figure 53: The Methods in PhaseTwoN3.py

5.4.5.1.1. Leaves Searching



(a) *Eight types of leaves in type one (Insertion Sort)*



(b) If there is an (X + 3) next to X, then there should be a leaf in type one

Figure 54: The Leaves in type one

Figure 54 (a) shows all types of type one leaves (Insertion Sort). By observing these eight leaves, we can conclude that if there is an X + 3 (Order in a Hamiltonian Cycle) next to X, there should be a leaf in type one. (Figure 54 (b)) This characteristic can be used in leaves searching.

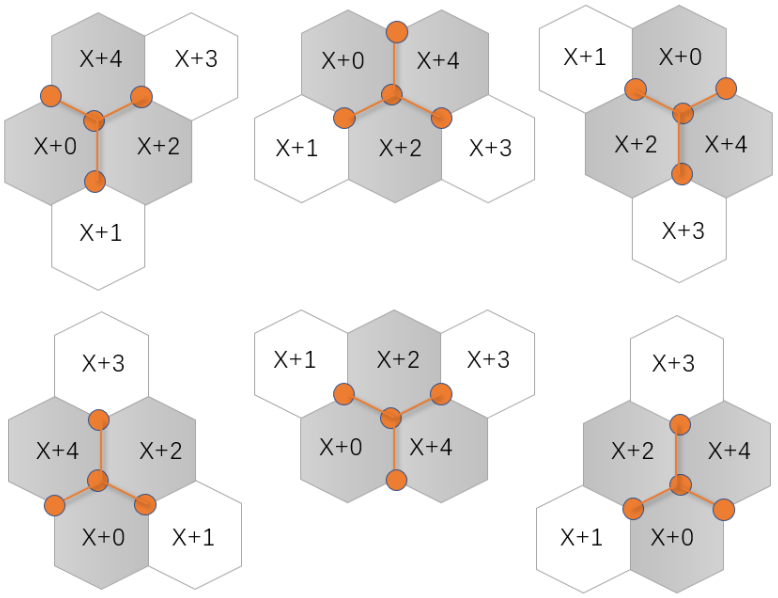


Figure 55: The Six types of leaves in type two (Bubble Sort)

Figure 55 shows all types of type two leaves (Bubble Sort). By observing these six leaves, we can conclude that if there is an X + 0 (Order in a Hamiltonian Cycle), an X + 2, and an X + 4 (Figure 55, grey cells) are connected to each others separately, there should be a leaf in type two.
This characteristic can also be used in leaves searching.

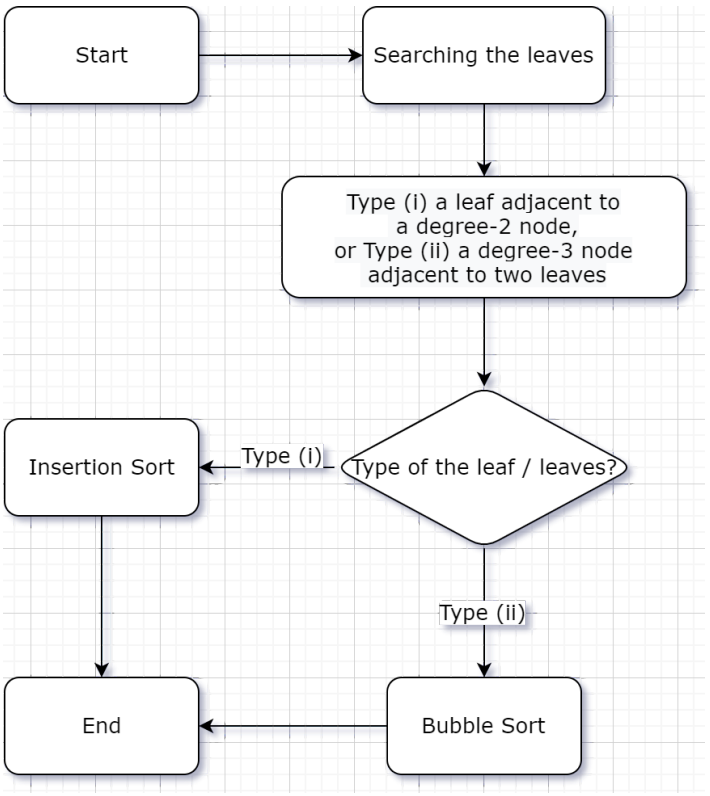


Figure 56: The Flowchart for leaves searching

```
# search type one
self.leafIndex, threeConnection = self.searchLeafInTypeOne()
if self.leafIndex != -1:
    self.leafType = 1
    print("\t", self.HCycleAux[self.leafIndex], "is the x of leaf type one")
    resultOfSort = self.typeOneInsertionSort(self.leafIndex)
    self.typeOneCheckTheDirectionOfGourds(self.leafIndex, threeConnection)

else:
    # search type two
    self.leafIndex = self.searchLeafInTypeTwo()
    if self.leafIndex != -1:
        self.leafType = 2
        resultOfSort = self.typeTwoBubbleSort(self.leafIndex)
```

Figure 57: The Realization for leaves searching

Figure 56 is the flowchart for leaves searching and Figure 57 is it's implementation which using the Figure 54's conclusion for searching leaves in type one and using the Figure 55's conclusion for searching leaves in type two.

