CSE537 Assignment-1 Report: Peg Solitaire

Remy Oukaour
SBU ID: 107122849
remy.oukaour@gmail.com

Jian Yang
SBU ID: 110168771
jian.yang.1@stonybrook.edu

Abstract

This is a report for the assignment 1 in course CSE537 Artificial Intelligence. Assignment 1 required to implement the ID-DFS(iterative deepening depth-first search) and two different heuristic A* methods. In this report, we would discuss the implementation details, and we also compared the six solutions, including the ID-DFS method, four A* methods and one baseline method.

1 Introduction

TODO: write the introduction

2 Implementation

2.1 Data Structure Optimazation

2.1.1 pegSolitaire

The original data structure to present pegSolitaire is a single state node to present a node. For the implementation for A* and the purpose to make the routine more clear, we separated the game wrapper and the game state to two different classes, the first one is the class *game* and the second one is the class *gameNode*.

2.1.2 gameState

The original data structure to save game state in given framework was a 2-dimensional integer array. The indexing to every single entry required two indexes, one is the column and another is the row. To simplify this, we flattened the array to 1-dimensional. According to that, the size of the array is 49, and we mapped the original entries to 1-dimenstional array by $new\_pos = old\_pos[0] \times 7 + old\_pos[1]$ where new_pos is the index in 1-dimensional array and old_pos is the indexes tuple in 2-dimensional array.

2.2 Required Implementation

There are four required implementation in file pegSolitaireUtils.py:

1. is_corner(self, pos):

```
def is_corner(self, pos):
    """
        Return whether a position is a corner/wall,
        i.e. not a hole or peg.
    """
    return 0 <= pos < 49 and self.state[pos] != -1
```

In implementation, we used another function $\_\_getitem\_\_(self, pos)$ instead:

```
def __getitem__(self, pos):
    """
    node[pos] is bounds-checked shorthand for node.state[pos].
    """
    return self.state[pos] if 0 <= pos < 49 else -1
```

2. *is_validMove*(*self*, *oldPos*, *direction*) :

```
def is_validMove(self, oldPos, direction):
    """
    Return whether it is valid to move a peg from a given position
    in a given direction. The position must have a peg, the
    destination must be empty, and the intermediate position
    must have another peg.
    """
    # Since self.getNextPosition and self.__getitem__ are
    # bounds-checked.
    # the self.is_corner check is redundant, so I commented it out
    ########################################
    # DONT change Things in here
    # In this we have got the next peg position and
    # below lines check for if the new move is a corner
    #newPos = self.getNextPosition(oldPos, direction)
    #if self.is_corner(newPos):
    #   return False
    ########################################
    # YOU HAVE TO MAKE CHANGES BELOW THIS
    # check for cases like:
    # if new move is already occupied
    # or new move is outside peg Board
    # Remove next line according to your convenience
    midPos = self.getNextPosition(oldPos, direction)
    newPos = self.getNextPosition(midPos, direction)
    # 1 is a peg, and 0 is a hole
    # Since oldPos is valid, don't use bounds-checked indexing
    # for it
    return (self.state[oldPos] == self[midPos] == 1 and\
                self[newPos] == 0)
```

3. getNextPosition(self, oldPos, direction):

```
@staticmethod
def getNextPosition(oldPos, direction):
    """
    Return the new position after moving in the given direction
    (7 is north/up, 1 is east/right, -7 is south/down, and -1 is
    west/left).
    Invalid movements outside the bounds of the board will return
    positions such that is_corner(position) returns True.
```

```
    """
    newPos = oldPos + direction
    # If direction is 7 or -7 (changing rows), newPos is already
    # correct
    return (newPos if direction == 7 or direction == -7 or
    # If direction is 1 or -1 (changing columns), the row should
    # not change
        newPos // 7 == oldPos // 7 else -1)
```

