

Hill-Climbing *Lights Out*: A Benchmark

Abstract

We introduce and discuss various theorems concerning optimizing search strategies for finding solutions to the popular game *Lights Out*. We then discuss how to implement a search both with and without optimization in order to benchmark observable effects of our optimization techniques. In particular, we are interested in discussing the effects of optimizations over a simple hill climbing approach. A graph search is used for our discussion since *Lights Out* is a sufficiently difficult problem for graph search, which makes graph search of particular interest for conducting optimization analysis.

Patrick F. Wilbur

*Department of Mathematics &
Computer Science,*

Clarkson University

December 13, 2006

Copyright 2006 Patrick F. Wilbur. Verbatim Copies Only.

Table of Contents

Introduction.....	3
Standard Puzzle.....	3
Other Variants.....	4
“Perfect” Solutions.....	4
Order of Moves Does Not Matter.....	4
Two Consecutive Identical Toggles are Equivalent to None.....	5
Cells Need Only Unique Toggles.....	5
Implementation.....	6
Heuristics.....	6
Goal State.....	6
General Heuristic.....	6
Better Heuristic.....	7
Transitioning Between States.....	8
Iterative Deepening and Hill Climbing.....	8
Experiment.....	11
Test Construction.....	11
Results.....	11
Unoptimized Version.....	11
Optimized Version.....	11
Discussion.....	12
Observations.....	12
Conclusions.....	13
Strengths.....	13
Weaknesses.....	13
Source.....	14

Introduction

Lights Out is a puzzle form originally popularized by Tiger Electronics as a fad electronic toy in the 1980s. It consists of an $n \times n$ boolean matrix play field, where each cell may be either “lit” or “unlit”—that is, either on or off. Cells are directly selected and toggled one-at-a-time by the player and each toggle results in a certain pattern of indirect resultant toggles in other cells, according to some pattern defined by the transition function of the puzzle. Whenever a cell is toggled, directly or indirectly, if it is on it becomes off and if it is off it becomes on.

In particular, a *Lights Out* puzzle is characterized by the three-tuple containing its initial start state, a set of acceptable goal states, and a static transition function to define the indirect toggles that result from each direct toggle made by the player. Its start state may be any like-degree matrix that is not a member of the set of goal states and that, for some sequence of direct toggles in agreement with the transition function of the puzzle, can eventually lead to at least one reachable state that is an element of the set of acceptable goal states. Its transition function may be any finite-parted function that defines the effects on any of the matrix's cells when a particular cell is toggled, provided that it never outputs a state from which a solution cannot be reached.

Standard Puzzle

The most well-known version is the standard puzzle, also known as the *Fiver Puzzle*, which is a fifth-degree *Lights Out* puzzle. Its start state may be any 5×5 matrix that, for some sequence of direct toggles in agreement with the transition

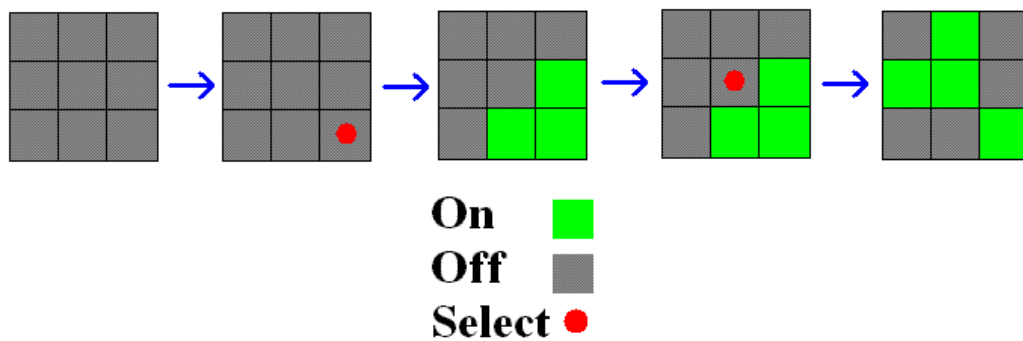


Illustration 1.1: A simple 3×3 example puzzle (Wikipedia, 2006)

function of the puzzle, can lead to at least one reachable state that is an member of the set of goal states.

