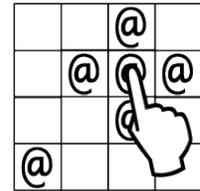# 15-122: Principles of Imperative Computation, Spring 2016

Programming Homework 10: Lights Out

Due: Sunday 10th April, 2016 by 22:00

In this programming assignment we will play a simple computational game: *Lights Out*. Your Lights Out solver will be a C program where you write the `main()` function from scratch. We will give you a helpful set of libraries, and you'll write some other libraries yourself. Make sure to look at the libraries we gave you before you start working on your Lights Out implementation in Task 5!

The code handout for this assignment is at

`qatar.cmu.edu/~srazak/courses/15122-s16/hws/lightsout-handout.tgz`

The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a *10 handin limit* for this assignment. Additional handins will incur a half-point penalty per handin.

For this assignment, it is permissible (and encouraged!) to share game boards (test input) that you're using, as long as you do this sharing publicly on Piazza.

Task 1 (0 points) This assignment will **not** be graded for style.
However you will still find it helpful to develop good style habits: reasonable contracts, at most 80-character lines, and comments that make it clear to a reader how your algorithm works and what invariants you expect to hold. You should use the libraries provided for you to make your code simpler and clearer. We expect you to write your own helper functions when appropriate. Bad style will have no direct effect on your grade but will make your life harder.

The Lights Out Game

Lights Out is an electronic game consisting of a grid of lights, usually 5 by 5. The lights are initially in some pattern of *on* and *off*, and the objective of the game is to turn all the lights off. The player interacts with the game by touching a light, which toggles its state and the state of all its cardinally adjacent neighbors (up, down, left, right).
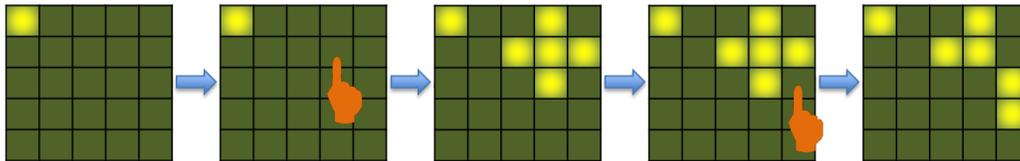
We represent boards as a vector of bits (see Section 1). When printing boards in ASCII we represent a square that is "on" (represented by the bit `1` or `true`) with a '`#`' and a square that is "off" (represented by the bit `0` or `false`) with a '`O`'. A touch is described by a 'row:column pair' as in previous assignments. On Andrew, we have provided an executable file, `loplayer`, that allows you to play Lights Out. The text in italics below represents the part that you would type in during this session.

```
% loplayer boards/board0.txt
#OOOO
OOOOO
OOOOO
OOOOO
OOOOO
1:3
Flipping 1:3
#OO#O
OO###
OOO#O
OOOOO
OOOOO
2:4
Flipping 2:4
#OO#O
OO##O
OOOO#
OOOO#
OOOOO
```

You can exit `loplayer` by typing in something invalid, by pressing Ctrl-D, or by winning the game and turning all the lights out.

1 Bit Vectors

The `pixel` type used a single 32-bit integer to efficiently store four 8-bit numbers. A single 32-bit integer could, by the same principle, store up to 32 true/false values. That is what the bit vector implementation that you will be implementing does. Bit vectors are an abstraction that assists us in efficiently manipulating small sequences of bits. We'll use these sequences of true/false bits to store the on/off squares in a Lights Out board.

The function `bitvector_new()` gives us a fresh bit vector containing only `false` bits, the function `bitvector_get(B, i)` tells us whether the `i`th bit of the vector is true, and the function `bitvector_equal(B1, B2)` returns `true` if all the bits in `B1` and `B2` are the same. Obviously, we could implement `bitvector_equal` by calling `bitvector_get` in a loop, but we expect that the library may be able to implement a more efficient equality check.

