CS 5200

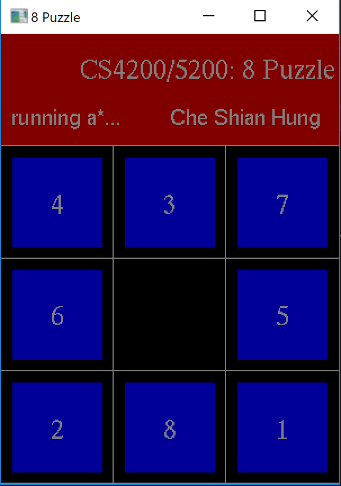
Che Shian Hung

Program # 1 – Eight Sliding Puzzle Analysis

The program use A\* search and two different heuristic functions to solve random 8 slide puzzles. Here are the steps to run the program and screenshots. First, when we run the program, one solvable puzzle will be automatically generated. There are three options: press j to run a\* search with h1(), press k to run a\* search with h2(), and press s to play the puzzle in manual mode.

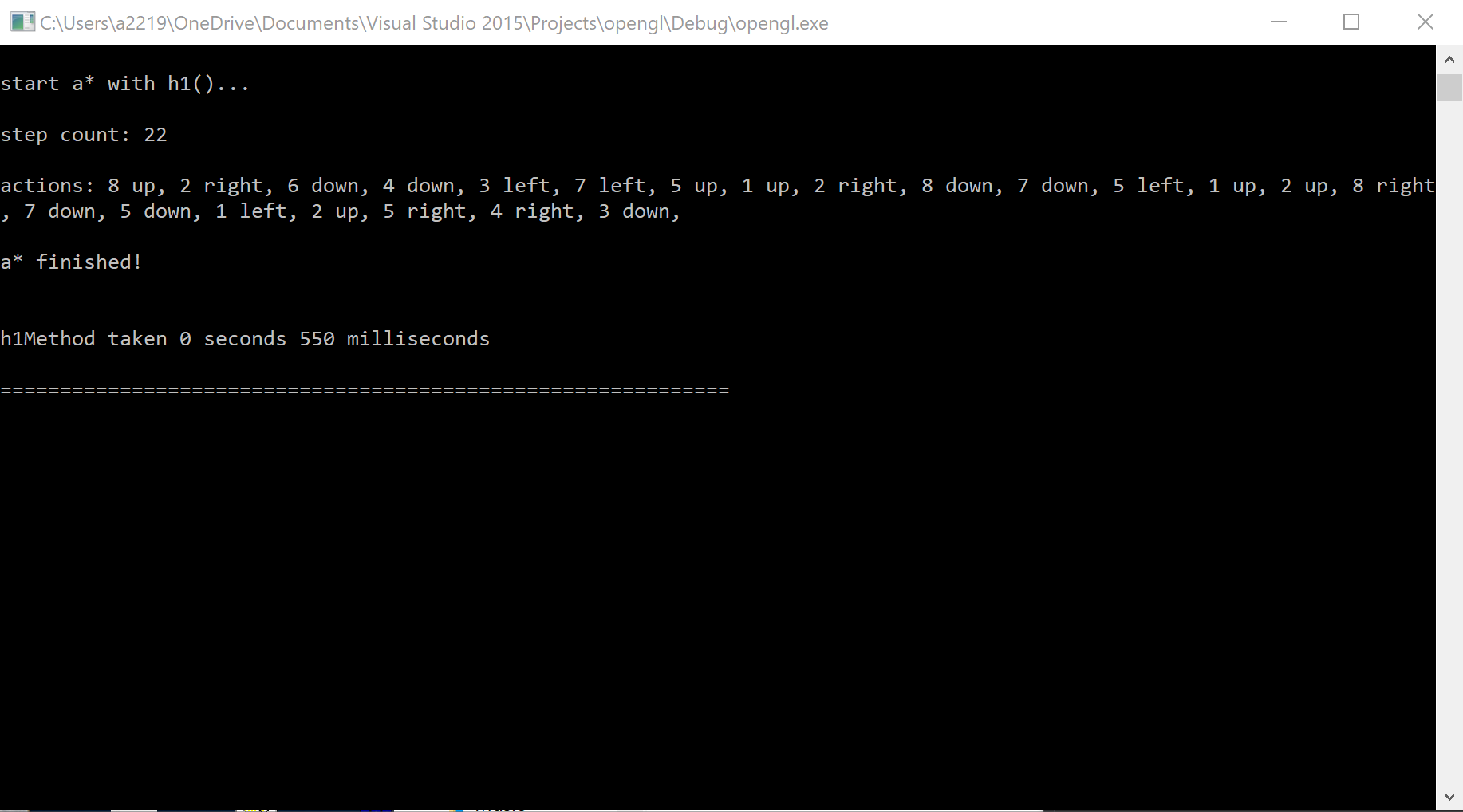


If we press either ‘j’ or ’k’, the text will change to “running a\*” until the searching is finished.