A simple example of a 3x3 puzzle with the same rules as the standard puzzle is illustrated in *Figure 1.1* to demonstrate the feel of the game.

Other Variants

Lights Out does not refer to simply the 5x5 puzzle, nor does it refer to a single type of start state, goal state, and transition function; it is actually a large classification of similar puzzles spanning an infinite number of variations. The most common variation is the degree of the puzzle—3x3, 4x4, 5x5, 6x6, and so on, using traditional-style start states, goal states, and transition functions. Other common variants include puzzles in which the transition function behaves like in the standard puzzle but with an added mirroring at the tail-end of the effects, puzzles that challenge a player to recreate a particular shape or piece of artwork, and puzzles that begin with any or all states off and challenge a player to turn all lights on (Baker, 2002). The infinite scalability and flexibility of these puzzles make them particularly noteworthy for discussion.

“Perfect” Solutions

Solving a *Lights Out* puzzle using a puzzle-solving agent that lacks more than very basic heuristic assumptions is not an ideal approach. There are many theorems for several of the puzzle variants that propose considerations to be made by an agent when choosing moves. Additionally, for some variants, there are even entire recipes for solving the puzzle, which can handle any start state within that particular variant and prevent the agent from making mistakes or making wasteful moves.

Recipes that avoid many mistakes are ideal for solving puzzles for which such recipes have been found, but often do not provide scalability for use with puzzles of different degrees nor flexibility for puzzles of different styles; however, some of the approaches that reduce the number of moves made by setting certain constraints are scalable and may be used in any-degree puzzles of a similar type.

It is these scalable constraints that are particularly useful for our benchmarking purposes, since for the benchmark we will vary the degree of our puzzles in order to accurately represent the time it takes the computer to solve each.

Order of Moves Does Not Matter

The puzzle positions in *Lights Out* form an Abelian group (Scherphuis).

Theorem 1.1

As an Abelian (or commutative) group, the sequence of puzzle positions toggled may be rearranged in any order while maintaining the same ultimate result state for that particular set of moves. Thus, each possibility for an agent's moves can be represented as a set, which is beneficial in that the number of possible unique sets of moves is far less than what would instead be the number of possible unique sequences of moves.

Two Consecutive Identical Toggles are Equivalent to None

Consider a cell in the unlit state being represented as the numeral 0 and a cell in the lit state the numeral 1. When a particular cell is toggled (either by the product of a direct selection by the agent or as an indirect output of the transition function given a direct selection) that cell's new current state is equal to its previous state plus 1, modulo 2. Thus,

Any two consecutive direct or indirect toggles on the same cell yield the same result state as having made no toggles on that cell. That is, two consecutive identical toggles cause an already lit cell to remain lit and an already unlit cell to remain unlit (Martin-Sanchez & Pareja-Flores, 2001).

Theorem 1.2

and an agent may ignore consecutive identical moves as they have the same effect as having not made such moves at all.

Cells Need Only Unique Toggles

From Theorem 1.1, we know that rearranging the toggling order of an attempt made by an agent has no effect on the outcome of that attempt. Thus, even if a particular attempt calls for two non-consecutive identical toggles, the toggling

order may be rearranged such that the identical toggles occur consecutively. By Theorem 1.2 we know that two consecutive identical toggles have the same effect as not making the two toggles in the first place. Thus,

No toggles ever need to be made on the same cell more than once in a single puzzle in order to find a solution (Scherpius).