5.4.5.1.2. Insertion Sort

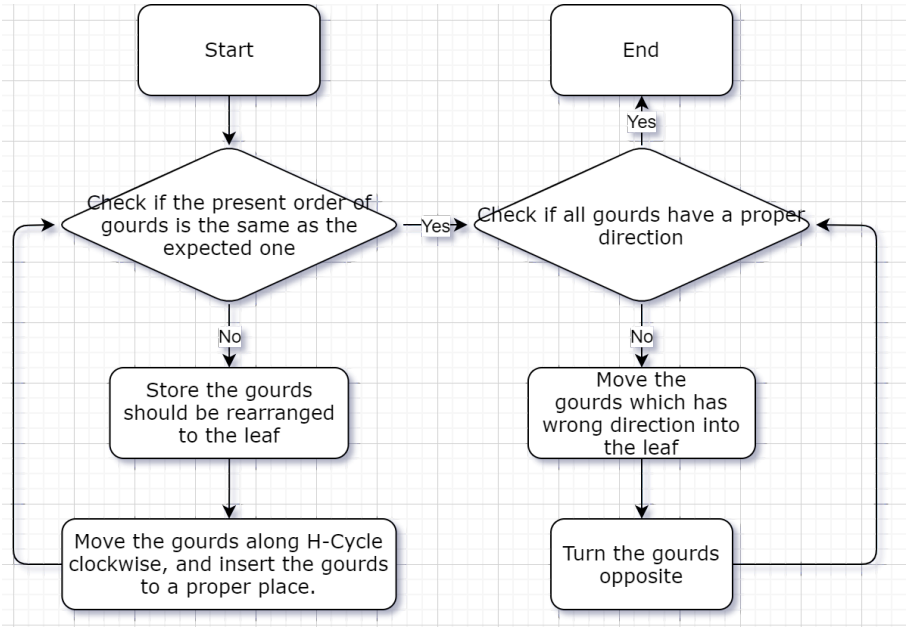


Figure 58: The Flowchart for leaves in type one (Insertion Sort)

Besides the Insertion Sort, this algorithm detects and change the gourds' direction for those in opposite (Figure 58).

```
# Insertion Sort
while(True):
    lenOfACycle = int(len(HCycleDup) / 2)

    # check if it is done (but with an offset)
    if self.gourdsOrderedWithOffset():...

    # insertion
    # Ensure Gourds In Proper Places
    self.typeOneEnsureGourdsInProperPlaces(HCycleDup, cellIndexInHCycle, HPrimeCycleDu

    # get the index of x + 1
    gourdsIndexAtXPlusOne = self.myGourdsConstructor.gourdsSearchingByIndex(HCycleDup[
    gourdsIndexShouldBeAtX = -1

    # don't align 0
    if gourdsIndexAtXPlusOne == 0:
        self.movesGourdsCClockwiseAlongACycle(HCycleDup)
        continue
    for i in range(lenOfACycle):...

    # get the index of x really be
    if self.cellChecking(HCycleDup[cellIndexInHCycle + 0])[0]:
        gourdsIndexAtX = self.myGourdsConstructor.gourdsSearchingByIndex(HCycleDup[cel
    else:
        gourdsIndexAtX = self.myGourdsConstructor.gourdsSearchingByIndex(HCycleDup[cel

    # Move the selected gourd to a proper place
    while not (gourdsIndexAtX == gourdsIndexShouldBeAtX):
        self.movesGourdsCClockwiseAlongACycle(HPrimeCycleDup)

        if self.cellChecking(HCycleDup[cellIndexInHCycle + 0])[0]:...
        else:...

    # move all gourds counter clockwise
    for i in range(lenOfACycle):
        if self.cellChecking(HCycleDup[i])[0]:
            self.movesAGourdAloneTheACycle(HCycleDup, i)
            break
    print("\tThe order of gourds now: ", self.gourdsPresentOrderAndDirectionInHCycleGe


return True
```

Figure 59: The Realization for leaves in type one

The Figure 59's codes realize the Insertion Sort for gourds on a board.

5.4.5.1.3. Bubble Sort

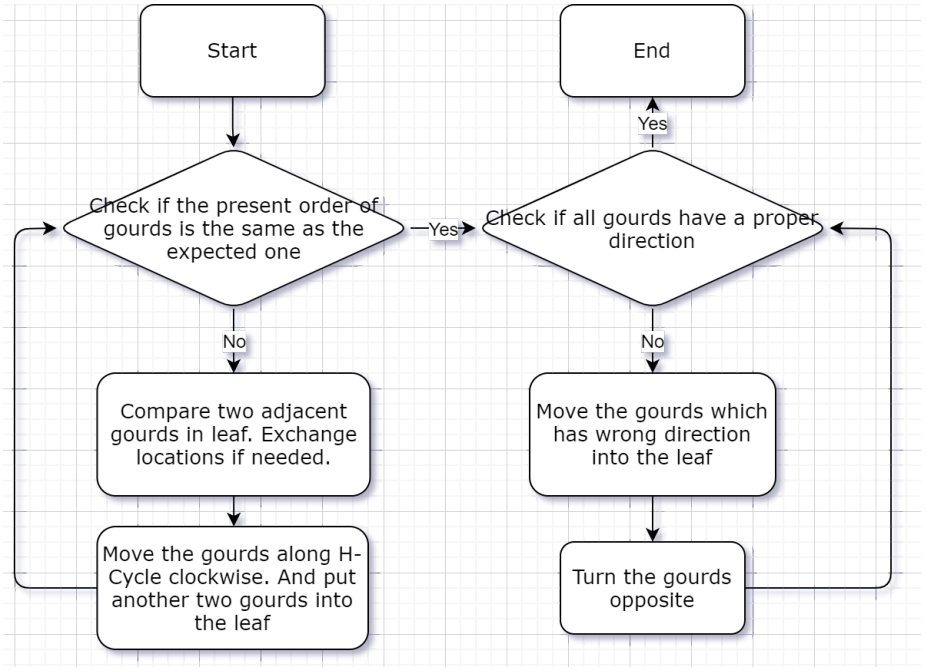


Figure 60: The Flowchart for leaves in type two (Bubble Sort)

However, the Bubble Sort algorithm shows in Figure 60 is still under-implementation.

