15-122 : Principles of Imperative Computation

Summer 1 2012

Assignment 5: Lights Out!

(Programming Part)

Due: Friday, June 15, 2012 by 23:59

For the programming portion of this week's homework, you'll implement a solver for a puzzle game called Lights Out. You'll reuse the hash table code that we discussed in class. This code reuse depends on the fundamental idea of abstraction: separating interface from implementation. You will write four C_0 files corresponding to different tasks:

- `board.c0` (described in Section 1.2),
- `bit-array.c0` (described in Section 1.3),
- `client.c0` (described in Section 1.4),
- `lightsout.c0` (described in Section 1.5)

You should submit these files electronically by the due date. Detailed submission instructions can be found below.

1 Assignment: Lights Out! (25 points)

Starter code. Download the file `hw5-starter.zip` from the course website. When you unzip it, you will find the following files

`bit-array.c0`	Bit array	Task 1
`board.c0`	Game board representation	Task 2
`client.c0`	Client-side code for the hash table	Task 3
`lightsout.c0`	Game solver	Task 4
`hw5-main.c0`	The driver program for testing	DO NOT TOUCH
`expected.txt`	Output of the test driver	DO NOT TOUCH
`hash-map.c0`	Generic implementation of a hash map	DO NOT TOUCH
`types.h0`	Typedefs ktype, vtype, and queue_elem	DO NOT TOUCH
`queue.c0`	Generic implementation of a queue	DO NOT TOUCH
`readfile.c0`	Code for reading words from a file	DO NOT TOUCH

You will also see a `boards/` directory with some sample text files you may use to test your code.

Testing your code: Boards from External Input When testing your code, it may be convenient to read starting boards from files. We can represent boards in files: each "off" light is an `0` character and each "on" light is a `#` character, and each row is a single string with no spaces on its own line. This format will be described in greater depth later in the writeup. We have provided a `read_board` function in `board.c0` that reads a board from a file in the format described above.

We have provided the HW5 main in hw5-main.c0. Compile and test with:

`cc0 -d -x hw5-main.c0` (-x immediately executes the compiled file)

This will only work once you've completed all tasks. Please test your individual functions as you work on this programming assignment.

Compiling and running. For this homework, use the `cc0` command as usual to compile your code. Don't forget to test your annotations by compiling with the `-d` switch to enable dynamic checking. **Warning:** *You will lose credit if your code does not compile with* `-d`.

Submitting. Once you have completed some files, you can submit them by running the command

```
handin -a hw5 board.c0 bit-array.c0 client.c0 lightsout.c0
```

The `handin` utility accepts a number of other switches you may find useful as well; try `handin -h` for more information.

You can submit files as many times as you like and in any order. When we grade your assignment, we will consider the most recent version of each file submitted before the due date. If you get any errors while trying to submit your code, you should contact the course staff immediately.

Annotations. Be sure to include appropriate `//@requires`, `//@ensures`, `//@assert`, and `//@loop_invariant` annotations in your program. You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct. **Annotations are part of your score for the programming problems; you will not receive maximum credit if your annotations are weak or missing.**

Style. Strive to write code with *good style*: indent every line of a block to the same level, use descriptive variable names, keep lines to 80 characters or fewer, document your code with comments, etc. We will read your code when we grade it, and good style is sure to earn our good graces. Feel free to ask on the course bboard (`academic.cs.15-122`) if you're unsure of what constitutes good style.

Task 0 (8 points) 8 points on this assignment will be awarded based on annotations and style, so be sure to follow the guidelines given above.

1.1 Lights Out: Overview

Lights Out is an electronic game consisting of a grid of lights, usually 5 by 5. The lights are initially presented in a random 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 of its cardinally adjacent neighbors. It is often quite tricky to see how to solve a given configuration, if it is solvable at all, since the path to the solution may involve seemingly-backward progress as lights must be turned on to point the way towards a final all-off state. Note that the coordinate (x, y) represents the xth column and the yth row, with indices starting at $(0, 0)$.

