

# Puzzle Solver

Tony Chahoud

for my GITHUB

## 1 Introduction

The Vox puzzle consists of a rectangular grid containing surveillance nodes that need to be destroyed, passive nodes that block the explosion of bombs, empty cells, and the bombs that the program will insert into the grid. The goal is to destroy all the surveillance nodes using the available number of bombs and remaining turns. Each game is given by a rectangular grid with a certain variable width and height, and the program returns all the possible solutions. Surveillance nodes are represented using the symbol '@,' which are positioned on the grid. Indestructible (or 'passive') cells have the symbol '#,' which are also present on the grid. A bomb is represented by [value], and the value could be anything from 1 to 3, indicating the time left for explosion, and null values are represented by an '.'. This symbolic representation is shown in the attached photo below.

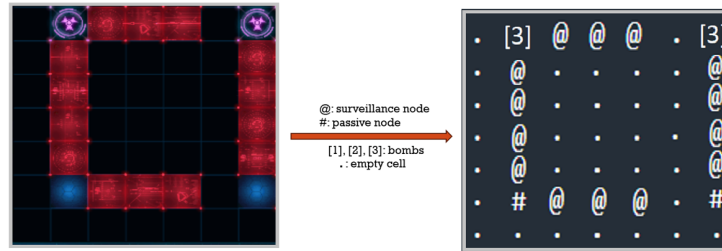


Figure 1: The grid of the puzzle

During each game turn, The puzzle can either place a bomb on the grid or wait. If no interference occurs, the bombs will explode after 3 turns once placed. A bomb might trigger another bomb to explode instantly, skipping the remaining turns it usually waits; more on this later. The bomb's explosion propagates in all four directions horizontally and vertically, with a range of 3 cells in each. It is only possible to place a bomb on empty cells and not on any other type of cell. If the number of turns allocated is surpassed and the surveillance nodes have not all been exploded, then the game is lost. Thus, for the program to win

the game, it must demolish all the surveillance nodes using the finite number of bombs and turns it has.

This puzzle presents an intriguing challenge that demands sophisticated programming techniques for an optimal solution. The implemented Prolog code updates grids and tree data structures using recursive exploration and back-tracking calls, making it a depth-first search algorithm implementation. Recursion plays a significant role in various predicates within the code, enabling the exploration of different actions, grid updates, and state evaluations.

The solve function, explained in detail below, uses recursion to iterate through different combinations of actions, like placing bombs or waiting, and to explore deeper into the search space. Prolog's capability to handle recursion and back-tracking provides a natural and expressive approach to defining recursive algorithms, simplifying the implementation of iterative processes involving repeated computations or exploration.

let's provide an example of a puzzle with the following 7x6 grid:

```
[.,.,.,.,.,.],[.,.,.,.,.,.],[.,.,.,.,.,.],[.,.#,.,.#,.,.],[#, @,., @, #,., @],[., #,.,.,., @],
```

where the number of bombs is 3, and the number of turns is 4. The output returned by the implemented Code is: Steps = [[1, 7], [5, 3], [3,7], wait] This is the list of steps representing the actual result of the puzzle, highlighting the optimal bomb positions required to achieve a win.

## 2 Algorithm

In our initial phase of development, we focused on implementing some logical rules that played a crucial role in the overall logic and integration of the project. These functions served as the foundation for building the entire system, enabling smooth connectivity and seamless execution of the algorithms.

### 2.1 Fundamental Functions

We initiated the implementation process by creating fundamental functions, such as *decr* for decrementing a variable and *incr* for incrementing. These functions were utilized during the propagation phase to update the coordinates of the cells while moving in the four directions (up, down, left, and right).

Another crucial function is *indexer*, responsible for validating the existence of an element at coordinates R and C within matrix M, identified as V.

We proceed to implement the *replace nth* function, utilizing Prolog's built-in *nth1* to replace specific list elements by index. This function proves critical in various sub-functions, enabling efficient replacement of bombs or cells during propagation under specific conditions. For example, during the propagation

process, upon eliminating surveillance nodes, we replace those nodes with null or  $\{.\}$  and transform [fire] into the letter "b," indicating its burnt state and signifying completion of the propagation phase. This optimization helps the program skip some combinations during the search.