The function `bitvector_flip(B, i)` returns a new bit vector that is just like `B` except that the ith bit is flipped. Unlike C and C0 arrays, bit vectors are a persistent data structure: when we flip a bit in a bit vector, the old bit vector stays the same and we return a new bit vector with the bit toggled.

Task 2 (4 points) Implement the `lib/bitvector.h` interface in a file called `bitvector.c`.

This task is all you need to do in order to successfully begin working on your Lights Out implementation. However, there are two additional challenges for your bit vector implementation. First, notice that the macro-defined constant `BITVECTOR_LIMIT` is set to be 25 in your version of `lib/bitvector.h`. Given that the type `bitvector` is a 32-bit unsigned integer, the constant `BITVECTOR_LIMIT` could just as easily be set to 8 or 32 (but not 33 or 42).

Task 3 (1 point) Make your implementation of `bitvector.c` work correctly regardless of what `BITVECTOR_LIMIT` is. You should assume that this limit is greater than 0 and less than or equal to the number of bits in the type `bitvector`.

The type `bitvector` is defined in `lib/bitvector.h` to be a 32 bit unsigned integer. But if we need bit vectors of length 33 or 42, it should be possible to use a larger unsigned integer type, and if we only need bit vectors of length 7, it should be possible to use a smaller unsigned integer type.

Task 4 (1 point) Make your implementation of `bitvector.c` work correctly regardless of what unsigned integer type `bitvector` is defined to be. You may assume that whatever unsigned integer type `bitvector` is defined to be has at least `BITVECTOR_LIMIT` bits in its representation.

Using Bit Vectors

Remember that, in the pixels assignment, we gave you a separate implementation of pixels that stored these four numbers in an array with length 4. It's similarly possible to use either a 32-bit integer or a **bool** array with length 25 to store 25 true/false values. Because we are

using C, an array implementation would presumably lead to unrecoverable memory leaks, because we don't expose a `bitvector_free` function.

We will compile your code that uses bit vectors against such other implementations, including an implementation that is a **bool** array. Aside from memory leaks, your code for tasks 4-6 should work with these other implementations. In other words, for the remainder of the assignment, you should respect the `bitvector` interface: you shouldn't assume you know anything about how a bit vector is implemented.

2   Hash Tables

The algorithm we describe for solving Lights Out is going to use hash tables which will store Lights Out boards that have already been looked at — this prevents you from doing computation for them again. We provide you with the generic version of automatically-resizing, separate-chaining hash dictionaries as described in `lib/hdict.h` — you do *not* have to implement hash tables from scratch.

However, instantiating the client interface and casting void pointers correctly in C is error-prone. For this task, you will implement, in `board-ht.h` and `board-ht.c`, a wrapper around the `hdict` interface that handles the tricky parts.

Specifically, the functions you will have to implement are:

- `ht_new`, which creates a new, initially empty, hashtable and correctly instantiates the client interface.

- `ht_insert`, which adds a new **struct** `board_data` to the table.

- `ht_lookup`, which looks up structs in the hash table based on the key, a `bitvector` stored in the `board` field of the struct.

When the `ht_insert` function is given a struct `DAT`, it needs to perform an insertion into the hash table using `hdict_insert`. The **value** stored in the hash table should be a pointer to the struct `DAT` that is passed to `ht_lookup`. The **key** should be a pointer to the `board` array of the same struct; you'll need to take the address of `DAT->board` to get this pointer.

There is no `ht_free` because it would work the same way as the free function provided by the `hdict` interface. Pointers added to the hash table with `ht_insert` are owned by the table, and should be freed when `hdict_free` is called

Task 5 (5 points)   Implement the `lib/board-ht.h` interface in a file called `board-ht.c`.

You're not required to use these hash tables in your Lights Out implementation, but we certainly suggest you do! Whether you use them or not, your implementation of the interface will be autograded. The elements in the hash table are pointers to **struct** `board_data`, a struct with two fields.

