Assignment 5 Report: Packing Puzzle

CSCE625, Spring 2022

Name: David van Wijk UIN: 932001896

---

1 Solution Approach

(a_0) Preamble Although we were asked to use local search explicitly in the assignment, I was inspired by the proof shown in [1] to see if I could determine if a packing existed, given only the height of the board H and the length of the board L. Using a graph theory coloring proof, it can be shown that a board will only have a solution if it's dimensions satisfy the following two conditions:

1. (H $\geq$ 3) and (L $\geq$ 3)
2. (H*L)/20 (k collections of pieces) is even

If we think of the board as an H*L checkerboard, with alternate light and dark squares, and the pieces are colored in this alternating way, we come to an interesting observation. It can be see that for all the tetrominoes except for the "T" shape, it is possible for each piece to be colored with precisely 2 light and 2 dark squares. The T shaped tetrominoe however, can only be colored with either 3 dark squares and 1 light, or the opposite - 3 light and 1 dark. This is illustrated in the figures below.

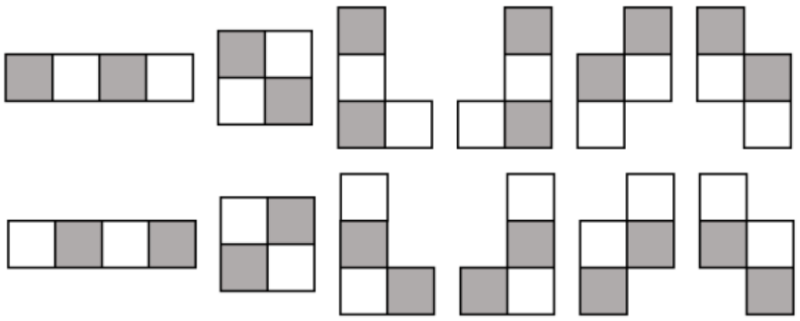

Figure 1: Pieces colored in checkerboard pattern, excluding T shaped piece [1]

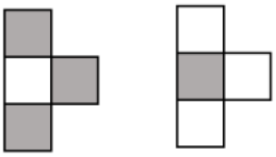

Figure 2: T shaped piece colored in checkerboard pattern [1]

When coloring the board in the checkerboard manner, the board will always have the same number of dark and light colored squares. Therefore, for a perfect packing solution to exist, we need to be able to have the same number of light pieces as dark pieces on in our set of puzzle pieces to fill the board exactly. Since in either configuration shown in Figure 2, the T shaped piece will have an odd number of light and dark pieces, the only way we can find a perfect packing for a given board, is if we have an even number of T shaped pieces. It follows then, that since a valid solution is only satisfied if we used exactly k collections of pieces, we have a valid packing solution for only even values of k. Therefore, to answer the prompt: "Is there a way to pack exactly k collections of tetrominoes into the rectangle?" we only need to satisfy the condition that k is even, and that both of the dimensions are greater than or equal to 3.

With all this said, I still implemented a local search algorithm to determine if a packing solution could be found, because if the prompt was changed to: "*How* can you pack exactly k collections of tetrominoes into the rectangle?" then the above method would get us nowhere. My code first determines if a packing solution can be found using the above criteria, displays this information to the user, and then proceeds in attempting to find the solution using local search.

(a) Representation Before diving into the details of my approach, I first wanted to explain the problem representation. As has been the case in many assignments throughout the course, selecting a good method to represent the problem is a very important step. I represented all of the possible rotations and reflections across the axis of symmetry of each of the tetrominoes as 4x4 matrices, populated by ones or zeros. For example, I would represent a T shape, with a single 90 degree rotation as:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The 4x4 matrix allows all the possible rotations and reflections of any of the pieces to be represented in a uniform format, which helps with generalizing the code. The only key point is that each piece should be contained in the 4x4 matrix as close to the top and left sides of the matrix as possible. This is because I will use the (1,1) position of the 4x4 matrices to describe the location of the pieces. Thus, all the possible orientations of the pieces can be represented with 19 4x4 matrices, and in my code I have kept these pieces separate to ensure that we are using k collections of pieces (where k is given by (H*L)/20) to fill the board. These 19 matrices are generated in the puzzlePieces.m function (subsection B.5).

