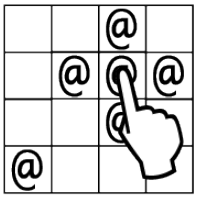
15-122: Principles of Imperative Computation, Fall 2014

Homework 12 Programming: Lights Out



Due: Thursday, November 20, 2014 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 3!

The code handout for this assignment is at

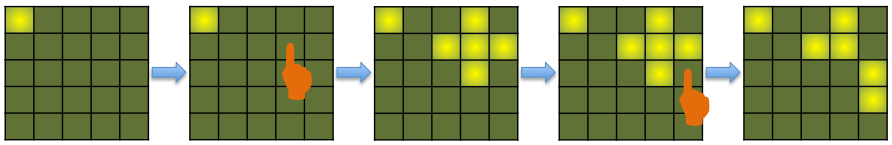
`http://www.cs.cmu.edu/~rjsimmon/15122-f14/prog/lightsout-handout.tgz`

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.

Task 0 (0pts) *This assignment will be graded for style. You should use reasonable contracts, have at most 80-character lines, and have comments that make it clear to a TA 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. Violations of style will cause your grade for this assignment to be zeroed out until you fix the issues described and review your code with a TA.*

The Lights Out Game

Lights Out is an electronic game consisting of a grid of lights, usually 5 by 5. The lights are initially pressed 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 an array of bits (see section 1). 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 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 arrays

Bit arrays are an abstraction that assists us in efficiently manipulating small arrays of bits. The function `bitarray_new()` gives us a fresh bit array containing only `false` bits, the function `bitarray_get(B, i)` tells us whether the `i`th bit of the array is `true`, and `bitarray_flip(B, i)` toggles the `i`th bit in the bit array. Unlike C and C0 arrays (but like pixels and ropes), bitarrays are a persistent data structure: when we flip a bit in a bit array, the old bitarray stays the same and we return a new bit array with the bit toggled.

The bit array interface file `lib/bitarray.h` constrains *both* the client and the library. The interface specifies that bit arrays must be unsigned integer types, and that these unsigned integers must contain at least `BITARRAY_LIMIT` bits.

When you are acting as the implementer of bit arrays by writing `bitarray.c`, respecting the interface means that you can't assume you know precisely what type a `bitarray` is. The type `bitarray` is defined in `lib/bitarray.h`, and your implementation in `bitarray.c` should assume only that this type is defined to be an unsigned integer type that contains at least `BITARRAY_LIMIT` bits. The `lib/bitarray.h` interface could change to define `bitarray` as another type and/or define `BITARRAY_LIMIT` to be a different positive number, and your bitarray implementation should still work. Because it might be undefined behavior to shift by more than `BITARRAY_LIMIT` bits, your contracts will have to mention this macro-defined constant.

When you are acting as the client of bit arrays, respecting the interface means using `BITARRAY_LIMIT` instead of assuming that its value is 25, 32, 8, or some other constant.[1] It also means *only* using `bitarray_new` to get an empty bitarray and *only* using `bitarray_get` and `bitarray_flip` to access individual bits in the bit array. We will compile your Lights Out implementation against bit array implementations that store bits in a different order than your implementation stores them. We will even compile your Lights Out implementation against bit array implementations where `bitarray_new()` does not return `0`. (Your implementation should probably return `0`.)[2] Your code that uses bitarrays must work even when given these strange implementations.

Clients *can* compare bit arrays `B1` and `B2` for equality using `B1 == B2`, and they can cast bitarrays to the unsigned type `size_t` and perform mathematical operations to the result. Both of these will be useful when you need to store bit arrays as the keys to a hashtable. (If it happens that `BITARRAY_LIMIT` is larger than the number of bytes in a `size_t`, then the cast from `bitarray` to `size_t` would truncate the bitarray to make it fit. This is not something you need to worry about here: even if it happened, it wouldn't be the end of the world for a hash function to behave this way.)

Our implementation of the `lib/boardutil.h` library function `file_read` checks that the board being read from a file has no more than `BITARRAY_LIMIT` bits, so this part of the client's responsibility is handled for you.

Task 1 (6pts) *Implement the* `lib/bitarray.h` *interface in a file called* `bitarray.c`.

[1] The `file_read` function we give you in `lib/boardutil.h` uses `BITARRAY_LIMIT` to fail if your Lights Out board is too big to fit in a `bitarray`.

[2] A puzzle to think about: how could you implement the bitarray interface if `bitarray_new` returned `~0` instead of the obvious value of `0` that your implementation will probably use?

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. In this part of the assignment, you are going to implement a wrapper around the generic hash tables implementation from lecture specifically for storing and looking up the bit arrays you implemented in the previous part. You're not required to use these hash tables in your Lights Out implementation, but we do suggest it!

