Report of Tangram Pieces Matching and Recognition Based on Fundamentals of Artificial Intelligence

Name:	Student ID:
CHEN,RUI(陈睿)	**1809853Z-I011-0017**
DAI,BEIQI(戴贝琪)	**1809853M-I011-0047**
QIN,MUPEI(秦沐霈)	**1809853J-I011-0040**

College:	**IT**
Major:	**C Major**
Instructor:	**Cheung-choy Eric**
Completion date:	**2021/04/25**

Report of Tangram Pieces Matching and Recognition Based on Fundamentals of Artificial Intelligence

Rui Chen, Beiqi Dai, Mupei Qin

【Abstract】

In order to achieve the goal of splicing 13 different templates of Tangram, our team chose to use the python to implement five different search algorithms and UI. We used matrices to represent tangram pieces and state space and use the algorithms to get the search results, which, subsequently, will be converted to UI-friendly form. Finally, the UI will handle the data and the results will be displayed with Qt library.

【Key words】

Artificial Intelligence, Depth-First Search, Greedy Search, A* Search, Uniform Cost Search, Iterative Deepening.

CONTENTS

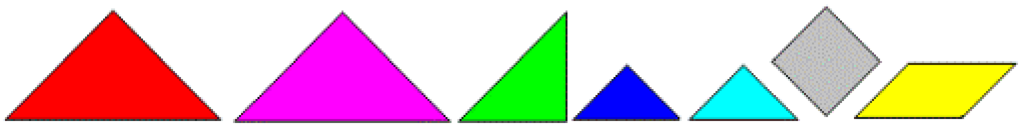
1. Introduction

1.1 Group introduction

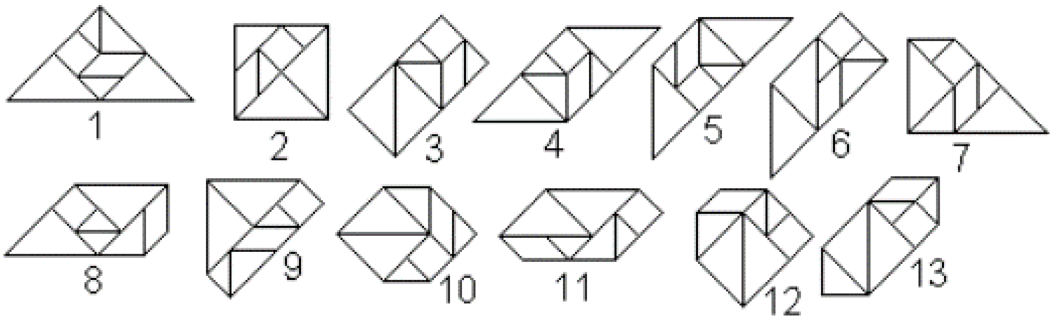
Class: D1
Group number :6
Chen, Rui 1809853Z-I011-0017
Dai, Beiqi 1809853M-I011-0047
Qin, Mupei 1809853J-I011-0040

1.2 Project introduction

Tangram is a dissection puzzle consisting of seven flat polygons: five isosceles right triangles (two small triangles, one medium triangle and two large triangles), one square and one parallelogram. The objective is to replicate a pattern, given only an outline, using all seven pieces without overlap



We need to use the tangram pieces above to splice out the 13 convex figures as shown in the figure below.



1.3 Development Environment

Environment: python 3.7 / Anaconda
IDE: PyCharm (Download form official website)
Tools: Qt designer (Use pip to install PyQt5 and PyQt5-tools), PyUIC

2. User Manual

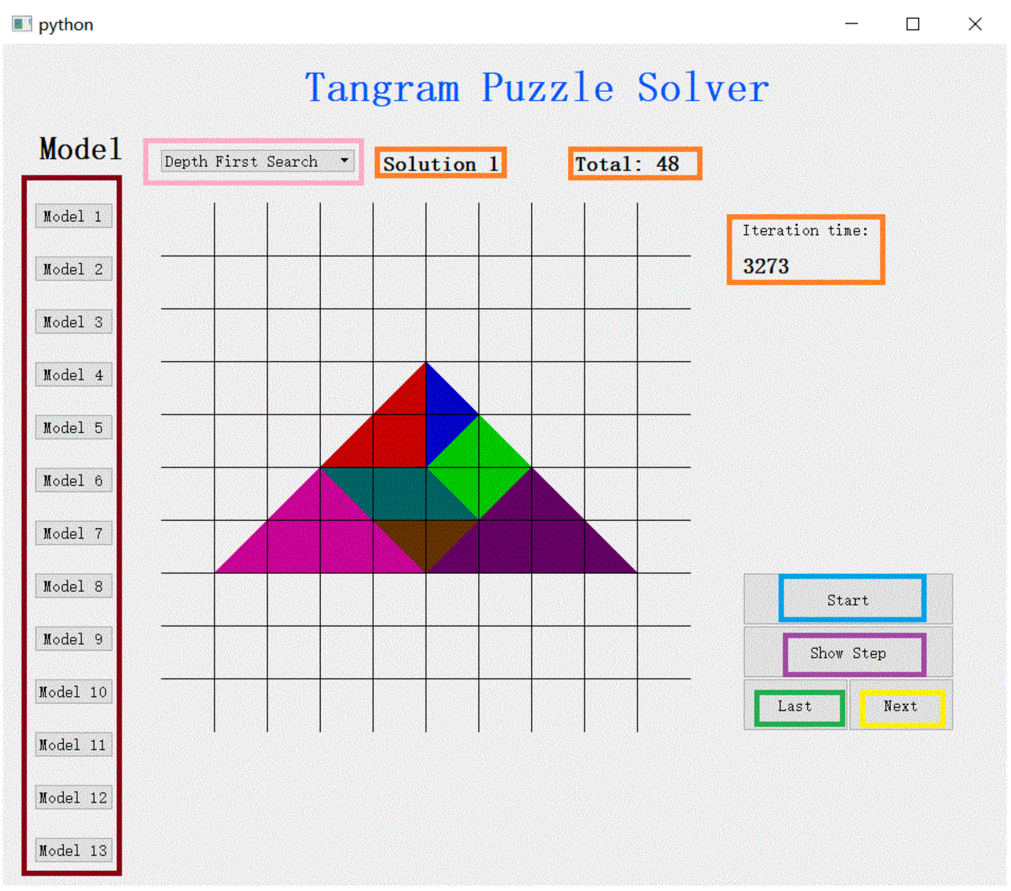


Figure 2.1

Steps:

1. Run "***main.py***" file, using python 3.7
2. Choose a button between Model1~Model13 buttons on the left.
3. Choose a search method in the dropdown box on the top.
4. Click the "start" button.
5. Wait a few seconds, the screen will display the first solution, number of total solution and total iteration time.
6. If you want to view the next solution, please click the "Next" button.
7. If you want to view the previous solution, please click the "Last" button.
8. If you want to view the piecing steps of the current solution, please click the "Show Step" button.

3. Algorithm

3.1 Problem Formulation

States: the remaining area that has not been covered by piece.

Initial state: the target figures of convex

Operators: Put a piece of tangram, covering the current state without conflicting with other existing pieces.