The board is represented by an (H+3) by (L+3) matrix. The pieces are placed on the board according to their grid location, and thus given the 4x4 matrices and their location on the board, it is fairly straightforward to generate the board (subsection B.2). The reason for the additional 3 rows and columns in the board matrix is to allow for "overhang" where the pieces may be located at row H or column L, and thus part of the piece could be "hanging off" of the H by L section of the board. With this representation, the goal state will be all ones in the H by L section of the board, and 0's in the remainder of the board. Throughout the code and report I call this overhang section of the board the "gutter".

(b) Algorithm To solve the packing puzzle problem, I used a stochastic local beam search algorithm, with some slight modifications that will be highlighted throughout the report. I wrote everything from scratch including all the necessary subfunctions to deal with creating the board representation, generating the puzzle pieces, getting the successors, and evaluating the board.

The main script (subsection B.1) will prompt the user for the vertical board dimension, H and the horizontal dimension, L. Then, before doing anything, the script checks the conditions described above, and if H and L don't satisfy those criteria, the script will return "No solution possible". If we pass this check, then we will initialize 50 random beams, where each beam represents k collections of puzzle pieces. Using an evaluation function that will be described in paragraph (d) Heuristic, each collection of initially random puzzle pieces is scored. Within each collection, we will have k*5 pieces, which are represented by 4x4 matrices, with an associated (x,y) position corresponding to the location of that particular piece on the board, and two identifiers which encode which type of piece it is (i.e. T, O, S/Z etc.) and which version of that piece it is (i.e. rotated 90°, reflected, etc.). I also then compute the score that would result in a successful packing using the evaluation function, which is used in our end condition.

After the initial set of random beams are generated, we enter a while loop that will only terminate upon finding a board that has the same score as the end condition score calculated above (or upon timeout). Now, we finally begin the local beam search. For each beam, we iterate through each piece it contains, and will generate 5 successors for each piece, generated through actions which are described in paragraph (c) Action Space. Each successor to the piece will result in a new, slightly different board, which will be evaluated using the evaluation function, and the score will be stored along with the new modified collection. The terminating condition is checked here, and if any of the successors form a solution board, the while loop will exit, and the script will display "Solution found in (number) beam iterations", where (number)

is a count that keeps track of our iterations. This is done for all the pieces in all of the collections for all of the beams, which will generate (number of beams)*(5*k)*5 successors. Each successor here is k collections of pieces, with an associated score.

The successors are then sorted in ascending order, since a lower score indicates a better board. The next set of beams are selected by choosing a certain proportion of beams randomly from the successors, and the remainder of the beams are selected greedily. The proportion of randomly selected beams will decrease with the number of iterations, with it starting extremely high, at a proportion of .9, and decreasing by .01 with each iteration, with a minimum random proportion of .1. I found this to be crucial to the performance of the algorithm, since when I had a purely greedy beam selection, I quickly converged on one collection of pieces. Of course this method doesn't work so well if the number of beams is not fairly high, which is why I generally kept the number of beams between 20 and 100 when testing the algorithm. The process continues until we either achieve the goal state, or until the minimum score has remained the same for 20 iterations. If this is the case, we perform a random restart, populate our beams with entirely random collections, and start again. This process will run until we reach the goal state, or until we reach the timeout time, which can be specified in line 30 of the code in subsection B.1.

(c) Action Space The successors in our local search space will be generated using 5 possible actions for each puzzle piece in k collections for each beam. For detailed code, please see subsection B.4.