5.4.5.2. An O(n^2) Sorting Algorithm

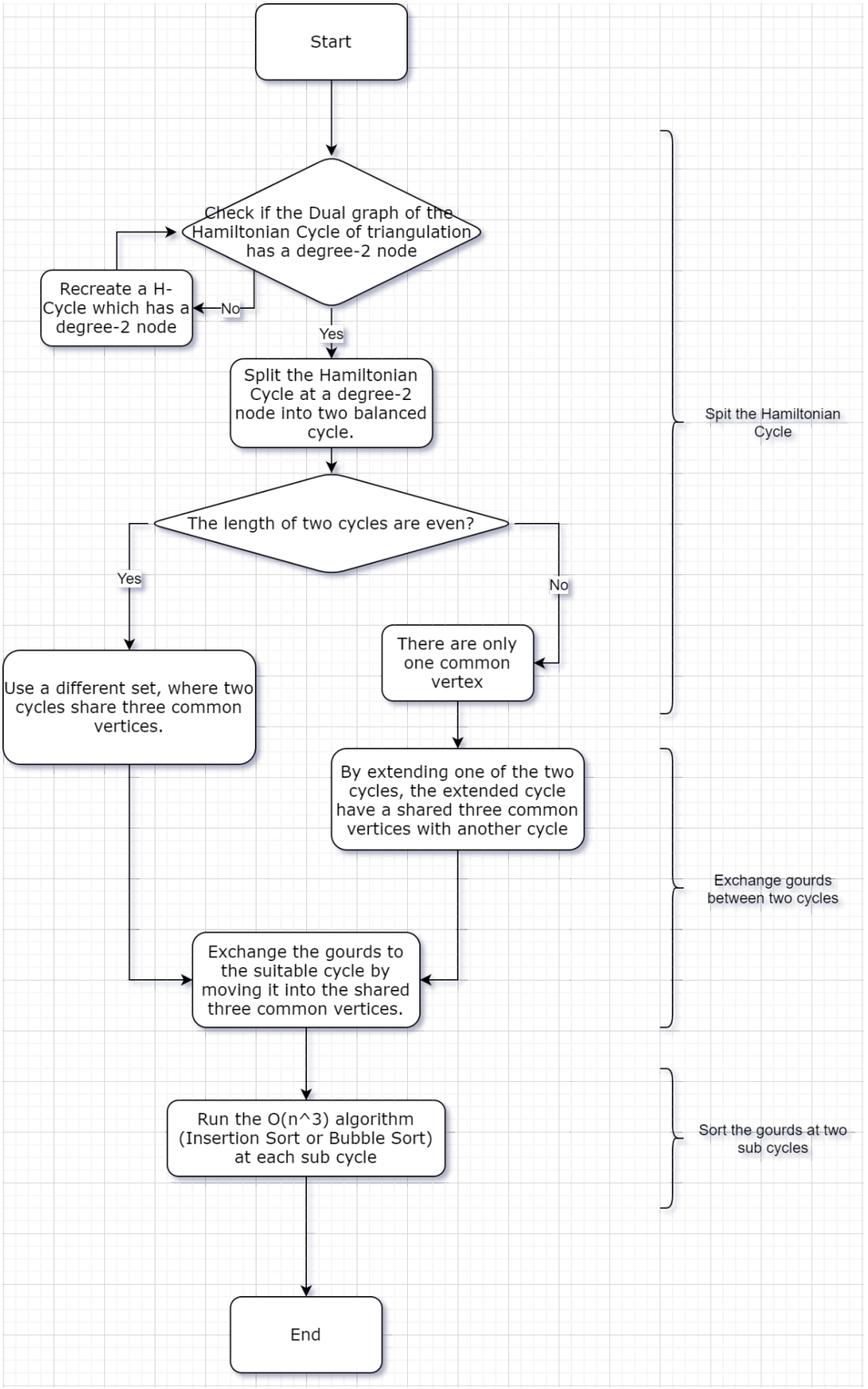


Figure 61: The Flowchart for O(n^2) Sorting Algorithm

However, the algorithm shows in Figure 61 is still under-implementation.