4. getNextState(self, oldPos, direction):

```
def getNextState(self, oldPos, dir, pegSol):
    """
    Return a child node of the current one, created by a given
    valid move.
    The given game has its count of expanded nodes incremented.
    """
    ################################################
    # DONT Change Things in here
    pegSol.nodesExpanded += 1
    if not self.is_validMove(oldPos, dir):
        print "Error, You are not checking for valid move"
        exit(0)
    ################################################
    # YOU HAVE TO MAKE CHANGES BELOW THIS
    # Update the gameState after moving peg
    # eg: remove crossed over pegs by replacing it's
    # position in gameState by 0
    # and updating new peg position as 1
    midPos = self.getNextPosition(oldPos, dir)
    newPos = self.getNextPosition(midPos, dir)
    # x[:] makes a copy of x (necessary to avoid mutating
    # self.state when updating childState)
    childState = self.state[:]
    childState[oldPos] = 0 # The peg moves from here, leaving a hole
    childState[midPos] = 0 # The jumped-over peg is removed
    childState[newPos] = 1 # The peg moves to this hole
    # Convert positions back into pairs for printing
    childTrace = self.trace + [(oldPos // 7, oldPos % 7), \
                        (newPos // 7, newPos % 7)]
    return gameNode(childState, childTrace, self.pegCount - 1,\
              self.heuristic)
```

2.3 Implementation of methods

2.3.1 ID-DFS

2.3.2 A* with heuristic method 1

```
        distances = [
```

```
                5, 4, 3, 2, 3, 4, 5,
                4, 3, 2, 1, 2, 3, 4,
                3, 2, 1, 0, 1, 2, 3,
                2, 1, 0, 1, 0, 1, 2,
                3, 2, 1, 0, 1, 2, 3,
                4, 3, 2, 1, 2, 3, 4,
                5, 4, 3, 2, 3, 4, 5
        ]
```

2.3.3 A* with heuristic method 2

```
        # difficulties = [
        #       0, 0, 4, 0, 4, 0, 0,
        #       0, 0, 0, 0, 0, 0, 0,
        #       4, 0, 3, 0, 3, 0, 4,
        #       0, 0, 0, 1, 0, 0, 0,
        #       4, 0, 3, 0, 3, 0, 4,
        #       0, 0, 0, 0, 0, 0, 0,
        #       0, 0, 4, 0, 4, 0, 0,
        # ]
```

2.3.4 A* with heuristic method 3

```
def heuristicThree(node):
        state = node.state
        # Count the number of dangling pegs
        num_dangling = 0
        for i in xrange(49):
                if state[i] != 1: continue
                for direction in [7, 1, -7, -1]:
                        if node[i + direction] == 1:
                                break
                else:
                        num_dangling += 1
        return node.pegCount * 2 + num_dangling
```

2.3.5 A* with heuristic method 4

```
        # corners = [
        #       0, 0, 1, 1, 1, 0, 0,
        #       0, 0, 1, 1, 1, 0, 0,
        #       1, 1, 0, 0, 0, 1, 1,
        #       1, 1, 0, 0, 0, 1, 1,
        #       1, 1, 0, 0, 0, 1, 1,
        #       0, 0, 1, 1, 1, 0, 0,
        #       0, 0, 1, 1, 1, 0, 0,
        # ]
```

2.3.6 Baseline for A*

We used the number of left pegs to be the cost function for a baseline methods.

2.3.7 A* with heuristic method mixed from 2 and 4

```
        difficulties = [
                0, 0, 4, 1, 4, 0, 0,
                0, 0, 1, 1, 1, 0, 0,
                4, 1, 2, 0, 2, 1, 4,
                1, 1, 0, 1, 0, 1, 1,
                4, 1, 2, 0, 2, 1, 4,
                0, 0, 1, 1, 1, 0, 0,
                0, 0, 4, 1, 4, 0, 0,
        ]
```

2.3.8 Other abandoned methods

We also tried Manhattan distances but this cost function doesn't work well as the above methods we used. So we didn't include it in the following performance disucssion.

2.4 Other Optimization for Performance

3 Benchmark

3.1 Manual Tests

We have written 10 manual cases from impossible cases to extreme cases.