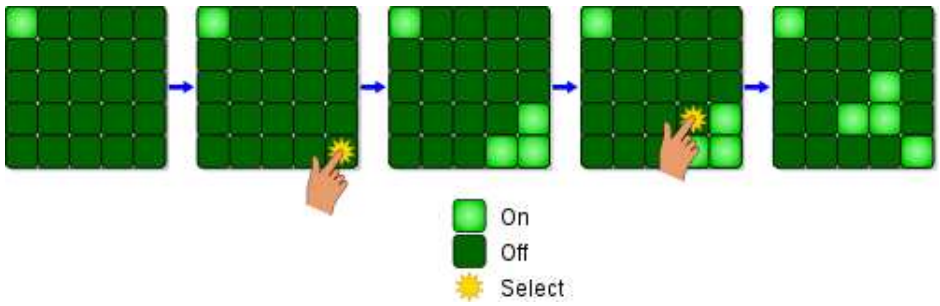

Figure 1: A sequence of moves in Lights Out. (Image: Wikipedia)

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 we touch a square that matters. These facts imply that, if the puzzle can be solved at all, it can be solved by touching some squares exactly once and the others not at all. Thus, a solution consists of indicating which squares to touch once.

A simple solver would proceed by a breadth-first search (using a queue) similar to the word ladder search you implemented in Homework 3, but with one essential difference: sequences of moves are not explicitly stored. In Homework 3, sequences of words were stored in stacks, and once the destination was found, the corresponding stack already contained all intermediate words. In this assignment, the process of finding the search-order predecessor of a game position must be handled externally.

1.2 Representing the Board

We represent a Lights Out board as a bit array along with its width and height.

```
typedef int bitarray;
typedef struct board* board;
struct board {
    int width; // the number of columns
    int height; // the number of rows
    bitarray lights;
};
```

Since the bit array representing the state of the lights is **given as a 32-bit `int`**, it is important that the total area of the board be less than or equal to 32, *an invariant that's checked by the specification function* `is_board`.

1.3 The bit array

The bit array is interpreted as a grid in the usual fashion: position (x, y) of a $w \times h$ grid is b_{wy+x}. For example, if `0` represents a light in the "off" state and `#` represents a light in the "on" state, the board

```
#0#
0##
#00
```

would be represented as 001110101, i.e., 117. We interpret the indices starting from the least significant bit, such that an index i refers to the 2^i bit of the integer when interpreted unsigned.

$$b_{31}b_{30}\ldots b_2b_1b_0$$

We represent the state of the board as an integer rather than as an array of boolean values for two reasons. First, we will soon wish to store the state in various data structures, and we want to be careful not to let any other piece of code change the state after its been stored in a data structure. By convention, arrays are treated as highly mutable, so we would not expect to count on an array not changing unless we made our own separate copy of it. And second, making many copies of an array puts a significant drain on our space performance, which can have a serious impact on run-time performance due to effects like cache locality.

Bit arrays support operations similar to ordinary arrays, but in a *persistent* rather than *mutable* fashion: to "update" a bit array, a function must return a new bit array. The operations of getting the value of a bit, setting a bit to a particular value, and flipping a bit are easily implemented using bitwise operators.

Task 1 (2 pts, `bit-array.c0`) *Implement the interface functions* `bitarray_get`, `bitarray_set`, *and* `bitarray_flip` *as specified in* ***`bit-array.c0`***.

Task 2 (6 pts, `board.c0`) *Implement the interface functions* `is_board`, `board_get`, `board_set`, `toggle_board`, `untoggle_board`, *and* `copy_board` *as specified in* ***`board.c0`***. *Several board functions (printing and reading in from a file) are already provided. Refer to 1.2 for this task.*

1.4 The Hash Table Abstraction

In this section, you'll write the necessary interface code to use a hash table to store the pieces of game state that we'll be tracking.

An essential concept in computational thinking is *abstraction*: the separation of *interface* from *implementation*. When a program is composed of many independent components whose boundaries are mediated by carefully specified interfaces, then the program can be

updated in a modular fashion: at any time, one implementation of a component can be replaced by another without changing the overall meaning of the program, provided that the new implementation adheres to the same interface as the old. Conversely, when interfaces are left unspecified or violated, things can go horribly awry: arguably, some of the most egregious software errors of all time were caused by a careless confusion of interface and implementation.