Theorem 1.3

Implementation

Heuristics

Goal State

Our goal in a standard-style game is to reach a state in which all lights are initially turned off. In order to verify that a goal state has been reached, we simply define our goal checker such that it returns true if all lights are off and returns false if the number of lights remaining lit is greater than zero.

```
(define isGoalState?  
  (lambda (state)  
    (cond ((null? state) #t)  
          ((= 0 (car state))  
           (isGoalState? (cdr state)))  
          (else #f))))
```

Illustration 2.1: Goal state verification

General Heuristic

Crudely speaking, as a player enters states with fewer lights on than the previous that player is making progress. We can construct a very basic, yet sensible, heuristic from this observation where our heuristic function takes a potential successor state as input and outputs a numerical value equaling the total number of lights that are lit in that state.

```

(define H
  (lambda (state)
    (cond ((null? state) 0)
          ((= 1 (car state))
           (+ 1 (H (cdr state))))
          (else
           (H (cdr state))))))

```

Illustration 2.2: Simple heuristic formation

This heuristic value will enable our algorithm to always choose the seemingly next-best successor node for a state while searching.

Better Heuristic

In order to accommodate for the advantages *Theorem 1.3* has for our algorithm, we redefine our heuristic function so that it dissuades our agent from toggling any element it has already toggled.

```

(define H
  (lambda (state currElement cellsVisited)
    (cond ((and (> currElement 0)
               (isIn? currElement cellsVisited)) -1)
          ((null? state) 0)
          ((= 1 (car state))
           (+ 1 (H (cdr state) 0 '())))
          (else
           (H (cdr state) 0 '())))))

```

```

(define isIn?
  (lambda (e ls)
    (cond ((null? ls) #f)
          ((= e (car ls)) #t)
          (else
           (isIn? e (cdr ls)))))

```

```
(else (isIn? e (cdr ls))))))
```

Illustration 2.3: Improved heuristic formation

This new heuristic function is given a potential successor state to investigate, the element that our agent proposes to toggle next, and a list of all nodes already toggled. It behaves like our previous heuristic except that it assigns a -1 value for any proposed toggle that is the same as a previous toggle, which will indicate to our agent to prune that successor from its list of available choices.

Transitioning Between States

The responsibility of our agent's transition function is to modify the current state according to the selected element to be toggled next.

```
(define transition  
  (lambda(state toggledElement degree)  
    ...
```

Illustration 2.4: Transition function formation

Our agent's purpose is to solve puzzles similar to the standard puzzle, so given a state and element to be toggled, its transition function should return the new state that results from toggling the given element plus the elements vertically and horizontally adjacent to that element.

Iterative Deepening and Hill Climbing

The algorithm our agent program will be using is a form of hill climbing. Hill climbing is used because all that our agent will know in each state is the presumed value of choosing any of the possible successor state nodes during its search. We will generate a list of pairs to represent the possible successor states for a given state, where the first variable in the pair will represent a particular successor node and the second variable will represent the perceived heuristic value if our agent transitions to that successor.

We sort this list by the value given to each successor state from our heuristic function, in ascending order so that the successors with the lowest (best) perceived value will be tried first.

```
(define genNextBestMoveList
  (lambda (state numElements degree cellsVisited)
    ...
```

Illustration 2.5: Sorting successors by next-best moves

Our algorithm will choose the first node on this list and succeed beneath it recursively, but if no solution can be reached beneath that node, it will pop the node off the front of our list and proceed with the next node. In order to prevent our agent from getting stuck in infinite loops, we ignore successors that are identical to the current node (to prevent loops of just consecutive toggles of the same node) and we also limit the depth at which our agent may succeed down the tree of successor nodes.

We do not know the maximum depth at which it is necessary to traverse such that all combinations for a particular-degree game are searched (although this is likely calculable), so we use iterative deepening instead of fixing this maximum depth to a constant. This will also help us to find solutions with few moves if they exist since they would be nearer the top of our tree of successor nodes, which can reduce the running time of our algorithm.