3.2 Random Tests

For test purpose, we implemented one random generator to generate a game board status. The idea is, based on the final state, we would randomly generate a valid backward move based on current board and apply the move and repeat this for random steps.
The pseudocode is:

Algorithm 1 Random Test generator

1: $num\_step \leftarrow$ a random number between 1 and 33
2: $board\_state \leftarrow$ initialized board state with only one peg in the center
3: **for** $j \leftarrow 1$ to num_step **do**
4: $\quad valid\_startpos, valid\_direction \leftarrow$ find a valid random move
5: $\quad board\_state[valid\_startpos] \leftarrow 0$
6: $\quad board\_state[valid\_startpos + valid\_direction] \leftarrow 1$
7: $\quad board\_state[valid\_startpos + valid\_direction + valid\_direction] \leftarrow 1$
8: **end for**
9: **return** $board\_state$

4 Result Study

4.1 Basic Cases

And the result for our different methods are as shown in table 1:

case/num nodeExpand	ID-DFS	$A*-1$	$A*-2$	$A*-3$	$A*-4$	$A*-Manhattan$	$A*-mix$	Baseline
given case	50	16	14	18	22	19	13	13

Table 1: This table shows some data

4.2 Impossible Case

We added one case with two noncontiguous pegs. There shouldn't be solutions for such a case and the program should quit without hanging. In our test, only ID-DFS expanded two nodes and other methods expanded only one, and then the program quilted properly without any issue.

4.3 Extreme Cases

The most extreme case for this problem is the "all full but the center" condition that showed as the figure 1. And the result for our different methods are as shown in table 2:

Figure 1: Full but the center case

case/#nodeExpand	ID-DFS	$A*-1$	$A*-2$	$A*-3$	$A*-4$	$A*-Manhattan$	$A*-mix$	Baseline
Extreme	MemErr	1,967,299	175	5,714,286	530,537	1,801,655	103,402	26,847

Table 2: This table shows some data

4.4 Mean and Median

For the whole benchmark, we calculated the mean and the median numbers of the expanded nodes in table 3. In comparison we chose $A*-1$ and $A*-mix24$" in our final submitted program.

#nodeExpand	ID-DFS	$A*-1$	$A*-2$	$A*-3$	$A*-4$	$A*-Manhattan$	$A*-mix$	Baseline
Mean	1,979,736,	47,084	4,398	46,408	8,937	19,201	3,388	5,146
Median	6,417	74	129	74	114	95	98	92

Table 3: This table shows some data

4.5 Performance Study with other tricks

In our testing, we found one interesting phenomenon. For several large cases, if the movement orders are different, for example, expand one node by the order ”East, West, North, South” might be quite different from the order ”East, North, West, South”.
There are 24 possible orders, where 24 = 4!, and we tested all the 24 possible orders with the above 6 methods, which included five A* methods and one baseline method. And we counted two sets of summations, one is the set of summations for ”opposite-order first” for the movements and another set of summations is for the ”non-opposite-order first”. Here ”opposite-order first” contained 12 conditions starting with opposite directions at the first two and the last two pairs such as ”East, West, North, South”. For ”non-opposite-order first”, such as ”East, North, West, South”. In a box plot in figure 2, it’s obvious that the ”non-opposite-order first” had better.

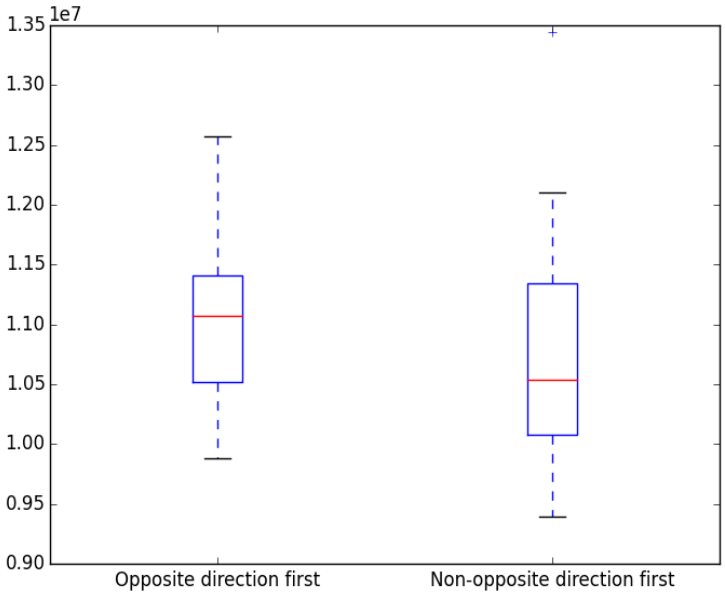

Figure 2: Box plot for two sets of summations

5 Conclusion