601.220 Intermediate Programming

Spring 2023, Day 18 (March 3rd)

Today's agenda

- Overview of midterm project
- Work on midterm project
    - This is a good chance to ask questions!

Announcements/reminders

- Midterm project is due Friday, March 17th by 11pm
    - Late submissions will not be accepted
- HW4 due this evening by 11pm (late submissions not accepted)

Midterm Project: 15-Puzzle

You will implement a program to simulate a “15-puzzle” (or similar puzzle of different dimensions.)

The puzzle is an N by N grid of tiles (where $N > 1$) with one tile missing. The goal is to slide the tiles until they are all in the correct positions.

Example (scrambled)

5	7	1	2
⬛	9	3	4
13	8	6	11
14	15	10	12

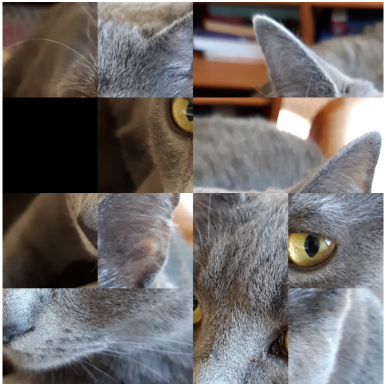

Figure 1: Scrambled 15-puzzle

Example (solved)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	■

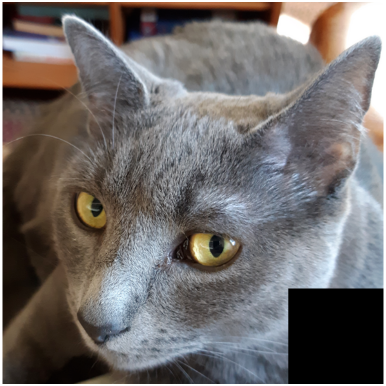

Figure 2: Solved 15-puzzle

Moving tiles

The player can slide one of the tiles adjacent to the missing tile, a.k.a. the “gap”.

If the gap is at the edge of the puzzle then a move won’t be possible in every direction.

Example moves

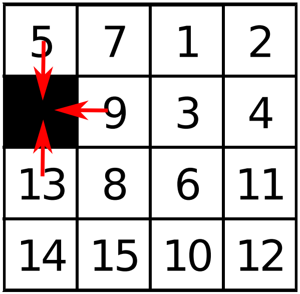

Figure 3: Example valid moves

In this example, the tiles above, right, and below the gap can be moved (respectively) down, left, and up.

There is no tile to the left of the gap, so a move to the right is not possible.

Representing the puzzle

You will define a `struct` data type to represent the puzzle:

```c
typedef struct {
  // ...fields go here...
} Puzzle;
```

You should declare and implement functions to allow the program to create, destroy, and work with instances of the `Puzzle` type. E.g.,

```c
Puzzle *puzzle_create(int size);
void puzzle_destroy(Puzzle *p);
void puzzle_set_tile(Puzzle *p, int col, int row, int value);
int puzzle_get_tile(const Puzzle *p, int col, int row);
// etc...
```

Program Input, Commands

The input to the program is a sequence of commands (read either from `stdin` or from a file.)

Command	Description
C *size*	create puzzle of specified size
T *tiles*	set initial state of puzzle
I *filename*	set the base image filename
P	print current state of the puzzle
W *filename1 filename2*	write image and puzzle representation files
S *dir*	slide free tile in given direction (u/d/l/r)
K	check whether puzzle is solved
V	attempt to find moves to solve the puzzle
Q	quit the program

Reading and Writing Images

The I command reads a background image. The background image is used by the W command to write a puzzle image (based on the current puzzle configuration.)

Code is provided to read and write PPM image files:

- `ReadImage` reads PPM image data from a file, and returns a dynamically allocated instance of the `Image` type, storing pixel color information in the `Image`'s `data` array
- `WriteImage` writes PPM data to a file based on the pixel data in an `Image`'s `data` array

Pixel and Image Data Types

```
typedef struct _pixel {
  unsigned char r;
  unsigned char g;
  unsigned char b;
} Pixel;

typedef struct _image {
  Pixel *data;  // pointer to array of Pixels
  int rows;     // number of rows of Pixels
  int cols;     // number of columns of Pixels
} Image;
```

Pixels are stored in the `data` array in row-major order (first the top row of pixels from left to right, then the next row of pixels from left to right, etc.)

Dynamic Allocation

You will need to use dynamic memory allocation for the representation of the `Puzzle` data type. The `Image` data type also uses dynamic allocation, for the `Image` instance and (importantly) the array of `Pixel` elements.

Make sure your program properly deallocates dynamic memory before the program exits, and make sure that you use `valgrind` to ensure that there are no memory errors.

Solving the Puzzle

The “V” command should attempt to determine a sequence of moves to solve the puzzle. Because this could require exponential time, we will only expect this to work for 3x3 puzzles.

The suggested approach is to use a backtracking search. Starting from the current puzzle configuration, for each possible move, see whether that move would recursively lead to a solution to the puzzle.

Teamwork!

You and your partner will be working collaboratively. Suggestions:

- Communicate frequently
- Plan times when you can meet and work together
- Pair programming: write the code together
- Add, commit, and push your work frequently
- Don't forget to `git pull` frequently to get your partner's most recent commits

We recommend that you do not rigidly divide up tasks (i.e., "you do A, B, C, I'll do X, Y, Z"). This approach tends to result in a lot of waiting for the other person to finish a task before progress is possible on a different task.

Testing

Two modes of testing:

- Run your program on the example inputs, verify that the output (to `stdout` and output files) matches the expected outputs
- Write unit tests for functions

Both of these approaches are useful. See the project description for details.

Notes

Notes

Notes

Notes

Notes