*You can edit **board-ht.h** to add any fields that you want to the **struct board_data**, but the autograder will require that the original two fields remain the first two fields in the struct.*

3   Lights Out

In this section, you will implement a solver that can decide whether or not a board is solvable, and that describes how to solve boards that can be solved.

A reference solution, which is about as efficient as your solution needs to be, can be found on Andrew with the name `lightsout-ref`.

```
% lightsout-ref boards/3x2-34.txt
Here's the board we're starting with:
#00
##0

Board is solvable.
Solution bitmap (the squares marked # need to be pushed to solve):
000
#00

% lightsout-ref boards/3x2-32.txt
Here's the board we're starting with:
#0#
#00

No solution was found!
```

Your implementation of a solver for Lights Out in `lightsout.c` should produce an executable that takes one command-line argument, the filename of a file containing a board which can be read with the `file_read` function from `lib/boardutil.h`. **The output from your solver will be a bit different than the output from the reference solution; read the instructions carefully.**

To take command-line arguments in C, you need to write a `main` function with two arguments, `argc` (*argument count*, the number of command-line arguments) and `argv` (*argument values* an array of strings, the actual command-line arguments). Our implementation begins like this:

```c
int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "Usage: lightsout <board name>\n");
    return 1;
  }
  char *board_filename = argv[1];
  ...
}
```

If you want to print out debugging information or error messages in your implementation, you should use `fprintf(stderr,...)` instead of `printf(...)` the way we do above. Do not use `printf`. Using `fprintf(stder,...)` ensures that all error messages or debugging

outputs go to *standard error*, which will be critical for Task 6. (Output from failed contracts automatically goes to standard error.)

3.1   Solving Lights Out

The return value of `main` in a C program is treated as meaningful to the operating system. Returning 0 means the program ran successfully and returning anything else means the program ran unsuccessfully. For your Lights Out solver, the `main()` function in your implementation must return `0` if the board can be solved, and should return `1` if there was an error **or** if the Lights Out board cannot be solved. A C program will not print out the returned integer like a C0 program does, so you probably want to print a helpful "board could be solved" or "board could not be solved" message to standard error.

Task 6 (5 points)   Implement a Lights Out solver in `lightsout.c` that takes a board and reports whether or not the board is solvable by returning `1` or `0` from the `main()` function.

Any valid implementation of Lights Out with reasonable performance will get points. You're welcome to be creative, but any creative idea you implement should be your own. (There are academic papers on the mathematics of Lights Out and its solutions, including a 1989 paper by the computer science department's own Klaus Sutner! But for this assignment, if you do something different than the strategy outlined in this writeup, it should be your own idea.) We suggest that you first implement the *breadth-first search* algorithm outlined in Section 3.4. But if you want a challenge, you're encouraged to try and figure out an algorithm on your own.

Your implementation should be free of memory leaks (as reported by `valgrind`) *regardless* of whether your `main` function returns `0` or `1`. Your implementation should also work regardless of the way bit vectors are implemented.

For this assignment, we have given you a `Makefile` to help you build your solver. If you type `make` after writing `bitvector.c`, `board-ht.c`, and `lightsout.c` it will compile two executables, `lightsout` and `lightsout-d` (the latter is compiled with `-DDEBUG`). You can change your Makefile to add new targets (for instance, to run unit tests for your bit vectors).

3.2   Returning the Solution

Task 7 (4 points)   Modify your Lights Out solver so that, when it is given a solvable board, it prints out, to standard output, the series of button pushes that solve the board.

The output of your program should not be like the reference solution's output. Instead, if a solution is found, all the moves leading to that solution should be printed, in order, to *standard output* (or `stdout`, which is what `printf` prints to). The output should be able to be re-routed into `loplayer` as input that solves the board. See Section 3.3 for an example. (If no solution is found, nothing should be printed to the standard output, and it's a good idea to print "No solution" to `stderr`.)

