Project Maze: Solver

Due: Wednesday, September 24, 2014

1 Introduction

Baby DJ is extremely pleased with his maze. He proudly shows it to all of the other members of his playgroup. One day, he went to get his favorite blocks, but realized he no longer remembered the way! Indeed, he has not yet even developed episodic memory, which presents a serious challenge in remembering the way to his beloved blocks.

Thankfully for him, young DJ is an algorithmic prodigy, and is certain that—with the help of a grown-up—he can devise a strategy to reunite him and his blocks. Use your C prowess to give him a hand!

2 Assignment

For the second half of the Maze assignment you will write a maze *solver*, which solves the mazes you generated in last week's project. Your solver program should take six arguments:

```
./solver <input maze file> <output path file> <starting x-position>
    <starting y-position> <ending x-position> <ending y-position>
```

Your solver should output its solutions to the specified file. As before, your code should be written to handle mazes of varying dimensions, but again you will use 10×25 mazes for this assignment — that is mazes with 10 rows and 25 columns. The upper-left corner will have coordinates $(0, 0)$ and the bottom-right corner of the maze will have coordinates $(24, 9)$.

Some of the information you may need for this assignment is in the Maze: Generator handout – hexadecimal representation of a maze, for example.

To get started, run

```
cs033_install maze_solver
```

to copy the stencil for this project into your home directory (*~/course/cs033/maze_solver*).

3 Translating Between Representations

In your `generator` you translated from the program represenation to the file representation. For your `solver` you must translate from the file representation to the program representation.

3.1 From File to Program

Before you can perform any operations on the hexadecimal character from a file, you must properly convert it back into an integer. Fortunately, you can combine both of these steps into one by using

the `fscanf()` function (described below) with the format string `"%1x"`.

Once you have the hexadecimal number back in integer form, use bit-level operations to extract the room data. Remember:

- the highest-order bit[1] represents the east connection;
- the next-highest bit represents the west connection;
- the next-lowest bit represents the south connection; and
- the lowest-order bit represents the north connection.

To extract a particular element, use a *bit mask*, or an integer whose binary representation consists entirely of zeroes except for a particular bit (or bits). For example, the C expression `x & 0x4` returns the third bit from an integer `x`.

Consult the Maze: Generator handout for more information on room representation.

4 Input and Output

In Generator, you learned how to write to a file. For Solver, you will need to read from a file.

4.1 Reading From a File

`<stdio.h>` defines many functions that you can use to read from files. The function that you will likely find most useful for this project is `fscanf()`.

```
int fscanf(FILE *stream, char *format, ...)
```

`fscanf()` is to `scanf()` as `fprintf()` is to `printf()`: it does the same thing as its counterpart function, but with a `FILE *` as the source of the input or output operation. To receive a hexadecimal number from a hexadecimal character, use the format string `"%1x"`, which will scan a single character into an `unsigned int` variable.

Consult the Maze: Generator handout for information about opening and closing files.

5 Solver

Your Solver program should solve a maze defined by the given representation, outputting to a file either the path to the exit or the order of its search (see Solution Output, below).

There are several ways to solve a maze. Your program should employ a *depth-first search*. Such a search begins at the maze's start room and explores adjacent, accessible rooms recursively.

[1] The highest-order bit of the hexadecimal number, *not* the highest-order bit of whatever C data type you use to store it.

```
dfs(x, y):
    if x, y are the coordinates of the goal
        return true
    mark the room at [x][y] as visited
    for each direction dir:
        neighbor = rooms[x + x-offset of dir][y + y-offset of dir]
        if the connection in direction dir is open and neighbor is unvisited:
            if dfs(neighbor.x, neighbor.y) returns true
                return true

    /* if the program reaches this point then each neighbor's branch
       has been explored, and none have found the goal. */

    return false
```

Beginning with the indicated room, this algorithm repeatedly chooses a path from each room and follows that path until it reaches a dead end, at which point it backtracks and tries a new path. This process continues until all paths have been explored or the destination is found.

5.1 Solution Output

We expect your solver to produce two different modes of output:

- Your program outputs the coordinates of only the final route from beginning to end. Your program should first print the line "`PRUNED`". The program should then print the coordinates of each room on the solution path.

  To do this, build a list of rooms as you search, and print out each room in the list when you reach the destination room. Programatically, you can accomplish this using pointers! Here are two good (and similar) solutions:

  - add a `next` pointer to your room structures. Use these pointers to maintain a linked list of rooms - when you move from room A to room B, set room A's pointer to room B.
  - define a `struct linked_list` and pass around a pointer to the last node of your list in your recursive `solve` function. When you visit a room, initialize a new list node and append it to the end of the list, using the pointer to the previous node.