5.4.6. Phase Three (Un-aligning)

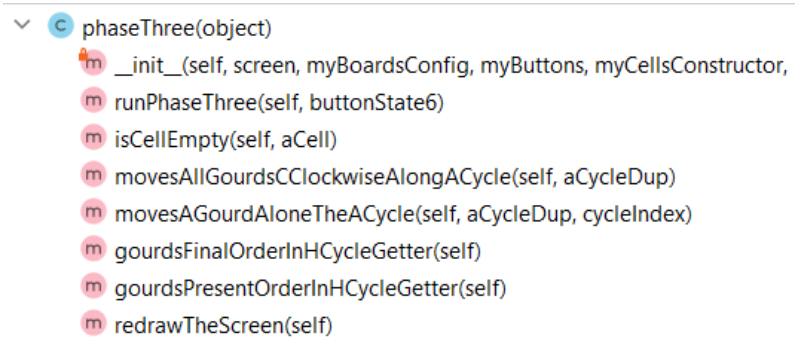


Figure 62: Methods in PhaseThree.py

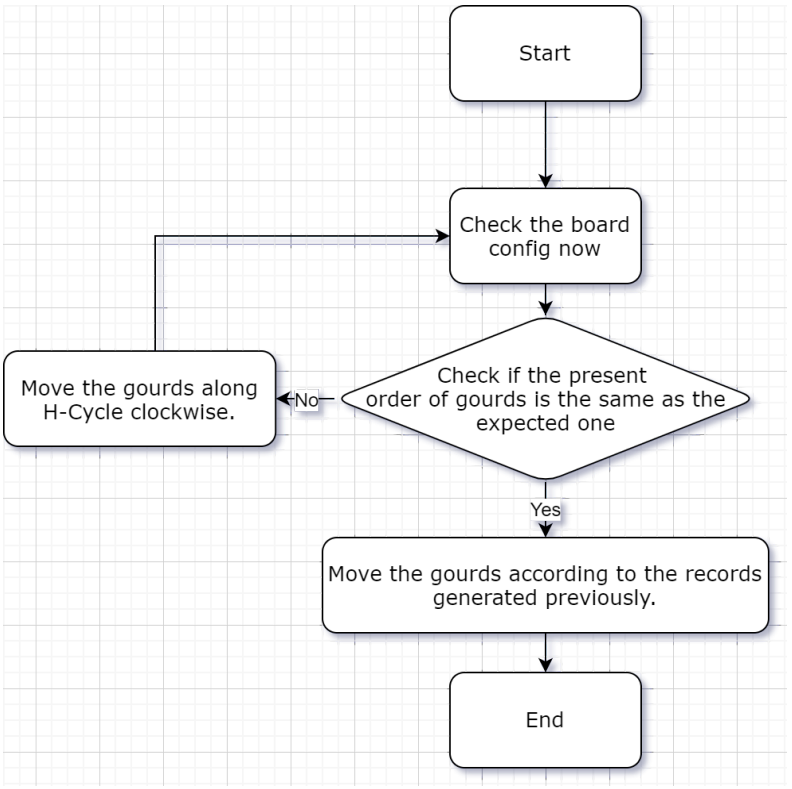


Figure 63: The Flowchart for Phase Three Un-aligning

```
# ensure the empty cell is at a proper place, if not true, then move gourds counter-clock-wise
while not (emptyCellAtAProperPlace and isTheFinalOrder):

    self.movesAllGourdsCClockwiseAlongACycle(self.HCycleDup)

    isTheFinalOrder = self.gourdsFinalOrderInHCycleGetter() == self.gourdsPresentOrderInHCycleGetter()
    emptyCellAtAProperPlace = self.isCellEmpty(self.HCycleDup[self.myFinalHCycleOrderConfig.emptyCellIndex])[0]

# un-aligning
footPrintAux = copy.deepcopy(self.myFinalHCycleOrderConfig.stackOfFootPrint)
while len(footPrintAux) > 0 :
    self.myGourdsConstructor.clickedMonitor(footPrintAux.pop(), 'al')
```

Figure 64: The Realization for Phase Three (Un-aligning)

The Phase 3 in implementation is different from the <Gourds> paper's description. In <Gourds>, the Phase 3 should check gourds direction and un-align (described in section 3.2.4.1).

However, in implementation, the direction checking was finished in the Phase 2. Thus, the Phase 3 only un-align gourds using the un-align steps generated in section 5.4.3.

