## Lights Out Puzzle

## Tomas Cespedes

Artificial Intelligence

**What is Lights out?**

Lights out is a puzzle game in which the goal is to turn off all the "lights" (1 tile is a single light). The rules are very simple. A light that is on is represented by a white tile and a light that is off is represented by a grey tile. When the player clicks on a light, the light that was clicked switches to it's other state (a light that is on turns off and vice versa). Additionally, it's direct neighbors (not including diagonals) also switch their states accordingly.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

*Puzzle starting state.* *Puzzle ending state.*

Above we can see the original state with the middle light and it’s neighbors turned on. To make things simple, all the other lights are turned off. After the player clicks the middle tile, we see the lights changing accordingly. Since the ending state has no lights on, the player wins the game!

Lets say that instead the player did not realize the middle tile was the solution. Instead, for some reason, they choose the top left tile.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

*Puzzle starting state.* *Puzzle ending state.*

Since the top left tile has no neighbor to it’s left and above him, nothing happens on that side. However, we still see the tile’s neighbors to the right and under him change states. In order to fix that last click, the player would have to once again click on the top left tile. This is a very important thing to remember because the player should only have to click on a single tile once. In other words, clicking the same tile twice is the same thing as never clicking it to begin with.

**How do we solve Lights Out?**

There are countless algorithms and different data structures we can choose to solve this puzzle. There are many aspects that we need to take into consideration when choosing our algorithm. Some algorithms give us the optimal solution every time but, as a consequence, suffer in time complexity. This is where we begin with Breadth First Search (BFS). The way BFS works is by utilizing a Search Tree and a queue which we will call the Frontier. It works by traversing through all the nodes of the tree to find a solution. If there is a solution, BFS will find it but it could take a very long time. However, if there is more than one solution, BFS will find the optimal solution. The frontier stores a list of nodes in which the algorithm will at some point visit. Therefore, the algorithm ends when the frontier is empty (means no more nodes need to be visited). For BFS to work, we need to have a root node in which the algorithm begins. The algorithm works by dequeuing a node from the frontier. We then find the children of that node and if it has not been visited, we add it to the frontier to analyze it later. It is extremely important to only add the child if it has not been visited before or else we will run into some problems down the road (traversing the same node, wasting time, etc...). If that child is not the solution, then we search the other child. This is where the name Breadth comes from because it is as if you are searching all the little bread crumbs. Below, you can see by following the numbers the path that BFS would take on a Search Tree.

1

0

2

4

3

6

5

14

13

12

11

10

9

8

7

*Figure of BFS on a SearchTree.*

The arrows and the circles represent the tree data structure. From now, we will refer to the circles as nodes and the arrows as vertices. The numbers represent the order in which we will visit each node. The dashed arrows represent that the tree continues. The root node (represented by a 0) is where we begin. We then traverse the first child of the root node and go to the node labeled as 1. Since that is not a solution, we then check node number 2. If that is not a solution, then we check node number 3. This process continues until we find our optimal solution.

**Sacrificing Accuracy for Time Complexity**

While Breadth First Search gets us an optimal solution, it is extremely time consuming. At this point, we can give up a little bit of accuracy in our puzzle and in exchange, we can solve it much faster. The next algorithm I used to solve Lights Out is called A Star Search (A\*). The goal of A\* is to approximate the shortest path to a solution determined by a heuristic value. Since we do not know how far away we are from our solution, we use a heuristic value as a rough estimate of how many conflicts we have until we reach our solution. In our puzzle, the value is the number of lights that are still on. However, it is important to remember that since it is all a rough estimate, the solution at the end will be correct but it will not always be in the minimal amount of moves. A\* uses BFS to plan the moves but also uses a finalized set to save moves that have already been visited to traverse the path for when we reach our solution. Since we have a heuristic value we can use, the algorithm chooses the child node that looks most promising. The value of the best node is the cost that it took to get to our current node plus it’s heuristic value. That node is then prioritized within the frontier to have the process repeated for it’s children.