## 2.2 Solve Function

Upon introducing fundamental functions, the puzzle-solving process was established, culminating in the creation of the main function, *Solve*. This central function serves as an efficient mediator, communicating with all the rules and constraints to seek the optimal solution for the puzzle.

This function takes four arguments: "Input-Grid," representing the puzzle to solve, "[3]or[2]or[1]" indicating the number of available bombs, "Turns," representing the number of available turns. and "Steps," the sequence of moves that will eventually lead to solving the game. The function returns all possible winning sequences, although only one may be needed to achieve the solution. The solution process hinges on three main logical components:

### 2.2.1 Determining the Next Move

The *Solve* function intricately assesses various possibilities to determine the next move in the puzzle-solving journey. It meticulously evaluates options such as placing a bomb  $\bar{X}, Y$  or waiting, aiming to advance toward the optimal solution. These decisions are governed by a set of conditional rules only if the number of remaining turns is greater than zero and, in the case of placing a bomb, if the number of remaining bombs is also greater than zero.

### 2.2.2 Updating Bomb Timers and Handling Explosions

A critical facet of the *Solve* function is the dynamic management of bomb timers and their explosions. For this purpose, the *wait grid* function was devised. This function actively monitors each bomb's actual status, appropriately decreasing timers. For instance, a timer with [3] diminishes to [2], [2] reduces to [1], and [1] transforms into "fire," each decrease accounting for one turn.

### 2.2.3 Handling Explosion Propagation

Another significant responsibility of the main function is the meticulous management of explosion propagation. Here, we designed the *propagate* sub-function. Its purpose is to adeptly govern the expansion of explosions throughout the grid, ensuring strict adherence to the game's rules and constraints. The "solve" function is thus repeatedly invoked with a simpler problem, each time updating the number of remaining bombs and turns and shortening the sequence of steps. This recursive process ultimately constructs a final winning sequence.

This screenshot summarizes the functionality of the 'solve' function when the Next move is placing a new Bomb:

```

solve(Grid,Bombs,Turns,[H|T]):-
    H=[X,Y],
    Bombs > 0,
    Turns > 0,
    replace_bombs(Grid,X,Y,[3],Temp_Grid),
    wait_grid(Temp_Grid,Future_Grid),
    New_Turns is Turns - 1,
    New_Bombs is Bombs - 1,
    propagate(Future_Grid,Updated_Grid),
    solve(Updated_Grid,New_Bombs,New_Turns,T).

```

Figure 2: Code of the Solve function

By employing recursive calls and updating the parameters with each step, the "solve" function effectively explores various combinations of moves, eventually leading to the discovery of the optimal solution for the puzzle. This recursive nature of the "solve" function and the careful handling of available resources play a crucial role in ensuring the program's success in solving complex puzzles.

### 3 Propagation Rule

The *propagate* function plays a central role in managing the explosion's propagation in all directions within the VOX puzzle code. It is a complex function that contains multiple sub-functions called recursively to iterate successfully three times in each direction. During each iteration, the function checks the conditions and acts accordingly. Figure3 summarizes the functionality of this function.

Now, let's delve into the details of the diagram. The bomb's state sequence: [3], [2], [1], "fire," and "b" is crucial in understanding the function's behavior. Each number represents the time left before explosion, "fire" indicates an active bomb ready to explode and propagate in all directions, while "b" denotes a burnt cell after propagation. Within the "propagate" function, cells marked as "fire" are given the highest priority.

To implement the "if fire, then propagate; else, do not propagate" logic in Prolog, the main "propagate" function call the sub-function "indexer" to check for the presence of fire. If fire is found, its coordinates in the grid are obtained and return them to the 4 recursive-functions: PropagateXR, PropagateXL, PropagateYU, and PropagateYD. This allows for multi-directional propagation: right, left, up, and down.