The hash table code we wrote was completely independent of the actual type of keys provided that they supported the operations of key equality, and key hashing, as well as custom printing. So to actually build a real hash table, the client has to specify an element type and definitions for these operations.

For this assignment we define the actual type of elements as a move to the current state and the location that was toggled to get there. The following data structure will be placed into the hash set, during algorithm execution:

```
typedef struct move* elem;
struct move {
    int x;           //coordinates of the toggle
    int y;
};
```

Notice that one can calculate the previous board for current, by simply undoing the toggle.

Task 3 (2 pts, `client.c0`**)** *Implement the "client code" functions required to use with the hash table:* `move_new`*,* `hmap_ktype_equal`*,* `hmap_hash`*.*

When implementing `hmap_hash`*, be sure to use some form of randomness to "smear" keys uniformly across the table.*

The appropriate abstraction to think of is this: you will begin with the starter board, and generate all boards that are one move from the starter board. Each valid move will be inserted into the hash table (the elements of the hash table are moves).

At some point, while exhaustively searching the set of moves, you will find a board with all the lights out. You then need to find the sequence of moves which got you to that board. All of this data is in your hashtable!

For this reason, the hash key for a move should be *the move's board.*

Recall that we used a pseudorandom number generator, not reproduced here but shown in the lecture notes, to "smear" strings uniformly over all possible hash values. The specially chosen constants `a = 1664525` and `b = 1013904223` ensure that small changes in the input string result in unpredictable changes in the hash value.

1.5 Solving the Puzzle

We can view the problem of searching for a solution to a Lights Out puzzle as the problem of searching for a path in a graph:[1] the nodes of the graph are board states, and the neighbors

[1]Recall that a graph is a collection of *nodes* and *edges*, where each edge connects a pair of nodes.

of a node are all the board states that are reachable with a single move. This game graph is enormous, though—there are 2^{25} different possible game states in the usual 5×5 game, and each state has 25 neighbors, one for each light. So when we search through this graph, we do not represent the graph explicitly in memory. Instead, we compute pieces of it lazily as we require them, and consequently, we must make use of some interesting data structures to support our search.

The idea behind the algorithm is quite close to what you implemented for searching for a word ladder in Homework 3, and your code will consequently look quite similar. Instead of searching for a sequence of words, though, you are searching for a sequence of moves $(x_1, y_1), \ldots, (x_n, y_n)$ that leads from an initial state to the all-"off" state. Instead of considering the next possible moves to be the set of words one letter different from a given word, you will compute every possible move you might make. And instead of maintaining a stack of words, you will maintain a hashtable from boards to the moves which led to them. Review the description of `struct move` in the previous section.

Throughout the algorithm you keep a queue of boards to later explore, and a hashtable from boards to moves.

1. Begin by enqueueing the initial board. Add it to the hash table as well, to note that it has been seen. The initial board can have arbitrary nonnegative toggle position.

2. Repeat the following as long as the work queue isn't empty: dequeue a board, compute all of its neighboring boards and their moves, and for each board **you haven't already seen**, enqueue it and add the corresponding move to the hash table.

3. If you ever encounter the winning board—with all lights in the "off" position—reconstruct and return a winning sequence of moves by working backward through the solution using the hash table.

Several important invariants clarify the behavior of the algorithm:

- Every board that has been seen should have an entry in the hash table describing the move that led to it.

- Every board in the to-be-explored queue should be recorded as having been seen.

- For every move `m` in the hash table—except the element representing the initial board—there is already another move m_0 in the hash table such that executing $(m->x, m->y)$ on $m_0->key$ should yield $m->key$.

- The final resulting move sequence should transform the initial board to the winning board.

Many of these invariants can be formally accounted for using contracts, and designing and implementing specification functions for them will help solidify your understanding of the algorithm and help you write correct code.

Task 4 (12 pts, `lightsout.c0`) *Implement the solver algorithm outlined above as the function* `solve`*. Be sure to include contracts for the function's preconditions, postconditions, and relevant invariants: part of your grade for this task will be based on the quality of your contracts and the specification functions you implement to express them. Your* `solve` *function should return* `NULL` *if given a board that has no solution, and you may assume that the given board is not already solved. The function should return a valid* `move list` *if one exists.*