6. Testing

6.1. Boards Construction

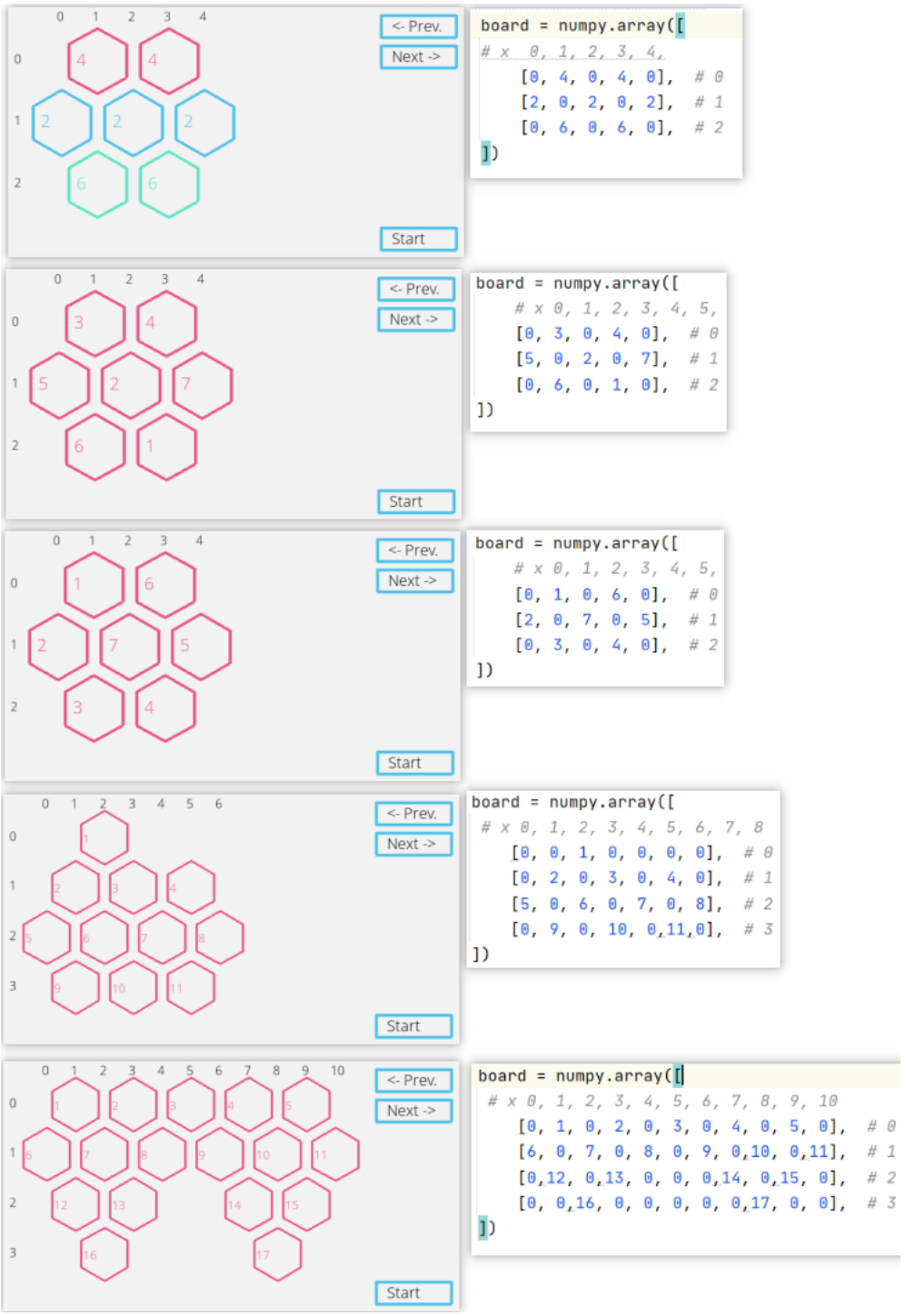


Figure 65: Test set 1

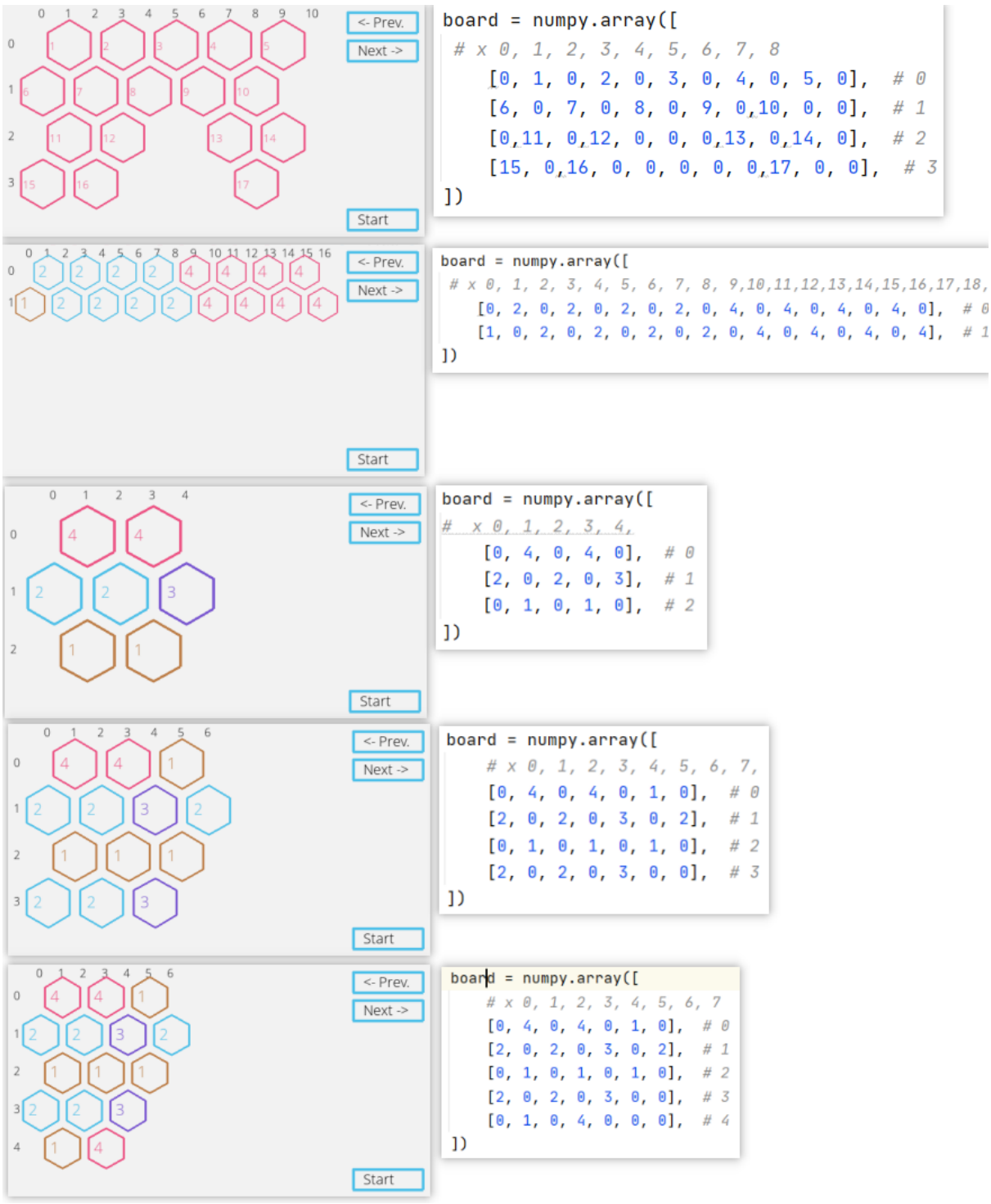


Figure 66: Test set 2

Figure 65 and Figure 66 shows ten different boards, the first seven boards were shown in the <Gourds> paper. Boards on the left-hand side are generated in Board Switcher window, the right-hand side are the boards data stored in the boardsConfigsContainer.py.