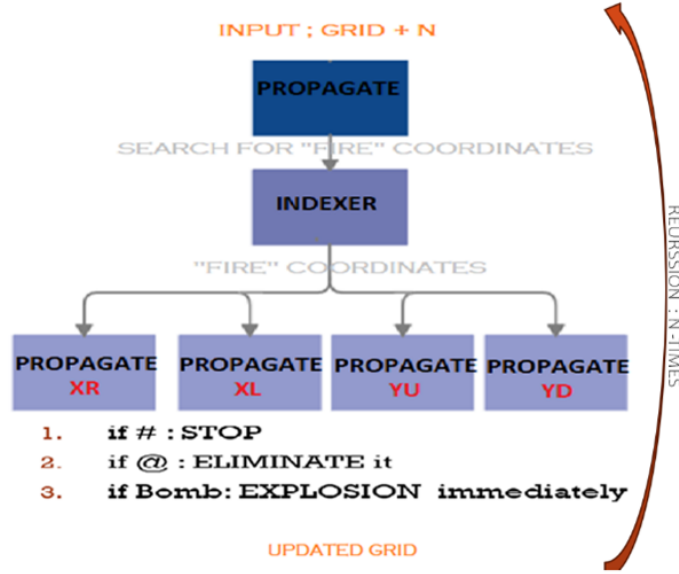


Figure 3: Propagation Rule Diagram

Each PropagateXR/XL/YU/YD function takes input N, Grid, R, and C. N corresponds to the explosion's length (set to 3 in our case) and decreases by one in each iteration until it reaches zero, effectively stopping the recursion. In certain cases, the loop is broken, and N is instantly assigned to zero, which will be further discussed.

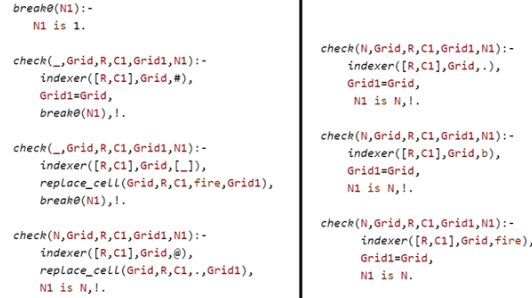
The "Grid" represents the state of the grid before the upcoming iteration, while "R" and "C" represent the row and column of the current index. The propagation in different directions occurs as follows:

- When propagating to the right using PropagateXR, the program increments the value of C by 1 to move one step to the right. The "check" function then evaluates the new index (R, C1) to efficiently and correctly perform the replacement.
- For propagation to the left (using PropagateXL), the program decrements the value of C by 1 to move one step to the left, updating the coordinates (R, C1).
- When moving upwards (using PropagateYU), the program decrements the value of R by 1 to take one step up, resulting in updated coordinates (R1, C).
- Similarly, for downward propagation (using PropagateYD), the program increments the value of R by 1 to move one step down, updating the coordinates (R1, C).

To prevent an index out-of-bounds error, a condition is implemented to increment/decrement the value of C or R only if it is smaller/greater than the overall number/lowest number of columns/rows in the grid. This process is then repeated depending on the length of propagation N (=3), resulting in three iterations for each direction. In each iteration, the coordinates are updated based on the logic discussed above.

These updated coordinates, in turn, update the grid based on specific conditional rules, executed by the sub-function "check," which is implemented six times to cover all possible scenarios. Depending on the encountered symbols, the "check" function determines whether to destroy a surveillance node, break the loop, trigger another bomb, or leave the cell unchanged.

This screenshot shows the code of the "check" function:



```

break0(N1):-
    N1 is 1.

check(_,Grid,R,C1,Grid1,N1):-
    indexer([R,C1],Grid,#),
    Grid1=Grid,
    break0(N1),!.

check(_,Grid,R,C1,Grid1,N1):-
    indexer([R,C1],Grid,[_]),
    replace_cell(Grid,R,C1,fire,Grid1),
    break0(N1),!.

check(N,Grid,R,C1,Grid1,N1):-
    indexer([R,C1],Grid,@),
    replace_cell(Grid,R,C1,.,Grid1),
    N1 is N,!.

check(N,Grid,R,C1,Grid1,N1):-
    indexer([R,C1],Grid,.),
    Grid1=Grid,
    N1 is N,!.

check(N,Grid,R,C1,Grid1,N1):-
    indexer([R,C1],Grid,b),
    Grid1=Grid,
    N1 is N,!.

check(N,Grid,R,C1,Grid1,N1):-
    indexer([R,C1],Grid,fire),
    Grid1=Grid,
    N1 is N.

