Black Box Consistency with SAT

[redacted]

CS 2800
November 9, 2020

Abstract

This paper explores the consistency of a board game called Black Box, in which players attempt to locate a number of atoms hidden within a square grid by observing the behavior of rays shot into the board. These rays give rise to "hints," which players can use to deduce the location of the atoms. We have written a program in Java which takes a set of these hints and determines if there exists a corresponding atom configuration that satisfies them. These encodings are individually generated with a recursive algorithm that identifies all possible ray traversals that lead to hint satisfaction. Unsurprisingly, the memory usage of this algorithm quickly balloons as a function of the board size. Going forward, the generation algorithm should be optimized to reduce the workload on the SAT solver and improve efficiency.

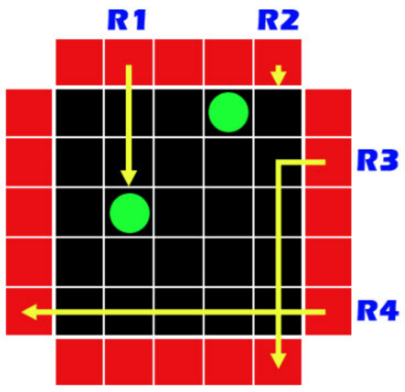


Figure 1. Four different examples of ray travel through the blackbox.

1 General Problem Area

Black Box is a fairly obscure single-player board-game that was created in 1976 by mathematician Eric Solomon, who was inspired by the invention of the CAT scanner in the previous decade [1]. The game is played with a square grid, the namesake "blackbox", in which a number of atoms are hidden. Players can probe the locations of the hidden atoms by shooting imaginary rays of light into the blackbox from the border of the grid. Rays will travel in their given direction until they are either deflected to a new direction by an atom, or until they are absorbed by an atom upon a direct collision (Fig. 1).

In practice, players "shoot" rays into the blackbox by revealing cells around its perimeter that contain information, called hints, about if and how the ray shot from that cell exits the blackbox. A ray may either directly hit an atom and never leave the blackbox, or it may exit the blackbox at some position, possibly after undergoing a series of deflections. In Fig. 1, the cell that R1 originated from would contain a Hit hint, and the cells for R2, R3, and R4 would all contain Exit hints that specify where the ray exits the blackbox. Using these hints in conjunction, players can deduce the locations of atoms, winning the game if they can successfully locate all atoms.

This paper explores the consistency of Black Box boards. A board is consistent if there exists a set of atoms that satisfies every hint. If there is some contradiction in the set of hints such that no set of atoms can satisfy every hint, then the board is inconsistent. Our goal is to algorithmically determine whether or not a given set of hints are consistent with one another by encoding each hint as a boolean expression and SAT-solving them together.

2 Approach

Overall, the game of Black Box has not been extensively studied through the lens computational theory. One of the only papers we have found that studies the game was written by student researchers at UC Irvine [2], who describe two algorithms that can be used to determine the locations of all atoms in a given Black Box board. In this paper, we take a novel approach by encoding Black Box boards as propositional logic statements that can be SAT-solved to determine board consistency. Our approach can be divided into two main steps. First, we generate a boolean expression for each hint on the board that represents whether or not the hint is satisfied. Then, we take these expressions in conjunction to form an expression that represents the consistency of the entire board. This larger expression is automatically converted into CNF and SAT-solved.

Generating the expression for each hint is done recursively by tracing out all possible paths a ray can take. At each position, the ray can continue straight, deflect 90° clockwise or counterclockwise, or reflect 180°. Each of these four moves arise only from particular atom configurations (Fig. 2).