Two figures shows that the board constructor works correctly.

6.2. Auto-solving Algorithms

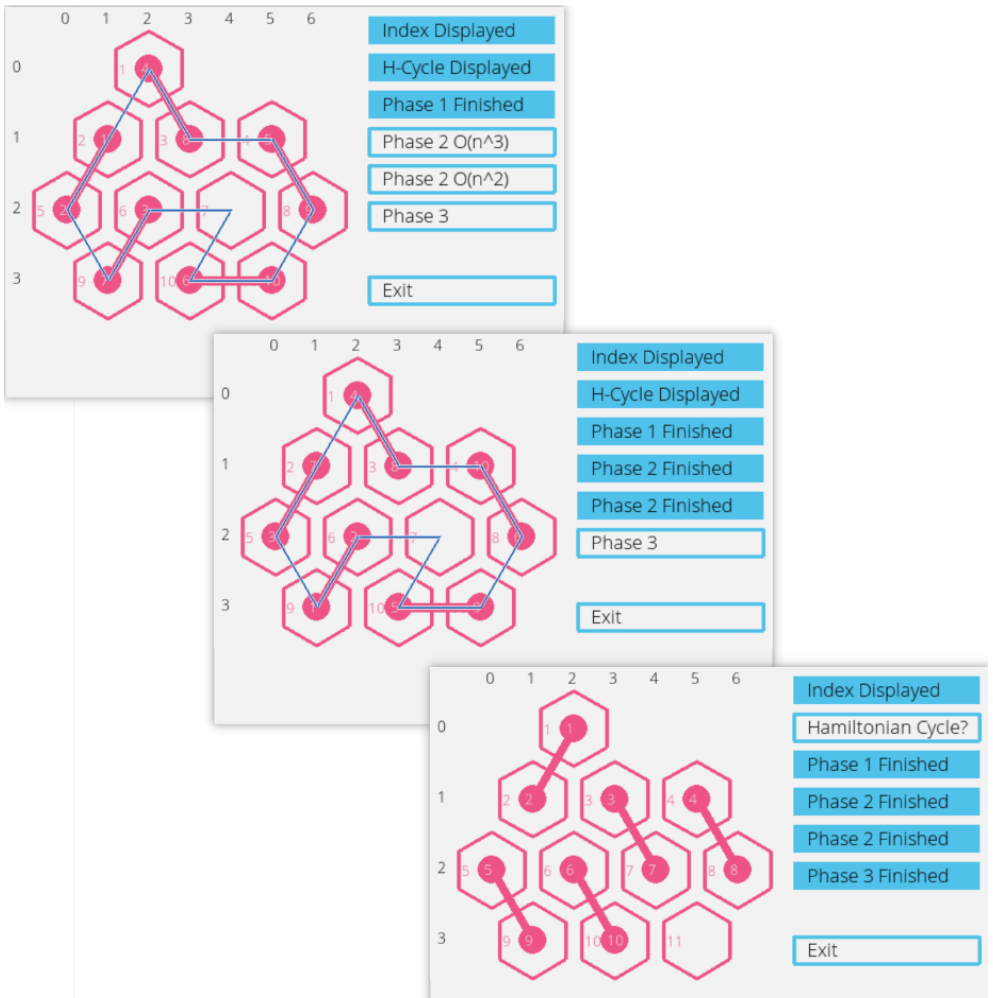


Figure 67: A numbered board runs the three phases;

top left: Phase 1 finished; middle: Phase 2 finished; down right: Phase 3 finished.

Figure 67 shows a numbered board takes reconfiguration process.

