

# CSCE625: Artificial Intelligence

## Programming Assignment 5

### Packing Puzzle

Name: Sun Yul Lee

UIN: 824007724

#### Implementation

To tackle the problem, I utilize numpy array to create tiles and a grid where each tile is placed. To implement this idea, I create classes for each tile. Tile classes contain the shape of tile as numpy array, the number of different shapes which can be produced from rotation, and the number of different shapes which can be produced from reflection. To identify the tiles, I assign different number to the cells of the tile (T – 1, I – 2, O – 3, J – 4, S – 5). Then, I implement a class of state in the problem. Since a grid and tiles which can be placed on a grid are changed for each state, I make a State class to contain a grid and a bag of tiles. A Bag class contains the number of remaining tiles for each type of tiles. Therefore, the program contains 5 types of tile classes, 1 State class, and 1 Bag class.

- class State()
  - `__init__(self, grid, bag)`: This function gets 2d numpy array and Bag object to define a condition of a state.
  - `is_goal(self)`: This function check whether a state is in goal condition or not. In this assignment, a goal state is a case where all tiles are perfectly placed on a grid. Therefore, I define a goal state as a state where all cells in a grid is not zero and the number of remaining tiles in a Bag object is zero.
  - `get_successors(self)`: If there are two tiles with the same shape, the successors of a current state generated from using the two tiles are the same. Therefore, I combine cases of using the same type of tile in generating successors. So, this function first gets a list of possible types of tiles. For each type, it gets a list of locations where a tile can be place by calling `can_fit()`. Then, it creates a list of successors.
  - `can_fit(self, tile)`: This function check whether overlapping occurs when a tile is placed on a grid for all zero cells. For each type of tiles, this function checks all shapes generated from rotation and reflection. This function also calculates sum of distances between each cell in a tile and the closest non-zero or boundary and create a tuple of possible location, shape of a tile, and the heuristic value.
  - `sum_dist_non_zeros(self)`: This function is one of heuristic functions that I implemented in this assignment. This heuristic function is generated from an idea that a grid where all placed tiles are close to each other might have a higher chance to find a solution. This function calculates sum of distances between non-zero cells in a grid. This heuristic function is used in local beam search algorithm.
  - `sum_min_dist(self, pos, shp)`: This function is the other heuristic function that I implemented in this assignment. This heuristic function is generated from the similar idea used in the other heuristic function. This function, however, calculates a distance between each cell of a tile and the closest non-zero cell on a grid when a tile is placed on a given position. This heuristic function is used in random hill climbing algorithm.
- class Bag()
  - `__init__(self, k)`: This function gets the number of tile set and set the given value as the number of each tiles.
  - `total(self)`: This function calculates the number of remaining tiles in a bag.
  - `copy(self)`: This function creates a copy of itself.
- class T, I, O, J, S: This class contains a shape of tile, number of different shapes from rotation and number

of different shapes from reflection.

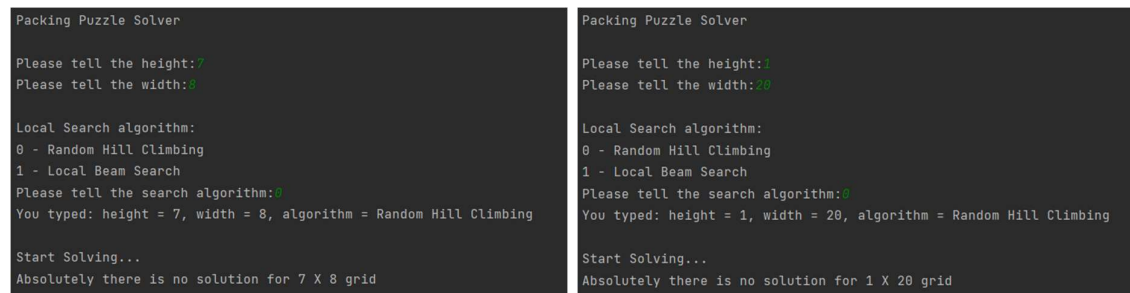
In the main function, the program asks a user to type the height and the width of a grid. It checks whether the inputs are positive integer or not. If a user types invalid input, the program ask a user to type the value again. Then, the program shows a type of local search algorithms. I implement random hill climbing algorithm and local beam search algorithm and the program shows two options. After getting all required input, the program creates process of generating a solution using selected algorithm with timeout. If the process does not finish the computation before the timeout, the process is terminated, and timeout message would be displayed. In addition, I make the program to display a grid of a state where the most of tiles in a bag are placed.

