Project Final Report - A Solver for Numbrix Games[☆]

Zhuoyuan Song[a,]

[a]*Department of Mechanical and Aerospace Engineering, University of Florida, Gainesville, Florida 32611, USA*

Abstract

In Numbrix puzzles, players are asked to fill a grid, on which one or more numbers are given, with consecutive numbers in a horizontal or vertical sequential path. A game play user interface is designed in Lisp enabling users play all twenty-six hard coded boards. An intelligent solver is also design to automatically solve the puzzle. The plain depth first search algorithm is implemented first to solve only easy boards. Through adding in more constrains to the depth first algorithm, the solver is able to solve more complicated boards and in a shorter time. For most boards that are given, the solve can finish the puzzle in less than one CPU minute. However, for extremely large and hard boards, the solver is not able to solve it in one CPU minute.

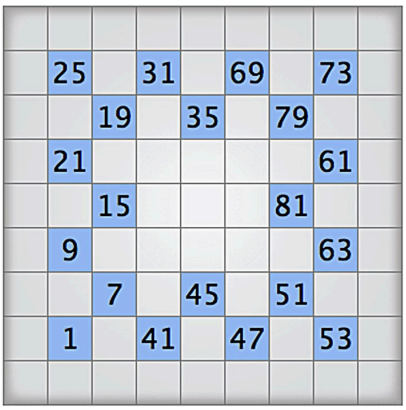

Figure 1: A 9×9 Numbrix example.

1. Introduction

The game Numbrix, created by Marilyn Vos Savant, is similar to Hidato, the goal of which is to fill the grid with consecutive numbers that connect horizontally or vertically. The only difference between Numbrix and Hidato is that diagonal moves are allowed in the latter. Numbrix grid boards are squire with one or more given numbers (Figure. 1). When the game finishes, an unbreakable sequential path from 1 to $N \times N$ (where N is the number of squares per row). The difficulty of the Numbrix is dependent on the size of the grid board, the number of given numbers and so forth.

In this project, a Numbrix game interface is designed and an automatic solver is provided in LISP. This report is organized as follows. Section 2 gives an introduction of the whole program. Section 3 describe the implementation approaches that were taken. A description of all programs and major variables used is presented in Section 4. A flow chart of the program is shown in Section 5. In Section 6, a calling hierarchy showing how the various functions call each other is illustrated. Section 7 talks about the intelligence that has been implemented. More details on potential solutions in designing a solver for Numbrix are shown in Section 8.

2. Description of the Entire Program

The entire program consists two major parts: manual play option and automatic play option. All available boards are hard coded in the program instead of reading from exterior files to make the submission folder neat. Some functional highlights of this program are itemized as follows:

1. Twenty-six boards are available to play ranging from 3×3 to 15×15;
2. An option between manual play and automatic play is provided;
3. During manual play, the player has options including "Restart the game", "Choose another board" or even "cheating moves" for some grid boards;
4. Invalid move checking is available to make sure the legitimate of inputs;
5. All moves performed by the player will be recorded and displayed on the user interface;
6. The program will recognize when the player successfully finish the play;
7. After each successful game, a re-game option is provided and even for some boards, the player is provided an optional to view the solution if he/she is not able to solve it;
8. In automatic play mode, the program will solve the Numbrix puzzle automatically and display the time it spends;
9. The option for viewing all the moves performed by the program is provided.

[☆]This work is submitted in partial fulfillment of the requirements for course project of *CAP 5635 Artificial Intelligence Concepts* (Fall 2013)

Email address: nick.songzy@ufl.edu (Zhuoyuan Song)

3. Implemented Approaches

In this section, some basic approaches used in the game play will be discussed. More focuses are placed on the methodology that has been applied in the automatic play mode.

3.1. Manual Play Mode

Manual play mode include a lot of useful options for the player and the methodologies are fairly simple. Here I will talk about only some of these functions that may be different from other designs.

3.1.1. Board Store and Print

All boards are stored as lists, the elements of which are lists of elements of each row of the grid board. All empty squares in the original grid board are represented by the number zero. When a particular game board is selected by the user, the program first replace all zeros on the board with spaces in order to print. Another variable recording the number of elements of each row is defined to be used in the formatting. When printing the board, the format of the board grid lines and borders are adjusted based on the size of the board. To visit a particular square on the board, the program converts the board to a 2D array and uses natural indexing.

