RECITATION 6: A high level view of the 8-puzzle.
17 March 2003

You should be able to download the WinZip file containing all the files and save it to some directory folder on your own disk. If you double click the file (and WinZip is on your machine), you'll get a WinZip archive window with a bunch of java files. Click on Extract to uncompress and separate the files. Then double clicking on the file with the .vjp ending should open MS J++ with the project in place.

The PTile class

The PTile class describes one sliding tile in the 8-puzzle. Variables of type PTile know where they are, because they keep track of their position using integer variables xpos and ypos. They also remember where they belong in the solved 8-puzzle, using the integer variables truex and truey. These variables are **private**, which means that they can only be accessed directly by methods inside the PTile class.

```
/* class defining a tile on the 8-puzzle board */
public class PTile
{
	/* where the tile is */
	private int xpos;
	private int ypos;

	/* where the tile should be */
	private int truex;
	private int truey;
```

In the main routine, where we actually play the 8-puzzle game using the PTile class, we'll need to know what these values are. This means that we need 4 routines, called GetXPos, GetYPos, GetTrueX, and GetTrueY that look up these properties for a particular tile variable; they're all similar, so only GetXPos is shown here.

```
	/* get the x position of a tile */
	public int GetXPos()
	{
		return xpos;
	}
```

Since each tile knows where it is and where it belongs, it can tell whether it's in its proper place. The function that decides this is:

```
	/* see if the tiles are in place */
	public boolean InPlace()
	{
		/* if the tile's current position matches its final position, it's right */
```

```
            if (xpos == truex && ypos == truey)
                return true;
            else
                return false;
        }
```

A tile can also use this information to calculate how far off from its proper place it is. This routine should be helpful when you write your board scoring routine. Notice that we have to call mathematical functions through the Math class, here.

```
        /* calculate the distance a tile is out of place from its proper position */
        public int ManhattanDist()
        {
            return (Math.abs(truex-xpos) + Math.abs(truey-ypos));
        }
```

Finally, we need a way to change the position of a tile when we move it. (This method is a bit silly, since the true x and true y coordinates don't really need to be reset every time.)

```
        /* set positional data for a particular tile */
        public void Set(int x, int y, int tx, int ty)
        {
            xpos = x;
            ypos = y;
            truex = tx;
            truey = ty;
        }
}
```

That's the PTile class.

<u>The PuzzleBoard class</u>

Here's the PuzzleBoard class, which contains PTiles. PuzzleBoards need to keep track of the tiles, and in this case, the board does that in two ways: by tile number and by position in the 8-puzzle. We use an array of PTiles to keep track of where each tile is. Once we make a new Puzzleboard, we can set tiles[0] to a PTile variable for the blank space, and tiles[1] to a PTile variable for the tile marked 1, …, and tiles[8] to a PTile variable for the tile marked 8. So tiles is a 1 x 9 array that uses the tile number to index the space and all the tiles. Then there's a variable called score, for the tiles-out-of-place measurement that we use to identify good boards. There's a variable called depth, which remembers how far down the tree the board is. At most 4 moves are possible for a board, since a blank in the middle allows you to move each of 3 tiles either left, right, up, or down. The array tilemoves stores (for each of these 4 possible move directions) what tile to move, and -1 if the move isn't possible. The array xmoves stores +1 or –1 for any possible move left or right (and stores 0 for up or down moves). The array ymoves stores +1 or –1 for any

possible move down or up (and stores 0 for left or right moves). Taken together, these three arrays define the moves you can make for a particular PuzzleBoard. Finally, there is a two-dimensional 3 x 3 array called matrix, which keeps track of which tile number is at which position in the board. Since we have this information in PTile, it may seem redundant to store it twice, but this makes it fast to find out what tile is at a certain position.

```
import java.io.*;

/* this class implements the entire 3 x 3 array of the 8-puzzle */
public class PuzzleBoard
{
        /* array of tiles, sorted by the number on the tile (0 is the blank space) */
        PTile [] tiles;

        /* how far each tile is from where it belongs, summed for all tiles
                (except the blank) */
        int score = 0;

        /* depth of this 8-puzzle configuration in the search tree, or how many moves to
                this configuration from the start */
        int depth = 0;

        /* 3 parallel arrays to keep track of possible moves (max of 4, may be less) we
                can make from this board  */
        int [] tilemoves;       /* which tile can move */
        int [] xmoves;          /* tile can move left or right by one space */
        int [] ymoves;          /* tile can move up or down by one space */

        /* 2-dimensional array to keep track of what tile is at each position on the 8-
                puzzle */
        int [][] matrix;
```