- `is_valid_grid(height, width)`: This function is used to check the validity of inputs. If multiplication of height and width can be divided by 20, it returns the result of division to inform the number of tile sets. If multiplication of height and width cannot be divided by 20, it returns 0 to inform that the inputs are invalid.
- `random_climbing_hill(state, shared_data)`: This function performs random hill climbing. I also implement random restart mechanism in the function. For given state, this function generates a list of successors. Using `sum_min_dist()` as heuristic function, this function creates a list of the best successors and randomly selects one successor from the list. Then, it makes the successor as current state and repeat the process until it finds a solution or there is no more successor. If there is no more successor, this function performs random restart. `shared_data` is used to store a state which is not a goal state and has no successors.
- `local_beam_search(state, k, shared_data)`: This function performs local beam search. This function also provides random restart mechanism. For given state, this function generates a list of successors. Using `sum_dist_non_zero()` as heuristic function, this function sorts the list in order of the heuristic value. Then, if the number of successors with the smallest heuristic value is larger than given k, this function randomly selects k successors from the successors with the smallest heuristic value. If not, this function selects first k successors from the list. When a list of successors contains less than k successors, the function chooses all successors. In this assignment, I set k = 5. `shared_data` is used to store a state which is not a goal state and has no successors.
- `generate_solution(height, width, search, shared_data)`: With the user inputs, this function creates a start state with a grid with all zero and bag of tiles. Then, it performs a selected local search algorithm. `shared_data` is used to store a state which is not a goal state and has no successors.

## Limitation

Although the program performs random hill climbing and local beam search without any error, the program cannot find a solution. I tried 4 X 10 grid where 2 sets of tiles can perfectly packed. However, the program could not find a solution with both search algorithms and reached timeout. The heuristic functions that I implemented in the program might be bad functions to achieve a goal state.

## Examples



The image contains two side-by-side screenshots of a terminal window running a program titled "Packing Puzzle Solver".

The left screenshot shows the following text:  
Please tell the height:7  
Please tell the width:8  
  
Local Search algorithm:  
0 - Random Hill Climbing  
1 - Local Beam Search  
Please tell the search algorithm:0  
You typed: height = 7, width = 8, algorithm = Random Hill Climbing  
  
Start Solving...  
Absolutely there is no solution for 7 X 8 grid

The right screenshot shows the following text:  
Please tell the height:1  
Please tell the width:20  
  
Local Search algorithm:  
0 - Random Hill Climbing  
1 - Local Beam Search  
Please tell the search algorithm:0  
You typed: height = 1, width = 20, algorithm = Random Hill Climbing  
  
Start Solving...  
Absolutely there is no solution for 1 X 20 grid

Figure 1. Outputs of program when a user types invalid inputs.



```

Packing Puzzle Solver

Please tell the height:5
Please tell the width:4

Local Search algorithm:
0 - Random Hill Climbing
1 - Local Beam Search
Please tell the search algorithm:1
You typed: height = 5, width = 4, algorithm = Local Beam Search

Start Solving...
Trial # 50 failed to find a solution.
Trial # 100 failed to find a solution.
Trial # 150 failed to find a solution.
Trial # 200 failed to find a solution.

Trial # 950 failed to find a solution.
Trial # 1000 failed to find a solution.
Time out! Fail to find a solution

```

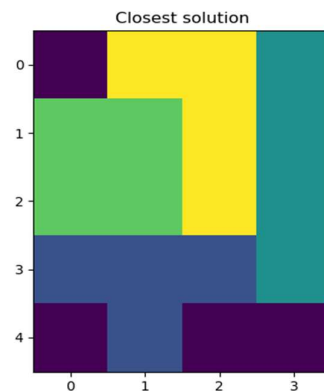


Figure 4. Local beam search on 5x4 grid

```

Packing Puzzle Solver

Please tell the height:4
Please tell the width:10

Local Search algorithm:
0 - Random Hill Climbing
1 - Local Beam Search
Please tell the search algorithm:1
You typed: height = 4, width = 10, algorithm = Local Beam Search

Start Solving...
Trial # 50 failed to find a solution.
Time out! Fail to find a solution

