BlackBox Solver

ALEXANDER JANNS, Eberhard-Karls-Universität Tübingen

ACM Reference format:
Alexander Janns. 2019. BlackBox Solver. 1, 1, Article 1 (February 2019), 3 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 ABSTRACT

A solver was written for the stated problem of a slightly modified BlackBox game, where every possible feedback was given as input and an arrangement of 'atoms' on a game board satisfying that feedback was desired as output. A brute-force solution was first developed and then improved upon by introducing a preprocessing step derived from game rules.

2 INTRODUCTION AND PROBLEM OVERVIEW

2.1 BlackBox as a Game

BlackBox was designed by Eric Solomon in the 1970's, inspired by the possibilities of the CAT scanner. Similar to the medical application, he designed a game where the objective is to derive a layout of 'atoms' in a square space divided into fields that can only be explored by examining the feedback of probing 'rays'. These rays can enter the board from any field on any side of the board, but cannot travel diagonally. The player is scored on how few rays they needed to correctly identify the hidden layout.
The feedback the player can receive is one of four possibilities (see Fig. 1):

- Hit: The ray has hit an atom directly. Only the entrance field is known.
- Miss: The ray has left the board on the tile directly opposite. Only the entrance and exit fields are known.
- Reflection: The ray has left the board from the exact field it entered from.
- Detour: They ray has left the board from any other field.

The player can infer the positions of atoms based on this feedback by using a process of elimination and applying the following rules of how rays and atoms interact (see Fig. 1):

(1) A ray is absorbed when it enters a field occupied by an atom.
(2) A ray is deflected at a 90° angle away from an atom when it enters one of the diagonally adjacent fields of an atom.
(3) In the case of a conflict, (1) takes precedence over (2).
(4) Nothing else interferes with a ray.

2.2 Problem Overview

The stated problem was to write a program that, given a full set of feedback plus board size and atom count, computes an arrangement of atoms that satisfies that feedback. The input is defined as a plain text file where each line represents one ray's feedback, except the first line, which represents the board size and atom count.
The feedback itself is either two numbers or just one. One number represents a Hit feedback and corresponds to the entrance point of the ray. Two numbers represent Miss, Detour and Reflection, which can be differentiated by examining their exit points. Entrance and exit points are interchangeable and thus each pair of numbers is only included in the input once.
Game rules are the same as described above, with one addition: An atom's deflection area can extend outwards of the board if it is positioned at an edge. This causes rays that try to enter the board from those fields to be immediately deflected, never even entering the board. Such rays have neither an entrance nor an exit field and are thus missing from the input file.
The desired output is a line for each atom giving its coordinates on the board. Board coordinates use the top numbers for the first value and the right numbers for the second (see Fig. 1). Thus, the top left

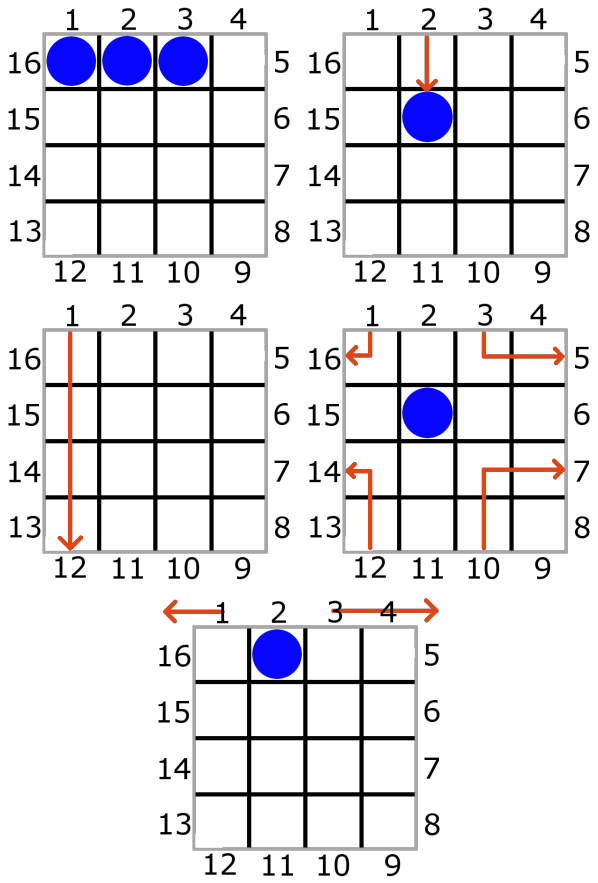

Fig. 1. Illustrations of (from left to right, top to bottom): a sample board with three atoms, a Hit, a Miss, four Detours caused by one atom and the special rule regarding interactions of rays and atoms at the edge of the board. Note that numbering is done clockwise around the entire board.