When we want a new PuzzleBoard, we have to request memory for all these arrays. So it's important to write a constructor to take care of that. The constructor gets memory to hold 9 tiles in an array, and adds 9 blank tiles to the array. It also asks for memory to hold the three arrays xmoves, ymoves, and tilemoves, which define the moves the board is allowed to do. Finally, it asks for memory for the 3 x 3 array defining which tile is at which position,

```
        /* construct a new variable of type PuzzleBoard, from scratch */
        public PuzzleBoard()
        {
                int k;
                /* reserve memory for the array of tiles */
                tiles = new PTile[9];
```

```
        for (k = 0; k < 9; k ++)
        {
                /* reserve memory for each Tile */
                tiles[k] = new PTile();
        }
        /* reserve memory for the 3 arrays that remember possible moves */
        xmoves = new int[4];
        ymoves = new int[4];
        tilemoves = new int[4];

        /* reserve memory for the 2-dimensional array of tile positions */
        matrix = new int[3][3];
}
```

You can also make a new PuzzleBoard type variable by copying from an old one (this way, to get a new board, we copy the old board over and then move a tile in the copy). The code still has to reserve memory, but now it also sets its variables to the same values as the board it's copying. Notice that this is our second constructor!

```
/* construct a new variable of type PuzzleBoard by copying an existing one */
public PuzzleBoard(PuzzleBoard p)
{
        int i, j, k;
        /* reserve memory for the 3 arrays that remember possible moves */
        tiles = new PTile[9];
        for (k = 0; k < 9; k ++)
        {
                tiles[k] = new PTile();
                /* now copy the tile variables over to the new board */
                tiles[k].Set(p.tiles[k].GetXPos(), p.tiles[k].GetYPos(),
                        p.tiles[k].GetTrueX(), p.tiles[k].GetTrueY());
        }
        /* reserve memory for the 2-dimensional array in the new board */
        matrix = new int[3][3];
        for (i = 0; i < 3; i++)
                for (j = 0; j < 3; j++)
                        /* and copy the entries over from the old board */
                        matrix[i][j] = p.matrix[i][j];
        /* reserve memory for the 3 arrays that remember possible moves */
        xmoves = new int[4];
        ymoves = new int[4];
        tilemoves = new int[4];
        /* copy the allowed move data over from the other board */
        for (i = 0; i < 4; i++)
        {
                xmoves[i] = p.xmoves[i];
```

```
                ymoves[i] = p.ymoves[i];
                tilemoves[i] = p.tilemoves[i];
        }
}
```

How does Java know the difference? Look at the top line of each method. If we create a new PuzzleBoard variable from scratch, we don't pass in any input. If we make a PuzzleBoard variable by copying, we pass in the board to copy. Java is smart enough to distinguish these two cases.

We have to be able to tell if two PuzzleBoard variables really represent the same configuration of tiles. For this, we write a method called equals. It checks whether the tiles are in the same place on each board:

```
/* test whether two puzzle boards are the same configuration */
public boolean equals(PuzzleBoard other)
{
        int i, j;
        for (i = 0; i < 3; i++)
        {
                for (j = 0; j < 3; j++)
                {
                        /* if any tile is different, the boards aren't the same */
                        if (matrix[i][j] != other.matrix[i][j])
                                return false;
                }
        }
        /* if no differences were found, the matrices must be the same */
        return true;
}
```

Whenever we move a tile, we change the information for that tile in the tiles array. But we also need to make sure that the 3 x 3 array matrix has the new information. This function just updates the matrix array to match the tile positions, and is used after a move has been made.

```
/* update the puzzleboard's 2-dimensional tile matrix after a move has been
        made */
public void IndexTiles()
{
        PTile tile;
        int k = 0;
        /* for every tile in the tiles array, */
        for (k = 0; k < 9; k++)
        {
                tile = tiles[k];
```

```
                /* update its position in the 2-dimensional matrix */
                matrix[tile.GetXPos()-1][tile.GetYPos()-1] = k;
        }
}
/* identify which tile occupies a position in the 2-dimensional array */
```

