ARTIFICIAL INTELLIGENCE REPORT

FALL 2014

SLITHER

Submitted By:

Pasupuleti Sravya Prasad

UFID: 21229828

Table of Contents

Topic 1: BASICS OF SLITHER

- Slither is a game which played on a 5x5 to 25x30 grid with “+” at each of the grid corners.

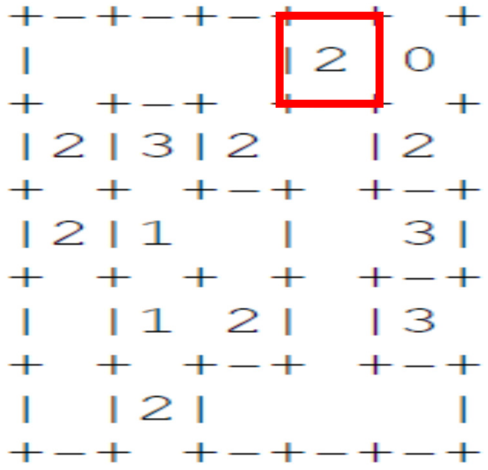
- Some of the squares contain a number between 0 and 3 inclusively. The number identifies the number of lines that will surround that particular square.

- Main objective is to find a single looped path with no crosses or branches.

- A player can either choose to load a file to solve or can load default puzzle.

- A player can choose to play manually or ask the computer to solve it.

- A sample grid is shown below with a cell marked in red.



- Each cell of grid can have four edges around it. The top edge is represented by “T”, bottom edge is represented by “B”, left edge by “L” and right edge by “R”.

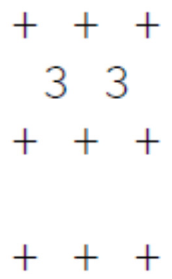
- Representation of the move is by “2 1 t” where 2 indicates the row element of cell in grid, 1 indicates the column element of cell in grid and “T “represents the edge on top.

- In manual mode, if the player enters the same move again, then the edge placed is removed. Player is prompted if the move entered is invalid.

- In automatic mode, the board is solved by computer and the result along with list of moves along with the time taken to solve is given to the user.

Topic 2: IMPLEMENTATION APPROACH

- First step which I took while implementing this game was to choose a representation for the game board. I used list representation for each row of the grid and thus the final board will be a list consisting of all grid rows as subsists. To illustrate this, I would show a simple board which is 2 X 2.

  ```
  +  +  +
    3  3
  +  +  +


  +  +  +
  ```

  Equivalent representation of this board in my program is:
  ((3 3) (NIL NIL))
  Here, when a cell contains no number which it is shown in list as NIL. Elements and there corresponding structure in list is as shown below.

+ (0 1)	(0 1)	+ (0 2)	(0 3)	+ (0 4)
(1 0)	(1 1)	(1 2)	(1 3)	(1 4)
+ (2 0)	(2 1)	+ (2 2)	(2 3)	+ (2 4)
(3 0)	(3 1)	(3 2)	(3 3)	(3 4)
+ (4 0)	(4 1)	+ (4 2)	(4 3)	+ (4 4)

- **Why this representation?**
  After fixing the representation for the board, I went ahead and implemented the same in lisp. And using functions in the later section I was able to generate the required representation. I used list representation because it seemed to be the most efficient of representing the board without using Arrays. Also, manipulation was easy for me as I worked mainly on lists before. An “n x m” grid is converted into 2n-1 x 2m-1 grid where.

- **Manual play with no intelligence**
  This step of implementation had two stages:
  1. *Take input from player and manipulate the board based on the input move*

This involved coding the program with no intelligence and to allow player to input into the board. Since, the player input format is a format “2 1 t” so I developed functions which could interpret the moves of users and place relevant edges on the board.

2. *Detect if a valid single loop is made by the move inputs from user.*

Firstly, I developed a logic for testing closed loop condition. I arrived at 3 conditions which need to be satisfied so that the closed loop is valid. If and only if all 3 are satisfied then a closed loop is formed across the grid.

- Sum of Horizontal “—” lines placed along each column of grid should be is even.
- Sum of vertical “|” lines placed along each row of grid should be even.
- Each cell in the grid should have edges around it equal to the number at the center. If there is no number then the maximum number of edges that can be placed around that cell is 3.

After developing the logic, I used functions to code this logic and detect the condition of closed loop.