```

Figure 4: Code of the Check Function

Now, let's delve into a detailed explanation of this code: For instance, if a passive node is encountered, the "check" function sets N to 1 and passes it to Propagate function, where it is automatically decreased by 1. Consequently, N becomes zero, leading to a call for PropagateXR/XL/YU/YD again without recursion, effectively breaking out of the loop. On the other hand, when another bomb is encountered, it is replaced with fire, and the loop breaks for optimization purposes, as remaining cells will be handled through recursion while dealing with succeeding bombs. Following the break, the main "propagate" function recursively checks for other fires, repeating the process if another fire is detected.

If a surveillance node symbol is encountered, it is replaced with a null value, ensuring smooth continuation of the iteration. When a "b" is encountered, the program leaves it unchanged to keep track of the locations where bombs have already been placed and exploded.

Finally, when a null or fire symbol is encountered, the program makes no changes, allowing the iteration to proceed naturally.

## 4 Optimizing Bomb Placement and Propagation

In our puzzle-solving algorithm, the process of placing a new bomb is carefully handled, considering specific conditional rules. When placing a bomb, we ensure that it cannot be placed in a cell already occupied by a surveillance node '@', an obstacle #, another bomb [ ], or a cell that has already exploded (b)/or Burnt. These conditions are crucial for optimizing the placement of new bombs and preventing unnecessary or invalid placements.

To achieve this optimization, we utilize the *replace\_bombs* predicate. This predicate evaluates the cell at the specified coordinates and checks for the presence of any of the aforementioned elements. If the conditions are not met, the bomb is placed in the grid. However, if any of the specified elements are present, the placement is skipped, and no further propagation occurs in that direction. This pruning of decision trees significantly reduces the computation required for bomb placement and propagation, leading to a more efficient solving process.

During the propagation process, we apply similar optimization techniques to control the propagation of explosions triggered by bombs. When a bomb explodes and triggers a new explosion in a neighboring cell, the propagation in that direction stops immediately by breaking the loop. The newly triggered bomb is directly replaced by fire, indicating its explosion, and it initiates a new propagation process. This strategy prevents the unnecessary continuation of propagation in directions where explosions have already occurred, saving computational resources, and improving the overall efficiency of our solver.

Additionally, the use of the cut (!) within some predicates was essential to commit to certain decisions and avoid unnecessary backtracking during the propagation process.

Optimization strategies for bomb placement and propagation in the puzzle-solving algorithm ensure that bombs are intelligently placed, and explosions are efficiently propagated. By adhering to specific conditional rules during bomb placement and strategically stopping propagation upon triggering a new bomb, our algorithm efficiently navigates grid-based puzzles, providing a highly optimized and reliable solution for solving a wide range of scenarios.

## 5 Conclusion

In conclusion, this report highlights the successful development of Vox Puzzle Solver, a powerful puzzle-solving algorithm tailored to navigate grid-based scenarios with a specific emphasis on bomb placement and propagation. By meticulously considering specific conditional rules, the algorithm achieves intelligent bomb placement and efficient explosion propagation, resulting in highly optimized solutions for eliminating surveillance nodes.

Looking ahead, the future work on the second version of Vox Puzzle Solver presents exciting prospects for further enhancement and expansion. A particularly intriguing avenue involves incorporating dynamic surveillance nodes capable of moving along columns or rows within the grid. This dynamic ele-

ment introduces an additional layer of complexity to the puzzle-solving process, necessitating extensive modifications and the generalization of our current logic to adapt seamlessly to these evolving surveillance nodes.

...