3.1.2. Invalid Inputs Check

The move by the player is formated as (row column number). Indexes of all predefined numbers on the original grid boards are stored in a list. Whenever a move is performed by the player, the program will check if the square that the player is trying to access has a predefined number in order to avoid these number being changed. Additionally, the program will check if the index that is inputed by the player exists on the board. Moreover, for $N \times N$ board, the number that can be put on the board should be in the range of $[1, N^2]$. If the player tries to put a number out of this range, the program will also inform the player that the input is not valid and ask for another input.

3.2. Automatic Play Mode

The intelligence of automatic play is based on the depth first search method. In this part, two versions of automatic play methods will be presented. The first method is the pure depth first search method. The efficiency of this method is very low. For large or complicated boards, this method either takes a very long time to find solution or is not able to find a solution in 1 CPU time, which is a requirement of this project. the second method tries to fill some blank squares use some heuristic before performing the search. Meanwhile, when apply the search, more constrains are added to check a validity of a leaf node in order to kill a wrong path as soon as possible.

3.2.1. Depth First Search (DFS)

Before the search starts, all predefined numbers are stored in a list. The staring point (root node) of the search is the last predefined numbers, which located at the most right-down. The solver will grow from this number to N^2 first then continue the search by growing from the starting number to 1. Two exceptions are also considered when the starting number is either 1 or N^2. When it is the turn to place a number that is predefined, a searching for that number in the four adjacent squares is performed to check if the path up to now is correct or not.

This method is able to solve small and easy board really fast. For large boards with more blank squares, the time taken by the solver is extremely long or the solve fails to find a solution.

3.2.2. Smart Depth First Search (SDFS)

In some Numbrix puzzles, some blank squares can be filled out based on the given numbers. Many heuristics can be applied to deterministically determine a square by checking the numbers around or near this square.If more squares can be determined heuristically, there will be more constrains on the leaf nodes in the DFS such that the number of branches can be reduced to improve the performance of the search. However, by observing the grid boards that the DFS can not generate a solution, it is usually the case that only a little or even none deterministic heuristics can be apply. The improvement by applying these heuristics are marginal. So the effort should be focused on the improvement of the search. For this reason, I only apply a simple heuristic to try to determine some squares using given numbers. When the linear distance between two given numbers is equal to the difference of these two numbers, there path goes through these two numbers can be uniquely determined.

After adding this heuristic, the DFS is improved by using the Manhattan distance between two determined numbers to form the SDFS. In SDFS, a list is created to store all the predetermined numbers and the numbers determined using the heuristic along with the indexes of each number. Each element of this list will be in the form of (number index-x index-y), where the index is the natural index using by arrays staring from 0. Then this list will be sorted into an increasing order based on the number. The SDFS starts from the first number in the list, denoted as x_1, which is the smallest number currently on the board, and try to grow a path to the next number in this list, denoted as x_2. When growing the path to adjacent squares to place $x_1 + 1$ (assume $x_1 + 1 < x_2$), the program will check if the Manhattan distance between $x_1 + 1$ and x_2 is smaller than or equal to the difference between these two numbers. Since if the Manhattan distance is larger than the difference between these two numbers, it is impossible this path can go through both $x_1 + 1$ and x_2. When a path from the first number to the last number in the list is found, SDFS further search from the last number to N^2 if the last number is not N^2 and then grow from x_1 to 1.

Buy add this simple heuristic and the Manhattan distance check in the DFS, the resultant SDFS has significant improvement in terms of searching efficiency. Searching time for small and simple boards reduces dramatically and most other large and complicated boards can also be solve in less than 1 CPU minute. The Manhattan distance check helps the program find out wrong path more quickly such that the size of the search tree reduces, which results in the decrease in searching time.

4. Functions and Major Variables Description

In this section, descriptions of all functions and major variables will be given. The entire program can be found in Appendix I.