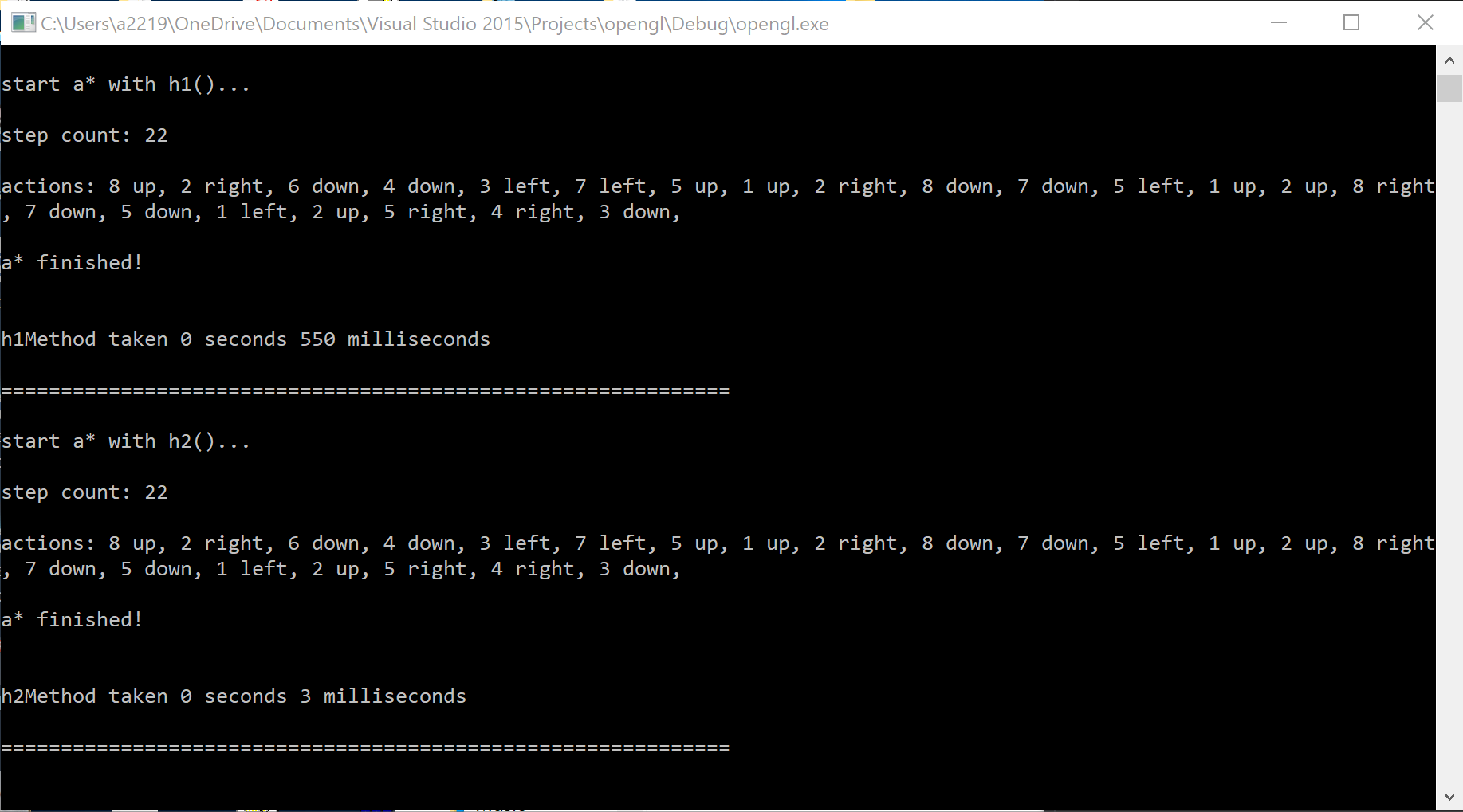
Once the searching is done, the text will back to the selection mode but ready to run the animated solution. The solution and the searching process will be printed in the terminal as well. In this case, the solution for with h1 function is ready to run.



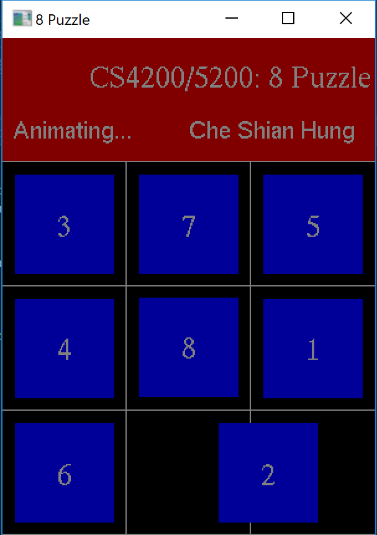


Here are the screenshots when both methods are ready. The solution animation will be available without finishing the corresponding a\* search. In this case, both algorithm found the optimal solution requires 22 steps to solve, and the list of solution actions are the same (sometimes they are different). The time performance is also shown after the a\* is finished.