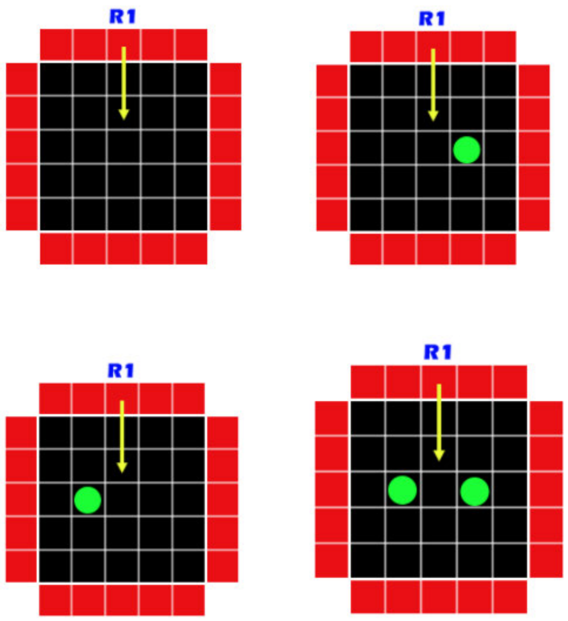


Figure 2. *Examples of ray deflection.* R1 travels in a straight line in the absence of any deflecting atoms. R2 is deflected clockwise by collision with the edge of an atom, and R3 counterclockwise. R4 is deflected by two atoms simultaneously, causing it to turn 180^o and exit the blackbox from its original cell.

At each cell, we can assume each of these deflections and find the next position of the ray in each case. The boolean expressions generated by each hint in this fashion can be used in conjunction to create a formula that represents the board.

We used a Java library called LogicNG to handle CNF conversion and SAT-solving. LogicNG is a publicly available repository which includes tools for memory-efficient boolean formula manipulation. An implementation of MiniSAT comes bundled with LogicNG. As such, we primarily focused on implementing the boolean encodings for Black Box boards that allow us to determine hint consistency, using LogicNG to handle the raw formula produced by our encoding.

Our main metric for success was program correctness, which was tested by using a set of home-made boards and a set of boards sourced from online Black Box implementations. Efficiency was a secondary objective, something that we had to sacrifice for the overall correctness of the algorithm. As a result, our testing data was limited to 5-by-5 and smaller boards only.

3 Methodology

3.1 Board Conventions

The convention we use sets the top left cell in the blackbox as position $(0, 0)$, with x increasing when travelling east and y increasing when travelling south. Each cell in the blackbox is assigned a boolean variable $P_{x,y}$, which is true if there is an atom in the cell at position (x, y) and false if not. Cells immediately around the blackbox are of particular interest to our encoding scheme since we are concerned with where rays enter and leave the blackbox, but these border positions are not assigned boolean variables because no atom can be present outside the blackbox.

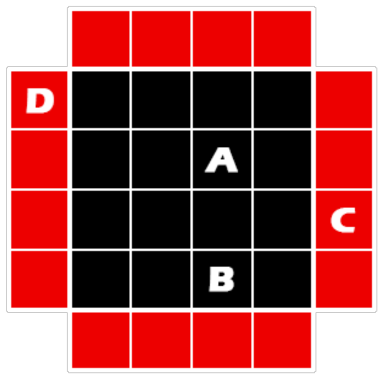


Figure 3. *Position Validity.* The 4-by-4 square grid gives some examples of positions and variables corresponding to the marked areas. Notice that C and D have positions outside of the board, and therefore do not have variables.

Cell	**Position**	**Variable**
A	(2,1)	$P_{2,1}$
B	(2,3)	$P_{2,3}$
C	(4,2)	
D	(-1,0)	

3.2 Rules for Hint Satisfaction

Our program models two types of hints: Hits and Exits. A Hit hint is satisfied if the ray it generates directly strikes an atom. An Exit hint is satisfied if the ray it generates manages to travel out of the board at a certain position, specified upon instantiation. Reflections are a special case of Exit hints in which the specified end position is the same as the start position.

With these conventions, we can outline a set of rules to determine whether or not a given Hit hint is satisfied:

1. If there are any atoms denying the ray's entry into the board, the Hit is immediately unsatisfied.
2. At every position during the ray's travel, the Hit is satisfied if
   (a) an atom occupies the ray's next position, or

   (b) travelling further (continuing in the same direction, getting deflected clockwise, or getting deflected counter-clockwise) results in a Hit.

3. A reflection reached at any point in the board immediately means the Hit is unsatisfied.

4. If the ray travels out of bounds, then the Hit is immediately unsatisfied.

A similar set of rules can be used to determine the satisfaction of any given Exit hint:

1. If there are any atoms denying the ray's entry into the board, the hint is satisfied iff the Exit is a reflection.

2. At every position during the ray's travel, the Exit is satisfied if

   (a) an atom does not occupy the ray's next position, and

   (b) travelling further will result in an Exit at the correct position.

3. If reflection occurs at any point during the ray's travel, the hint is satisfied iff the Exit is a reflection.

4. If the ray travels out of bounds, then it must have exited at the correct position for the hint to be satisfied.

The process of algorithmically generating boolean expressions to correspond to these rulesets is discussed in further detail in the following sections.