3. Implementation of Search and Intelligence for automatic play

This was the most crucial and I would say the most difficult part of the project. I selected depth first search because it seemed most intuitive one to use. Another reason was that in this grid we have to traverse a path and then if it not correct we need to backtrack and then proceed or explore other paths.

I implemented code for depth first search. Initially my code had issue with backtracking, it could explore paths but was not able to remember which paths were explored and hence backtrack from there.

I did iterations with the code to make the backtracking work and finally could implement the depth first search. After, this the priority was to include as much as possible intelligence to make the search efficient. To not explore paths already visited. I changed my technique from what I had mentioned in the initial Intelligence report because I could not develop systematic approach for exploring nodes with that approach.

Topic 3: PROGRAMS AND PROCEDURES

Firstly, I am giving a brief of main functions of the program.

1. Slither
   This function is the main function of the code. It gives game instructions to the player.
   It allows the user to either load file by default or specify a file name. Depending on which option is selected, the encoded board representation is loaded. Also, it gives option to user to solve the loaded board manually or automatically.

2. Game in progress (brd)
   Game in progress is called by the main function slither. It is main function for the manual play. It creates the initial board configuration and then depending on the board selected, necessary data is populated into the lists. It also asks player to enter moves and every time a move is made the board is updates. When, the player has reached the correct solutions for the board, it informs player that they have won the game and list of moves made by the user are printed out.

3. Cloosedlooptest (initialbrd)
   This function is used to check the closed loop condition when a move is made. Whenever the player makes a move, if a valid closed loop is formed is it returns a true value or else NIL to indicate that the solution has been reached by the user. It has been coded to check the closed loop logic described in previous section.

4. Solve (brd)
   This function is the main function for automatic play and is called from the main function slither. It sets up the initial board configuration and initialized the board with the board that is being solved. This function first scans the whole board for a 3 in the board. When a 3 is found, it selects one corner of this cell as starting point for carrying out the DFS as we know for certainty that all the four "+" around a cell containing 3 will be path for solution for a closed path.

5. Processnode (i j NIL)s
   This is the function for depth first search. The starting node which was obtained in the step 4 is the starting point for depth first search. It starts the search for valid loop from starting point. It adds this node into node path. Then the neighbors are processed, all it four nodes are processed one by one. The first one which is encountered is selected and is checked for its validity, in order to eliminate out of index neighbors. When the neighbor is selected. It is then checked if this neighbor is same as the starting point, if it is then we check if a valid closed loop is being formed. If a valid closed loop is formed we just move out of the loop and return the path. If the neighbor is not the start point then we go ahead and check if it is already in the node path. If it is then this neighbor is eliminated. Then again, if this node is

not in node path, then process node is again carried out on this neighbor. In this way the recursive functions proceeds till a valid loop is formed.

This section covers all the functions which have been used for Manual and Automatic play. Calling hierarchy for these function is shown in the calling hierarchy section.

DESCRIPTION OF FUNCTIONS FOR MANUAL PLAY

1. **Slither ():** Main function where game rules are explained, input board is read and manual or automatic play is selected.
2. **Readboard (pathname)** : Reads the data from the text file
3. **Filenameloop ()**: Reads input from the player for the name of text file from which board is to be loaded.
4. **Game-in-progress (brd)**: Main function for the manual play. It passes the representation of the board as an argument to help create board in lisp.
5. **Grid-order (brd)** : Calculates the order of the grid to be create
6. **Make-board-list (initialbrd brd):** Inserts the data value to create initial configuration of the board.
   This function calls two functions:
   I. **Plus-row (row):** This function prepares the row with plus signs
   II. **Face-row (row):** This function prepares the row with no plus signs and thus has actual values
7. **Colprint() :** This function is simply used to print column numbers for the ease of player
   This function calls one function:
   I. **Spacing ():** This function is called to manage spacing
8. **Print-board-list (initialbrd):** Used to print the created board in the previous step.
   This function calls two functions:
   I. **Rowprint ()** : This prints the row number for the ease of player to locate cell
   II. **Print-board-row ():** This function is used to print each individual row with required spacing
9. **Move-from-user** (): This function is used to take input from the user for the move.
   I. **Isvalidmove()** : Checks if a valid move is entered by player.
   II. **Move-update-board ():** It updates the board with a valid move entered by the player
      A. **Evenrows (row ele):** Updates the even row elements according to move made
      B. **Oddrows (row ele):** Updates odd rows elements according to move made
10. **Print-board-list (initialbrd)** : prints board after the move is update in the board
11. **Closedlooptest(move)** : This function is used to check if the move made by the player results in a valid closed loop or not. If it leads to a valid closed loop, player is prompted that he/she has won the game.