.

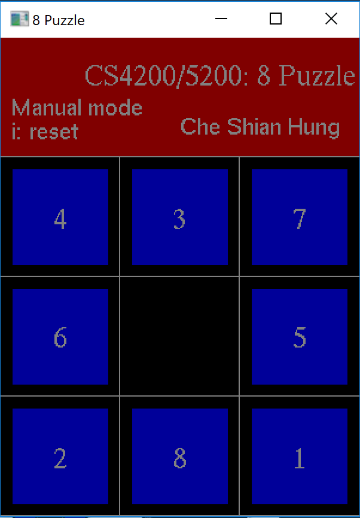
Once the solution has been generated, we can press either ‘n’ for the h1 version or ‘m’ for the n2 version to run the solution animation. Here is the screenshot when one of the solutions is running its animation.



Once the animation is done. The program will change back to selection mode and looks like the following picture. The tiles will be in the correct order after the solution is applied.



The user can also press ‘s’ to play the puzzle in manual mode. Even after the puzzle has been solved after running the animation, the puzzle will return to its beginning state for the user to play. The user can click the tiles to trigger the action. During the manual mode, the player can always press ‘I’ to forfeit and play a new random puzzle (‘i’ key applies in other modes too).



When the puzzle is solved in the manual mode, it will appear like this.



To analyze the perform for each heuristic functions, I ran both methods on 15 different test cases. The following table shows the performance results. Each row represents each distinctive puzzle, and for each puzzle, I focus on the solution step count, the time performance for h1 method and h2 methods.

|  |  |  |
| --- | --- | --- |
| Step Count (Optimal Solution Steps) | h1 Method Performance (s) | h2 Method Performance (s) |
| 16 | 0.006 | 0.005 |
| 18 | 0.01 | 0.002 |
| 19 | 0.019 | 0.011 |
| 19 | 0.074 | 0.035 |
| 21 | 0.309 | 0.024 |
| 22 | 0.55 | 0.003 |
| 23 | 1.346 | 0.048 |
| 23 | 2.159 | 0.052 |
| 24 | 2.236 | 0.035 |
| 24 | 4.268 | 0.034 |
| 24 | 2.495 | 0.186 |
| 26 | 85.356 | 0.148 |
| 26 | 56.755 | 0.194 |
| 26 | 229.609 | 0.194 |
| 26 | 110.841 | 0.396 |

In the result, we can see that regardless the number of the step count, the performance of h2 Method (Manhattan distance) is always better than the performance of h1 Method (unmatched tiles), and the difference of the performances becomes greater when the step count is bigger. It makes sense because h1 method will only output cost number from 0 to 9, but h2 method can output cost number from 0 to more than 20, which is easier to identify which state is closer to the goal state. With h1 method, there will be more inefficient node selections because the costs are too similar to identify. Especially when the optimal step count is greater, since we are using tree to store all the states, the greater the step count, the taller the tree and more children will be generated and stored into the tree. That is also why the exponential growth of time performance for h1 method when the step count increases.

/\*

\*\*\* FILE NAME : eight\_slide\_ex.c

\*\*\* DATE : Sep. 26th 2017

\*\*\* WRITTEN By : Che Shian Hung, JKL

\*\*\* PURPOSE : The program use A\* search and two different heuristic functions to

solve random 8 slide puzzle.

\*\*\* DESCRIPTION : The program uses four structures to carry out entrie process.

MyPoint is used to declare the coordinates for each tile. The program

will construct and keep track of the coordinates for each tile, so that

it can use these coordinates to move the tile. The Node struct contains

all the information for a state, like the path cost and the heuristic cost,

current values' position, what is the last action been made, the pointers

to its children and parent. The program use Nodes to create the tree and

search for the goal state. The list struct contains all the nodes that have

been created and sorts the nodes with the total cost (g + h). The solution

list struct contains a list of actions and moved values information, like

tile 2 moves down or tile 9 goes left. Inside aStar() function, it

will first create a tree from the initial state and create a list to sort

the cost. Then it will extract the node with the least code from the list

and check if it is the goal state. If not, it will expend the leaf into a

subtree and add the new leafs to the list. Once a goal state is found,

the it will trace the goal state back to the root and add the actions to

the solution list along the way. With the info in solution list, the

program can use the info to animate the action one by one.