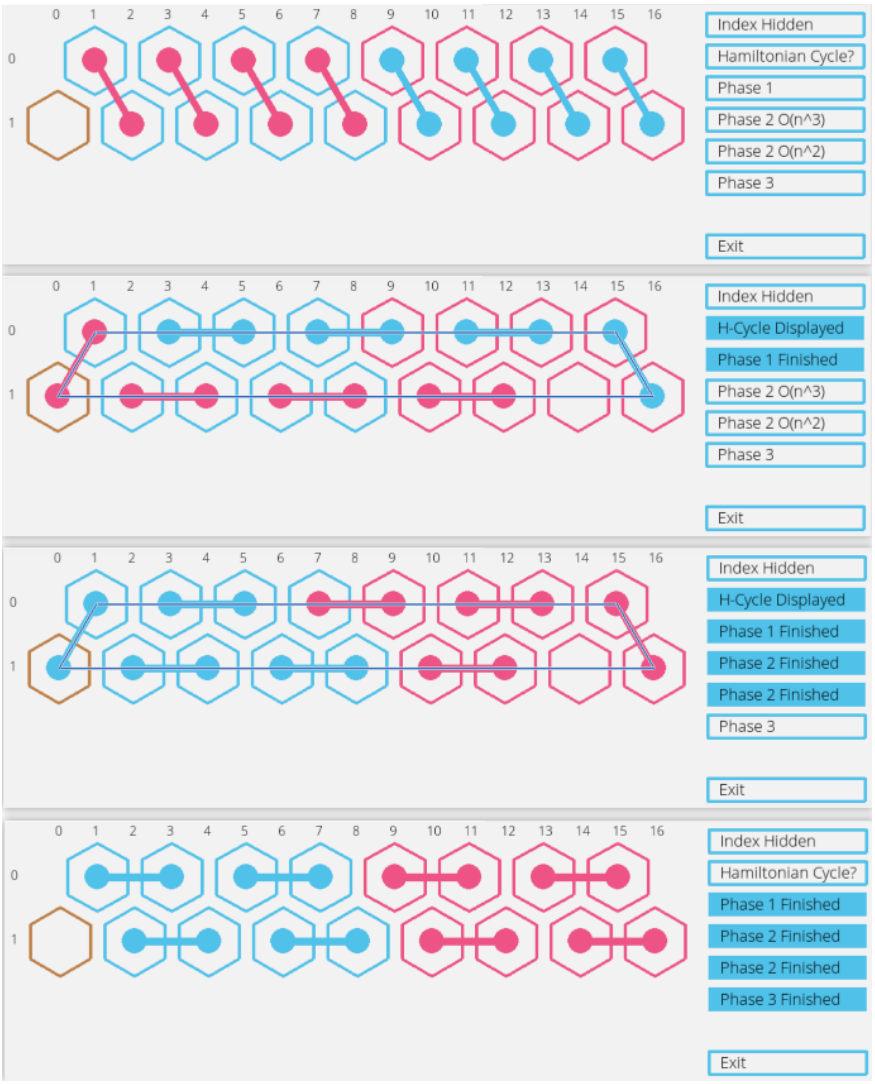


Figure 68: A coloured board runs the three phases.

Figure 68 shows a coloured board takes reconfiguration process.

In a coloured board, there might be several acceptable final configs, however this program can only produce one final config of it, but this final config might take lot more steps than others.

Also, the gourds in this program's algorithms can only go counter-clockwise, but it might be the worst case in some scenario.

For example, the third graph in Figure 68 is a board just finished Phase 2. It can go to finished Phase 3 by moving 5 gourds clockwise a step each gourd. However,

the algorithm with counter-clockwise only should take n^2 steps to moves all n gourds counter-clockwise until reach the final config.

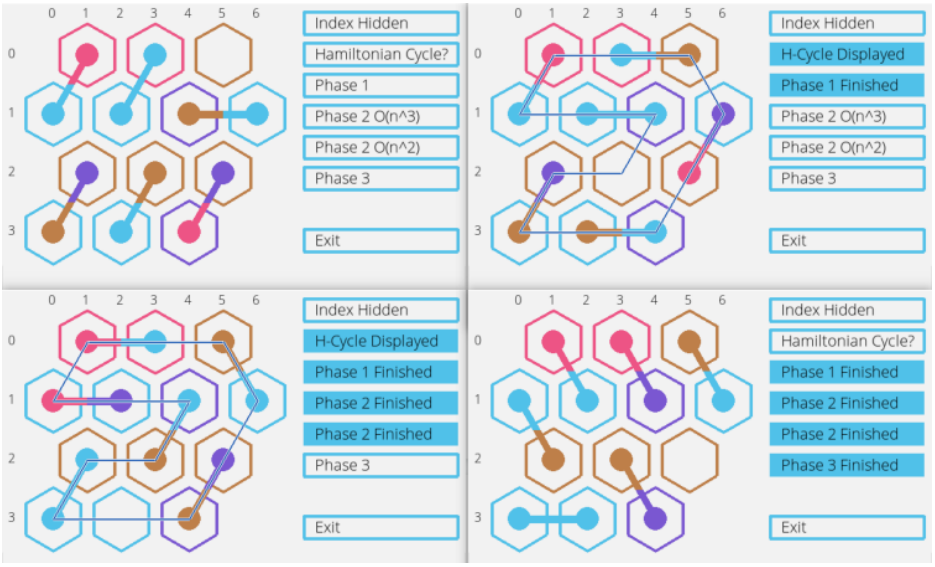


Figure 69: Another coloured board runs the three phases

Finally, It is easy to figure out that these three examples (Figrue 67, Figure 68, Figure 69) runs correctly.

7. Evaluation

7.1. Freeze while Running Phases

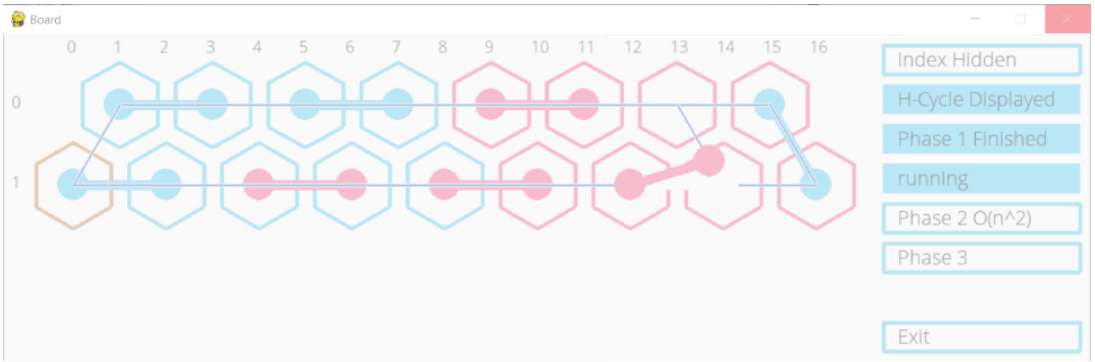


Figure 70: No Response while running Phase 2

It only happens on Windows 10 (Figure 70). When a board has too many cells and take lots of time to run phases (especially in Phase 2) on it, it won't respond to any mouse click, in Windows 10, the system would label this program "Not Responding" and looks like freezing.