Goal: Cover the whole convex figure. No remaining areas is left.

Type of problem: configuration search

3.2 Mapping/Converting Methodology

All squares, which are consisted of two small triangles, have 7 different conditions. (show as figure 3.2.1)

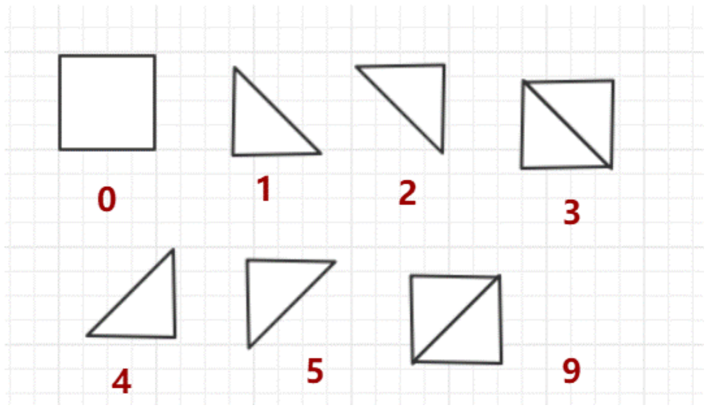


Figure 3.2.1

Each condition is assigned a given value: 1,2,3 and 4,5,9. 0 representing empty.

Therefore, for a convex figure, we can map it to a matrix. For example:

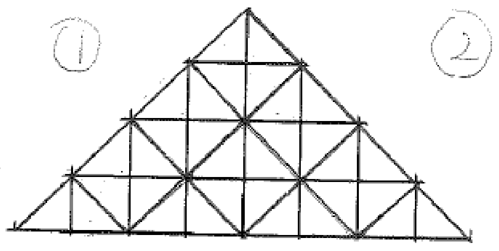


Figure 3.2.2

```
.   initialState1 = np.array([[0,0,0,0,0,0,0,0],
.                               [0,0,0,0,0,0,0,0],
.                               [0,0,0,4,1,0,0,0],
.                               [0,0,4,3,9,1,0,0],
.                               [0,4,3,9,3,9,1,0],
.                               [4,3,9,3,9,3,9,1]])
```

In addition to represent all possible convex, the shape of the convex figure is defined as **6*8**. The all presenting matrices can be found in file, *matrixData.py*.

Similarly, for all pieces of tangram, we can map them into matrices respectively.

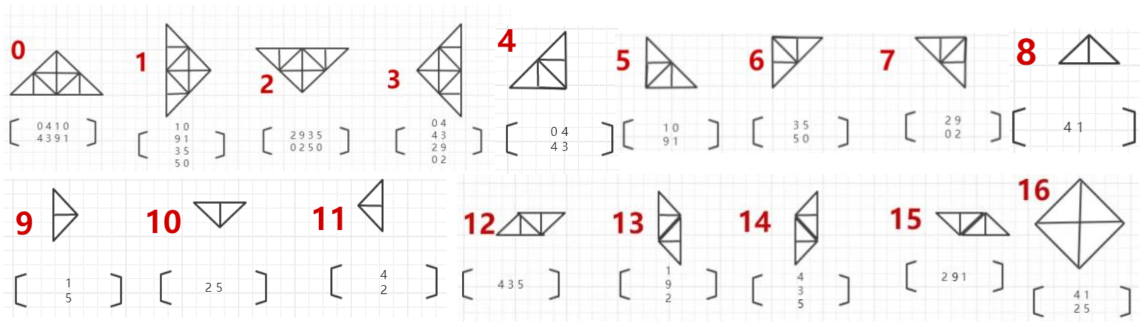


Figure 3.2.3

3.3 Operation Implementation

The operation of putting a piece on the current state is required to avoid conflicts with other existing pieces. Thus, the design of sliding window is invoked in our project.

From the left-top most corner of the current state matrix, to the right-bottom most corner of the current state matrix, every sub-matrix that has the same shape of the putting piece will be subtracted by matrix of putting piece. If all values of the result matrix are bigger than one and the subtraction procedure obeys the following two rules:

1) 4, 5, 9 cannot be subtracted by 1, 2, 3.

2) 2 cannot be subtracted by 1 and 5 cannot be subtracted by 4.

A new state will be generated. (figure 3.4)

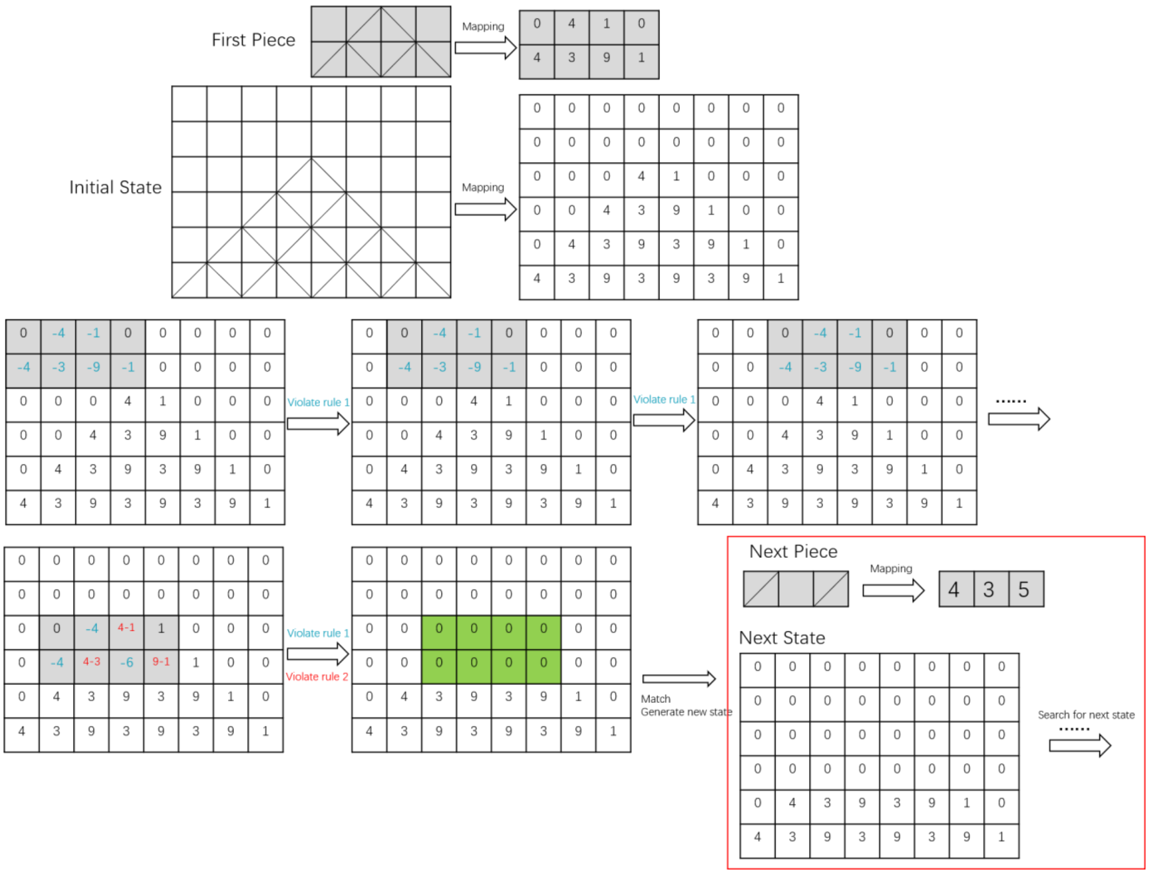


Figure 3.3.1

3.4 Uninformed Search Method

3.4.1 Depth-First Search

1. Reason to Use

Depth-First Search is an algorithm that expands the deepest node first. In our design, the goal node is always the deepest node.