2019. XXXX-XXXX/2019/2-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

coordinate would be (1,6) on a 5-by-5 board and (1,7) on a 6-by-6 board.

3 SOLUTION

3.1 Basic Idea: Brute-Force

The principle of this is, of course, that we will eventually arrive at a viable solution if we just try every single possibility. The first question is then: What is the part we iterate? The answer is that in each step we generate a new configuration of atoms, since we will eventually find one configuration that matches the input, given that the input was valid. The next step is to verify that the current configuration is or is not the one we are looking for. We do this by the same process by which the input would have been generated: 'Firing' every probing ray and recording its feedback. Then we compare that feedback list to the input. If they are the same, we have found the solution, if they are not, we generate the next configuration and try again.
This approach is viable only for small boards and low atom counts, as the number of possible configurations follows the binomial coefficient $\binom{n}{k}$, where n is the number of fields on the board and k is the number of hidden atoms. This means that a 5-by-5 board with 5 atoms has 53, 130 different configurations, while an 8-by-8 board with 5 atoms already has 7, 624, 512. Additionally, for each of these configurations, $4n$ rays have to be evaluated to yield the complete feedback.

3.2 First Optimization: Feedback Mismatch Detection

Any ray has to be traced across the game board to produce its feedback. Since these paths can potentially be long, it seems sensible to reduce the number of fired rays per configuration. To achieve this, a check is introduced to see if the original input contains a feedback matching the one generated by any ray. If it does not, it follows that this configuration cannot be used to generate an identical feedback to the input and is thus not a viable solution. This approach saves the most time if the mismatch is detected after only a few or even the first ray. However, the assumption was made that most configurations should create a mismatch early on, as a configuration far from the solution would be expected to create a very different feedback list. As such, it is estimated to be overall preferable to iterate the input list once per ray instead of once after every ray has been fired.

3.3 Second Optimization: Preprocessing

By examining the input, it is possible to identify where some atoms are or at least narrow down their positions. This is possible by one key observation:

> It is impossible for a ray travelling along an edge to be deflected inwards into the board.

From this observation follows that any Miss feedback reported by such a ray is only possible when the path of the ray and the spaces directly adjacent to it are clear of atoms. As such, these spaces can be ignored when testing possible configurations, which significantly reduces the amount of cases to test (by reducing n). See Fig.2 for an example. Additionally, once a Miss feedback is detected along an edge, the neighboring fields can now be interpreted as new "edge" fields and the process repeated until something other than a Miss feedback is reported.

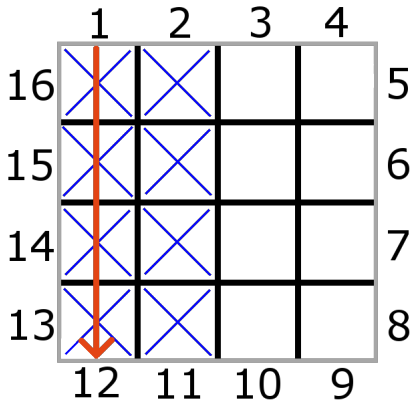

Fig. 2. Spaces that cannot contain any atoms, as determined by the orange ray (Miss feedback). A ray entering at 2 can now also be interpreted as an edge ray.

In a similar matter, any Detour feedback created by a ray travelling parallel to an edge (be it the real edge or an inferred edge as seen above) can only be caused by an atom in one spot (see the Detour example of Fig. 1). That means we can already place that atom on the board permanently and in doing that, also reduce the amount of permutations needed to test (by reducing k).
The final feedback that allows to narrow down atom positions is the special feedback of a ray being deflected before it enters the board. However, this feedback only narrows the position down to two possibilities, as the feedback naturally does not contain any information on the direction the ray was redirected to. To represent this, "XOR atoms" are introduced, that iterate between two possible positions when configurations are produced. There is a special case here (see Fig. 3): If a Reflection feedback is flanked by two Hits and at least one "instant deflection", we can deduce that there have to be two atoms at the edge here. Overall, these reduce the number of possible configurations to
$\binom{n-e}{k-f-x} \cdot 2^x$, where e is the number of excluded fields, f is the number of fixed atoms and x the number of XOR atoms. In the best case, this reduces the number to 1, but the worst case is still better than not preprocessing, as there is no configuration of hidden atoms where the analysis yields no excluded fields, no fixed atoms and no XOR atoms.