The next function looks up a tile in the matrix array. Since it's conventional to define tile positions in the 8-puzzle by numbers, like (2, 2) for the center position (that is, we start counting at 1) and Java starts counting arrays from 0, we have to be careful to subtract one from the conventional position before using it to look up stuff in matrix, the two-dimensional array.

```
public int TileAt(int x, int y)
{
        return matrix[x-1][y-1];
}
```

To play this game, we need to be able to set up a starting board. This method asks the user for the horizontal and vertical position of tile 0 (the space), then tile 1, tile 2, and so on to tile 8. It then sets the current position of each PTile in the tile array, and calculates the place where that tile really belongs (don't worry about the details of this) and sets that too.

```
/* get a starting board from the user */
public void GetInitialState() throws IOException
{
        PTile tile;
        int k;
        /* input from keyboard requires this line */
        BufferedReader stdin = new BufferedReader(new
                InputStreamReader(System.in));
        int i, j, xpos, ypos, tx, ty;

        /* before the tiles are placed, no moves are possible */
        for (i = 0; i < 4; i++)
        {
                xmoves[i] = 0;
                ymoves[i] = 0;
                tilemoves[i] = -1;
        }
        k = 0;

        /* get the user to give a position for each tile */
        for (i = 0; i < 3; i++)
        {
                for (j = 0; j < 3; j++)
```

```
                {
                        /* ask the user for the positions of each tile (1,1) to (3,3) */
                        System.out.print("Enter x position for tile " + k + ": ");
                        xpos = Integer.parseInt(stdin.readLine());

                        System.out.print("Enter y position for tile " + k + ": ");
                        ypos = Integer.parseInt(stdin.readLine());

                        tile = tiles[k];
                        /* tile 0 (the blank) belongs at the end (the 3,3 position) */
                        if (k == 0)
                                tile.Set(xpos, ypos, 3, 3);
                        /* otherwise the tile position can be calculated from the tile
                                number */
                        else
                        {
                                if (k%3 == 0)
                                {
                                        tx = k/3;
                                        ty = 3;
                                }
                                else
                                {
                                        tx = (k/3+1);
                                        ty = k%3;
                                }
                                tile.Set(xpos, ypos, tx, ty);
                        }
                        k++;
                }
        }
}
```

It's always good to be able to print out different boards, so you can see if they are behaving the way you expect. This function just prints out the tile arrangement on a particular PuzzleBoard.

```
/* print out the puzzle board */
public void PrintBoard()
{
        int i, j;
        for (i = 0; i < 3; i++)
        {
                for (j = 0; j < 3; j++)
                        System.out.print(matrix[i][j]+ " ");
```

```
                System.out.println();
            }
            System.out.println();

        }
```

The next function is a bit complicated, so don't worry about the details. Since the blank square's position determines possible moves, we need to figure out whether the blank square moves—that is, whether a numbered tile gets shifted into the blank space. We just figure out where the blank space can move (left, right, up or down); then we find the adjacent tiles that can move into the blank space, then we set up the three arrays that define the allowed moves for these tiles. The allowed moves array have four slots; if the move isn't possible, then we set the tile it corresponds to –1, which indicates that the move isn't allowed.

```
        /* identify moves that this board allows us to make */
        public void PossibleMoves()
        {
            boolean left = false;
            boolean right = false;
            boolean up = false;
            boolean down = false;
            /* if the blank's not on the left edge of the puzzle, we can move it left */
            if (tiles[0].GetXPos() > 1)
                left = true;
            /* if the blank's not on the right edge of the puzzle, we can move it right */
            if (tiles[0].GetXPos() < 3)
                right = true;
            /* if the blank's not on the top edge of the puzzle, we can move it up */
            if (tiles[0].GetYPos() > 1)
                up = true;
            /* if the blank's not on the bottom edge of the puzzle, we can move it d
                down */
            if (tiles[0].GetYPos() < 3)
                down = true;
            /* if down's a legal move, figure out what tile's above the blank */
            if (down)
            {
                tilemoves[0] = TileAt(tiles[0].GetXPos(), tiles[0].GetYPos()+1);
                xmoves[0] = 0;
                ymoves[0] = 1;                  /* this tile moves down one space */
            }
            else
            {
                tilemoves[0] = -1;
                xmoves[0] = 0;
                ymoves[0] = 0;
```

```
        }
        /* if up's a legal move, figure out what tile's below the blank */
        if (up)
        {
                tilemoves[1] = TileAt(tiles[0].GetXPos(), tiles[0].GetYPos()-1);
                xmoves[1] = 0;
                ymoves[1] = -1;                 /* this tile moves up one space */
        }
        else
        {
                tilemoves[1] = -1;
                xmoves[1] = 0;
                ymoves[1] = 0;
        }
        /* if left's a legal move, figure out what tile's right of the blank */
        if (left)
        {
                tilemoves[2] = TileAt(tiles[0].GetXPos()-1, tiles[0].GetYPos());
                xmoves[2] = -1;                 /* this tile moves one right */
                ymoves[2] = 0;
        }
        else
        {
                tilemoves[2] = -1;
                xmoves[2] = 0;
                ymoves[2] = 0;
        }
        /* if right's a legal move, figure out what tile's right of the blank */
        if (right)
        {
                tilemoves[3] = TileAt(tiles[0].GetXPos()+1, tiles[0].GetYPos());
                xmoves[3] = 1;                  /* this tile moves one left */
                ymoves[3] = 0;
        }
        else
        {
                tilemoves[3] = -1;
                xmoves[3] = 0;
                ymoves[3] = 0;
        }
}
```