In `lib/ba-ht.h`, you have been given the interface for the hash table for bit arrays that you will be implementing in the file `ba-ht.c`. Note that you do not have to implement hash tables from scratch. We provide you with the generic version of automatically-resizing, separate-chaining hash tables as described in `lib/hset.h`. Specifically, the functions you will have to implement are:

- `ht_new`
- `ht_insert`
- `ht_lookup`
- `ht_free`
- Any other client interface functions for underlying hash table

Note that this hash table stores as its element a pointer to `struct board_data` which has two fields. The field `board` is the current board. The field `last_move` is the index of the button that was pressed to get to the current board `board`. You should write your implementation so that the hash table owns the structs that have been stored in the table, and accordingly the hash table should free those structs when the hash table is freed.

Task 2 (5pts) *Implement the* `lib/ba-ht.h` *interface in a file called* `ba-ht.c`.

3 Lights Out

Your implementation of a solver for Lights Out in `lightsout.c` should produce an executable that takes one command-line argument, the name of a board, and writes out to standard output *only* the moves necessary to successfully turn the lights out if the board has a solution. This means 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:

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

There are two important things to note. First, 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. *Your implementation should return* `0` *if a solution was found, and should return* `1` *if there was an error or if the Lights Out board cannot be solved.* Second, because we don't want to print anything to standard output that isn't a valid move, we print out the "You didn't give us a board!" error message to *standard error* instead. If you want to print out debugging information in your implementation, it would be a good idea to use `fprintf(stderr,...)` instead of `printf(...)`, because the latter prints to standard output.

Task 3 (9pts) *Implement your Lights Out solver in* `lightsout.c`*.*

Your implementation should be free of memory leaks (as reported by `valgrind`) *regardless* of whether your `main` function returns `0` or `1`.

For this assignment, we have given you a `Makefile` to help you build your solver. If you type `make` after writing `bitarray.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 arrays).

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 above, it should be your own idea.) We suggest that you first implement the *breadth-first search* algorithm outlined on the next two pages. If you want more of a challenge, you're welcome to try and figure out an algorithm on your own, but in that case you should still take a look at the testing instructions on the last page.

3.1 Suggested Algorithm: Breadth-First Search

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.

The algorithm we will sketch out here for solving Lights Out is called *breadth-first search*, and uses a hash table and a queue. The algorithm works like this: we start with just the initial board in the queue, and then 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 25 boards we can get to with one touch, then 625 boards we can get to with two touches, then 15625 boards we can get to with three touches. . .

This approach will always find a solution if one exists, but it is very wasteful, because we can get to the rightmost board in the introduction 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 above described in the previous paragraph unnecessarily considers both of these possibilities separately.

One way to solve the problem more efficiently is by using a hashtable to store all the boards we have already seen. Then, inside the loop, we can compute all 25 possible moves but only enqueue (and add to the hashtable) the ones that we haven't previously considered.

A challenge in this assignment will be figuring out what data you need to store in your queue and what data you need to store in your hashtable. Your ba-ht implementation allows you to store a bitarray and an additional integer, which represents the index of the previous move. The keys are pointers to the board state that the entry represents. Once you find a winning solution, you must also recover the moves you used to get to that board. You can do this by working backwards: When you notice you have a board with all lights out, work backwards: 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.

3.2 Pseudocode

Here's another presentation of the algorithm we sketched out on the previous page:

```
H is a ba-ht containing only the starting board
Q initially is a queue containing only the starting board

// H contains all board states which have already been
// added into the queue at some point

// Q contains board states which have not been processed

while(!queue_empty(Q)) {
  // Find a board that we haven't looked at yet
  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);
      bitarray newboard = press_button(...)

      if (number of lights of newboard == 0) {
        Compute and print solution
        Free memory
        return 0
      }

      if (hashtable H doesn't contain newboard) {
        Allocate memory for hashtable element N
        Set last move to i
        Set current board to newboard
        Insert N into the hashtable H
        Enqueue N into the queue Q
      }
    }
  }
}
Free Memory
return 1
```

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/3x3-5.txt | loplayer boards/3x3-5.txt
O##
#OO
#O#
Flipping 0:2
OOO
#O#
#O#
Flipping 2:0
OOO
OO#
O##
Flipping 2:2
OOO
OOO
OOO
You got all the lights out!
```

Every 2x2 board and an assortment of 3x2 and 4x4 boards are distributed with the handout. Your solution should be able to solve all the 2x2 boards instantly and solve any 3x2 and 4x4 board in seconds at most. The included 5x5 boards included may be too challenging for your implementation, but you should be able to solve 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.*