3.3 Encoding Hits into Boolean Expressions

The algorithm to generate a boolean expression for a Hit is based on the idea that Hits are satisfied if there is an atom in the next cell that the ray will travel to, or if there is an atom somewhere else in the ray's path. The core of the method used to generate the boolean expression is based on the following two rules defined in the previous section:

Rule 2a: For the Hit to be immediately satisfied by a direct collision, $P_{a,b}$ must be true, where a and b are the coordinates of the ray's next position.

Rule 2b: For the Hit to be satisfied somewhere else in the ray's path, rule 2a must be satisfied for some later position. Satisfaction by rule 2b is determined by considering each of the four deflection cases recursively.

For each of the four deflections types (no deflection, 90º clockwise deflection, 90º counterclockwise deflection, and 180º reflection), the ray will travel to a different position. Each new position can be evaluated for Hit satisfaction by the same process: checking if its next position contains an atom, or if there is an atom somewhere later down the path of any of the four new deflections. This case analysis for each type of deflection is captured in our boolean expression by a recursive conjunction of implications. Each antecedent represents one of the four possible deflection cases, and each corresponding consequent represents the appropriate condition for Hit satisfaction given that deflection.

A conjunction of these implications is an appropriate form for case analysis on the four types of deflections because exactly one antecedent will be true at every position. The three false antecedents will immediately cause their respective implications to evaluate to true by constant propagation, so the only relevant expression in the conjunction is the one that represents the actual deflection case, as desired.

Combining Rules 2a and 2b yields an expression for Hit satisfaction for a ray that is traveling in the blackbox. For example, consider a ray at position (x, y) travelling east (Fig. 4). The boolean expression that represents whether or not this ray will satisfy a Hit is given by the function $h(x, y, E)$, which evaluates to:

$$\begin{aligned} P_{x+1,y} \vee (&[(\neg P_{x+1,y\ \ 1} \wedge \neg P_{x+1,y+1}) \rightarrow h(x+1, y, E)] \wedge \\ &[(P_{x+1,y\ \ 1} \wedge \neg P_{x+1,y+1}) \rightarrow h(x, y+1, S)] \wedge \\ &[(\neg P_{x+1,y\ \ 1} \wedge P_{x+1,y+1}) \rightarrow h(x, y-1, N)] \wedge \\ &[(P_{x+1,y\ \ 1} \wedge P_{x+1,y+1}) \rightarrow 0]) \end{aligned}$$

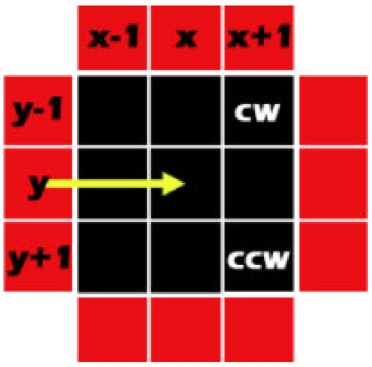


Figure 4. *Example ray traveling east.* The ray will undergo one of four deflection cases depending on the values of the variables for the cells marked CW and CCW.

The first term, $P_{x+1,y}$, accounts for the Rule 2a case in which the ray will collide with an atom in the next cell; if the ray continues to travel east, its next position will be (x + 1, y), so if an atom occupies that position, the hit is satisfied.

The positions to consider for clockwise and counterclockwise ray deflections are functions of direction and position. An atom that results in a clockwise deflection is represented by P_{CW}; its counterpart is P_{CCW}. In the example above, P_{CW} is $P_{x+1,y\ \ 1}$, while P_{CCW} is $P_{x+1,y+1}$.

For the no deflection case (i.e. $\neg P_{CW} \wedge \neg P_{CCW}$), the algorithm recurs by calling h on the next position reached by travelling in the same direction. In the case that exactly one of the CW or CCW atoms exists, the ray is deflected in a new direction (respectively S and N in this example). The algorithm recurs by calling h on the next position and direction resulting from the deflection. Finally, if both the CW and CCW atoms are present (i.e. $P_{CW} \wedge P_{CCW}$), the hit immediately becomes unsatisfied by rule 3 – no recursion is necessary.