I. **Countedgesh (intialbrd):** Calculates the number of horizontal lines in each vertical column of the grid and places into a counter.
   A. **Countrow (y)** : Iterative to calculate sum of horizontal lines in each vertical column of grid
   It calls a function Sqr (row col brd) to access particular elements of the board.

II. **Countedgesv (initialbrd):** Calculates the number of vertical lines in each horizontal row of the grid and places into a counter.
   A. Countrow (y) : Iterative to calculate sum of vertical l lines in each horizontal row of grid
   It calls a function Sqr (row col brd) to access particular elements of the board.

III. **Counttotalline (initialbrd):** Checks if the numbers of edges around a cell in a grid exceeds the number at the center of the grid.
   A. Countforsquares (val y x) : Accesses each grid, sees the numbers at center and calculates sum of edges around it. Sqr (row col brd) is also called by it.
   B. It calls a function Sqr (row col brd) to access particular elements of the board.

IV. **Checkforevenrow (row-counter)** : Checks if the counter set by countedgesh is even

V. **Checkforevencol (row-counter 1):** Checks if the counter set by countedgesv is even

VI. **Checkfortotallline (row-counter 2):** Checks if all the cells in grid satisfy the condition set by countotallines function.

DESCRIPTION OF FUNCTIONS FOR AUTOMATIC PLAY

1. **Slither ():** Main function where game rules are explained, input board is read and manual or automatic play is selected.
2. **Readboard (pathname)** : Reads the data from the text file
3. **Filenameloop ():** Reads input from the player for the name of text file from which board is to be loaded.
4. **Game-in-progress (brd):** Main function for the manual play. It passes the representation of the board as an argument to help create board in lisp.
5. **Grid-order (brd) :** Calculates the order of the grid to be create
6. **Make-board-list (initialbrd brd):** Inserts the data value to create initial configuration of the board.
   This function calls two functions:

I. **Plus-row (row):** This function prepares the row with plus signs

II. **Face-row (row):** This function prepares the row with no plus signs and thus has actual values

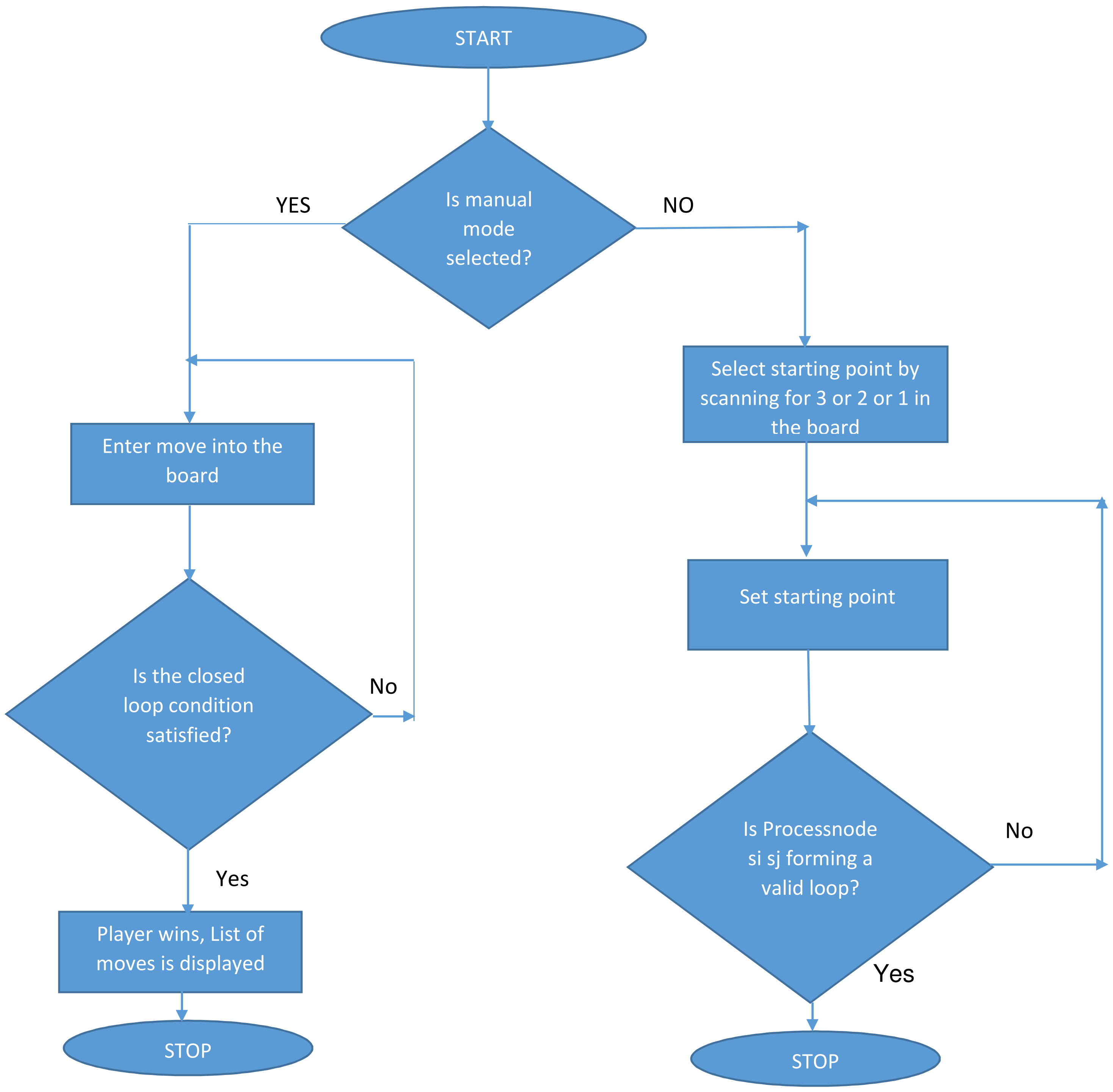
7. **Colprint() :** This function is simply used to print column numbers for the ease of player
   This function calls one function:
   II. **Spacing ():** This function is called to manage spacing
8. **Print-board-list (initialbrd):** Used to print the created board in the previous step.
   This function calls two functions:
   III. **Rowprint () :** This prints the row number for the ease of player to locate cell
   IV. **Print-board-row():** This function is used to print each individual row with required spacing
9. **Check3 (initialbrd):** It scans for 3 in the whole grid and then uses the value of remembers the grid position of the latest 3 encountered in the grid.
   Check 31 (list) function is called for this purpose
   I. **Check2 (initialbrd)** : It scans for 2 in the whole grid and then uses the value of remembers the grid position of the latest 3 encountered in the grid
   Check 21 (list) function is called for this purpose
   II. **Check1 (initialbrd):** It scans for 1 in the whole grid and then uses the value of remembers the grid position of the latest 3 encountered in the grid
   Check 11(list) function is called for this pupose
10. **Solvetime (si sj):** This function is has as arguments the starting point of the depth first search. It calls the function (processnode i j NIL) which is the main recursive function or depth first search.
11. **Processnode (i j NIL)**: This functions calls various other functions in order to traverse the graph from the start point systematically. It also starts with neighbor list of node (i j) to NIL and when it is explored neighbors are pushed into this list
   I. **Isloop (i j):** This function checks whether there is a loop. That is the coordinates of start point and the node (i j) called same, that means there is a closed loop.
   II. **Isvalidloop (i j):** This calls the function Closedlooptest on board to find out, if the loop formed is valid. The logic for closed loop is same as described in previous section of manual play.
   III. **Inpath1 (i j):** Checks if the node (i j) is already in the path in the node-path list or not.
   IV. **Isvalidnbr (list):** Checks if the node is out of index of the grid or not.
   V. **Checkedge (list i j):** Checks if there exists an edge between node (i j) and the node represented in the list form.
   VI. **Placedge (list i j) :** Places an edge between node (i j) and the node represented in the list form
   VII. **Inpath2 (list):** Checks if the node in list form is already in the node-path list or not.
   VIII. **Checkresult (result):** Checks if the processnode function when applied on node (i j) returns a result of 1 or 0.
   IX. **Removeedge (list i j):** Removes an edge between node (i j) and the node represented in the list form-