Every time we examine a new PuzzleBoard, we need to decide whether it matches the solved state. We can do this by using the InPlace method defined in the PTile class to see whether each tile in the board's tile array is in its proper place. We count up every

misplaced tile in the board. You'll write a function something like this to count how far out of place each tile is, in homework 3.

```
/* count how many tiles are not in place */
public int MisplacedTiles()
{
        PTile tile;
        int k, sum;
        sum = 0;
        for (k = 0; k < 9; k++)
        {
                tile = tiles[k];
                if (!tile.InPlace())
                        sum += 1;
        }
        return sum;
}
```

Finally, we need to actually move tiles. Moving a tile means swapping its position with the blank tile. We get the new x and y positions by using the three arrays that define the allowed moves. We have to check that the move does not shift a tile off the board. When we change the position of a tile, we use the Set method in the PTile class to change its position. Remember that the tile position variables are private, so we can't change them directly, but can set them using PTile class methods.

```
/* move a tile on the board */
public boolean MoveTile(int direction)
{
        int dx, dy;
        PTile tile;
        int tileindex;

        /* if the move isn't valid, send back an error value */
        if (tilemoves[direction] == -1)
                return false;
        /* choose the tile to move */
        tile = tiles[tilemoves[direction]];

        dx = xmoves[direction];
        dy = ymoves[direction];

        /* sanity check--we can't move off the board */
        if (tile.GetXPos() - dx > 3 || tile.GetXPos() - dx < 1)
                return false;
        if (tile.GetYPos() - dy > 3 || tile.GetYPos() - dy < 1)
                return false;
```

```
                /* we also have to make sure we move the blank space */
                tiles[0].Set(tile.GetXPos(), tile.GetYPos(), 3, 3);
                tile.Set(tile.GetXPos() - dx, tile.GetYPos() - dy, tile.GetTrueX(),
                        tile.GetTrueY());

                return true;
        }
}
```

Woof! That's the PuzzleBoard class. Aren't you glad that's over?

The CleverSearch class

The CleverSearch class takes care of searching the boards. There's a difference between this class and the PTile and PuzzleBoard classes. You can imagine a particular PTile, like tile 3 at position (3,3) on the board when it belongs at position (1,3). Or you can imagine a particular PuzzleBoard, with tiles in certain positions and a particular depth away from the starting board. But we don't define a particular kind of CleverSearch. If the class method gets called without a particular example of the class, then it's called by the class name instead, of the variable name. Methods called in this way are **static**. (In Java, mathematical functions are static; that's why you can say Math.sqrt(5.3) to get the square root of a number, or Math.abs(-3) to get a number's absolute value. You're calling these functions through the Math class, rather than making a variable of type Math and calling them though that variable.

The first thing we should do is make sure we're not repeating boards, which is done by the NotExplored method. We compare the board we're curious about to every board we've inspected so far. If it matches a board we've already looked at, then we don't want to examine it again. If it's new, then we want to put it on the list of boards to check.

```
public class CleverSearch
{
        /* check that a puzzleboard we're using is really a new configuration */
        public static boolean NotExplored(PuzzleBoard currentboard, PuzzleBoard []
                closed)
        {
                int i;
                boolean unexplored = true;
                if (closed.length > 0)
                {
                        /* search the closed list for a matching board */
                        for (i = 0; i < closed.length; i++)
                        {
                                /* if we find one, then our board is not new */
                                if (currentboard.equals(closed[i]))
```

```
                                unexplored = false;
                        }
                }
                /* else we should go ahead and use the board */
                return unexplored;
        }
```