4.1. Variables

- ***First-submission-board-1*** : Original grid board with 0 as blank squares
- ***First-submission-board-1-length*** : Number of squares per row in *First-submission-board-1*
- **rowcount** : Counts for the number of row, used in board plotting
- **flag** : Completing check flag
- **boardArray** : Original board stored as a 2D array (read only)
- **brdArr** : 2D array stores the board being played
- **nonCell** : A list stores all the indexes that have predefined numbers
- **move** : A move made by the player
- **num** : Selects between *First-submission-board-1* and *First-submission-board-1* for some features
- **ncFlag** : Flag for actions on squares with predetermined numbers
- **regame** : Flag for regame
- **ele** : Temporary variable for a number on board
- **compCheckArr** : Board stored as an array used in completing check
- **movRecord** : A list of move history placed by the player
- **movCount** : A counter for valid moves
- **temp** : Temporary storage of values
- **listBrd** : Board stored as a simple list with all atom elements
- **counter** : A counter for the number of blank squares that have been filled
- **threshold** : The minimum number of moves required to finish a board
- **solu** : A flag indicating if the player want to take a look at the result
- **prdefNum** : A list of predefined numbers
- **step** : The step when grow a search tree from a number to another
- **endFlag** : A flag indicating if the solution is found (used in automatic mode)
- **foundFlag** : A flag indicating if the target number is found in an adjacent cell (used in the DFS solver)
- **failFlag** : A flag indicating if a search tree node fails (used in automatic mode)
- **AIchoose** : An option for the player to choose from manual play and automatic play
- **board** : The board that has been chosen to play
- **endPts** : A list consisted by all predetermined numbers and numbers determined using the heuristic along with their indexes
- **tS** : Starting time of the automatic play
- **tF** : Ending time of the automatic play
- **length** : The number of squares in each row of the board being played
- **firstR** : The row index of the starting square of the search (used in the DFS solver)
- **firstC** : The column index of the starting square of the search (used in the DFS solver)
- **firstNum** : The number in the starting square of the search (used in the DFS solver)
- **fRow** : The row index of the adjacent square where the target number is found (used in the DFS solver)
- **fCol** : The column index of the adjacent square where the target number is found (used in the DFS solver)
- **AIbrdArr** : The board being play store as a 2D array
- **distance** : The Manhattan distance between two squares
- **moveHistory** : The move history of the automatic play
- **moveHistoryChoose** : An option provided to the player to decide if he/she want to take a look at the move history performed by the SDFS solver

4.2. Functions

4.2.1. General Functions

- **(numbrix ( ))** : Main function of the program which displays all general information of the game, rules, boards options and reset some variables
- **(start ( ))** : Provides the option of manual play or automatic play and steps through the automatic play mode when selected
- **(replace0 (brd length))** : Replaces all zeros on the original boards with spaces. **brd** is the board and **length** is the number of squares in each row of the board

- **(replace0_row (row length))** : Replaces all zeros in a row with spaces. **row** is the row to be processes and **length** is the number of squares in each row of the board
- **(p_board (brd length))** : Prints the board with grid lines, borders and square numbering based on the size of the board. **brd** is the board and **length** is the number of squares in each row of the board
- **(p_row (row length))** : Prints a row. **row** is the row to be printed and **length** is the number of squares in each row of the board
- **(predef (brd))** : Creates a list of all the numbers currently on the board along with their indexes. The list is sorted in an increasing order based on the number and the indexes follow natural indexing convention used in arrays
- **(list< (a b))** : Sorts a list whose elements are all lists based on the first elements of inner lists in an increasing order
- **(regame? ( ))** : An option to re-game when the last game is finished (manually or automatically) or terminated