1. Moves the piece up on the board by a random step between 1 and k slots
2. Moves the piece down on the board by a random step between 1 and k slots
3. Moves the piece right on the board by a random step between 1 and k slots
4. Moves the piece left on the board by a random step between 1 and k slots
5. Pick the next version of the current piece, in a cyclic pattern

It should be noted that the moves will only be executed if they do not take the piece out of the board (i.e. the new piece location calculated by the action must be within the H x L matrix making up the valid board). To expand further on action 5, we examine the piece we are generating successors for, and from an ordered list of possible versions of that piece, we select the next in the list, or loop back to the first possible version of the piece. For the T piece, we have 4 versions of this piece, and thus if the piece we are generating successors for is the 3rd version of the T shape, one of the successors of that piece will be the next version of that piece, without changing the location of the piece. In this way we are able to capture all the possible types of movements, rotations and reflections as actions. The reason for making the step of the movement of pieces range from 1 to k is so that the successors can be slightly more diverse, which helped with addressing local minima. I also found it useful for the range of steps to scale with board size for greater generality.

(d) Heuristic The heuristic is a cost function that is used to score each k collection of pieces that make up a valid board. Here, the lower the score, the better the board, or closer to the solution the board is. There are four components to the score which are enumerated below.

1. (c_1): Cost associated with overlapping pieces which is calculated by the maximum number on the board, times a tunable scalar value
2. (c_2): Cost associated with gaps in the board (excluding the gutter). Sums up all the zeros on the board, times some tunable scalar value
3. (c_3): Cost associated with overlap in the gutter. This cost starts off very low and gradually increases with the number of iterations. The idea here is to first allow pieces to move around more freely with some overhang but slowly converge onto the tightly packed board
4. (c_4): Reward associated with filled rows, which is largest for the lowest row. The purpose is to incentivize filling from the bottom row first, which I found to be helpful. Since this is a reward, it subtracts from the cost, as can be seen in Equation 1

$$c_{total} = c_1 + c_2 + c_3 - c_4 \tag{1}$$

It took quite a lot of trial and error to tune the scalars associated with each of the score components, and I tried to do what I could to reduce the number of tunable parameters, in an attempt to make my algorithm as general as possible. The specific values for the scalars associated with each component can be found in subsection B.3.

2 Examples of Code

In Figure 3, we can see a few examples of the code being run where a solution is found just using local search. We can also see in Figure 4 we display that a solution was found, but we're unable to reach the solution within 5 minutes using local search. In Figure 5 we are immediately able to discern that there is no solution using the criterion outlined in section 1. I was able to directly find solutions for up to k = 3, but exceeding this was very rare. I worked a lot on tweaking various parts of the functions to try to get out of local minima, but these local minima were extremely persistent. I will discuss this more in the next section. Technically though, I am able to answer the prompt for any H by L board.

```
>> assignment5
Please enter the vertical dimension, H: 4
Please enter the horizontal dimension, L: 10
Beam is stuck at score of 19.2... randomly restarting
Beam is stuck at score of 17.8... randomly restarting
Beam is stuck at score of 15... randomly restarting
Beam is stuck at score of 11.7... randomly restarting
Beam is stuck at score of 19.95... randomly restarting
Beam is stuck at score of 15.15... randomly restarting
Beam is stuck at score of 17.55... randomly restarting
Beam is stuck at score of 15.75... randomly restarting
Beam is stuck at score of 17.15... randomly restarting
Beam is stuck at score of 18... randomly restarting
Beam is stuck at score of 18... randomly restarting
Beam is stuck at score of 12.15... randomly restarting
Beam is stuck at score of 15.15... randomly restarting
Beam is stuck at score of 14.15... randomly restarting
Beam is stuck at score of 15.15... randomly restarting
Solution found in 62 beam iterations
```

```
>> assignment5
Please enter the vertical dimension, H: 5
Please enter the horizontal dimension, L: 8
Beam is stuck at score of 11.9... randomly restarting
Beam is stuck at score of 14.55... randomly restarting
Beam is stuck at score of 12.15... randomly restarting
Beam is stuck at score of 14.75... randomly restarting
Beam is stuck at score of 15... randomly restarting
Beam is stuck at score of 14.75... randomly restarting
Beam is stuck at score of 11.55... randomly restarting
Beam is stuck at score of 12.7... randomly restarting
Beam is stuck at score of 15.15... randomly restarting
Beam is stuck at score of 14.2... randomly restarting
Beam is stuck at score of 9.05... randomly restarting
Beam is stuck at score of 12.55... randomly restarting
Beam is stuck at score of 8.9... randomly restarting
Beam is stuck at score of 8.5... randomly restarting
Beam is stuck at score of 11.3... randomly restarting
Beam is stuck at score of 12.15... randomly restarting
Beam is stuck at score of 15.25... randomly restarting
Beam is stuck at score of 11.75... randomly restarting
Beam is stuck at score of 10.95... randomly restarting
Beam is stuck at score of 11.3... randomly restarting
Beam is stuck at score of 13.3... randomly restarting
Beam is stuck at score of 15... randomly restarting
Beam is stuck at score of 11.3... randomly restarting
Beam is stuck at score of 11.3... randomly restarting
Beam is stuck at score of 12.75... randomly restarting
Solution found in 104 beam iterations
```