Whenever we want to look at an open board, we take the first item from the array of open boards. Since after that, we don't want to look at this board again, we must remove it from the open list. The method below just copies the open array to a new array without the first element of open.

```
        /* shorten the list of boards by one, by removing the first item */
        public static PuzzleBoard [] PopBoard(PuzzleBoard [] boardlist)
        {
                int i;
                PuzzleBoard oldboard = new PuzzleBoard();
                PuzzleBoard firstboard;
                if (boardlist.length > 0)
                {
                        /* reserve memory for the shortened list */
                        PuzzleBoard [] newboardlist = new PuzzleBoard[boardlist.length -
                                1];
                        for (i = 1; i < boardlist.length; i++)
                        {
                                /* copy the second item from the old list to the first item in
                                        the new one, and so on till the end */
                                newboardlist[i-1] = boardlist[i];
                        }
                        return newboardlist;
                }
                /* if this failed, send back an error value */
                else
                {
                        return null;
                }
        }
```

The next function is long in code, but represents a simple idea. Given a particular PuzzleBoard, we make an array containing all the 'child' boards that can be obtained in one move from that board. This means figuring out which moves can be made, and then checking whether any of the boards resulting from these moves are ones we've already checked. At the end, this method sends back a list of the child boards, sorted by (depth + score).

```
        /* given a board, figure out what child boards are worth exploring */
```

```
public static PuzzleBoard [] ExpandBoard(PuzzleBoard currentboard,
        PuzzleBoard [] closed)
{
        int k = 0;
        int i;
        int min;
        int min_index=-1;
        int count = 0;
        int possboards = 0;
        boolean [] legalmoves = new boolean[4];
        PuzzleBoard board;
        PuzzleBoard [] boards = new PuzzleBoard[4];
        currentboard.PossibleMoves();
        /* check the possible moves */
        while (k < 4)
        {
                /* if this move is valid, then let's create a new puzzleboard for that
                        board */
                if (currentboard.tilemoves[k] > -1)
                {
                        board = new PuzzleBoard(currentboard);

                        /* move the tile we're planning to move */
                        board.MoveTile(k);
                        board.IndexTiles();
                        board.QuantMisplacedTiles();

                        /* if the board is one we haven't seen before, then we want
                                to explore it */
                        if (NotExplored(board, closed))
                        {
                                possboards++;
                                board.QuantMisplacedTiles();
                                board.depth = currentboard.depth + 1;
                                boards[k] = board;
                                legalmoves[k] = true;
                        }

                        /* if it's not new, we don't want to check it again */
                        else legalmoves[k] = false;
                }
                else
                        legalmoves[k] = false;
                k++;
        }
```

```
        /* make a sorted list of the child boards; this is important but I don't
                want to get into the gory details here--ask me if you're curious */
        PuzzleBoard [] childboards = new PuzzleBoard[possboards];
        int [] checked = new int[4];
        for (k = 0; k < 4; k++)
                checked[k] = -1;
        int [] order = new int[possboards];
        for (i = 0; i < possboards; i++)
        {
                min = 1000000;
                for (k = 0; k < 4; k++)
                {
                        if (legalmoves[k] && checked[k] == -1 &&
                                boards[k].score + boards[k].depth < min)
                        {
                                min = boards[k].score + boards[k].depth;
                                min_index = k;
                        }
                }
                order[i] = min_index;
                checked[min_index] = 1;
        }
        for (i = 0; i < possboards; i++)
        {
                childboards[i] = boards[order[i]];
        }

        /* print the child boards */
        System.out.println("Child boards");
        for (k = 0; k < possboards; k++)
        {
                System.out.println("Child score: " + (childboards[k].score +
                        childboards[k].depth));
                childboards[k].PrintBoard();
        }

        /* send back the list of child boards to add to the search */
        return childboards;
}
```