4.2.2. Manual Play Functions

- **(mov (brd))** : Displays all options allow during the game play, move history, warning/error messages and control the manual play procedure. **brd** is the board being played
- **(set_sqr (row clo val brd))** : Place new numbers in to a square. **row** is the row number of the square, **col** is the column number of the square, **val** is the number and **brd** is the board being played
- **(check (cell noncelllist))** : Checks if the square being placed number in by the player has a predefined number and avoids these number being changed. **cell** is the index of the square the player wants to make changes to and **noncelllist** is a list of indexes of all predefined numbers
- **(checkValInput (moveI brd))** : Checks if the number being placed by the player is valid, triggers the completion check and triggers other options made by the player including re-game, exit, cheat, etc. **moveI** is the move made by the player (different format based on the option) and **brd** is the board being played
- **(checkInputRange (moveR brd)** : Checks if the number being placed is between 1 and N^2. **moveR** is the number being placed and **brd** is the board being played
- **(solu? (num))** : An optional solution display provided to the player when he/she gives up in the game. This option is only available for *First-submission-board-1* and *First-submission-board-2* so **num** is the number of these two boards respectively
- **(cheat (row column num brd))** : A cheat in manual play to allow the program to fill in a given square for the play. **row** is the row number of the square to be filled, **column** is the column number of the square to be filled and **brd** is the board being played. This option is only available for *First-submission-board-1* and *First-submission-board-2* so **num** is the number of these two boards respectively
- **(listRm (brd))** : Removes all first level inner "( )" in a list. **brd** is the list to be processed
- **(dupliCheck (brd))** : Checks all numbers on the board to see if there are duplicated numbers. **brd** is the board being played
- **(compCheck (brd))** : Checks if the game is finished successfully. First makes sure there is no duplications on the board. Then checks if for all number x ($1 < x \leq N^2$), $x-1$ is in an adjacent cell. **brd** is the board being played
- **(neighber (array element row col))** : Checks if the number $x-1$ ($1 < x \leq N^2$) is in an adjacent cell of the number x. **array** is the board being played stored as an array, **element** is the number x, **row** is the row index of the adjacent square of x being examined and **col** is the column index of the adjacent square of x being examined
- **(recMov (move1))** : Records moves played by the player. **move1** is the move
- **(prtMov (movrcd))** : Prints the move history placed by the player. **movrcd** is the list stores the entire move history

4.2.3. Automatic Play Functions

- **(checkCell (row col array))** : Checks if the square being accessed is in the border of the board. **row** is the row index of the square, **col** is the column index of the square and **array** is the board being played store as a 2D array
- **(findAround (row col x brd))** : Finds a number in the adjacent squares. This function is only used in the simple DFS solver. **row** is the row index of the center square, **col** is the column index of the center square, **x** is the number being looked for in adjacent squares and **brd** is the board being played
- **(grow (r c num step AIbrdArr))** : DFS solver. **r** is the row index of the starting square, **c** is the column index of the starting square, **num** is the number in the starting square, **step** is the step of grow (1 or -1) and **AIbrdArr** is the board being played stored as a 2D array
- **(2d-array-to-list (array))** : Converts a 2D array to a list, whose elements are lists of elements in each row. **array** is the array to be converted
- **(moreConstrains (endPts))** : Applies a simple heuristic to connect two numbers in a line. **endPts** is a list of predefined numbers with their indexes in an increasing order based on the value of the number

- **(lineUp (num1 num2))** : Checks if two numbers in a line should be lined up based on whether the Manhattan distance between these two squares is equal to the difference between these two numbers. **num1** is a list consisted by the first number and its index and **num2** is a list consisted by the second number and its index
- **(connect (num1 num2 index step))** : Fills in all the blanks between two numbers in a line. **num1** is a list consisted by the first number and its index, **num2** is a list consisted by the second number and its index, **index** is an indicator for orientation of the line (index=1 if two numbers are in the same row and index=2 if two numbers are in the same column) and **step** determines if the path is increasing or decreasing (step=1 when number1 < number2 and step=2 when number1 > number2)
- **(growTo1 (Num)** : Searches for a path from the a number to 1. **Num** is a list consisted by the starting number and its index
- **(growToMax (Num))** : Searches for a path from the a number to N^2. **Num** is a list consisted by the starting number and its index
- **(manhattan (num1 num2))** : Calculates the Manhattan distance between two squares. **num1** is the index of the first square and **num2** is the index of the second square
- **(smartGrow (Num endPtsList))** : SDFS solver. **Num** is a list consisted by the starting number and its index and **endPtsList** is a list of all determined numbers currently on board along with their indexes

5. Program Flow Chart

A brief flow chart of the program with the SDFS solver is shown in Figure. 2.

6. Function Calling Hierarchy

A function calling hierarchy illustration is shown in Figure. 3, where rounded square (blue) are recursive functions.