Figure 3: Solution found using beam search for (4 by 10) and (5 by 8)

```
>> assignment5
Please enter the vertical dimension, H: 28
Please enter the horizontal dimension, L: 70
A solution exists! Trying to find it via beam search...
A solution is possible for this board! However, local search timed out after 5 (min)
```

Figure 4: Solution found using criterion outlined in section 1

```
>> assignment5
Please enter the vertical dimension, H: 10
Please enter the horizontal dimension, L: 14
No solution possible
```

```
>> assignment5
Please enter the vertical dimension, H: 15
Please enter the horizontal dimension, L: 28
No solution possible
```

Figure 5: No solution possible

3 Running the Code

Running the code is very straightforward. If all of the required subfunctions are in the same directory as the main script, all of which can be located in Appendix B, then the user must simply run assignment5.m,

and enter the desired H and L. If the aspect ratio is low enough that we're able to find a solution, or even if we don't find a solution, the board can be visualized by the user by entering the following command into the command prompt:

```
[board] = createBoard(k_beam_collections{1,1},H,L)
```

Doing this will display lowest scored (i.e. best) collection on the (H+3) by (L+3) matrix with all the pieces placed on the board. A solution will of course be a matrix of 1's in the H by L area of the matrix, and 0's in the remainder of the matrix elements. Additionally, if you would like to know which pieces are placed where, navigate to the workspace, and click on the cell array named k_beam_collections, where the first element will be the "best" collection of pieces found. The cell array structure is documented throughout the code.

4 General Notes

There were a number of things that I had to tune within my code to obtain some valid solutions. Initially, I had a purely greedy search. However, I found that the beams converged on a single set of k collections very quickly, and in order to prevent this I had to add stochasticity. I also had to tune the heuristic quite a bit. Initially I had fixed scalars associated with the costs, but I found that adding dependence on a "time" of some sort, like a counter associated with iterations was helpful for situations like the gutter, where at first it is favorable to have pieces move around freely, but once we try to converge on a solution, (ie. later in the beam search) we want to increase the gutter cost. I also added random restarts which I found to be useful for smaller k collections, since sometimes the initial random configuration makes it very difficult to find a solution, and the best thing may be to just restart all of the beams randomly from scratch. Additionally, I had to tune the number of beams, and this was also quite important in finding a solution. I later also added stochasticity to how the successors were generated because this too helped get out of local minima. I also had some initial results using backtracking, but since Dr. Shell specified we use local search, I didn't use any backtracking in an attempt to solve this problem without keeping track of a "path" of any sort. In theory this would also mean that my solution (if tuned better) would be able to handle much larger board sizes that would crush backtracking solutions such as DFS. Overall, I think that my solution suffers from having too many tunable parameters. I think this makes it more prone to local minima, which is a downside to the solution.

5 Acknowledgments

For this assignment I discussed high-level concepts and methods to approach this problem with Matthew Kocmoud and Moeez Akmal. I also discussed the proof for finding no solutions for certain values of k with Hannah Lehman.