\*/

#include <stdio.h> // standard C libraries

#include <stdlib.h>

#include <math.h>

#include <time.h>

#include <string.h>

#include <GL/glut.h> // GLUT library

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

// Constants

#define WINDOW\_XS 300 // Window size

#define WINDOW\_YS 400

#define WINDOW\_NAME "8 Puzzle" // Window name

#define ANI\_MSEC 5 // gap between frames

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

// Structures

typedef struct pt

{

GLfloat x, y;

}MyPoint;

// Node for each state

struct node;

typedef struct node {

int g;

int h;

int zeroPos;

int lastAction; //1: up, 2: down, 3: left, 4: right

int childrenSize;

int values[9];

struct node \*parent;

struct node \*children;

} Node;

// List to sort the nodes

struct list;

typedef struct list {

Node \*n;

struct list \*next;

} List;

// List to store the solution

struct solutionList;

typedef struct solutionList {

int action;

int movePosition;

struct solutionList \*next;

} SolutionList;

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

// Global Variables

MyPoint bottomLeftPt;

MyPoint boxCordinates[8];

SolutionList \*h1Solution = NULL;

SolutionList \*h2Solution = NULL;

SolutionList \*currentSolution = NULL;

int values[9];

int initValues[9];

int gap = 1; // step for animation

int h1Mode = 0; // -2: manual finished, -1: manual, 0: before run, 1: running, 2: ready, 3: runningAnimation

int h2Mode = 0; // -2: manual finished, -1: manual, 0: before run, 1: running, 2: ready, 3: runningAnimation

int hMethod = 0; // 1: h1(), 2: h2()

int offSet = 10;

int recLength = 80; // size of puzzle

int doAnimation = 0; // 0: no animation, 1: up, 2: down, 3: left, 4: right

int selectedValue = -1;

int currentPosition;

int animationCounter;

int findingSolution = 0;

int runningAnimation = 0;

int solutionAnimationCounter = 0;

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

// Function prototypes

void output(int x, int y, int mode, char \*string);

void mouse\_func(int button, int state, int x, int y);

void init\_setup(int width, int height, char \*windowName);

void display\_func(void);

void keyboard\_func(unsigned char c, int x, int y);

void animation\_func(int val);

void reshape\_handler(int width, int height);

void aStar();

void drawTile(char \*val, float x, float y);

void setupPuzzle();

void newProblem();

void reloadProblem();

void drawAllTiles();

void switchPosition(int \*ary, int a, int b);

int isGoal(int \*ary);

int solvable(int \*ary);

int heuristic(int \*ary);

int getDirection(int p);

int getPosition(int x, int y);

void destroyTree(Node \*n);

void expendTree(Node \*n);

void displayList(List \*h);

List\* addToList(List \*h, Node \*n);

List\* removeFront(List \*h);

void displaySolutionList(SolutionList \*h);

SolutionList\* addToSolutionList(SolutionList \* h, Node \*n);

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

int main(int argc, char \*\*argv)

{

glutInit(&argc, argv);

init\_setup(WINDOW\_XS, WINDOW\_YS, WINDOW\_NAME);

newProblem();

bottomLeftPt.x = 0;

bottomLeftPt.y = 0;

glutDisplayFunc(display\_func);

glutKeyboardFunc(keyboard\_func);

glutMouseFunc(mouse\_func);

glutTimerFunc(ANI\_MSEC, animation\_func, 0);

glutMainLoop();

return 1;

} // end of main()

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void reshape\_handler(int width, int height)

{

glViewport(0, 0, width, height); // sets the viewport

glMatrixMode(GL\_PROJECTION); // projection matrix

glLoadIdentity(); // loads identity matrix

gluOrtho2D(0.0, (GLdouble)width, 0.0, (GLdouble)height); // 2D orthographic projection

} // end of reshape\_handler()

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void init\_setup(int width, int height, char \*windowName)

{

//glutInitDisplayMode(GLUT\_SINGLE | GLUT\_RGB); // single buffer, rgb color

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB); // double buffer, rgb color

glutInitWindowSize(width, height); // init. window size

glutInitWindowPosition(5, 5); // init. window position

glutCreateWindow(windowName); // window name

glutReshapeFunc(reshape\_handler); // sets the reshape call back

} // end of init\_setup()

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void display\_func(void)

{

int i;

glClearColor(0.0, 0.0, 0.0, 1.0); // background color (yellow)

glClear(GL\_COLOR\_BUFFER\_BIT); // clearing the buffer not to keep the color

// draw top of the puzzle board

glColor3f(0.5, 0.0, 0.0); // setting pen color

glBegin(GL\_POLYGON);

glVertex2i(0, 300);

glVertex2i(300, 300);

glVertex2i(300, 400);

glVertex2i(0, 400);

glEnd();

// banner

glColor3f(0.5, 0.5, 0.5); // setting pen color

output(70, 360, 1, "CS4200/5200: 8 Puzzle");// puzzle number

output(150, 320, 2, "Che Shian Hung"); // puzzle number

// h1Mode, h2Mode

// -2: manual finished, -1: manual, 0: before run, 1: running, 2: ready, 3: runningAnimation

if (h1Mode == -2 && h2Mode == -2) {

output(10, 335, 2, "Good job!");

output(10, 315, 2, "i: reset");

}

else if (h1Mode == -1 && h2Mode == -1) {

output(10, 335, 2, "Manual mode");

output(10, 315, 2, "i: reset");

}

else if (h1Mode == 1 || h2Mode == 1) {

output(10, 320, 2, "running a\*...");

}

else if (h1Mode == 3 || h2Mode == 3) {

output(10, 320, 2, "Animating...");