7. Results and Discussions

The program is tested with all 26 boards hard-coded in the program including 2 first submission boards, 7 additional boards and 11 tournament boards. The testing machine is a Macbook Pro with a 2.3 GHz Intel Core i5 processor. The testing results using DFS are shown in Table 1 and the results using SDFS solver are shown in Table 2. It can be noticed that the SDFS solver is much more robust that the DSF solver. For most of the boards, the SDFS solver works very well and the solution can be found in less than 2 seconds. However, the solve is not be able to generate a solution in one CPU minute. By observing these two unsolved boards, it can be found that both boards are very large boards (15×15) and the given numbers spread very well. This indicates that the SDFS solver is not robust in solving boards with large dimensions.

Board	Time (CPU seconds)
First submission board 1	0.000856
First submission board 2	0.221476
Additional board 1	0.003432
Additional board 2	0.146830
Additional board 3	0.070998
Additional board 4	2.422119
Additional board 5	Unsolve in 60 seconds
Additional board 6	Unsolve in 60 seconds
Additional board 7	Unsolve in 60 seconds

Table 1: Testing results of SDFS with all hard-coded boards.

Board	Time (CPU seconds)
First submission board 1	0.000045
First submission board 2	0.026839
Additional board 1	0.000108
Additional board 2	0.028133
Additional board 3	0.005807
Additional board 4	0.000948
Additional board 5	0.058154
Additional board 6	1.853056
Additional board 7	Unsolve in 60 seconds
Tournament board 1	0.000086
Tournament board 2	0.000180
Tournament board 3	0.000158
Tournament board 4	0.000318
Tournament board 5	0.000193
Tournament board 6	0.000101
Tournament board 7	0.000386
Tournament board 8	0.000724
Tournament board 9	0.000421
Tournament board 10	0.022629
Tournament board 11	0.092747
Tournament board 12	0.000581
Tournament board 13	0.010907
Tournament board 14	1.729533
Tournament board 15	Unsolved in 60 seconds
Tournament board 16	0.067125
Tournament board 17	0.059566

Table 2: Testing results of SDFS with all hard-coded boards.