2. Implementation

Recursive function is utilized in implementation.

```
def DFS(depth):
    while True:
        piece ← next_piece()
        current_state ← get_current_state()
        if depth == 7 and is_all_zeros(current_state) == True:
            save_data()
            return
        for i in 0...3 :
            isMatch ← is_find(piece, current_state, depth)
            if isMatch==True:
                DFS(depth+1)
            else:
                piece ← turn_90_degrees(piece)
            if i==3:
                return
```

Return when it finds a solution, then save the data and search for the next solution. The detailed implementation is in ***uitl.py***, ***dfs()*** function

3. Properties

1) Completeness

Yes, by traversing the whole state space, DFS will finally find every possible solution.

2) Optimality

No, DFS will traverse the whole state space and will not be optimal.

3) Time Complexity

Assume the branching factor of the first piece is b1, the second is b2 ..., the seventh is b7. Then the time complexity is $\boldsymbol{O(b1 * b2 * b3 * b4 * b5 * b6 * b7)}$.

4) Space Complexity

Assume the branching factor of the first piece is b1, the second is b2, ..., the seventh is b7. Then the Space complexity is $\boldsymbol{O(\mathrm{b1} + \mathrm{b2} + \mathrm{b3} + \mathrm{b4} + \mathrm{b5} + \mathrm{b6} + \mathrm{b7})}$.

** Note that the value of b1 to b7 is changed when the sequence of pieces to use is different. It may be very huge when the two smallest triangles are used first.*

4. Result

Here is an example of the procedure of solution search.

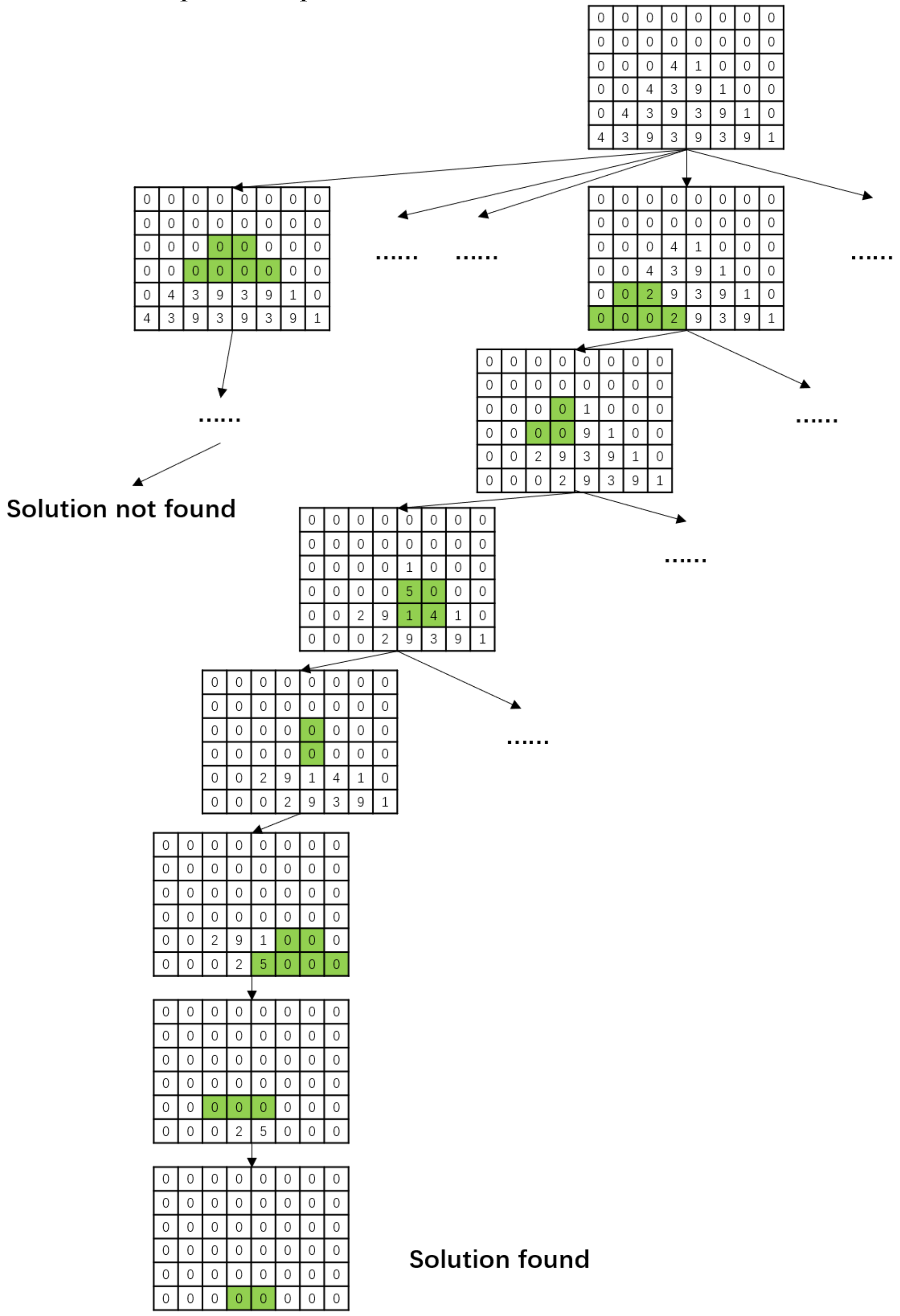


Figure 3.4.1

The piece sequence is **big triangle 1, medium triangle, square, small triangle 1, big triangle 2, parallelogram, small triangle 2.** Store the data of the solution and keep searching, finally, all solutions will be found.

3.4.2 Iterative Deepening Algorithm

1. Reason to Use

The iterative deepening algorithm is easy to implement with the completed DFS function. It combines the advantages of the depth-first and breath-first search with only moderate computational overhead.

2. Implementation

Progressively increases the cut-off from 1 to 7, with increment equals to 1.

```
def iterate(depth, cutoff):
    while True:
        piece ← next_piece()
        current_state ← get_current_state()
        if depth == 7 and is_all_zeros(current_state) == True:
            save_data()
            return
        for i in 0...3 :
            isMatch ← is_find(piece, current_state, depth)
            if isMatch==True:
                DFS(depth+1)
            else:
                piece ← turn_90_degrees(piece)
            if i==3:
                return
def iterate_deepening():
    for cutoff in 1...7:
        iterate(0,cutoff)
```

Return when it finds a solution, then save the data and search for the next solution. The detailed implementation is in ***uitl.py***, ***iterating_deepening_start()*** and iterate() function.