A References

References

[1] Talwalkar, Presh (2018). *Can You Solve The Tetris Riddle?*. Retrieved from `shorturl.at/tDHNX`

B Matlab Code

B.1 assignment5.m

```
%% Programming Assignment 5: Packing Puzzle
% CSCE625: Artificial Intelligence
% David van Wijk

%% Puzzle Piece Representation
% Represent pieces and all possible rotations using 4x4 matrices using 1'
   s
% and 0's. Take rotations and reflections into account

[T_blocks,I_blocks,O_blocks,J_L_blocks,S_Z_blocks] = puzzlePieces();
All_blocks = {T_blocks,I_blocks,O_blocks,J_L_blocks,S_Z_blocks};

%% Local Beam Search Algorithm
% Local search is not required to solve the problem - we can show that a
% packing solution exists if [k is even] AND [(H >= 3) AND (L >= 3)]
%
% Use local beam search to determine if pieces can be packed within given
% dimensions. Actions are for each block type, can select the type of
   variation
% of block (ie. rotated, translated etc) or the (x,y) location on the

H = input('Please enter the vertical dimension, H: ');
L = input('Please enter the horizontal dimension, L: ');
k = (H*L)/20;
k_beam = 50;
pseudo_time = 0;

% How many times in a row we can repeat min score before random restart
max_repeated_min_count = 30;

% Timeout in seconds
maxTime = 60*5;
tic

base_board = zeros(H+3,L+3);
base_board(1:H,1:L)=ones(H,L);
[end_score] = evalBoard(base_board,H,L,pseudo_time);

% We know there is a solution if it enters this loop
if (mod(k,2) == 0) && ((H >= 3) && (L >= 3))
    disp('A solution exists! Trying to find it via beam search...')
    % Intialize k_beam collections randomly

    % Each collection represents k sets of puzzle pieces (ie. k*5 pieces)
    % Each collection has a score associated with it (2nd row)
    k_beam_collections = cell(2,k_beam);
    for z = 1:k_beam
        % Contains a bunch of pieces and their (x,y) grid locations
        collection = {};
        % For k collections of pieces
```

```
    for i = 1:k
        % For each type of piece
        for j = 1:size(All_blocks,2)
            % Pick location on the grid that each piece will have
            grid_location = [randi([1 L],1,1),randi([1 H],1,1)];
            % Pick orientation for each piece
            num_orientations = size(All_blocks{1,j},2);
            orientation_idx = randi([1 num_orientations],1,1);
            piece = All_blocks{1,j}{1,orientation_idx};
            piece_idx = j;
            rand_piece_mtrx = {grid_location,piece,piece_idx,
               orientation_idx};
            collection{end+1} = rand_piece_mtrx;
        end
    end
    k_beam_collections{1,z} = collection;
    [board] = createBoard(collection,H,L);
    [score] = evalBoard(board,H,L,pseudo_time);
    k_beam_collections{2,z} = score;
end

% Perform local beam search
% We will generate k_beam*(5*k)*5 successors in each iteration

totalSuccessors = cell(2,k_beam*(5*k)*5);
doRun = 1;
last_best_score = 999;
count_repeated_sc = 0;

while doRun == 1 && toc < maxTime

    % For each beam
    for z = 1:k_beam

        % Get 5 successors for each piece in each collection
        for i = 1:size(k_beam_collections{1,z},2)

            collection = k_beam_collections{1,z};
            scores = zeros(1,5);

            % Get the piece information
            piece2Change = collection{1,i};
            [successors] = getSuccessors(piece2Change,All_blocks,H,L)
               ;
            successor_collections = cell(1,5);
            for j = 1:size(successors,2)
                collection{1,i} = successors{1,j};
                successor_collections{1,j} = collection;
                [board] = createBoard(collection,H,L);
                [score] = evalBoard(board,H,L,pseudo_time);

                % End condition!