Our algorithm will keep recursively evaluating h in the consequent of each implication until the ray reaches a pair of coordinates that are out of bounds, at which point h will return false by Rule 4. For a board of r rows and c columns and a ray with the current position (x, y) and direction D, the complete expansion of $h(x, y, D)$ yields a boolean expression in terms of the variables P_{00} through $P_{r\;\;1,c\;\;1}$ that will return true if the Hit is satisfied for a given set of atoms, and false if not. This expression correctly represents the satisfaction of a Hit by this ray only if (x, y) is a cell in the blackbox; for a ray starting from outside the blackbox, there are certain initial cases that must be considered (discussed further in Section 3.5).

3.4 Encoding Exits into Boolean Expressions

The algorithm generates boolean expressions for Hits and Exits in a similar fashion, and in some ways, a Hit can be thought of as "not an Exit." After performing some initial checks to ensure that the ray correctly enters the board, the algorithm then moves into recursive steps that map out all possible atom placements that lead to hint satisfaction, just as with Hits.

Using the same abstract board in Fig. 4, the recursive expression $e(x, y, E)$ is constructed in much the same way as $h(x, y, E)$. The expression $e(x, y, E)$ evaluates to:

$$\begin{aligned}\neg P_{x+1,y} \wedge ([(\neg P_{x+1,y\;\;1} \wedge \neg P_{x+1,y+1}) &\rightarrow e(x+1, y, E)] \wedge \\ [(P_{x+1,y\;\;1} \wedge \neg P_{x+1,y+1}) &\rightarrow e(x, y+1, S)] \wedge \\ [(\neg P_{x+1,y\;\;1} \wedge P_{x+1,y+1}) &\rightarrow e(x, y-1, N)] \wedge \\ [(P_{x+1,y\;\;1} \wedge P_{x+1,y+1}) &\rightarrow \mathit{isReflection?}])\end{aligned}$$

The next position must be unoccupied (signified by the term $\neg P_{x+1,y}$) so that the ray does not result in a hit. Additionally, recursively calling e, using the same case-switch scheme as for Hits, must result in an exit at the correct location. Deciding whether a ray exits at the right location is computed dynamically whenever the ray has moved out of bounds.

Hits and Exits differ in their start and end behavior, but as seen in the expression above, the core recursion scheme has the same structure for both types of hints.

3.5 Entry Conditions

Before the recursion can begin, each ray must satisfy certain initial conditions. Black Box defines a special rule for deflections that occur on the border: for atoms that would ordinarily deflect rays 90°, the deflection is changed to 180° if the ray is just entering the blackbox. In Figure 5, R1 is immediately reflected, and its end position is the same as its start position. A Hit that is border reflected is immediately unsatisfied (Rule 1). An Exit in the same situation would only be satisfied if it was a reflection (i.e. if its start position was the same as its end position, as per Rule 1).

If there are two atoms on the border such that one would result in a hit and the other would result in a border reflection, Black Box rules state that the hit should be counted first. Therefore, R2 in Figure 5 would satisfy a Hit, but not an Exit.

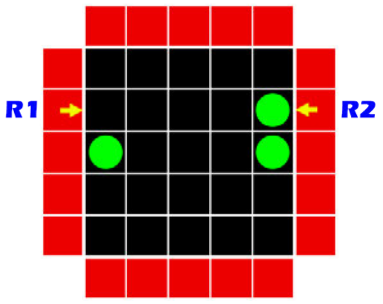


Figure 5. *Example of border reflection.* Neither ray can enter the board due to the border atoms. By definition, R1 is an Exit (specifically, a reflection) and R2 is a Hit.

These reflections are easily resolved when constructing the boolean formulas. For Hits, these can be accounted for by including an extra sub-expression before generating the bulk of the expression recursively:

$$(\neg P_{CCW} \wedge \neg P_{CW}) \wedge (P_{hit} \wedge h(x, y, D))$$

Exits are more complex, since the behavior is different depending on whether or not the Exit is a reflection. For reflections, P_{CCW} and P_{CW} are valid atom placements that satisfy the hint. However, the ray can still be obstructed by an atom directly in front of it. Therefore, the expression that must be included before generating the rest of the expression for a reflection Exit is:

$$\neg P_{hit} \wedge ((P_{CW} \vee P_{CCW}) \vee e(x, y, D))$$

And for an Exit that is not a reflection:

$$(\neg P_{CW} \wedge \neg P_{CCW} \wedge \neg P_{hit}) \wedge e(x, y, D)$$

3.6 Border Cases

The generation algorithm is greatly simplified when rays travel along the border, since it limits what kind of deflections can occur. A ray travelling along the border can only move forward or be deflected out of the board; a deflection towards the center of the board would have to be caused by an out-of-bounds atom. These scenarios allow us to ignore conditions where deflections are caused by an out-of-bounds atom, decreasing the number of recursive calls made.