3.4 Special Case: Indistinguishable Boards

In addition to the special cases mentioned above, there exist boards that can yield identical feedback to other boards with different atom configurations (Shepard et al., 2013). The solver only reports

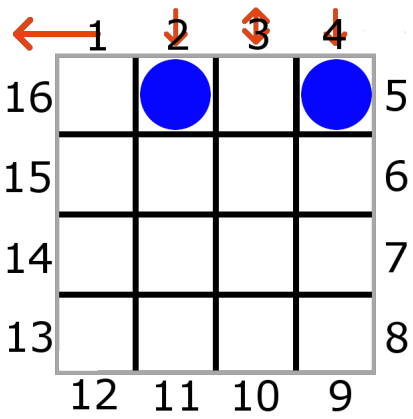

Fig. 3. Special case where two atoms can be fixed based on all combined feedback.

the first matching configuration it finds. Since there is no way of telling which of the configurations is the intended one, this cannot be resolved and the reported configuration has to be considered correct.
A case of indistinguishable boards arises, for example, from atoms arranged in such a way that no ray can enter specific parts of the board, and so no information about that part can be gleaned from ray feedback. Conversely, this means any atoms in such areas do not contribute to the feedback output at all and can be freely placed.

4 IMPLEMENTATION

4.1 Choice of Programming Language

The solver was written in Java. As an object-oriented language, Java allowed for simple abstraction of the game elements as objects, enabling an easy-to-understand approach to the solution.

4.2 General

The program is comprised of several classes, which can be grouped by their purpose: Representation (Board, Cell, Coordinates, Marble, Ray, Solver), Support (CoordinateListParser, MarbleArray, XORMarble) and Analysis (Analyser, Checker). Note that the ACM problem calls atoms "marbles", which is reflected in the naming of the respective classes.

4.3 Representation

These classes are used to either provide "graphical" (read, ASCII) output of the game board or an intuitive way to handle elements of the game.
The first is handled by the class **Board**, which both holds the game board (represented as a two-dimensional array of Cells) and provides a method to output that board to the console in ASCII format. It also features methods to create random boards, which is intended for testing.
The **Cell** class is a representation of what a field can contain (marbles ("o"), excluded fields ("x") and XORMarble positions ("?")).
The **Coordinates** class holds pairs of values representing board coordinates, but also input pairs. Since input can also be only one value, but never negative, negative values are used in Coordinates to represent "empty".
The **Marble** class represents a single marble, but can represent all types (free, fixed, XOR). The method advance() returns a new Marble at its new position (depending on type).
The **Ray** class represents a single probing ray. It holds a reference to the board it is "traversing" and updates its position based on what is directly ahead of it. The feedback is reported as Coordinates.
The **Solver** class acts as the front end for the Analyser and Checker classes and provides output of the results, especially that of the preprocessing step.

4.4 Support

These classes provide additional functionality.
The **CoordinateListParser** class reads the plain text file and parses the content into an ArrayList of Coordinates usable by the rest of the program.
The **MarbleArray** class manages an array of Marbles. It has its own advance() method, which coordinates the calls to Marble.advance() and ensures that no collisions occur.
The **XORMarble** class serves as a container for the two positions that marble can inhabit. It is used only to initialize a Marble in XOR mode.

4.5 Analysis

The **Analyser** class does the bulk of the preprocessing work. It starts on the left edge and performs the rules outlined in section 3.3 for all four edges.
The **Checker** class is initialized with an ArrayList of Coordinates representing the original input to test against. It takes the currently created solution (a Board) and casts Rays to determine if the given Board is a valid solution.

4.6 Code

The source code of the program can be found on GitHub.

5 LITERATURE

SHEPARD, Joey; MONJI, Archie; TEJEDA, Helio. Black Box The Ultimate Game of Hide and Seek. 2013.