```

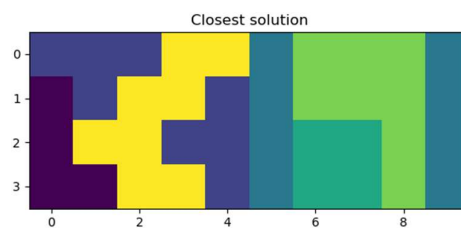


Figure 5. Local beam search on 4x10 grid

```

Packing Puzzle Solver

Please tell the height:8
Please tell the width:10

Local Search algorithm:
0 - Random Hill Climbing
1 - Local Beam Search
Please tell the search algorithm:1
You typed: height = 8, width = 10, algorithm = Local Beam Search

Start Solving...
Time out! Fail to find a solution

```

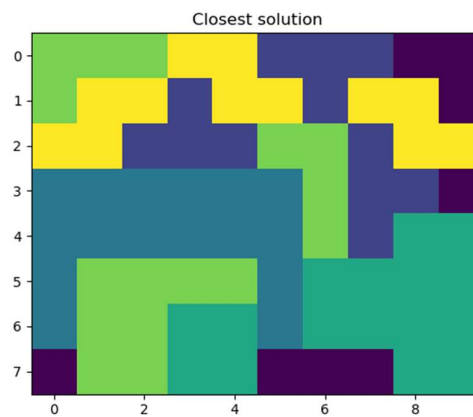


Figure 6. Local beam search on 8x10 grid

## Appendix

### Code

```
import numpy as np
from scipy import ndimage
import random
import matplotlib.pyplot as plt
import multiprocessing
import time

'''
class for a state
holds grid and bag of remaining tetrominoes of
current state
'''
class State:
    # initialization
    def __init__(self, grid, bag):
        self.grid = grid
        self.bag = bag
    # check whether the state is in goal condition
    def is_goal(self):
        if np.all(self.grid) and self.bag.total == 0:
            return True
        else:
            return False
    # get a list of successors
    # successor = (State, sum_min_dist)
    def get_successors(self):
        tiles = []
        if self.bag.num_t != 0:
            tile = T()
            tiles.append(tile)
        if self.bag.num_i != 0:
            tile = I()
            tiles.append(tile)
        if self.bag.num_o != 0:
            tile = O()
            tiles.append(tile)
        if self.bag.num_j != 0:
            tile = J()
            tiles.append(tile)
        if self.bag.num_s != 0:
            tile = S()
            tiles.append(tile)

        successors = []
        for tile in tiles:
            possible = self.can_fit(tile)
            bag = self.bag.copy()
            if type(tile) is T:
                bag.num_t -= 1
            if type(tile) is I:
                bag.num_i -= 1
            if type(tile) is O:
                bag.num_o -= 1
            if type(tile) is J:
                bag.num_j -= 1
            if type(tile) is S:
```

```

        bag.num_s -= 1

    for p in possible:
        pos = p[0]
        shp = p[1]
        grid = self.grid.copy()
        for row in range(shp.shape[0]):
            for col in range(shp.shape[1]):
                grid[pos[0] + row][pos[1] + col] += shp[row][col]
        successor = State(grid, bag)
        successors.append((successor, p[2]))
    return successors

# check a given tile can be placed on the grid
# return the list of possible cases
def can_fit(self, tile):
    loc_shape = []
    shp = tile.shape
    zeros = np.argwhere(self.grid == 0)
    for i in range(tile.num_reflect):
        shp = reflectTile(shp)
        for j in range(tile.num_rotate):
            shp = rotateTile(shp)
            for index in zeros:
                if index[0] + shp.shape[0] > self.grid.shape[0]:
                    continue
                if index[1] + shp.shape[1] > self.grid.shape[1]:
                    continue
                overlap = False
                for row in range(len(shp)):
                    for col in range(len(shp[row])):
                        if shp[row][col] != 0 and self.grid[index[0] +
row][index[1] + col] != 0:
                            overlap = True
                            break
                if overlap:
                    break
                if not overlap:
                    min_dist = self.sum_min_dist(index, shp)
                    loc_shape.append((index, shp, min_dist))
    return loc_shape

# heuristic function
# calculates sum of all distances between non zero elements on the grid
def sum_dist_non_zeros(self):
    nonzero_idx = np.transpose(np.nonzero(self.grid))
    sum_dist = 0
    for idx_1 in nonzero_idx:
        for idx_2 in nonzero_idx:
            dist = abs(idx_2[0] - idx_1[0]) + abs(idx_2[1] - idx_1[1])
            sum_dist += dist
    return sum_dist

# heuristic function
# calculates sum of distances between each cell in the tile
# and the closest non-zero or boundary
def sum_min_dist(self, pos, shp):
    result = 0

    if not np.any(self.grid):