3. Properties

1) Completeness

Yes, similarly to depth-first search, the IDA will get all solutions when the cutoff is equal to 7.

2) Optimality

No, there is no solution when the cutoff is smaller than 7. It will cost a lot to find out all goal states.

3) Time Complexity

Assume the branching factor of the first piece is b1, the second is b2 ..., the seventh is b7. Then the time complexity is $\boldsymbol{O(b1 + b1*b2 + b1*b2*b3 + b1*b2*b3*b4 + b1*b2*b3*b4*b5 + b1*b2*b3*b4*b5*b6 + b1*b2*b3*b4*b5*b6*b7)}$.

4) Space Complexity

Assume branching factor of the first piece is b1, the second is b2 ..., the seventh is b7. Then the Space complexity is $\boldsymbol{O(7 * b1 + 6 * b2 + 5 * b3 + 4 * b4 + 3 * b5 + 2 * b6 + b7)}$.

** Note that the value of b1 to b7 is changed when the sequence of pieces to use is different. It may be very huge when the two smallest triangles are used first.*

4. Result

Here is an example of the procedure of solution search.

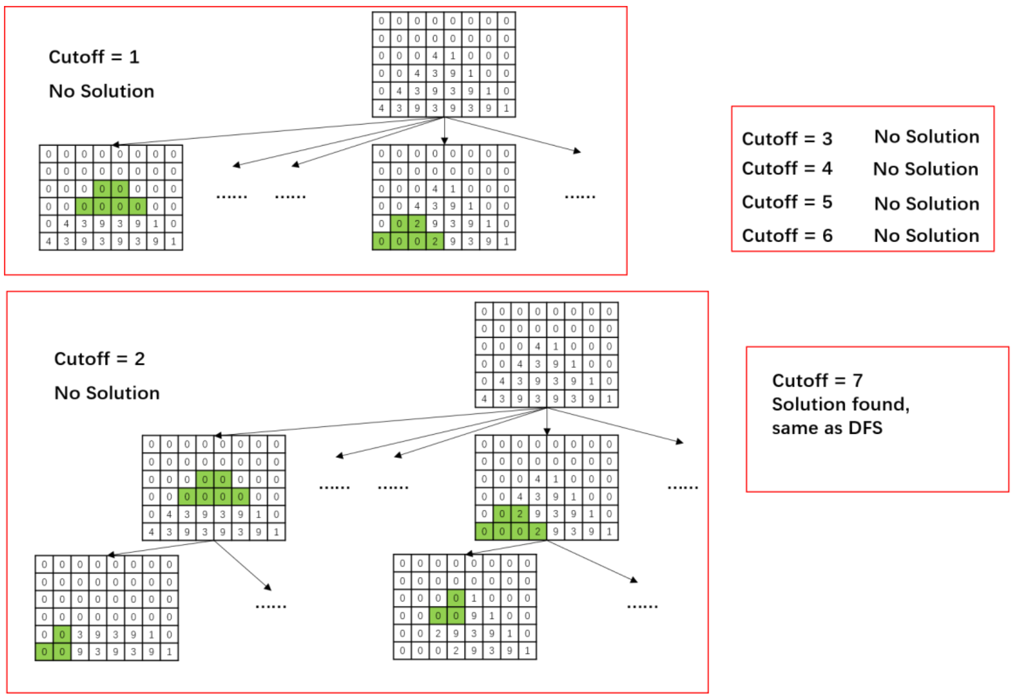


Figure 3.4.2

The result of Iterating Deepening is similar to DFS when cutoff=7, however, there is no solution existing when cutoff is smaller than 7.

Store the data of the solution and keep searching, finally, all solutions will be found.

3.4.3 Uniform Cost Search

1. Reason to Use

Uniform cost search is an algorithm for finding the minimum cost path. it will search the node with minimum path cost first.

2. Cost Function

We define:

i : the row number of the target convex figure
j : the column number of the target convex figure
m : the row number of the piece.
n : the column number of the piece.
pieceLen[index] : the length of index-th piece.

$$\boldsymbol{cost}[\boldsymbol{index}] = (\boldsymbol{i} - \boldsymbol{m} + \boldsymbol{1}) * (\boldsymbol{j} - \boldsymbol{n} + \boldsymbol{1}) * \boldsymbol{pieceLen}[\boldsymbol{index}] \,/\, (\boldsymbol{m} + \boldsymbol{n})$$

The longer **pieceLen** is, the harder it to find a proper position that can reach the goal state. Therefore, it is cost more for a piece that has longer **pieceLen**

3. Implementation

Calculate the cost of rest pieces and choose the lowest-cost piece as the next.

```
def Uniform_Cost_Search(depth):
    while True:
        calculate_cost_of_rest_piece()
        piece ← next_lowest_cost_piece()
        current_state ← get_current_state()
        if depth == 7 and is_all_zeros(current_state) == True:
            save_data()
            return
        for i in 0...3 :
            isMatch ← is_find(piece, current_state, depth)
            if isMatch==True:
                DFS(depth+1)
            else:
                piece ← turn_90_degrees(piece)
            if i==3:
                return
```

Return when it finds a solution, then save the data and search for the next solution. The detailed implementation is in **uitl.py**, **uniform_cost_search()** and **search()** function

4. Properties

1) Completeness

Yes, there is no loop exists and it will traverse all possible solutions

2) Optimality

Yes, it follows the least cost path and will search in a least-cost means.

3) Time Complexity

Time complexity is approximately the number of solutions multiply by the number of tangram pieces.

4) Space Complexity

Similar to time complexity, the space complexity is approximately the number of solutions multiply by the number of tangram pieces.

5. Result

Here is an example of the procedure of solution search.

