# Project 1-1, Pentominoes

GROUP 16: Alex, Ali Tarik, Elyas, Gabriel, Ranjani and Husam

# Changes on the brute force code

# Search.java

## User can input desired pentominoes' formats:

### Original

**On search class:**

```
public static final char[] input = { 'W', 'Y', 'I', 'T', 'Z', 'L'};
```

### Modified

**On search class:**

```
static char[] input;
```

**On the main: (added)**

```
Scanner scanner = new Scanner(System.in);
    System.out.println("Please enter the number of pieces you want to add: ");
    int NumberOfPieces = scanner.nextInt();
    input = new char[NumberOfPieces];
    for (int a = 0; a < NumberOfPieces ; a++ ) {
      System.out.println("Please enter piece number " + (a+1));
            input[a] = scanner.next().toUpperCase().charAt(0);
```

# Checking whether the complete field is filled

Original

Modified

```
outerloop:
      for (int a = 0; a < horizontalGridSize ; a++) {
        for (int b = 0; b < verticalGridSize ; b++ ) {
          if (field[a][b] == -1) {
            solutionFound = true;
            break outerloop;
          }
        }
      }
```

# Pentominos.csv

Database duplicates were removed (sample case on X)

Original

0 0 3 3 0 1 0 1 1 1 0 1 0

0 1 3 3 0 1 0 1 1 1 0 1 0

0 2 3 3 0 1 0 1 1 1 0 1 0

0 3 3 3 0 1 0 1 1 1 0 1 0

0 4 3 3 0 1 0 1 1 1 0 1 0

0 5 3 3 0 1 0 1 1 1 0 1 0

0 6 3 3 0 1 0 1 1 1 0 1 0

0 7 3 3 0 1 0 1 1 1 0 1 0

Modified

0 0 3 3 0 1 0 1 1 1 0 1 0

# Branching Algorithm

Depth-first search, backtracking and heuristic

# ADD PIECES AFTER CHECKING IF THE MOVE IS LEGAL

```
private static int[][] addPiece(int[][] grid, int[][] piece, int id, int x, int y) {
  int[][] newGrid = grid;
  int counter = 0;
  boolean noOverlap = true;

  for(int i=0; i<piece.length; i++) {
    for(int k=0; k<piece[i].length; k++) {
      if(piece[i][k] == 1)
        if(grid[x+i][y+k] == -1)
          counter++;
    }
  }
  if(counter == 5)
    noOverlap = true;
  else
    noOverlap = false;
  if(noOverlap == true) {
    for(int i=0; i<piece.length; i++) {
      for(int k=0; k<piece[i].length; k++) {
        if(piece[i][k] == 1)
          newGrid[x+i][y+k] = id;
      }
    }
    return newGrid;
  }
  return null;
}
```

# IF FINDS THE SMALLEST AREA EMPTY AVAILABLE

```
private static int smallestHole(int[][] grid) {
   int smallestArea = (horizontalGridSize * verticalGridSize) + 1;
   int emptySpots = 0;

   for(int[] row: grid) {
     for(int pos: row) {
       if(pos == -1) {
         emptySpots++;
       }
     }
   }
   int[][] gridSearch = new int[grid.length][grid[0].length];
   for(int i=0; i<grid.length; i++) {
     for(int k=0; k<grid[0].length; k++) {
       gridSearch[i][k] = grid[i][k];
     }
   }
```

```
int count = 0;
   while(count != emptySpots) {
     for(int i=0; i<gridSearch.length; i++) {
       for(int k=0; k<gridSearch[0].length; k++) {
         if(gridSearch[i][k] == -1) {
           int area = emptyArea(gridSearch, i, k);
           count += area;
           smallestArea = Math.min(smallestArea, area);
         }
       }
     }
   }
   count = 0;
   return smallestArea;
 }
```

# USING THE FLOOD FILL ALGORITHM WE GET THE SIZE OF THE EMPTY AREA AT (x,y)

```
private static int emptyArea(int[][] grid, int x, int y) {
  int area = 0;

  if(grid[x][y] == -1) {
    grid[x][y] = -2;
    area++;
  }
  if(x!=0)
    if(grid[x-1][y] == -1) {
      area += emptyArea(grid, x-1, y);
    }
  if(y!=0)
    if(grid[x][y-1] == -1) {
      area += emptyArea(grid, x, y-1);
    }
    if(y != grid[0].length-1)
        if(grid[x][y+1] == -1){
            area += emptyArea(grid, x, y+1);
        }
    if(x != grid.length-1)
        if(grid[x+1][y] == -1){
            area += emptyArea(grid, x+1, y);
        }
    return area;
}
```

THIS METHOD GENERATES ALL POSSIBLE PERMUTATIONS OF A PIECE ON THIS GRID

```
private static ArrayList<int[][]> genPossib(int[][] grid, int id){
   ArrayList<int[][]> positions = new ArrayList<int[][]>();
   for(int i=0; i<PentominoDatabase.data[id].length; i++) {
     int[][] piece =  PentominoDatabase.data[id][i];


     for(int k=0; k<grid.length - piece.length; k++) {
       for(int j=0; j<grid[0].length - piece[0].length; j++) {
         int[][] gridCopy = new int[grid.length][grid[0].length];
         for(int z = 0; z < grid.length; z++){
             for (int s = 0; s < grid[0].length; s++){
                 gridCopy[z][s] = grid[z][s];
             }
         }
         if(addPiece(gridCopy, piece, id, k, j) != null) {
           positions.add(addPiece(gridCopy, piece, id, k, j));
         }
       }
     }
   }
   return positions;
 }
```

THIS METHOD ELIMINATES ILLEGAL MOVES AND GOES BACK TO THE PREVIOUS RECURSIVE LAYER TO CHECK FOR ANOTHER MOVE

```
  private static boolean eliminateIllegal(int[][] grid, int layer) {
  Random randGen = new Random();
  int nextPent = characterToID(input[layer]);
  ArrayList<int[][]> positions = genPossib(grid, layer);
  ArrayList<int[][]> legalPos = new ArrayList<int[][]>();
  for(int[][] board: positions) {
    if(smallestHole(grid) == (horizontalGridSize*verticalGridSize) +1) {
      ui.setState(board);
      return true;
    }
    else if(smallestHole(grid) >= 5 && smallestHole(grid)%5 == 0) {
      legalPos.add(board);
    }
  }
  int newLayer = layer +1;
  while(!legalPos.isEmpty()) {
    int randSelected = randGen.nextInt(legalPos.size());
    int[][] nextGrid = legalPos.get(randSelected);
    legalPos.remove(randSelected);

    if(eliminateIllegal(nextGrid, newLayer)) {
      return true;
    }
  }
  return false;
}
```