}

else {

if (h1Mode == 0) {

output(10, 346, 2, "j: h1");

}

else if (h1Mode == 2) {

output(10, 346, 2, "n: run h1");

}

if (h2Mode == 0) {

output(10, 325, 2, "k: h2");

}

else if (h2Mode == 2) {

output(10, 325, 2, "m: run h2");

}

output(10, 308, 2, "s: manual");

}

// update text and mode and run aStar

if ((h1Mode == 1 && findingSolution == 0) || (h2Mode == 1 && findingSolution == 0)) {

findingSolution++;

}

else if ((h1Mode == 1 && findingSolution == 1) || h2Mode == 1 && findingSolution == 1) {

clock\_t start= clock(), diff;

aStar();

diff = clock() - start;

int msec = diff \* 1000 / CLOCKS\_PER\_SEC;

printf("h%dMethod taken %d seconds %d milliseconds\n\n", hMethod, msec / 1000, msec % 1000);

printf("=============================================================\n\n");

}

// update mode and run animation

if ((h1Mode == 3 && runningAnimation == 0) || (h2Mode == 3 && runningAnimation == 0)) {

runningAnimation++;

}

else if ((h1Mode == 3 && runningAnimation == 1) || h2Mode == 3 && runningAnimation == 1) {

doAnimation = 5;

}

// draw the grids for the puzzle board

glColor3f(0.5, 0.5, 0.5); // setting pen color

glBegin(GL\_LINES);

for (i = 0; i < WINDOW\_XS; i += 100)

{

glVertex2i(i, 0); // vertical grid lines

glVertex2i(i, 300);

glVertex2i(0, i); // horizontal grid lines

glVertex2i(300, i);

}

glVertex2i(0, 300);

glVertex2i(300, 300);

glEnd();

drawAllTiles();

glFlush();

glutSwapBuffers(); // double buffering

} // end of display\_func()

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void keyboard\_func(unsigned char c, int x, int y)

{

switch (c)

{

case 'j': // run h1()

if ((h1Mode == -2 || h1Mode == 0) && h2Mode != 1 && h2Mode != 3) {

reloadProblem();

hMethod = 1;

findingSolution = 0;

h1Mode = 1;

}

break;

case 'k': // run h2()

if ((h1Mode == -2 || h2Mode == 0) && h1Mode != 1 && h1Mode != 3) {

reloadProblem();

hMethod = 2;

findingSolution = 0;

h2Mode = 1;

}

break;

case 'n': // animate solution from h1()

if (h1Mode == 2 && h2Mode != 1 && h2Mode != 3) {

reloadProblem();

runningAnimation = 0;

h1Mode = 3;

hMethod = 1;

solutionAnimationCounter = 0;

}

break;

case 'm': // animate solution from h2()

if (h2Mode == 2 && h1Mode != 1 && h1Mode != 3) {

reloadProblem();

runningAnimation = 0;

h2Mode = 3;

hMethod = 2;

solutionAnimationCounter = 0;

}

break;

case 's': // manual mode

if (h1Mode != 1 && h1Mode != 3 && h2Mode != 1 && h2Mode != 3) {

reloadProblem();

h1Mode = -1;

h2Mode = -1;

glutPostRedisplay();

}

break;

case 'i': //get new problem

newProblem();

break;

case 'Q':

case 'q':

//Destroy the lists

while (h1Solution) {

List \*temp = h1Solution->next;

h1Solution = h1Solution->next;

temp = NULL;

free(temp);

}

while (h2Solution) {

List \*temp = h2Solution->next;

h2Solution = h2Solution->next;

temp = NULL;

free(temp);

}

printf("Good Bye !\n");

exit(0); // terminates the program

} // end of switch

} // end of keyboard\_func()

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void mouse\_func(int button, int state, int x, int y)

{

// when in manual mode, done animation, and press down the mouse

if (state == GLUT\_DOWN && doAnimation == 0 && h1Mode == -1 && h2Mode == -1) {

int p = getPosition(x, WINDOW\_YS - y); // the position user clicks

if (p != -1 && p != currentPosition){ // if the position is valid and not zero

selectedValue = values[p];

int d = getDirection(p); // get direction of sliding

doAnimation = d; // do animation

switch (d) { // swap values

case 1:

switchPosition(values, currentPosition, currentPosition + 3);

currentPosition += 3;

break;

case 2:

switchPosition(values, currentPosition, currentPosition - 3);

currentPosition -= 3;

break;

case 3:

switchPosition(values, currentPosition, currentPosition + 1);

currentPosition++;

break;

case 4:

switchPosition(values, currentPosition, currentPosition - 1);

currentPosition--;

break;

}

animationCounter = 0;

glutPostRedisplay();

}

}

}

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void animation\_func(int val)

{

// up

if (doAnimation == 1) {

boxCordinates[selectedValue - 1].y += gap;

animationCounter++;

}

// down

else if (doAnimation == 2) {

boxCordinates[selectedValue - 1].y -= gap;

animationCounter++;