                if score == end_score
```

```
                disp(['Solution found in ' num2str(pseudo_time) '
                    beam iterations'])
                doRun = 0;
                break
            end

            scores(1,j) = score;
        end

        piece_idx = 5*k*(z-1)*5 + (5*i)-4;
        piece_idx_next = 5*k*(z-1)*5 + 5*i;

        totalSuccessors(1,piece_idx:piece_idx_next) =
           successor_collections;
        totalSuccessors(2,piece_idx:piece_idx_next) = num2cell(
           scores);
    end

end

% STOCHASTIC BEAM SEARCH
% chooses k successors at random, with the probability of
   choosing
% a given successor being an increasing function of its value

% Random selection by percentage -- not scaled
rand_prop = .9-(.01)*pseudo_time;
if rand_prop < .1
    rand_prop = .1;
end
rand_beams = floor(k_beam*rand_prop);
greedy_beams = k_beam-rand_beams;

scores = cell2mat(totalSuccessors(2,:));
[sorted_scores,sorted_idxs] = sort(scores,'ascend');

% Get k_beam*(1-rand_prop) best collections
selected_greedy_scores = 9999*ones(1,greedy_beams);
for i = 1:greedy_beams
    for j = 1:size(sorted_scores,2)
        idx = sorted_idxs(j);
        value = sorted_scores(j);
        if ~ismember(value,selected_greedy_scores)
            break
        end
    end
    selected_greedy_scores(1,i) = value;
    k_beam_collections{1,i} = totalSuccessors{1,idx};
    k_beam_collections{2,i} = value;
end
for i = (greedy_beams+1):k_beam
    idx = randi([1 size(totalSuccessors,2)],1,1);
    k_beam_collections{1,i} = totalSuccessors{1,idx};
```

```
        k_beam_collections{2,i} = totalSuccessors{2,idx};
    end

    % For testing only:
    [board] = createBoard(k_beam_collections{1,1},H,L);

    min_score = min(scores);
    if min_score == last_best_score
        count_repeated_sc = count_repeated_sc + 1;
    else
        count_repeated_sc = 0;
        last_best_score = min_score;
    end

    % RANDOM RESTART -- get k_beam new random beams
    if count_repeated_sc > max_repeated_min_count
        [board] = createBoard(k_beam_collections{1,1},H,L);
        disp(['Beam is stuck at score of ' num2str(min_score) '...
           randomly restarting'])
        for z = 1:k_beam
            % Contains a bunch of pieces and their (x,y) grid
               locations
            collection = {};
            % For k collections of pieces
            for i = 1:k
                % For each type of piece
                for j = 1:size(All_blocks,2)
                    % Pick location on the grid that each piece will
                       have
                    grid_location = [randi([1 L],1,1),randi([1 H
                       ],1,1)];
                    % Pick orientation for each piece
                    num_orientations = size(All_blocks{1,j},2);
                    orientation_idx = randi([1 num_orientations],1,1)
                       ;
                    piece = All_blocks{1,j}{1,orientation_idx};
                    piece_idx = j;
                    rand_piece_mtrx = {grid_location,piece,piece_idx,
                       orientation_idx};
                    collection{end+1} = rand_piece_mtrx;
                end
            end
            k_beam_collections{1,z} = collection;
            [board] = createBoard(collection,H,L);
            [score] = evalBoard(board,H,L,pseudo_time);
            k_beam_collections{2,z} = score;
        end
        pseudo_time = 0;
    end

    % Keep a pseudo-time (ie. iteration count)
    pseudo_time = pseudo_time + 1;
end
```

```
else
    disp("No solution possible")
end

if toc > maxTime
    disp(['A solution is possible for this board! However, local search
       timed out after ' int2str(toc/60) ' (min)'])
end
```