For example, if a ray were traveling south along the left edge, it would be forced out of bounds by a clockwise deflection (Fig. 6). If this ray were searching for a Hit, the simplified expression for this border case would check that no deflection out of bounds occurs and that the ray either collides with an atom at the next position or at some later position. Notice that only one recursion is necessary, as opposed to the three recursive branches in Section 3.3:

$$\neg P_{x+1,y+1} \wedge (P_{x,y+1} \vee h(x, y+1, S))$$

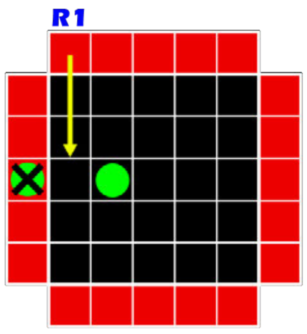


Figure 6. *Border travel.* The southward travelling ray R1 cannot be deflected counter-clockwise, because any P_{CCW} position would be outside the board. R1 is forced to keep travelling southwards or exit the board by some P_{CW}.

3.7 Dealing with Cycles

Due to the exhaustive nature of our case-analysis method, which considers all possible ray moves during recursion, there is a possibility that one branch of the algorithm reaches a cycle, trying to check a ray position that it had earlier come across. Left unaccounted for, this can lead to infinite recursion and stack overflow in non-trivial boards. We got around this error by continuously updating a list of "checked" cells that holds the positions that the algorithm has already visited. If a cycle has been detected, the recursion down that branch immediately terminates.

However, to maintain algorithm correctness, we found that we had to let each recursive branch use and maintain separate copies of the checked-squares-list to ensure that they did not cause one another to prematurely terminate; if each recursive branch referred to the same, centralized list of checked squares, breaking out of cycles resulted in incorrect solutions for some boards.

3.8 Considering Black Boxes

Individual hints are instantiated as objects of the abstract AHint class, from which the Hit and Exit sub-classes extend. Hits are created with an initial position and direction, whereas Exits are constructed with an additional exit position. Initializing an AHint is all that is required to be able to generate its characteristic formula. However, considering individual hints will not yield any interesting results; most Black Box games only become inconsistent when multiple hints are in contradiction.

To determine the consistency of an entire game of Black Box, a BBGame object can be created to encapsulate a list of hints. The BBGame constructor will ensure that it has received a well-formed array of AHints – specifically, it checks that each hint has been initialized with proper start positions and directions, and that the list of hints has a uniform board size.

To check for consistency, the BBGame will combine each individual hint expression with conjunctions to create a single characteristic boolean formula for the entire board. This formula is then passed to the LogicNG CNF converter, and then into MiniSAT to check consistency. If MiniSAT concludes satisfiability, one potential model is recovered and parsed into a plain-text board depiction showing empty and occupied squares. Our program is therefore capable of acting as a solver for consistent boards.

4 Metrics

The algorithm outlined above was checked for correctness using a set of four home-made boards and six boards sourced from online Black Box games. These boards were constrained to 5x5 and smaller to avoid efficiency concerns.

Tests were conducted on a Windows 10 machine with an Intel i5-6600k processor, running at 4.3 GHz, and with 16 GB RAM.

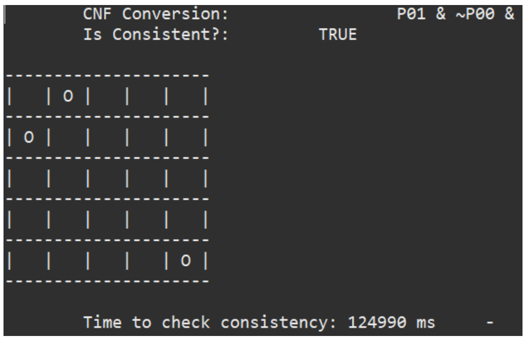
Runtime did not prove to be as big of an issue as memory usage, even though our recursive scheme added a lot of bulk to our boolean formulas. For boards exceeding a side length of 5, trying to run the formula generation algorithm would sometimes run into JVM OutOfMemory exceptions, due to how lenient the algorithm is when detecting cycles. In the future, this algorithm can definitely be improved from a memory usage perspective by only computing the recursive analysis for each square once, and allowing future recursive calls to access the already-computed result rather than completely recomputing the recursion for itself.