```

```

        for row in range(shp.shape[0]):
            for col in range(shp.shape[1]):
                if shp[row][col] != 0:
                    x_dist = min(pos[0] + row, self.grid.shape[0] -
(pos[0] + row))
                    y_dist = min(pos[1] + col, self.grid.shape[1] -
(pos[1] + col))
                    min_dist = x_dist + y_dist
                    result += min_dist
            return result

    for row in range(shp.shape[0]):
        for col in range(shp.shape[1]):
            min_dist = self.grid.shape[0] + self.grid.shape[1]
            if shp[row][col] != 0:
                for r in range(self.grid.shape[0]):
                    for c in range(self.grid.shape[1]):
                        if self.grid[r][c] != 0:
                            dist = abs(pos[0] + row - r) + abs(pos[1] +
col - c)
                            if dist < min_dist:
                                min_dist = dist
                    result += min_dist
            return result

'''
class for a bag of tetrominoes
'''
class Bag:
    # initialization
    def __init__(self, k):
        self.num_t = k
        self.num_i = k
        self.num_o = k
        self.num_j = k
        self.num_s = k

    # get the number of tiles in the bag
    def total(self):
        return self.num_t + self.num_i + self.num_o + self.num_j +
self.num_s

    # copy the bag
    def copy(self):
        copied = Bag(0)
        copied.num_t = self.num_t
        copied.num_i = self.num_i
        copied.num_o = self.num_o
        copied.num_j = self.num_j
        copied.num_s = self.num_s
        return copied

'''
class for tile with shape T
    num_rotate - num of possible shape from rotation
    num_reflect - num of possible shape from reflection
'''
class T:
    def __init__(self):

```

```

        self.shape = np.array([[1, 1, 1],
                                [0, 1, 0]])

        self.num_rotate = 4
        self.num_reflect = 1

'''
class for tile with shape I
    num_rotate - num of possible shape from rotation
    num_reflect - num of possible shape from reflection
'''
class I:
    def __init__(self):
        self.shape = np.array([[2, 2, 2, 2]])
        self.num_rotate = 2
        self.num_reflect = 1

'''
class for tile with shape O
    num_rotate - num of possible shape from rotation
    num_reflect - num of possible shape from reflection
'''
class O:
    def __init__(self):
        self.shape = np.array([[3, 3],
                                [3, 3]])

        self.num_rotate = 1
        self.num_reflect = 1

'''
class for tile with shape J
    num_rotate - num of possible shape from rotation
    num_reflect - num of possible shape from reflection
'''
class J:
    def __init__(self):
        self.shape = np.array([[0, 4],
                                [0, 4],
                                [4, 4]])

        self.num_rotate = 4
        self.num_reflect = 2

'''
class for tile with shape S
    num_rotate - num of possible shape from rotation
    num_reflect - num of possible shape from reflection
'''
class S:
    def __init__(self):
        self.shape = np.array([[0, 5, 5],
                                [5, 5, 0]])

        self.num_rotate = 2
        self.num_reflect = 2

# rotating a tile
def rotateTile(tile):
    return np.rot90(tile)

# reflecting a tile