You may want to add fields to **struct** `board_data` to do this task.

An examination of the puzzle leads to interesting observations — changing the state of a square an even number of times is equivalent to not changing it at all; changing the state an

odd number of times is equivalent to changing it only once. Furthermore, the order in which we touch various squares is unimportant. It is only the number of times that we touch a square that matters. These facts imply that, if a puzzle can be solved at all, it can be solved by touching some squares exactly once and others not at all. Thus, a solution consists of indicating which squares to touch once.

Therefore, one way to complete this task is to store boards in the hash table alongside a bit vector representing *all* the moves needed to produce that board. Another option is to store boards along with the index of the move that got you to that board. Then you can find the solution by working backwards. When you notice you have a board with all lights out, re-apply the last move to get the previous board. Then look up that previous board in the hashtable, which will also give you the move that allowed you to reach *that* board, and so on and so forth until you get back to your original board.

3.3 Testing

You'll want to be sure to test your solver on boards that aren't symmetric and examples that aren't squares. A good way to test your program is to use a Unix *pipe*, redirecting the standard output of your Lights Out solver to the standard input of the `loplayer` program:

```
% ./lightsout boards/3x2-34.txt
1:0
% ./lightsout boards/3x2-34.txt | loplayer boards/3x2-34.txt
#00
##0
Flipping 1:0
000
000
You got all the lights out!
```

When testing your solution on the boards we provide, be aware that if you have good contracts, your code may be too slow to solve the harder boards in reasonable time. Once you are confident of the correctness of your code, run your tests on `./lightsout`, not `./lightsout-d`.

Every 2x2 board and an assortment of 3x2 and 4x4 boards are distributed with the handout. **When compiled without -D,** your solution should be able to solve all the 2x2 boards (or report no solution) instantly and do the same for any 3x2 and 4x4 board in seconds at most. The included 5x5 boards may be too challenging for your implementation, but you should be able to search any 5x5 board that takes less than 6 touches to finish without much difficulty. *You can share test boards and solutions to individual boards on Piazza.*

3.4 Suggested Algorithm: Breadth-First Search

The algorithm we will sketch out here for solving Lights Out is called *breadth-first search*. The basic version of breadth-first search uses a queue. We start with just the initial board in the queue, and then we begin a loop. As long as the queue is not empty, one iteration of the loop removes a board from a queue, computes the effect that each of the 25 possible button-pushes will have on the board, and then inserts all 25 modified boards back onto the queue. If we do this very naively, the algorithm will first consider the one board that we can get to with zero touches, then the 25 boards we can get to with one touch, then the 625 boards we can get to with two touches, then the 15,625 boards we can get to with three touches, and so on.

This approach will always find a solution if one exists, but it is very wasteful, because we can get to the rightmost board on page 2 in two different ways: by touching the square in row 1, column 3 and then the square in row 2, column 4 and also by touching the square in row 2, column 4 and then by touching the square in row 1, column 3. The naive algorithm described in the previous paragraph unnecessarily considers both of these possibilities separately.

We solve the problem more efficiently with a hash table. When we start the loop, the initial board is also present in the hash table. Inside the loop, we can compute all 25 possible moves but we *only* enqueue and add to the hash table the ones that we haven't previously considered. Here is pseudocode for the resulting algorithm:

```
while (!queue_empty(Q)) {
  // Find a board that we haven't looked at yet from the queue
  B = deq(Q);

  // Consider all the moves
  for (row = 0; row < height; row++) {
    for (col = 0; col < width; col++) {
      i = get_index(row, col, width, height);
      bitvector newboard = press_button(...)

      if (number of lights of newboard == 0) {
        Free all memory, return 0
      }

      if (hash table H doesn't contain newboard) {
        Allocate memory for hashtable element N
        Set the field N->board to newboard, set other fields
        Insert N into the hashtable H
        Enqueue N into the queue Q
      }
    }
  }
}
Free all memory, return 1
```