The runtimes (in milliseconds) for each of the 10 examples are given in the table below.

Board	**Side Length**	**Time (ms) fast-enabled**	**Time (ms) slow-enabled**
unsatEx1	3	328	296
satEx2	3	212	218
bigGame	5	11820	22734
bigUnsat	5	2057	2960
online1	5	9480	17570
online2	5	12506	20895
online3	5	10840	21156
online4	5	10049	21498
online5	5	10103	19454
online7	5	11298	21721

The average fast-enabled runtime between the six online boards was about 10713 ms, whereas the average slow-enabled runtime was about 20382 ms. Anecdotally, these runtimes varied greatly depending on machine specifications. For instance, on an Intel i7 processor @ 2.2 GHz, 16 GB RAM, solving the 5-by-5 online boards took about 2 to 3 minutes each.

For all 10 test results, the program returned board consistency as expected. Because the online boards originate from playable Black Box instances, all six of of them are by definition consistent – they each have a valid configuration of balls that satisfies all hints since they have solutions. In addition to ensuring that our algorithm did return that these boards were indeed consistent, using playable boards as test data exercised our algorithm's board-solving capability. For each of the consistent boards, our algorithm gave a valid solution, but not necessarily the expected one. Two examples of our algorithm recovering solutions from consistent boards are are shown in figures 7 and 8 below.

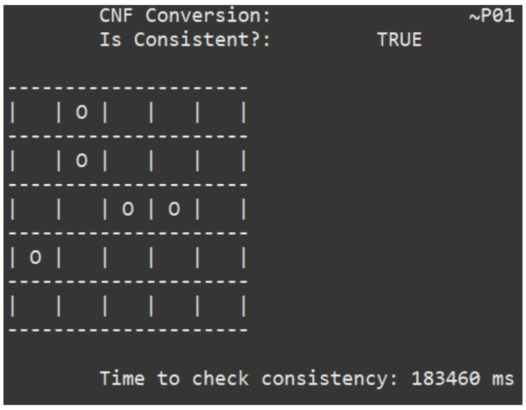


```
        CNF Conversion:                   P01 & ~P00 &
        Is Consistent?:         TRUE

---------------------
|   | o |   |   |   |
---------------------
| o |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------
|   |   |   |   | o |
---------------------

        Time to check consistency: 124990 ms      -
```

Figure 7. *Result of solving the online1 Board.* Our algorithm found the exact solution expected for the 5-by-5 hint-set online1.



```
        CNF Conversion:                          ~P01
        Is Consistent?:         TRUE

---------------------
|   | o |   |   |   |
---------------------
|   | o |   |   |   |
---------------------
|   |   | o | o |   |
---------------------
| o |   |   |   |   |
---------------------
|   |   |   |   |   |
---------------------

        Time to check consistency: 183460 ms
```

Figure 8. *Result of solving the online2 Board.* Our algorithm did not find the exact solution, but the solution depicted above was hand-checked with each of the hints and determined to be equally valid under our ruleset. The true solution only has three balls; it does not have balls at positions (1, 1) or (2, 2). We verified by substitution that the true solution does indeed satisfy the boolean formula generated for this board, even if it was not the depicted solution.

5 Summary

Black Box is an obscure puzzle game in which players shoot rays into a square board and analyze the end behavior to guess the locations of atoms hidden inside it. Abstracting the shooting mechanic as a "hint" for the locations of atoms in a board, we can analyze all hints for a given board and determine if there exists a corresponding atom configuration that satisfies it. We can naively compute all possible ray traversals inside a board that lead to complete hint satisfaction as a boolean expression and SAT-solve the expression to determine the board's consistency. For consistent boards, we can recover a model that depicts a valid set of atom placements, giving a solution to the board.

Our algorithm was tested on a set of ten boards: four home-made and six sourced from online Black Box games, but all 5-by-5 or smaller. In all cases, our program reached the correct conclusion regarding board satisfiability, and returned valid solutions for all consistent boards. However, for large enough boards, we found that generating the characteristic boolean formula led to an exorbitant number of recursive calls and memory usage. In the future, a better system to keep track of previously-computed recursive calls should be implemented in order to optimize the program's memory usage and enable it to run on larger board sizes.

6 Source Code

All our code is available [redacted]

References

[1] M. Barral, “Blackbox, the First Scientific Board Game.”

[2] J. Shepard, A. Monji, and H.Tejeda, “Black Box: The Ultimate Game of Hide and Seek,” , 2013.