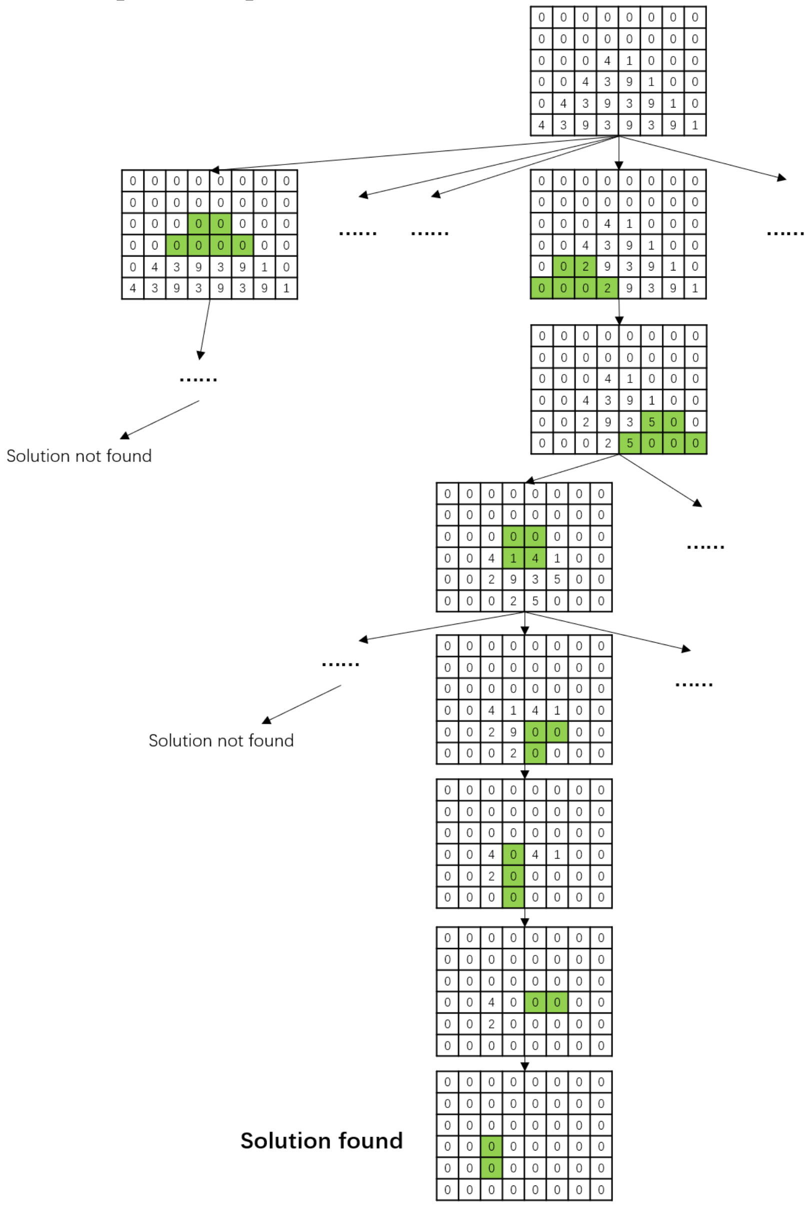


Figure 3.4.3

The piece sequence is **big triangle 1, big triangle 2, square, medium triangle, parallelogram, small triangle 1, small triangle 2.** Store the data of the solution and keep searching, finally, all solutions will be found.

3.5 Informed Search Method

3.5.1 Greedy Search

1. Reason to Use

Greedy search is simple to implement and has good efficiency.

2. Evaluation Function

f(n) = h(n)

h(n) = the remaining area that is uncovered by the piece

3. Implementation

Greedy search will choose the pieces that have larger areas. The larger a piece's area is, the smaller remaining area will be, which result in a smaller f(n).

```
def Greedy_Search(depth):
    while True:
        calculate_fn()
        piece ← next_lowest_fn_piece_in_rest_pieces()
        current_state ← get_current_state()
        if depth == 7 and is_all_zeros(current_state) == True:
            save_data()
            return
        for i in 0...3 :
            isMatch ← is_find(piece, current_state, depth)
            if isMatch==True:
                DFS(depth+1)
            else:
                piece ← turn_90_degrees(piece)
            if i==3:
                return
def calculate_fn():
    for piece in rest_pieces:
        fn[piece] = calculate_gn(piece)
```

Return when it finds a solution, then save the data and search for the next solution. The detailed implementation is in **uitl.py**, **greedy_start()** and **search()** function.

4. Properties

1) Completeness

Yes, there is no repeated state in the tangram state space. The every possible solution will be found when the algorithm goes through the entire state space.

2) Optimality

No, due to the areas of the parallelogram, medium-size triangle and square are the same, the greedy search cannot always find the optimal path of searching.

3) Time Complexity

Assume the branching factor of the largest piece is b1, the second largest piece is b2 ..., the seventh biggest is b7. Then the time complexity is $\boldsymbol{O(b1 * b2 * b3 * b4 * b5 * b6 * b7)}$.

4) Space Complexity

Assume the branching factor of the biggest piece is b1, the second biggest piece is b2 ..., the seventh biggest is b7. Then the Space complexity is $\boldsymbol{O(b1 + b2 + b3 + b4 + b5 + b6 + b7)}$.

** Note that the value of b1 to b7 is changed when the sequence of pieces to use is different. It may be very huge when the two smallest triangles are used first.*

5. Result

Here is an example of the procedure of solution search.

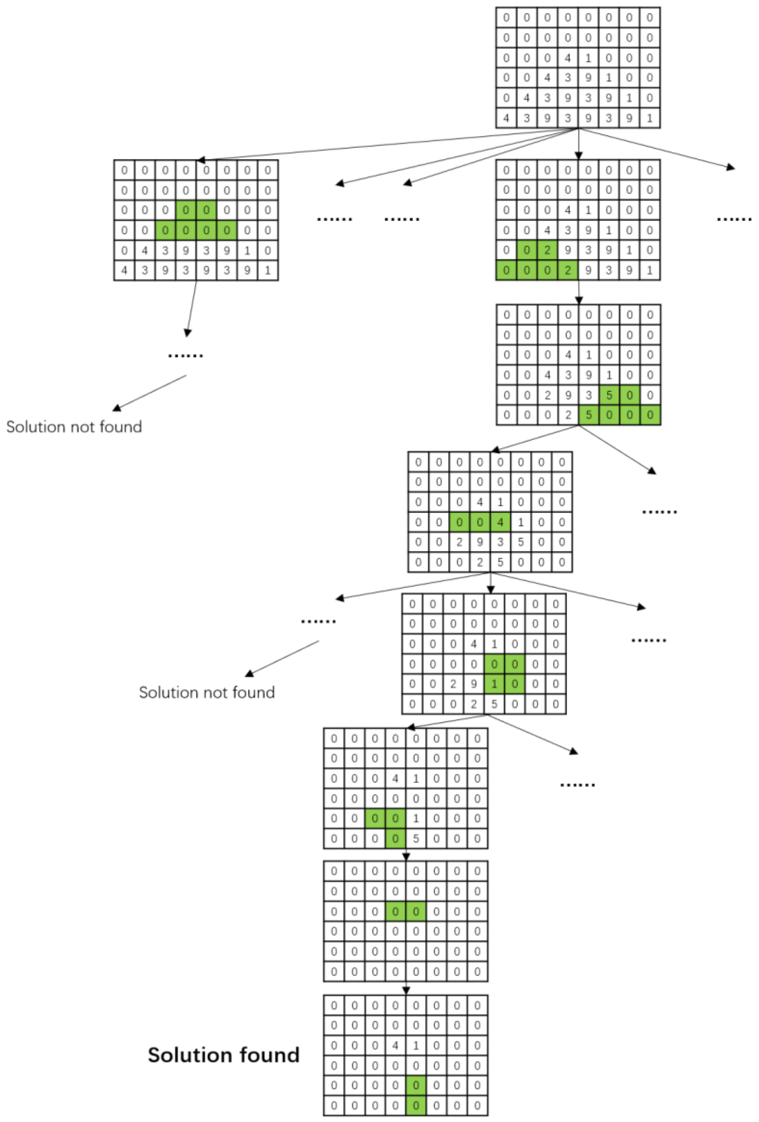


Figure 3.5.1

The piece sequence is **big triangle 1, big triangle 2, parallelogram, square, medium triangle, small triangle 1, small triangle 2.** Store the data of the solution and keep searching, finally, all solutions will be found.