With these considerations in mind, we can construct the following general Scheme algorithm for solving games of a given degree:

```
(define LightsOut          ; Iterative deepening
  (lambda (game degree)
    (letrec ((solver (lambda (maxDepth)
      (cond ((> maxDepth
        (* degree degree)) '())
        (else
          (let ((solution
            (solveLightsOut game
              (* degree degree)
              degree 0 maxDepth '()))))
            (cond ((null? solution)
```



```

      (cond ((null? result)
            (makeTry (cdr
                     nextBestMoveList)
                    ))
            (else result)))))))))
(makeTry (genNextBestMoveList game
        numElements degree cellsVisited))))))

```

Illustration 2.6: Algorithm formation

Experiment

Test Construction

The algorithm is prototyped using randomly-generated games. These games that are supplied to the algorithm for testing are all solvable, and are generated on-the-fly by recursively applying the transition function to the goal state on randomly selected elements in the grid. The transition function is called on random elements a considerably sufficient number of times for testing—the square of the degree of the game board.

Both unoptimized and optimized versions of our heuristic were tested with our algorithm simultaneously over several separate, identical lab computers that were dedicated to our algorithm's calculations. Corresponding results from identical tests performed on multiple machines were concatenated into individual files for those types of tests and then evaluated.

Results

Unoptimized Version

One execution of the algorithm is with our original heuristic that does not make considerations based on *Theorem 1.3*. The duration of the algorithm searching for a solution for 3x3, 4x4 and 5x5 games are as follows (note that no solution was ever found within the duration of the experiment for a 5x5 puzzle):

Nx N	Number of unsolved puzzles	Number of solved puzzles	Average Duration (seconds)
3x3	0	282	31.52
4x4	0	2	5394.00
5x5	(1 timeout)	0	(14000.00)

Optimized Version

Another execution of the algorithm is with our original heuristic that does consider the rule shown in *Theorem 1.3*. The duration of the algorithm searching for a solution for 3x3, 4x4 and 5x5 games are as follows (note that still no solution was found within the duration of the experiment for a 5x5 puzzle):

Nx N	Number of unsolved puzzles	Number of solved puzzles	Average Duration (seconds)
3x3	0	281	8.47
4x4	0	6	593.50
5x5	(1 timeout)	0	(14000.00)

Discussion

Observations

An interesting observation can be made from our agent's output logs in the solution field for 3x3 puzzles. In both the unoptimized and optimized version,

after nearly 300 3x3 puzzles were solved not a single puzzle solution ever contained a toggle on the ninth cell (the cell at the bottom right of the 3x3 grid). The ninth cell was lit originally at the start of a large number of games, and all the solutions verified to be accurate solutions for the problem that was given, so it appears that the ninth cell did not need to be toggled.

Although it may still be possible that there are puzzles that the random puzzle generator did not query the agent with but that would require the ninth cell to be toggled to reach a solution, perhaps in a 3x3 game a ninth cell that is lit can always be dealt with by toggling the adjacent sixth or eighth cell. It would be interesting to try to verify or counter this by formal proof. Although all results for 3x3 games did not include solutions with ninth cells in them, both the unoptimized and optimized solutions for 4x4 puzzles included solutions containing the ninth cell.

Conclusions

Strengths

The data contained in our results does support *Theorem 1.3*, and the optimization was noticeable in 3x3 and 4x4 puzzles. In either case, the 5x5 puzzle could not be solved by our algorithm even after 14,000 seconds, or nearly four hours.

Hill climbing behaved as could be expected during these experiments, and was ideal for illustrating the effects of our optimizations.

Weaknesses

There were a number of variations between durations for puzzles in our experiment, some over 100 seconds for 3x3 puzzles and 1000 seconds for 4x4 puzzles. In all cases these variations occurred with solutions to moderately difficult puzzles that began with moves causing more lights to be left lit than any other move.

This is very expectable due to the nature of our heuristic function and hill climbing, and can be considered a similar problem to that of temporary local maxima/minima typically encountered in hill climbing. Our algorithm does not

break in this context because of its iterative nature, and these only occurred less than three percent of the time.