}

// left

else if (doAnimation == 3) {

boxCordinates[selectedValue - 1].x -= gap;

animationCounter++;

}

// right

else if (doAnimation == 4) {

boxCordinates[selectedValue - 1].x += gap;

animationCounter++;

}

// solutions from h1() or h2()

else if (doAnimation == 5) {

// select solution list

if (!currentSolution) {

if (hMethod == 1) currentSolution = h1Solution;

else currentSolution = h2Solution;

}

else {

// after a step

if (solutionAnimationCounter != 0 && solutionAnimationCounter % 101 == 0) {

// get next action

if (currentSolution->next) {

currentSolution = currentSolution->next;

}

// animation finished

else {

if (hMethod == 1) h1Mode = 2;

else h2Mode = 2;

doAnimation = 0;

runningAnimation = 0;

currentSolution = NULL;

}

}

// sliding tile

else {

switch (currentSolution->action) {

case 1:

boxCordinates[currentSolution->movePosition - 1].y += gap;

break;

case 2:

boxCordinates[currentSolution->movePosition - 1].y -= gap;

break;

case 3:

boxCordinates[currentSolution->movePosition - 1].x -= gap;

break;

case 4:

boxCordinates[currentSolution->movePosition - 1].x += gap;

break;

}

}

solutionAnimationCounter++;

}

}

// counter for manual mode

if (animationCounter >= 100) {

doAnimation = 0;

if (isGoal(values) == 1) {

h1Mode = -2;

h2Mode = -2;

}

}

glutPostRedisplay();

glutTimerFunc(ANI\_MSEC, animation\_func, 0);

} //end animation\_func

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void output(int x, int y, int mode, char \*string)

{

int len, i;

glRasterPos2i(x, y);

len = (int)strlen(string);

if (mode == 1)

{

for (i = 0; i < len; i++)

{

glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_24, string[i]);

}

}

else if (mode == 2)

{

for (i = 0; i < len; i++)

{

glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, string[i]);

}

}

}

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void drawTile(char \*val, float x, float y) {

// draw a puzzle piece

int x2 = (int)x;

int y2 = (int)y;

glColor3f(0.0, 0.0, 0.6); // setting pen color

glBegin(GL\_POLYGON);

glVertex2i(x2 + offSet, y2 + offSet);

glVertex2i(x2 + offSet + recLength, y2 + offSet);

glVertex2i(x2 + offSet + recLength, y2 + offSet + recLength);

glVertex2i(x2 + offSet, y2 + offSet + recLength);

glEnd();

glColor3f(0.5, 0.5, 0.5); // setting pen color

output(x2 + offSet + 34, y2 + offSet + 31, 1, val); // puzzle number

}

//@@ get the position from x, y coordinates\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

int getPosition(int x, int y) {

if (x < 95 && x > 5) {

if (y < 95 && y > 5) return 6;

else if (y < 195 && y > 105) return 3;

else if (y < 295 && y > 205) return 0;

}

else if (x < 195 && x > 105) {

if (y < 95 && y > 5) return 7;

else if (y < 195 && y > 105) return 4;

else if (y < 295 && y > 205) return 1;

}

else if (x < 295 && x > 205) {

if (y < 95 && y > 5) return 8;

else if (y < 195 && y > 105) return 5;

else if (y < 295 && y > 205) return 2;

}

return -1;

}

//@@ get direction (1 - 4) from a position and the sero position\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

int getDirection(int p) {

//up

if (p == currentPosition + 3) return 1;

//down

else if (p == currentPosition - 3) return 2;

//left

else if (p == currentPosition + 1 && p % 3 != 0) return 3;

//right

else if (p == currentPosition - 1 && p % 3 != 2) return 4;

else return 0;

}

//@@ set up random problem\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void setupPuzzle() {

int seed = (int)time(NULL);

int usedValues[9];

srand(seed);

// generating new problem if not solvable

do {

for (int i = 0; i < 9; i++) {

usedValues[i] = i;

}

for (int i = 0; i < 9; i++) {

int r = rand() % (9 - i);

values[i] = usedValues[r];

usedValues[r] = usedValues[8 - i];

usedValues[8 - i] = values[i];

//Testing

/\*

if (i == 0) values[i] = 1;

else if (i == 1) values[i] = 2;

else if (i == 2) values[i] = 0;

else if (i == 3) values[i] = 3;

else if (i == 4) values[i] = 4;

else if (i == 5) values[i] = 5;

else if (i == 6) values[i] = 6;

else if (i == 7) values[i] = 7;

else if (i == 8) values[i] = 8;

else values[i] = i;

\*/

}

} while (solvable(values) == 0);