3.5.2 A* Search

1. Reason to Use

A* Search will combine the information of greedy search and uniform cost search. It will combine the exploration information in uniform cost search and heuristic information in greedy search.

2. Evaluation Function

f(n) = g(n) + h(n)

h(n) = the remaining area that is uncovered by the piece

g(n) = cost of calculation

We define:

i : the row number of the target convex figure

j : the column number of the target convex figure

row : the row number of the piece.

column : the column number of the piece.

pieceLen[index] : the length of index-th piece.

(The longer the pieceLen is, the harder to find a proper place to put)

$$\boldsymbol{g}(\boldsymbol{n}) = (\boldsymbol{i} - \boldsymbol{row} + \mathbf{1}) * (\boldsymbol{j} - \boldsymbol{column} + \mathbf{1}) * \boldsymbol{pieceLen}[\boldsymbol{index}]/(\boldsymbol{row} + \boldsymbol{column})$$

3. Implementation

```
def A_star_Search(depth):
    while True:
        calculate_fn()
        piece ← next_lowest_fn_piece_in_rest_pieces()
        current_state ← get_current_state()
        if depth == 7 and is_all_zeros(current_state) == True:
            save_data()
            return
        for i in 0...3 :
            isMatch ← is_find(piece, current_state, depth)
            if isMatch==True:
                DFS(depth+1)
            else:
                piece ← turn_90_degrees(piece)
            if i==3:
                return
def calculate_fn():
    for piece in rest_pieces:
        fn[piece] = calculate_gn(piece)
        fn[piece] += calculate_hn(piece)
```

Return when it finds a solution, then save the data and search for the next solution. The detailed implementation is in **uitl.py**, **A_star_start()** and **search()** function.

4. Properties

1) Completeness

Yes, the A* search will not stick in an infinite loop and will get solutions eventually.

2) Optimality

Yes, with an admissible heuristic function, the A* search will never overestimate the distance to the goal. Therefore, it is optimal.

3) Time Complexity

Time complexity is approximately the number of solutions multiply by the number of tangram pieces.

4) Space Complexity

Similar to time complexity, the space complexity is approximately the number of solutions multiply by the number of tangram pieces.

5. Result

Here is an example of the procedure of solution search.

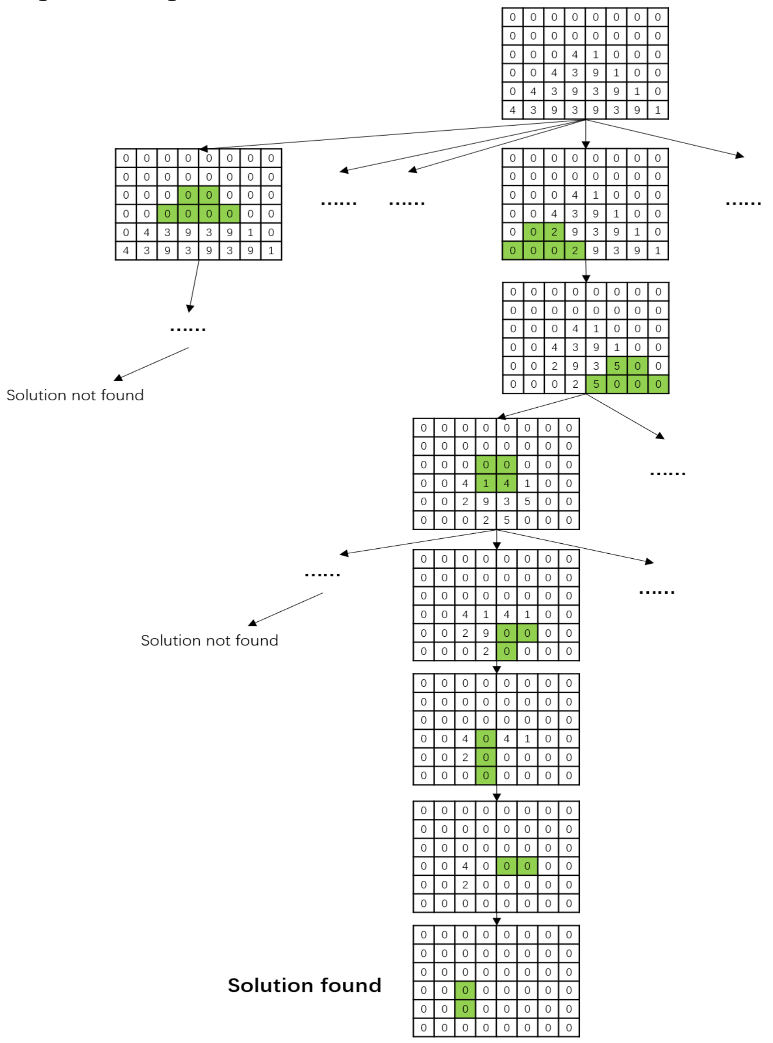


Figure 3.5.2

The piece sequence is **big triangle 1, big triangle 2, square, medium triangle, parallelogram, small triangle 1, small triangle 2.** Store the data of the solution and keep searching, finally, all solutions will be found.

4. UI design

4.1 Grey background figures

There are 13 grey background figures to display 13 kinds of target convex figures.

We Set a variable, self.Draw in *UI.py*, it is used to show the shadow of target figures.

```
def Model_1(self):
```

```
def paintEvent(self, e):
    if self.Draw == 1:
    ...... #  see detail in UI.py
```

The *paintEvent()* function will be used to display the UI and the change in Self.Draw value will show the different figures. (Figure 4.1)

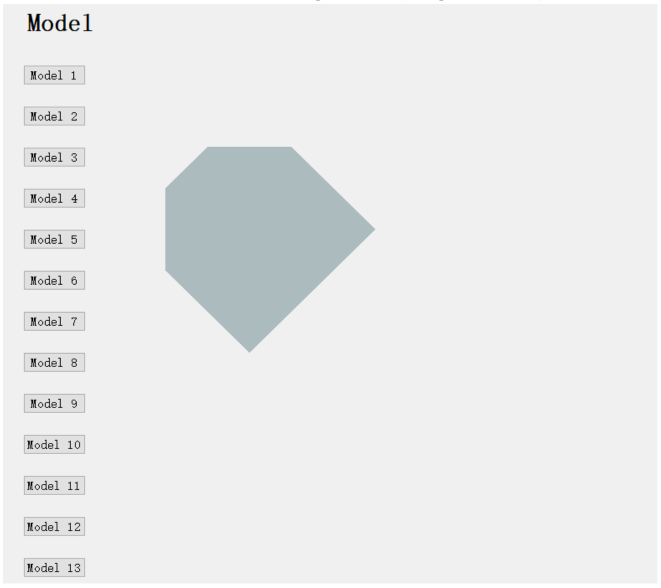


Figure 4.2

4.2 Filling color in pieces

There are 7 pieces of tangram in different colors, we use the following design to pass search data to user interface and draw them in different color respectively.