Source

Scheme source code is available at <http://pdub.net/projects/lightsout/> and <http://www.clarkson.edu/~wilburpf/lightsout/lightsout.scm> .

This source should be used with MIT-Scheme. It was tested using MIT-Scheme.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;
;; LightsOut Benchmark Program
;; Copyright 2006, Patrick F. Wilbur
;;
;; Patrick F. Wilbur
;; Department of Mathematics and Computer Science
;; Clarkson University
;; Potsdam, NY USA
;;
;; http://pdub.net/projects/
;;
;; This program is released under the Creative Commons Attribution-Share Alike 2.0
Generic
;; license, available online at http://creativecommons.org/licenses/by-sa/2.0/
;;
;; You may share (copy, distribute, and transmit) this work, and remix (adapt) this
work,
;; as long as you attribute this work to the author and share adapted works under the
same
;; or similar license by leaving this entire notice in place (including the original
;; author's name/contact information/URL and this license notice).
;;
;;
```

```

;; General usage, for NxN games:
;; =====
;;
;; (LightsOut listContainingGame N) ; Display solution sequence for given game
;; -----
;;
;; (prototypeGames N1 NN R) ; Run LightsOut over R repetitions of random
;; ----- ; N1xN1 games, followed by N2xN2 games,
followed ; followed
;; ; by N3xN3, ... , up to NNxNN games
;; ;
;; ; Displays CSV output, one per line:
;; ;
;; ; N,listContainingGame,solution,duration
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Type-of-game behavioral configuration
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Standard-Style Game ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; checks for "no lights are on" goal state, returns #t or #f
;

```

```

(define isGoalState?
  (lambda (state)
    (cond ((null? state) #t)
          ((= 0 (car state))
           (isGoalState? (cdr state)))
          (else #f))))

; transitions from current state to new state given toggle,
; returns new state
;
(define transition
  (lambda(state toggledElement numElements degree)
    (alterElement
     (alterElement
      (alterElement
       (alterElement
        state
         (+ toggledElement 1)
         numElements
         degree)
        (- toggledElement 1)
        numElements
        degree)
       (+ toggledElement degree)
       numElements
       degree)
      (- toggledElement degree)
      numElements

```

```
degree)
toggledElement
numElements
degree)))
```

```
; heuristic function
; returns count of number of lights still lit
;
;; non-optimized algorithm:
;;
;;(define H
;; (lambda (state currElement cellsVisited)
;; (cond ((null? state) 0)
;; (= 1 (car state))
;; (+ 1 (H (cdr state) currElement cellsVisited)))
;; (else
;; (H (cdr state) currElement cellsVisited))))
;
; optimized algorithm:
(define H
  (lambda (state currElement cellsVisited)
    (cond ((and (> currElement 0)
                (isIn? currElement cellsVisited)) -1)
          ((null? state) 0)
          ((= 1 (car state))
           (+ 1 (H (cdr state) 0 '())))
          (else
           (H (cdr state) 0 '())))))
```

```

(define isIn?
  (lambda (e ls)
    (cond ((null? ls) #f)
          ((= e (car ls)) #t)
          (else (isIn? e (cdr ls))))))

; generates what we call the goal state (a "blank" game field)
; returns this game field
;
(define genGoalGameField
  (lambda (degree)
    (genGoalGameField-helper '() (* degree degree))))

(define genGoalGameField-helper
  (lambda (game degree)
    (cond ((= 0 degree) game)
          (else (cons 0 (genGoalGameField-helper game (- degree 1))))))

; generates a random, *solvable* game
; returns the game field
;
(define genRandomGame
  (lambda (degree)
    (genRandomGame-helper (genGoalGameField degree) degree (* degree degree))))

(define genRandomGame-helper
  (lambda (game degree numPasses)
    (cond ((= 0 numPasses) game)
          (else (genRandomGame-helper (genGoalGameField degree) degree (* degree degree))))))

```