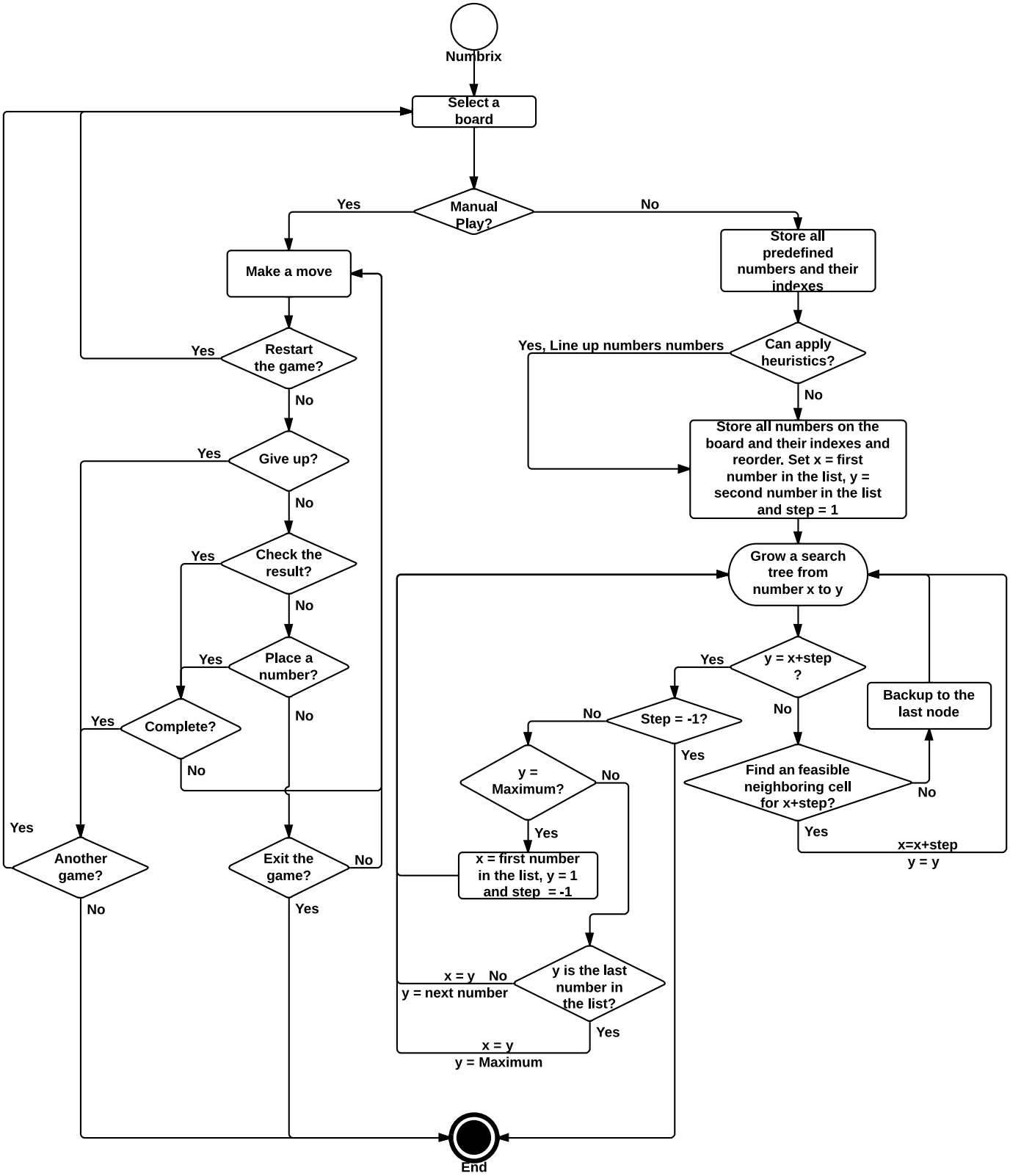

Figure 2: Program flow chart.