✧ **Data passed to the UI**

[row, column, pid, num]

row: the number of row where the top left corner of the graph is located.

column: the number of columns where the top left corner of the graph is located.

pid: corresponding Tangram plate, we give every piece of tangram a pid from 0 to 15, including rotated graphics.

num: number of pieces (0~6).

The following data is a complete variable called parameters

```
[[5, 1, 0, 0],
 [5, 5, 0, 1],
 [2, 4, 13, 2],
 [3, 6, 4, 3],
 [4, 3, 16, 4],
 [3, 5, 9, 5],
 [4, 2, 10, 6]], # solution1]
```

One 1*4 matrix can display one piece of the tangram and one 7*4 matrix can display one solution.

Here is the schematic diagram of pid:

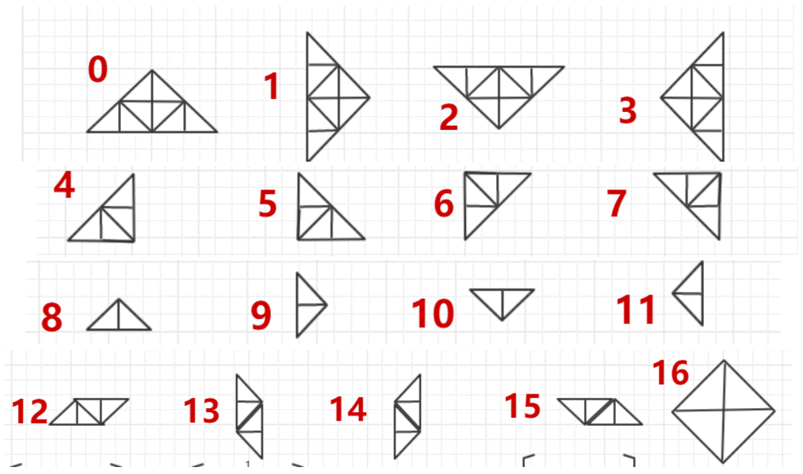


Figure 4.3

Here is an example to draw one big triangle:

```
if (m[j][num][2] == 0):
    ini_2 = QPolygon()
    ini_2.setPoints(m[j][num][1] * 50 + 250, m[j][num][0] * 50 + 150, .
.....)
```

UI get the coordinates of upper left corner of one piece from the algorithm, and the coordinates can determine other vertex coordinates. Then use the drawPolygon, the UI can show this piece of triangle at the specific location.

m[j][num][1] means the x-axis of the starting point and m[j][num][0] means the y-axis of the starting point. And four points (starting point and ending point need to coincide) can determine three sides of a triangle and form a triangle.

Here is the code that can color the pieces according to the fourth parameter:

```
def drawImage(self, painter, int):
```

The fourth parameter can determine the color of each piece: 0 red, 1 green, 2 blue, 3 purple, 4 dark green 5 brown and 6 pink.

Here is the schematic diagram of top left corner of the figure 4.2:

The blue point a is the top left corner of the graph and the coordinates of it is (5,4) which corresponds to the first two parameters [row, column].

The parameter of triangle shown here and it's is **[5,4,0,0].**

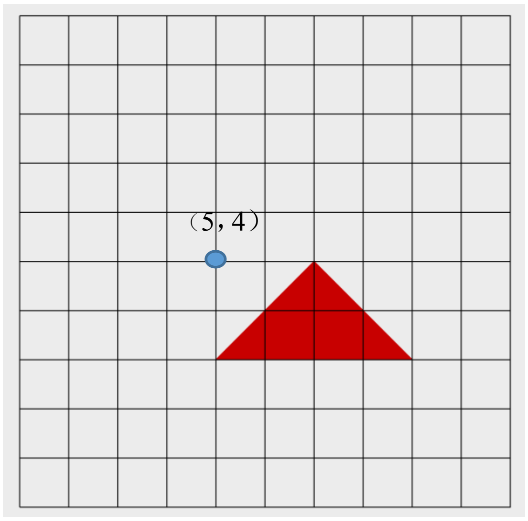


Figure 4.4

4.3 The implementation of showing steps

In this model, we need to use the QTimer to set a timer of 1000ms, to control every piece is displayed by one second. Then, we set one button in the *UI.py*, named "show step". Clicking on it can show the emergence of each plate (every one second show one piece).

```
def clickButton_step(self)
```

Then connect the button to a slot function (cliskButton_step), when the condition of clicking the button is triggered, the timer will start and stop after 1000ms.

```
self.timer = QTimer(self)
self.timer.timeout.connect(self.timeout)
```

Connecting timer and timeout.

```
def timeout(self):
```

```
for num in range(0, 7):
if num > self._step - 1:
    break
```

In the function *paintevent()*, we set two for loops, and the inner loop controls the diplay of each pieces of tangram. After 1000ms, the function timeout will be called, and the parameter _step will plus 1 and restart the timing of 1000ms, then start next loop to display next piece. At last, after 7 loops the function will stop.