At present, the child boards you find above are added to the end of the array of open boards to consider. This function takes care of that. It does this by figuring out how many items are in the open list and how many items are in the child boards list, then reserving an array big enough to hold all the items. Then it copies the boards listed in open to the first parts of the new array, and copies the boards in the child boards to the

end part of the array. Your homework involves doing this same thing, but copying the boards to the new array so that they end up in sorted order.

```
/* less sophisticated searching function (why?). It should give you an idea about
        what to do for you smart sorting function */
public static PuzzleBoard [] EnqueueAtEnd(PuzzleBoard [] oldboard,
        PuzzleBoard [] addboard)
{
        int i = 0;
        int j = 0;
        int k = 0;

        /* make a new array to hold the concateneated list */
        PuzzleBoard [] newboard = new PuzzleBoard [oldboard.length +
                addboard.length];
        /* copy the first list over */
        while (i < oldboard.length)
        {
                newboard[k] = oldboard[i];
                k++;
                i++;
        }
        /* copy the second list over */
        while (j < addboard.length)
        {
                newboard[k] = addboard[j];
                k++;
                j++;
        }
        return newboard;
}
```

The last routine is the search for the right board. Inside what appears to be an infinite loop, we do the following: take the first board off the open list, check if it's solved. If not, we generate all the new boards we can get in one move from this board, and add those to the open list to check later. Finally, we throw away the board we were checking. There are only two ways out of the 'infinite' loop. One way is that we find the solved board. The other is that we run out of boards to check. Eventually, one thing or the other will happen, so the loop really does terminate. We keep track of how many boards we've checked in the integer variable k, to see how efficient the search is.

```
/* the heart of the search routine */
public static PuzzleBoard SmartSearch(PuzzleBoard [] open, PuzzleBoard []
        closed) throws IOException
{
        PuzzleBoard [] queue;
```

```
PuzzleBoard [] discarded;
PuzzleBoard currentboard = new PuzzleBoard();
BufferedReader stdin = new BufferedReader(new
        InputStreamReader(System.in));
/* this loop would be infinite except for the return statement, which break
        out */
int k = 0;
while (true)
{
        /* if there are no boards to search, we need to stop */
        if (open.length == 0)
        {
                System.out.println("Search failed");
                return null;
        }

        /* get the first board in the list of boards to search */
        currentboard = open[0];
        open = PopBoard(open);
        System.out.println("score: " + (currentboard.score +
                currentboard.depth));
        currentboard.PrintBoard();

        /* if the board's already solved, it's fine, we're done */
        if (currentboard.MisplacedTiles() == 0)
        {
                System.out.println("Found a solution at " +
                        currentboard.depth + " in " + k + " tries");
                return currentboard;
        }

        /* else we need to check the children of this board; you will call
                your sorted list function here instead of EnqueueAtEnd,
                once you write it */
        queue = EnqueueAtEnd(open, ExpandBoard(currentboard,
                closed));

        System.out.println("Queue length: " + queue.length);

        /* add this board to the closed list */
        discarded = new PuzzleBoard[1];
        discarded[0] = currentboard;
        closed = EnqueueAtEnd(closed, discarded);

        System.out.println("Closed length: " + closed.length);
        currentboard.PrintBoard();
```

```
                    /* update to the new queue */
                    open = queue;
                    k++;
                }
        }
}
```

The EightPuzzle class

The main function is contained in EightPuzzle.java. As you can see, it's quite short, because all the functionality is contained in the other code modules. We define an open list for boards we want to check, and a closed list for boards we've already checked (at the start, both are empty). Then we get a board, add the board to the open list, and send the open and closed lists off to the search method described above. The call to SmartSearch is static, meaning we call it through the class CleverSearch rather than though a variable of type CleverSearch.

```
import java.io.*;
public class EightPuzzle
{

        /**
        Solves the 8-puzzle by breadth-first search
        **/
        public static void main (String[] args) throws IOException
        {
                BufferedReader stdin = new BufferedReader(new
                        InputStreamReader(System.in));

                int i;
                /* list of boards to consider */
                PuzzleBoard [] open;
                /* list of boards we're done worrying about */
                PuzzleBoard [] closed = new PuzzleBoard[0];

                /* start a new list for the open boards */
                open = new PuzzleBoard[1];

                /* reserve memory for the starting board */
                PuzzleBoard currentboard = new PuzzleBoard();

                /* get the initial tile positions */
                currentboard.GetInitialState();

                /*
```

```
                currentboard.QuantMisplacedTiles();
                */

                /* the starting board is 0 moves away from itself */
                currentboard.depth = 0;

                /* make sure the board has its tiles in order */
                currentboard.IndexTiles();

                /* find possible moves on the board */
                currentboard.PossibleMoves();

                /* add the starting board to the open list */
                open[0] = currentboard;

                /* now start searching */
                CleverSearch.SmartSearch(open, closed);
        }
}
```