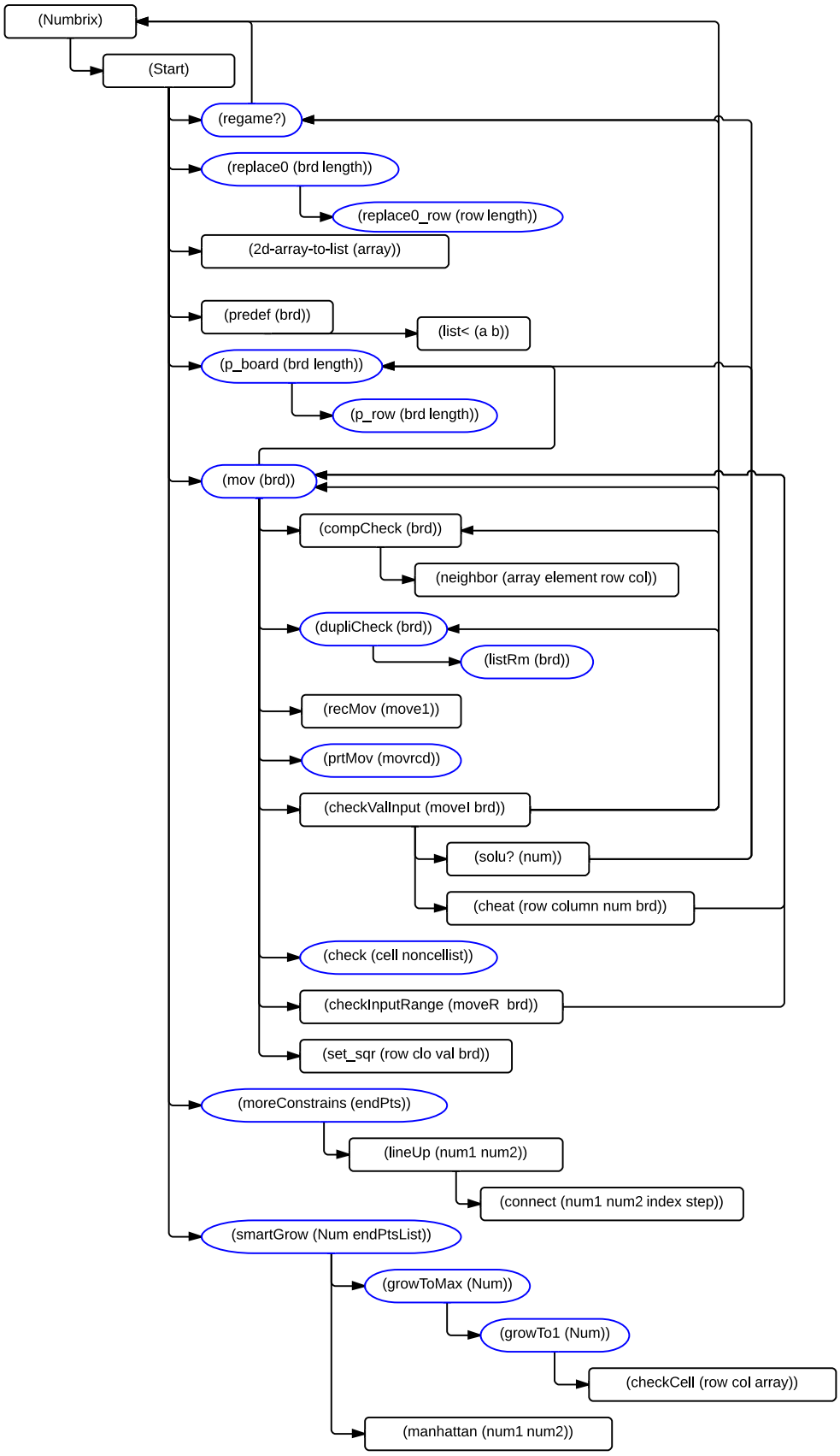

Figure 3: Function calling hierarchy illustration.

8. Conclusions

In this project, a Numbrix program is designed. This program provides the players with both manual play option and automatic play option. For automatic play, a simple depth first search (DFS) solver is designed. The DFS is able to solve small easy boards in a satisfactory speeds but fails in complicated and large dimension boards (> 9). Then a smart depth first search (SDFS) solver is designed by adding in a simple heuristic before the search starts and the Manhattan distance check when make a hypothesis. The SDFS has significant improvement in efficiency and capability compared with the DFS solver. Only two 15×15 can not be solved in one CPU minutes.

Some improvements are not implemented mainly due to the time limitation. In addition, the goal of this project is to solve all the boards in less that one CPU minutes. It is found that for the boards that simple DFS can not solve, very little heuristics can be applied to improve the search. The performance improvement trough implementing such heuristics is marginal. However, to further improve the solving speed, some of these heuristics can contribute a lot for some board.

One option is to categorize all the numbers predefined as closed cells (both $x + 1$ and $x - 1$ are in neighboring cells), [1] semi-closed cells (only $x+1$ or $x-1$ is in the neighboring cells.) and open cells (neight $x + 1$ nor $x - 1$ is in the neighboring cells.). For different type of predefined numbers, the number of blank cells around it can help determining the next number. For example, if only one blank neighboring cell is available and the number is a semi-closed cell, then the blank cell can be filled easily. Another potential heuristic is based on the shape of the path. Since the path should not be broken in the middle, a circled path with blank cells in the middle should not be taken.

There are other heuristics that can be applied to further improve the speed of the search. Nevertheless, a better searching algorithm is the key to really improve the performance of the solver. If necessary, a new solver design should start from the constrains of the search. The earlier the wrong path is deleted, the faster and more powerful the solve will be. Similar to the Manhattan distance checking, similar hypothesis guidance method can be used to generated a better prediction. Currently, the path growing sequence is fix to up, down, left, right. A dynamic search sequence can be applied adapted to specific scenarios. Moreover, probabilistic methods can be applied to add different weights to blank cells representing the likelihood of given cell to be on the path as the next step.

Due to the limitation of knowledge of Lisp, program codes are not very efficient. The confusion on the scope of variables may cause some inefficiency in real-time execution. Many other high-level functions may also be able to help improving the overall performance of the program. Finally, the entire program code is included in Appendix for the readers reference.

[1] Only $x + 1$ or $x - 1$ is needed for $x = 1$ or $x = N^2$. These two conditions will be ignored in later discussions for the simplicity of illustration but will be considered in practical implementation.