B.2 createBoard.m

```
function [board] = createBoard(pieces,H,L)
% createBoard: Given k collections of pieces and (x,y) locations, will
   generate a
% board which will be a (h+3)-by-(l+3) matrix
%
%   INPUTS
%     pieces          (1-by-k*5) cell array, with each cell containing a
%                     (1-by-2) cell with (x,y) grid location and (4-by-4)
%                     matrix for piece representation
%
%   OUTPUTS
%     board           (h+3)-by-(l+3) matrix representing orientation of k
%                       collection of puzzle pieces on the board
   dimensions, plus
%                       additional room for the "gutter"

board = zeros(H+3,L+3);

for i = 1:size(pieces,2)
    piece = pieces{1,i}{1,2};
    grid_location = pieces{1,i}{1,1};

    board(grid_location(2):grid_location(2)+3, grid_location(1):
       grid_location(1)+3)...
        = board(grid_location(2):grid_location(2)+3, grid_location(1):
           grid_location(1)+3) + piece;
end

end
```

B.3 evalBoard.m

```
function [score] = evalBoard(board,H,L,pseudo_time)
% evalBoard: Score the inputted board made up of k collections of puzzle
% pieces
%
%   INPUTS
%       board           (h+3)-by-(l+3) matrix representing orientation of
    k
%                       collection of puzzle pieces on the board
   dimensions, plus
%                       additional room for the "gutter"
%       H               y dimension
%       L               x dimension
```

```
%       pseudo_time     keeps track of "time"
%
%   OUTPUTS
%       score           Scalar value evaluating board based on heuristic
%

%% Max overlap in main space
scalar_main = 1;
maxOverlap = sum(max(board(1:H,1:L)));
overlap_score = scalar_main*(maxOverlap);

%% Zeros in the main board are gaps which are bad
scalar_zeros = 2;
numZeros_main = length(find(board(1:H,1:L) == 0));
zeros_score = scalar_zeros*numZeros_main;

%% Overlap in gutter
scalar_gutter = .5;
maxGutter = sum([max(max(board(H+1:H+3,1:L))),max(max(board(1:H+3,L+1:L
   +3)))]);
gutter_score = (scalar_gutter*pseudo_time)*(maxGutter)^2;
% gutter_score = (scalar_gutter)*(maxGutter)^2;

%% Reward (ie. negative score) for filled rows
filled_score = 0;
filled_scalar = .05;
full_row = ones(1,L);
for i = 1:H
    if board(i,1:L) == full_row
        filled_score = filled_score + i^2*filled_scalar;
    end
end

%% Sum up and return total score

score = overlap_score + zeros_score + gutter_score - filled_score;

end
```

B.4 getSuccessors.m

```
function [successors] = getSuccessors(piece,All_blocks,H,L)
% getSuccessors: Will return valid sucessors given a piece and location
%
%   INPUTS
%     piece           (1-by-4) cell with (x,y) grid location and (4-by-4)
%                     matrix for piece representation, idx corresponding
   to
%                     the type of piece (ie. T,O,etc) and orientation idx
%                     corresponding to the orientation of type of piece
%
%     All_blocks
%
%     H
```

```
%
%     L
%
%   OUTPUTS
%                       (1-by-5) cell array containing 5 new sucessors of
   the
%                       inputted piece, with each cell containing a (1-by
   -4)
%                       cell with (x,y) grid location and (4-by-4)
%                       matrix for piece representation, idx corresponding
   to
%                       the type of piece (ie. T,O,etc) and orientation idx
%                       corresponding to the orientation of type of piece
%
% Performs check to see what x, y translation moves are valid
% (i.e. moves on edges of board are restricted)

grid_location = piece{1,1};
pieceIdx = piece{1,3};
orientationIdx = piece{1,4};

successors = cell(1,5);

% Change the grid location only
% Jump sizes based on grid size
max_step = (H*L)/20;
step = [1 max_step];
gridChanges = [0 randi(step,1,1);randi(step,1,1) 0;0 -randi(step,1,1);-
   randi(step,1,1) 0];

for i = 1:4
    successors{1,i} = piece;
    newGridLoc = grid_location + gridChanges(i,:);
    % Check x
    if newGridLoc(1) <= 0
        newGridLoc(1) = 1;
    elseif newGridLoc(1) > L
        newGridLoc(1) = L;
    end
    % Check y
    if newGridLoc(2) <= 0
        newGridLoc(2) = 1;
    elseif newGridLoc(2) > H
        newGridLoc(2) = H;
    end

    successors{1,i}{1,1} = newGridLoc;
end

% Cycle through possible orientations
successors{1,5} = piece;
newOrientationIdx = orientationIdx + 1;
```

```
if size(All_blocks{1,pieceIdx},2) < newOrientationIdx
    newOrientationIdx = 1;
end
successors{1,5}{1,2} = All_blocks{1,pieceIdx}{1,newOrientationIdx};

end
```