- Your program outputs the entire path traversed up until the goal is reached. Your program should first print the line "`FULL`". The program should then print each room's coordinates when first visiting that room, and after each recursive call that returns false. This will print the path from start to finish, including "backtracking" after dead ends.

The choice should be made when your program is compiled. This is done using preprocessor macros.

```
#ifdef FULL
printf(<something>);
#else
printf(<something else>);
#endif
```

The above code fragment executes the `printf(<something>)` statement only if the macro `FULL` is defined, and executes the `printf(<something else>)` statement otherwise. You can toggle this definition by using the `gcc` compiler flag, `-D<macro>`, which defines `<macro>` for the preprocessor. You can also use the macro `#ifndef <macro>` to execute code only if `<macro>` is not defined.

Your program should print the entire search if a macro `FULL` is defined and print only the path to the exit otherwise. Rooms should be printed with format `x, y` on its own line with no parentheses with the upper-left corner of the maze corresponding to coordinate `(0, 0)`. If the start and end rooms happen to be the same, your output should contain the room only once.

6 Compiling and Running

As with Generator, you've been provided with a Makefile to compile your program for you. This Makefile contains two targets:

- `solver`, which builds your solver program with no macros defined (i.e. it should print pruned output).
- `solver_full`, which builds your solver with the `FULL` macro defined (i.e. it should print its full exploration path).

You can build each target separately, or run `make all` (or just `make`) to build both.

If you want to add extra files to your Solver, add them to the file list defined by `SOL_OBJS`.

Once you have compiled your solver, you can run it with the command

```
./solver <input maze file> <output solution file> <starting x-position>
    <starting y-position> <ending x-position> <ending y-position>
```

or

```
./solver_full <input maze file> <output solution file> <starting x-position>
    <starting y-position> <ending x-position> <ending y-position>
```

7 Support

As with Generator, you are provided with some programs that will assist with this project:

- `cs033_solver_demo <maze file> <output file> <start_x> <start_y> <end_x> <end_y>`

  This is a demo program for the Solver project; it produces pruned output, i.e. just the path from the start coordinate to the end coordinate.

- `cs033_solver_full_demo <maze file> <output file> <start_x> <start_y> <end_x> <end_y>`

  This is a demo program for a Solver's full output. It produces a list of rooms in the order in which they were explored.

- `cs033_maze_validator <maze file> <solution file> <start_x> <start_y> <end_x> <end_y>`

  This program will also validate a solution file (if you provide it one) for a maze. Make sure the end-point coordinates correspond to those given to your Solver.

- `cs033_maze_vis <maze file> <solution file> <speed>`

  This program will visualize your solution path in the provided maze. The `<speed>` argument to `cs033_maze_vis` must be an integer between 1 and 3, inclusive; higher values make Baby DJ move faster. If either the maze or the solution file is improperly formatted, the visualizer will exit.

  This program will also check your input files for correctness, and will print out a list of errors if problems are found. If there are problems in the maze file, the viewer will still be launched, giving you the opportunity to find the errors in your maze graphically. However, if problems are found in a solution file, the `cs033_maze_vis` script will terminate.

8 Grading

Your grade for this project will be composed as follows:

Functionality	40%
Code Correctness	40%
Style	20%
Total	100%

- **Functionality**: your code produces correct output, performs error checking on its input, and does not crash for any reason. It does not terminate due to a segmentation fault or floating point exception.
- **Code Correctness**: no part of your code relies on undefined behavior, uninitialized values, or out-of-scope memory; your program compiles without errors or warnings.
- **Style**: your code should look nice! Use appropriate whitespace and indentation and well-named variables and functions. Your code should be reasonably factored, and functions should not be too long.

Your programs should perform error checking on their input, with one exception: if your program successfully opens a maze file, you may assume that the file contents form a correctly-formatted maze. Your program should not crash for any reason; before you hand in your project, make sure that your program does not terminate due to a segmentation fault or floating point exception.

Consult the C Style document (which is on the website) for some pointers on C coding style.

9 Handing In

To hand in your project, run the command

```
cs033_handin maze_solver
```

from your project working directory. You should hand in your source code, a Makefile and a README; you need not include any other files in your handin. Your README should describe the organization of your program and any unresolved bugs.

If you wish to change your handin, you can do so by re-running the script. Only your most recent handin will be graded.