```



```

def reflectTile(tile):
    return np.flip(tile, 1)

# check whether a given values form valid grid
def is_valid_grid(height, width):
    area = height * width
    if area % 20 == 0:
        return int(area / 20)
    return 0

# perform random hill climbing
def random_climbing_hill(state, shared_data):
    i = 1
    min_tiles = state.bag.total()
    while True:
        current = state
        while not current.is_goal():
            best_successors = []
            successors = current.get_successors()
            if len(successors) == 0:
                if i % 50 == 0:
                    print("Trial # %d failed to find a solution." % i)
                i = i + 1
                if current.bag.total() < min_tiles:
                    min_tiles = current.bag.total()
                    shared_data[0] = current
                break
            min_dist = successors[0][1]
            for successor in successors:
                if successor[0].is_goal():
                    solution = successor[0]
                    return solution
                dist = successor[1]
                if dist < min_dist:
                    min_dist = dist
            for successor in successors:
                if successor[1] == min_dist:
                    best_successors.append(successor[0])
            random_idx = random.randint(0, len(best_successors) - 1)
            current = best_successors[random_idx]

# perform local beam search
def local_beam_search(state, k, shared_data):
    i = 1
    min_tiles = state.bag.total()
    while True:
        current = [state]
        while True:
            all_successors = []
            for current in current:
                successors = current.get_successors()
                for successor in successors:
                    if successor[0].is_goal():
                        solution = successor[0]
                        return solution
                    sum_dist = successor[0].sum_dist_non_zeros()
                    all_successors.append((successor[0], sum_dist))
            if current.bag.total() < min_tiles:
                min_tiles = current.bag.total()
                shared_data[0] = current

```

```

        if len(all_successors) == 0:
            if i % 50 == 0:
                print("Trial # %d failed to find a solution." % i)
            i = i + 1
            break
        all_successors.sort(key=lambda x: x[1])
        min_value = all_successors[0][1]
        min_successors = list(filter(lambda x: min_value in x,
all_successors))
        if len(all_successors) > k:
            currentts = []
            if len(min_successors) < k:
                for successor in all_successors[:k]:
                    currentts.append(successor[0])
            else:
                for j in range(k):
                    random_idx = random.randint(0, len(min_successors)
- 1)

                    currentts.append(min_successors[random_idx][0])
                    min_successors.remove(min_successors[random_idx])
        else:
            currentts = []
            for successor in all_successors:
                currentts.append(successor[0])

# perform local search algorithm to generate solution
def generate_solution(height, width, search, shared_data):
    if height == 1 or width == 1:
        print("Absolutely there is no solution for %d X %d grid" % (height,
width))
        return

    num_set = is_valid_grid(height, width)

    if num_set == 0:
        print("Absolutely there is no solution for %d X %d grid" % (height,
width))
        return

    # create a grid height x width
    grid = np.zeros((height, width))
    # create a bag of k set of tetrominos
    bag = Bag(num_set)

    startState = State(grid, bag)
    closest_sol = startState
    shared_data.append(closest_sol)
    if search == 0:
        solution = random_climbing_hill(startState, shared_data)
    if search == 1:
        solution = local_beam_search(startState, 5, shared_data)
    print("Found a solution!")
    plt.imshow(solution.grid, interpolation='none')
    plt.title("Solution")
    plt.show()

# create process for local search and add timeout on the function
def main():
    print("Packing Puzzle Solver\n")

```

```

while True:
    height = input("Please tell the height:")
    if height.isnumeric():
        if int(height) > 0:
            height = int(height)
            break
    print("Please type a positive integer\n")

while True:
    width = input("Please tell the width:")
    if width.isnumeric():
        if int(width) > 0:
            width = int(width)
            break
    print("Please type a positive integer\n")

print("\nLocal Search algorithm:")
print("0 - Random Hill Climbing")
print("1 - Local Beam Search")
algo = ["Random Hill Climbing", "Local Beam Search"]
while True:
    search = input("Please tell the search algorithm:")

    if search.isnumeric():
        if int(search) in range(len(algo)):
            search = int(search)
            break
    print("Please type a valid selection\n")

    print("You typed: height = %d, width = %d, algorithm = %s\n" % (height,
width, algo[search]))

    print("Start Solving...")

    mgr = multiprocessing.Manager()
    l = mgr.list()
    p = multiprocessing.Process(target=generate_solution, args=(height,
width, search, l))
    p.start()
    p.join(100)
    if p.is_alive():
        p.terminate()
        print("Time out! Fail to find a solution")
        plt.imshow(l[0].grid, interpolation='none')
        plt.title("Closest solution")
        plt.show()
        p.join()

if __name__ == '__main__':
    main()

```