B.5 puzzlePieces.m

```
function [T_blocks,I_blocks,O_blocks,J_L_blocks,S_Z_blocks] =
   puzzlePieces()
% puzzlePieces: Will output all the different types of puzzle pieces
% represented as 1's and 0's, taking account all possible reflections or
% rotations
%
%   INPUTS
%
%   OUTPUTS
%       T_blocks
%       I_blocks
%       O_blocks
%       J_L_blocks
%       S_Z_blocks

T_base = [1 1 1 0;
    0 1 0 0];

T_blocks = {};
for i = 1:4
    base = zeros(4,4);
    A = rot90(T_base,i);
    base(1:size(A,1),1:size(A,2)) = A;
    if i == 1
        % Row
        a = base;
        a(1,:)=[];
        base = [a; zeros(1,4)];
    end
    if i == 2
        % Column
        a = base;
        a(:,1)=[];
        base = [a zeros(4,1)];
    end
    T_blocks{i} = base;
end

% I block
I_base = [1 1 1 1;
    0 0 0 0];

I_blocks = {};
for i = 1:2
    base = zeros(4,4);
```

```
    A = rot90(I_base,i);
    base(1:size(A,1),1:size(A,2)) = A;
    if i == 2
        % Row
        a = base;
        a(1,:)=[];
        base = [a; zeros(1,4)];
    end
    I_blocks{i} = base;
end

% O block
O_blocks = {[1 1 0 0;
    1 1 0 0
    0 0 0 0
    0 0 0 0]};

% J/L block
J_base = [1 1 1 0;
    0 0 1 0];

L_base = [0 0 1 0;
    1 1 1 0];

J_L_blocks = {};
for i = 1:4
    base = zeros(4,4);
    A = rot90(J_base,i);
    base(1:size(A,1),1:size(A,2)) = A;
    if i == 1
        % Row
        a = base;
        a(1,:)=[];
        base = [a; zeros(1,4)];
    end
    if i == 2
        % Column
        a = base;
        a(:,1)=[];
        base = [a zeros(4,1)];
    end
    J_L_blocks{i} = base;
end
for i = 1:4
    base = zeros(4,4);
    A = rot90(L_base,i);
    base(1:size(A,1),1:size(A,2)) = A;
    if i == 1
        % Row
        a = base;
        a(1,:)=[];
        base = [a; zeros(1,4)];
    end
```

```
    if i == 2
        % Column
        a = base;
        a(:,1)=[];
        base = [a zeros(4,1)];
    end
    J_L_blocks{end+1} = base;
end

% S/Z block
Z_base = [1 1 0 0;
    0 1 1 0];

S_base = [0 1 1 0;
    1 1 0 0];

S_Z_blocks = {};

for i = 1:2
    base = zeros(4,4);
    A = rot90(Z_base,i);
    base(1:size(A,1),1:size(A,2)) = A;
    if i == 1
        % Row
        a = base;
        a(1,:)=[];
        base = [a; zeros(1,4)];
    end
    if i == 2
        % Column
        a = base;
        a(:,1)=[];
        base = [a zeros(4,1)];
    end
    S_Z_blocks{i} = base;
end
for i = 1:2
    base = zeros(4,4);
    A = rot90(S_base,i);
    base(1:size(A,1),1:size(A,2)) = A;
    if i == 1
        % Row
        a = base;
        a(1,:)=[];
        base = [a; zeros(1,4)];
    end
    if i == 2
        % Column
        a = base;
        a(:,1)=[];
        base = [a zeros(4,1)];
    end
    S_Z_blocks{end+1} = base;
```

```
end

end
```