However, the program is still running in the background (just looks freezing), when this phase finished, the program will back to normal.

Therefore, it is better to run it on a Mac (OS X 11).

7.2. Data Validation

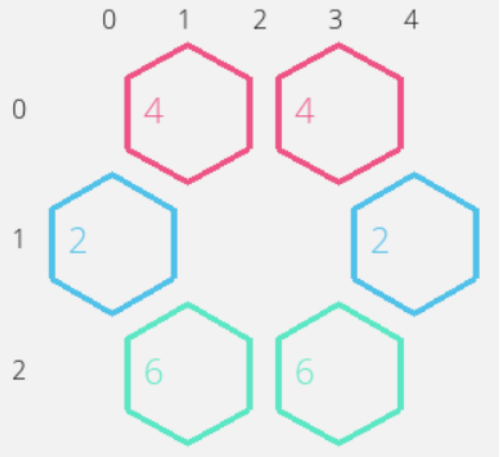


Figure 71: Not a Proper Board

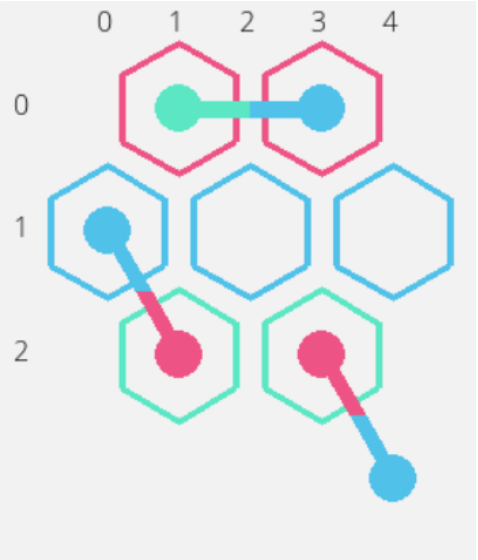


Figure 72: Invalid data in Gourds List

This program does not have data validation for the data in BoardsConfigsContainer.py.

Therefore, when a board (Figure 71) is not “Proper” (described in section 3.2.1) or gourds are not compatible to the board (Figure 72), the program would stop working.

7.3. Robustness

When the data in BoardsConfigsContainer.py are valid, this program would very stable and only take a few memories (25 MB typically) even it running for a very long time or loading a very large board.

7.4. Optimization

There are many parts could be optimized. For example, in the Implementation of Insertion Sort (Section 5.4.5.1.2.). The gourds can only a round counter-clockwise but not clockwise, which would wast a lot of time to wait for gourds to go around for a cycle. By adding a clockwise movement for all gourds in Phase 2, which could reduce the steps in this phase.

Another example is that in the Implementation of Final Configuration Generator (Section 5.4.2). The Final Configuration generated quite randomly, it cannot choose a configuration that is best for auto-solve while it might have several acceptable final configs.

The last problem is that this program cannot generate a Proper Board itself and building a board in the BoardsConfigsContainer.py manually is quite difficult.

8. Conclusion

This dissertation gives an overview of the final year project of Gourds Implementation, with detail descriptions of the background, software design, realization, testing, and evaluation.

Although the O(n^2) algorithm (section 5.4.5.2) and Bubble Sort (section 5.4.5.1.3) in Phase 2 are still under implementing (the flowcharts of the algorithms' implementation are given), the aims are basically achieved.

There still some aspects could be improved for this project. For example, in the design phases, this project has not considered developed as a mobile app or a web (HTML5) application which might be better for spreading and more easy for instalment.

However, this project is generally successful.

9. BCS Criteria & Self-Reflection

In this project, the BCS guidelines and in particular below shows how six outcomes achieved.

1. This project includes a software development process, which requires practical and analytical skills in programming, system analysis and scheduling and meets the requirements of apply practical and analytical skills gained during the degree program.
2. To meets the requirements of innovation and creativity, non-trivial algorithms in this puzzle games have been achieved.
3. This project reviewed many kinds of literature, collected numerous puzzle algorithms and implementation related information, and finally proposed a high-quality solution, and included the evaluation of the solution, which demonstrates the ability of synthesis of information, ideas and practices.
4. This project could help researchers and learners have a better understanding of the gourds puzzle; in the meantime, this program can easily be extended to other new-raised algorithms and other puzzles.
5. During the project, the project plan and shows the self-management ability.
6. For the process of this project, there will be plans to continuously conduct self-evaluation in each version to ensure the quality of the project, which shows the critical self-evaluation ability.

10. References

[1] J. Hamersma, M. van Kreveld, Y. Uno, and T. C. van der Zanden, “Gourds: A sliding-block puzzle with turning,” arXiv. 2020.

[2] “Hamiltonian Cycle -- from Wolfram MathWorld,” [Online]. Available: https://mathworld.wolfram.com/HamiltonianCycle.html. [Accessed May 8th, 2021]