12

1

2

3

4

8

9

11

5

3

10

6

11

7

5

2

0

3

7

2

1

0

*Actual Path A\* would take. Potential better path.*

As you can see, the A\* would choose the path to the left and not even continue down the right path (coded as grey). It would continue to follow the shortest path until it found the best solution there. If it were to be able to have some more information, it could check the other path and see that there was a better solution one step earlier (coded as yellow). The dashed vertices represent a path that was not visited by A\*. The run times will show that it might be worth it to give up some accuracy in return for a faster answer.

There is an algorithm called Iterative Deepening A\* which acts as a median for BFS and A\*. It works by running A\* with a limit in which the algorithm does not look past. As a result, it checks all the possible solutions on one level before checking any solutions in the next level. Due to the increased amount of nodes that the algorithm must check, the run time increases but not by as much as it would for a normal BFS or an Iterative deepening search. If we refer to the diagram above for A\*, IDA\* would find the other path that is optimal and as a result return a better answer. To get a better image of how IDA\* would run, you can imagine the diagram but the difference is that it would check everything on level 1. If the solution was not found, then it would run it again but this time the limit would be 2. This continues until an answer is found.

**The Results**

BFS:

|  |  |  |
| --- | --- | --- |
| Puzzle Number | Computational Time (sec) | Number of Moves (Expected) |
| 1 | 459.24 | 8 (8) |
| 2 | 1446.45 | 10 (10) |
| 3 | 3429.96 | 12 (12) |
| 4 | 6.89 | 5 (5) |
| 5 | 2.48 | 5 (5) |

*Data for Breadth First Search.*

As we can see from the data from BFS, the run times for the algorithm are not ideal for the harder puzzles. For the hardest puzzle, it took about 3429.96 seconds (58 minutes) to run to solve it in the best possible amount of moves (12). The easiest puzzle was solved in only 2.48 seconds which appears to be fairly fast. However, we can improve on that number with A\*. If we take a look at the number of moves it took to solve each puzzle versus their expected number of moves, we see that all puzzles were solved in the most optimal way. The reason for such a high run time for the harder puzzles is due to the nature of our Search Tree. The run time complexity for BFS is O(V + E) where V is the number of vertices and E is the number of edges. Since every move has 24 other possible moves from there, the tree’s width grows very quickly so it takes a while for BFS to search all children, grandchildren, etc...

A-Star Search:

|  |  |  |
| --- | --- | --- |
| Puzzle Number | Computational Time (Seconds) | Number of Moves (Expected) |
| 1 | 0.51 | 8 (8) |
| 2 | 12.84 | 12 (10) |
| 3 | 9.82 | 12 (12) |
| 4 | 0.014 | 5 (5) |
| 5 | 0.0078 | 5 (5) |

*Data for A-Star Search.*

When we compare A\* to BFS, we can see that the time complexity for A\* has significantly improved from where we were with BFS. For the hardest puzzle, we go from 3429.96 seconds (BFS) to a mere 9.82 seconds (A\*). In the easiest case, we saw BFS run the puzzle in 2.48 seconds but here we see it run almost instantly at 0.0078 seconds. However, as we discussed earlier, A\* is a rough estimation of the best path. If you take a look at the number of moves it took to solve the second puzzle, you can see that it solved it in 12 moves while the ideal number of moves is 10.

IDA\*:

|  |  |  |
| --- | --- | --- |
| Puzzle Number | Computational Time (Seconds) | Number of Moves (Expected) |
| 1 | 1.08 | 8 (8) |
| 2 | 42.31 | 10 (10) |
| 3 | 43.31 | 12 (12) |
| 4 | 3.06 | 5 (5) |
| 5 | 2.96 | 5 (5) |

*Data for Iterative Deepening A-Star Search.*

Finally, we have the ability to combine the pros from BFS and A\* which we expected a slightly worse time complexity but better algorithm. The data follows our expectation as we see most of the run times increase from A\*. However, with BFS it took our hardest puzzle 3429.96 minutes to solve for the ideal move but IDA\* does it in only 43.31 seconds. Although this is an extra 33.49 seconds from how A\* solved it, we can detect a difference in the number of moves in which they solved the puzzle. IDA\* solved that same puzzle in only 10 moves where as regular A\* solved it in 12 moves. With the first puzzle, we got lucky that A\* solved it an optimal number of moves but if we compare IDA\* to how BFS solved it we see a 458 second increase and are guaranteed the optimal solution.