# Project for Languages and Algorithms for Artificial Intelligence - Module 1

Andrea Virgillito & Sfarzo El husseini
University of Bologna

## Introduction

Vox codei Episode 1 is an interesting puzzle that requires quite complex programming to be solved efficiently. The following program manipulated grids and tree data structures to find all the possible solutions for the game and implemented the Brute-Force approach. The game consists of surveillance nodes that should 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 of width and height[1] 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 presented on the grid, a bomb is represented by `[value]` the value could be anything from 1 to 3 indicating the time left for explosion, and the null values are represented by an "o". Each game turn it can either be placed a bomb on the grid or wait. If no interference occurred, the fork-bombs will explode after 3 turns after having been placed. A bomb might trigger another bomb to explode instantly skipping the remaining turns it usually waits, more on this later. The explosion of the bomb propagates in all 4 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 than the game is lost. Thus, for the program to win the game all the surveillance nodes must be demolished using the finite number of bombs and turns it has.

## The steps followed

The very first step for the project has been an internet research to get the most basic functions[2] that would have come in hand when in need; for instance the functions `decr` for decrementing a variable or the

[1] Passed manually alongside the rectangular grid

[2] In this document the clauses and the rules used are called functions because one of the arguments can be seen as the output of a function which takes as input the remaining arguments

function `indexer` to check whether the element with coordinates $R$ and $C$ of the matrix[3] $M$ is $V$. Among the first functions built there is `clean_grid` used to assess whether a grid is freed from surveillance nodes.
The function `replace_nth` (it replaces a list element by index by mean of the built-in `nth1`) was used inside the functions `replace_row_col` and `replace_row_col_bombs` needed respectively to replace an element of a matrix with another whatever the previous element was and if some conditions are met.
The latter is useful to avoid placing bombs in cells occupied by surveillance nodes, passive cells, another bomb or in case in the past another bomb has been placed there; indeed the program is able to remember this by marking a cell with the constant b[4] right after a bomb has exploded on it. The reasoning behind this is that if on a certain cell a bomb is needed to destroy a surveillance node, only one and one bomb is needed and it is never required to place a second bomb on the same cell, not even in order to exploit a domino effect for saving turns, indeed a second bomb cannot accelerate the explosions of already placed bombs; therefore this is for optimization purposes as the program will skip some combinations during the search. Once this have been obtained the backbone (comprising functions not already implemented) of the `solve` function, the one to invoke to actually solve puzzles, has been built. It became then clear that a function to update the timers of the bombs was necessary, and this summarizes the purpose of `wait_grid` which looks for elements of the type `[_]` and decreases the number inside of them, with the exception that the token `[1]` is substituted with the constant `fire`, representing an explosion that needs to be handled according to the rules of the game. The explosion handling is the result of the `propagate` function. In retrospect, making treasure of the experience gained, also solving the second episode of Vox Codei would have been possible: in that case it would be sufficient to generalize the `wait_grid` so that the surveillance nodes move along columns or rows, paying attention to their current direction.

## The solve function

The main body of the program is constituted by the `solve` function. This function take as arguments `Grid`[5], the puzzle to solve, `Bombs`, the number of available bombs, `Turns` the number of available turns and `Steps`, the sequence of moves to solve the game, this will return all the possible winning sequences although just one may be needed. The function in practice is made up by three definitions. The first one is true whatever the number of remaining bombs and turns if `Grid` does not contain any surveillance node, and in this case the sequence of `Steps` is the empty list. The other two definitions decide whether the next move in the sequence of steps should be `wait`, i.e. just wait, or `[X,Y]`, that means placing a new bomb in the cell of co-

[3] A list of lists
[4] It stands for "burnt"
[5] The upper left corner has coordinates (1,1)

ordinates (X,Y). Those two definitions can be true only if the number of remaining turns is greater than zero and in the case of placing a bombs also the number of remaining bombs should be greater than zero. A part from placing a new bomb or not these two definitions are identical as they call in order the functions `wait_grid`, `propagate` and `solve`. The `solve` function is thus invoked recursively with a simpler problem each time with the updated number of remaining bombs and turns and a shorter sequence of steps. At the end a final winning sequence will be built recursively.

# The propagate function

The function `propagate` was used to handle the propagation of the explosion in all directions. This function is quite complex, it contains multiple sub functions that are being called recursively to successfully iterate 3 times in each direction. During every iteration in each direction the function checks what are the conditions and acts accordingly. It is important to note that a bomb states sequence is the following: [3], [2], [1], fire, and b. The number indicates the time left before explosion, "fire" indicates that the bomb is on fire, and it will explode propagating in all directions, and "b" stands for burnt, and it is placed after the propagation is done. To obtain the (if "fire" then propagate else do not propagate) logic in Prolog the function `propagate` was written twice with different predicates. The rule that had its predicates satisfied was initiated and the process would carry on from there. For example, the main propagate function uses a sub function to check whether there is fire or not which is `indexer`. The input parameters for the main `propagate` function is the current grid, and it returns the grid after propagation if there were any. The `indexer` function which is the first predicate to consider inside the main propagate function takes as input the current grid, a certain token (in this example fire), and if there exists any it returns the coordinates of fire. In case the `indexer` returned true the coordinates of the fire in the grid are obtained and then the program calls `PropagateXR`, `PropagateXL`, `PropagateYU`, and `PropagateYU`. By doing so the propagation occurs from those coordinates of `fire` obtained in all directions right, left, up, and down. Now the iteration should proceed for 3 times, which is the length of the explosion. Hence, it is necessary to implement the concept of "for loop" in Prolog which is done by recursion. Each of those functions are written twice as well and based on their input parameters the program decides whether to proceed with the rule that will continue recursion or the one that will break out of the loop. For instance, `PropagateXR` takes as input `N`, `Grid`, `R`, and `C`. `N` corresponds to the length of the explosion and decreases by one through every iteration until it reaches zero and recursion stops, there are cases where the loop is broken and assign N to zero instantly which will be further discussed later. `Grid` corresponds to the grid before the iteration that will follow. `R` and `C` correspond to the row and column of the current index. When propagating to the right using `PropagateXR` the program increments the value of `C` by 1 to move one step to the right and then the function `check` sees what

is it at this new index to do the replacement efficiently and correctly. To avoid index out of bound error a condition is put to only increment the value of C if it is smaller than the overall number of columns in the grid. The function `check` takes as input the `N`, `Grid`, `R`, and `C1`. The letters correspond to the same meaning as they did in the `propagateXR` function, `C1` is `C` +1. The rule of `check` is written 6 times, once for each possible token. In each rule the first predicate determines which `check` definition to consider and this is given by the `indexer` as well. Based on the value of the index at `R`, `C1` the program decides whether to destroy a surveillance node, break, trigger another bomb, or does nothing. If a passive node is encountered, the rule `check` turns the value of `N` to 1 and passes it to the `propagateXR` where it is automatically decreased by 1; thus, `N` becomes zero and based on that input the program will call again the `propagateXR` that does not apply recursion and breaks out of the loop. If another bomb is encountered, it is replaced with `fire` and breaks out of the loop for optimization purposes because the remaining cells will be taken care of through recursion when handling the succeeding bomb; after breaking out of `propagateXR` the main propagate will recursively check for other `fire`s and if it will detect another `fire` the process repeats. If a surveillance node symbol is encountered it is replaced with a null value and the iteration continues normally. In case a `b` is encountered the program does not touch it to keep note of the places where bombs have already been placed and exploded. Finally, when a null or `fire` encountered, the program changes nothing and continues the iteration naturally. Propagation in other directions are handled analogously.