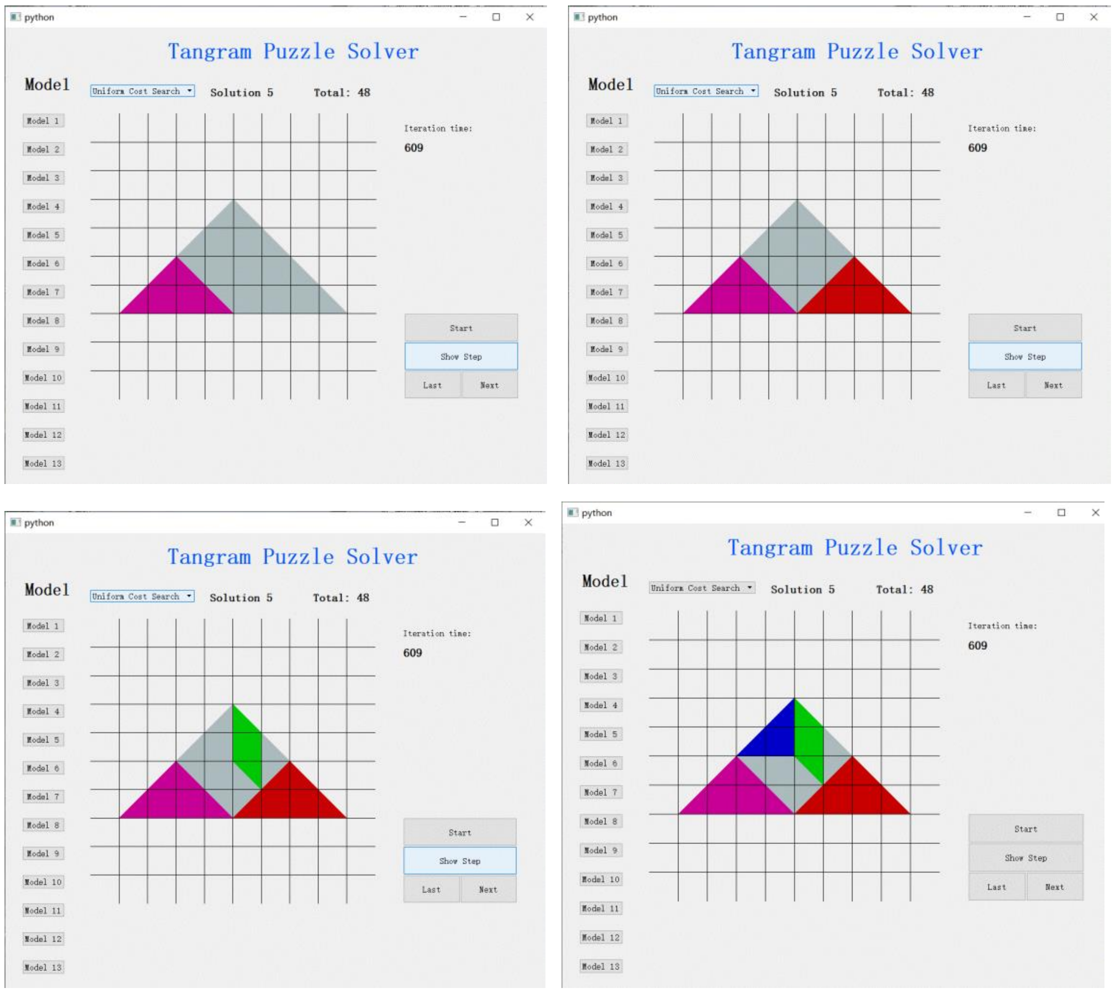


Figure 4.5

4.4 The implementation of next and last buttons

Every time, the UI can show one solution which is determined by one 7*4 matrix of parameters.

First, we set two buttons in the *UI.py*, one is last and one is next. Clicking on it can show last or next solution of the tangram.And we define a variable self. Index = 1 in *UI.py* for the next function.

```
def clickButton_last(self):
def clickButton_next(self, i):
```

Then connect each button to a slot function, when the condition of clicking the button is triggered, the _index +1(next) or -1(last).

```
def paintEvent(self,e):
   d = len(m)
   if self._index > d:
       self._index = d
   if self._index < 1:
       self._index = 1
   x = self._index
   for j in range(x-1, x):
```

In the function *paintevent()*, we set two for loops, the outer loop controls the display and page turning of each solution. If _index +1 then x will plus 1, the displayed solution will jump to the next matrix to display the next solution. If it's at the last solution, clicking on "next", and the solution won't change. The same goes for clicking button "last", it will display previous solution.

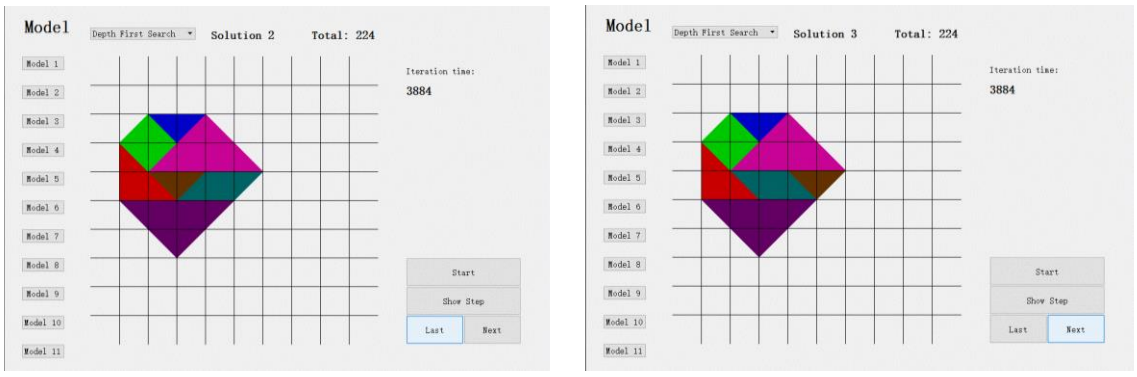


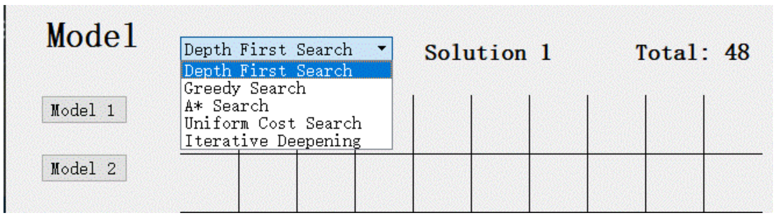
Figure 4.6

4.5 The dropdown box of search method

First, we set up a dropdown box with five choices and connect to the algorithm module in *UI.py.*

```
def algorithm(self,int):
```

Return the value after selecting the box



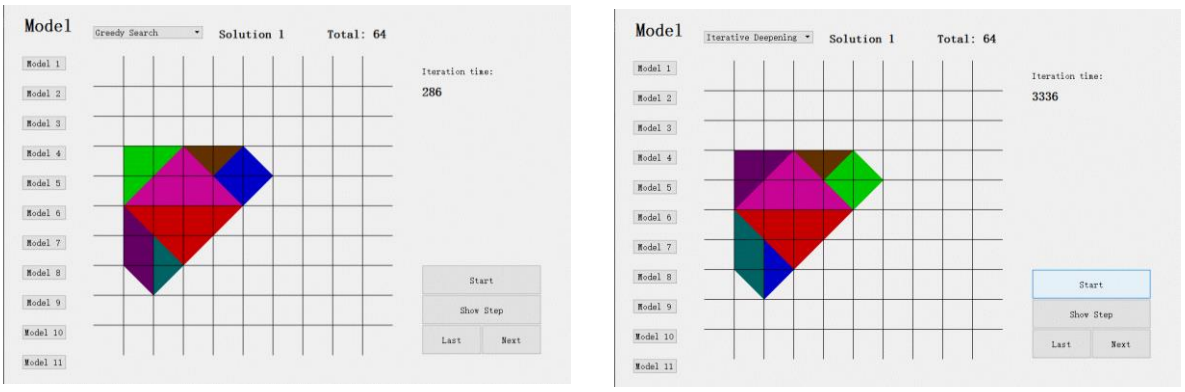


Figure 4.7

5. Discussion

5.1 Difficulty in Algorithm

#1: It is hard to implement a recursive function. The process of debugging is torturing.

Solution: With patience, cautiousness and the help of debug tools, the program finally run as same as designed.

#2: The data representation of tangram pieces and states.

Solution: With several times of attempts and inspiration of linear algebra, the matrix is used in the data representation.

#3: The heuristic function for A* search and the cost for uniform cost search.

Solution: Combining guess and practice, we finally get the heuristic function for A* search and the cost for uniform cost search.

5.2 Difficulty in UI design

#1: When we write code, we always think that all drawing codes must be encapsulated in *paintevent()*, which leads to a lot of duplicate code in the *paintevent()*module.

Solution: We use other slot functions in *paintevent()* to execute repeated code, which reduces a lot of repeated code.

#2: Ordinary slot functions cannot be called by *paintevent()*.

Solution: The slot function called by *paintevent()* must introduce the parameters used in *paintevent()*.

6.Contribution

Chen, Rui design the matrices data representation for tangram pieces. Also, he implemented the five search algorithms and related report content.

Dai, Beiqi and Qin, Mupei are responsible for UI design and related report content.