// set up coordinates

for(int i = 0; i < 9; i++){

initValues[i] = values[i];

if (values[i] != 0) {

GLfloat xPos, yPos;

switch (i) {

case 0:

xPos = 0;

yPos = 200;

break;

case 1:

xPos = 100;

yPos = 200;

break;

case 2:

xPos = 200;

yPos = 200;

break;

case 3:

xPos = 0;

yPos = 100;

break;

case 4:

xPos = 100;

yPos = 100;

break;

case 5:

xPos = 200;

yPos = 100;

break;

case 6:

xPos = 0;

yPos = 0;

break;

case 7:

xPos = 100;

yPos = 0;

break;

case 8:

xPos = 200;

yPos = 0;

break;

}

boxCordinates[values[i] - 1].x = xPos;

boxCordinates[values[i] - 1].y = yPos;

}

else

currentPosition = i;

}

}

//@@ draw all eight tiles and numbers\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void drawAllTiles() {

char str[20];

for (int i = 0; i < 9; i++) {

if (values[i] != 0) {

int num = values[i];

snprintf(str, 20, "%d", num);

drawTile(str, boxCordinates[num - 1].x, boxCordinates[num- 1].y);

}

}

}

//@@ switch positions in an array\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void switchPosition(int \*ary, int a, int b) {

int temp = ary[a];

ary[a] = ary[b];

ary[b] = temp;

}

//@@ h1() and h2()\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

int heuristic(int \*ary) {

int count = 0;

if (hMethod == 1) {

for (int i = 0; i < 9; i++) {

if (ary[i] != i) {

count++;

}

}

}

else if (hMethod == 2) {

for (int i = 0; i < 9; i++) {

if (ary[i] != i) {

count += abs(ary[i] / 3 - i / 3) + abs(ary[i] % 3 - i % 3);

}

}

}

return count;

}

//@@ running a\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void aStar() {

printf("start a\* with h%d()...\n\n", hMethod);

// create root

Node \*r = (Node\*)malloc(sizeof(Node));

for (int i = 0; i < 9; i++) {

r->values[i] = initValues[i];

}

r->childrenSize = 0;

r->g = 0;

r->h = heuristic(values);

r->zeroPos = currentPosition;

r->parent = NULL;

Node \*root = r;

// create exploring list

List \*hList = NULL;

hList = addToList(hList, root);

// searching

while (hList->n->h != 0) {

Node \*fn = hList->n;

hList = removeFront(hList);

expendTree(fn);

for (int i = 0; i < fn->childrenSize; i++) {

hList = addToList(hList, &fn->children[i]);

}

//displayList(hList);

}

// input solution

Node \*t = hList->n;

while (t->parent) {

if (hMethod == 1) {

h1Solution = addToSolutionList(h1Solution, t);

}

else if (hMethod == 2) {

h2Solution = addToSolutionList(h2Solution, t);

}

t = t->parent;

}

printf("step count: %d\n\n", hList->n->g);

// display solutions

if (hMethod == 1) {

displaySolutionList(h1Solution);

}

else if (hMethod == 2) {

displaySolutionList(h2Solution);

}

// destroy the tree

destroyTree(root);

// destroy the list

while (hList) {

List \*temp = hList->next;

hList = hList->next;

temp = NULL;

free(temp);

}

if (hMethod == 1) h1Mode = 2;

else h2Mode = 2;

findingSolution = 0;

printf("a\* finished!\n\n\n");

}

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void destroyTree(Node \*n) {

if (n != NULL) {

n->parent = NULL;

for (int i = 0; i < n->childrenSize; i++) {

destroyTree(&n->children[i]);

}

n = NULL;

free(n);

}

}

//@@\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void expendTree(Node \*n) {

int zeroPos = n->zeroPos;

int actions[4];

for (int i = 0; i < 4; i++) {

actions[i] = 0;

}

// counting children size

if (zeroPos + 3 <= 8 && n->lastAction != 2) {

n->childrenSize++;

actions[0] = 1;

}

if (zeroPos - 3 >= 0 && n->lastAction != 1) {

n->childrenSize++;

actions[1] = 1;

}

if ((zeroPos + 1) % 3 != 0 && n->lastAction != 4) {

n->childrenSize++;

actions[2] = 1;

}

if ((zeroPos - 1) % 3 != 2 && zeroPos != 0 && n-> lastAction != 3) {

n->childrenSize++;

actions[3] = 1;

}

// create and intialize children

n->children = (Node\*)calloc(n->childrenSize, sizeof(Node));

int counter = 0;

for (int i = 0; i < 4; i++) {

if (actions[i] == 1) {

for (int i = 0; i < 9; i++) {

n->children[counter].values[i] = n->values[i];

}

switch (i) {

case 0:

n->children[counter].zeroPos = zeroPos + 3;

switchPosition(n->children[counter].values, zeroPos, zeroPos + 3);

break;

case 1:

n->children[counter].zeroPos = zeroPos - 3;

switchPosition(n->children[counter].values, zeroPos, zeroPos - 3);

break;

case 2:

n->children[counter].zeroPos = zeroPos + 1;

switchPosition(n->children[counter].values, zeroPos, zeroPos + 1);

break;

case 3:

n->children[counter].zeroPos = zeroPos - 1;

switchPosition(n->children[counter].values, zeroPos, zeroPos - 1);

break;