```

        (let ((nextBestMove (car (car nextBestMoveList))))
          (let ((result (solveLightsOut (transition game nextBestMove
numElements degree)
                                         numElements
                                         degree
                                         (+ 1 depth)
                                         maxDepth
                                         (cons nextBestMove cellsVisited))))
            (cond ((null? result) (makeTry (cdr nextBestMoveList))
                  (else result))))))
      (makeTry (genNextBestMoveList game numElements degree cellsVisited))))))

```

```

(define genNextBestMoveList
  (lambda (state numElements degree cellsVisited)
    (genNextBestMoveList-sorter
     (genNextBestMoveList-helper state numElements degree cellsVisited 1)
     numElements)))

```

```

(define genNextBestMoveList-helper
  (lambda (state numElements degree cellsVisited currElement)
    (cond ((> currElement numElements) '())
          ((and (not (null? cellsVisited)) (= currElement (car cellsVisited)))
           (genNextBestMoveList-helper state numElements degree cellsVisited (+ 1
currElement))))
          (else
           (cons
            (cons
             currElement
              (H (transition state currElement numElements degree) currElement
cellsVisited))
            ))))

```

```
      (genNextBestMoveList-helper state numElements degree cellsVisited (+ 1
currElement)))))))))
```

```
(define genNextBestMoveList-sorter
  (lambda (NBML numElements)
    (genNextBestMoveList-sorter-helper NBML numElements 0)))
```

```
(define genNextBestMoveList-sorter-helper
  (lambda (NBML numElements currVal)
    (cond ((null? NBML) '())
          ((> currVal numElements) '())
          (else
           (append (filterNextBestMoveList NBML currVal)
                    (genNextBestMoveList-sorter-helper NBML numElements (+ currVal 1)))))))
```

```
(define filterNextBestMoveList
  (lambda (NBML currVal)
    (cond ((null? NBML) '())
          ((= currVal (cdr (car NBML)))
           (cons (car NBML) (filterNextBestMoveList (cdr NBML) currVal)))
          (else
           (filterNextBestMoveList (cdr NBML) currVal))))
```

```
(define LightsOut
  (lambda (game degree)
    (letrec ((solver (lambda (maxDepth)
                      (cond ((> maxDepth (* degree degree)) '()))
```

```

      (else
        (let ((solution
              (solveLightsOut game
                            (* degree degree)
                            degree
                            0
                            maxDepth
                            '()))))
          (cond ((null? solution)
                 (solver (+ 1 maxDepth)))
                (else solution))))))
    (solver 3)))

```

; An example solver that does not use iterative deepening

```

; (define LightsOut
;   (lambda (game degree)
;     (solveLightsOut game (* degree degree) degree 0 (* degree degree) '())))

```

(define prototypeGames

```

  (lambda (lowDegree highDegree numTimesEach)
    (letrec ((prototype (lambda (low t)
                          (cond ((> low highDegree) #t)
                                ((= 0 t) (prototype (+ 1 low) numTimesEach))
                                (else
                                 (let ((randomGame (genRandomGame low)))
                                   (let ((beginTime (get-universal-time)))
                                     (begin
                                       (display low) (display ",")
                                       (display randomGame) (display ",")
                                       (display (LightsOut randomGame low)) (display ",")

```


Works Cited

- Baker, M. (2002). *Changing the rules*. October 10, 2006.
<http://www.haar.clara.co.uk/Lights/changing.html>
- Martin-Sanchez, O. & Pareja-Flores, C. (2001). Two reflected analyses of Lights Out. *Mathematics Magazine*, 74, 295-304.
- Scherphuis, Jaap. *Lights out*. October 28, 2006.
<http://www.geocities.com/jaapsch/puzzles/lights.htm>
- Wikipedia. (2006). *Lights out (puzzle)*. November 28, 2006.
http://en.wikipedia.org/wiki/Lights_Out_%28puzzle%29