}

n->children[counter].lastAction = i + 1;

n->children[counter].h = heuristic(n->children[counter].values);

n->children[counter].childrenSize = 0;

n->children[counter].g = n->g + 1;

n->children[counter].parent = n;

n->children[counter].children = NULL;

counter++;

}

}

}

//@@ add new node to the list and sort by total cost\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

List\* addToList(List \*h, Node \*n) {

List \*newNode = (List\*)malloc(sizeof(List));

newNode->n = n;

newNode->next = NULL;

// if the list is empty

if (!h) {

h = newNode;

}

// if none empty list

else {

List \*l = h;

List \*pre = NULL;

// if the cost is greater than the current node's cost

while (l && n->g + n->h > l->n->g + l->n->h) {

pre = l;

l = l->next;

}

// insert at front

if (!pre) {

newNode->next = h;

h = newNode;

}

// insert at the end

else if (!l) {

pre->next = newNode;

}

// insert in middle

else {

pre->next = newNode;

newNode->next = l;

}

}

return h;

}

//@@ remove the front of the list\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

List\* removeFront(List \*h) {

List\* temp = h;

h = h->next;

free(temp);

return h;

}

//@@ display the list for testing\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void displayList(List\* h) {

if (h) {

printf("[g, h]: ");

List \*l = h;

while (l) {

printf("[%d, %d] ", l->n->g, l->n->h);

l = l->next;

}

printf("\n");

}

else {

printf("empty list\n");

}

}

//@@ check if the array matchs the goal state\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

int isGoal(int \*ary) {

int same = 1;

for (int i = 0; i < 9; i++) {

if (ary[i] != i) same = 0;

}

return same;

}

//@@ add the node information to the solution list\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

SolutionList\* addToSolutionList(SolutionList \*h, Node \*n) {

SolutionList \*newSolution = (SolutionList\*)malloc(sizeof(SolutionList));

newSolution->action = n->lastAction;

switch (n->lastAction) {

case 1:

newSolution->movePosition = n->values[n->zeroPos - 3];

break;

case 2:

newSolution->movePosition = n->values[n->zeroPos + 3];

break;

case 3:

newSolution->movePosition = n->values[n->zeroPos - 1];

break;

case 4:

newSolution->movePosition = n->values[n->zeroPos + 1];

break;

}

newSolution->next = h;

h = newSolution;

return h;

}

//@@ display the solutions to the user\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void displaySolutionList(SolutionList \*h) {

if (h) {

printf("actions: ");

SolutionList \*sl = h;

while (sl) {

switch (sl->action) {

case 1:

printf("%d up, ", sl->movePosition);

break;

case 2:

printf("%d down, ", sl->movePosition);

break;

case 3:

printf("%d left, ", sl->movePosition);

break;

case 4:

printf("%d right, ", sl->movePosition);

break;

}

sl = sl->next;

}

printf("\n\n");

}

else {

printf("empty list\n\n");

}

}

//@@ check if the problem is solvable\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

int solvable(int \*ary) {

int counter = 0;

for (int i = 0; i < 9 - 1; i++) {

for (int j = i + 1; j < 9; j++) {

if (ary[i] != 0 && ary[j] != 0 && ary[i] > ary[j])

counter++;

}

}

if (counter % 2 == 0) return 1;

else return 0;

}

//@@ create a new problem and initialize variables\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void newProblem() {

setupPuzzle();

h1Mode = 0;

h2Mode = 0;

doAnimation = 0;

findingSolution = 0;

runningAnimation = 0;

solutionAnimationCounter = 0;

currentSolution = NULL;

//reset solution

if (h1Solution) {

while (h1Solution) {

SolutionList \*temp = h1Solution->next;

h1Solution = h1Solution->next;

temp = NULL;

free(temp);

}

}

if (h2Solution) {

while (h2Solution) {

SolutionList \*temp = h2Solution->next;

h2Solution = h2Solution->next;

temp = NULL;

free(temp);

}

}

h1Solution = NULL;

h2Solution = NULL;

}

//@@ reload the same problem\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@@

void reloadProblem() {

for (int i = 0; i < 9; i++) {

values[i] = initValues[i];

if (values[i] != 0) {

GLfloat xPos, yPos;

switch (i) {

case 0:

xPos = 0;

yPos = 200;

break;

case 1:

xPos = 100;

yPos = 200;

break;

case 2:

xPos = 200;

yPos = 200;

break;

case 3:

xPos = 0;

yPos = 100;

break;

case 4:

xPos = 100;

yPos = 100;

break;

case 5:

xPos = 200;

yPos = 100;

break;

case 6:

xPos = 0;

yPos = 0;

break;

case 7:

xPos = 100;

yPos = 0;

break;

case 8:

xPos = 200;

yPos = 0;

break;

}

boxCordinates[values[i] - 1].x = xPos;

boxCordinates[values[i] - 1].y = yPos;

}

else

currentPosition = i;

}

}