X. **Removenode (i j):** Removes node (i j) from the node-path list.

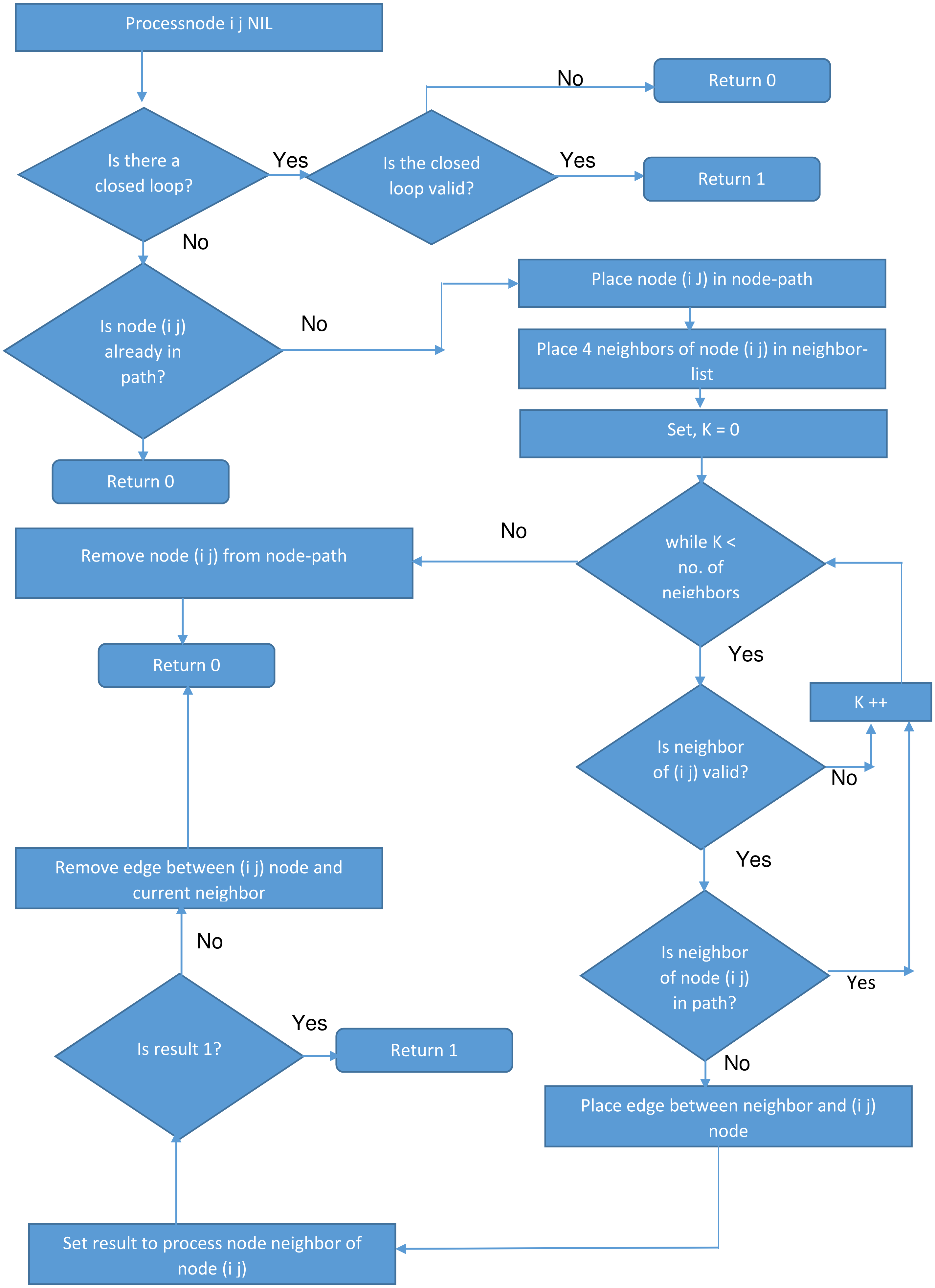
12. **Convertedgelist (edge-list):** It changes the representation of the edges in the board to desired format that is “2 2 t” form. Functions called for this are:

    I. **Edge-convert-list (list1):** Converts edges in form (0 3) to (1 2 T). Calls function :
    Checknumbereven (i j): Converts horizontal edges in desired format
    Checknumberodd (i j): Converts vertical edges in desired format

Topic 4: FLOWCHARTS

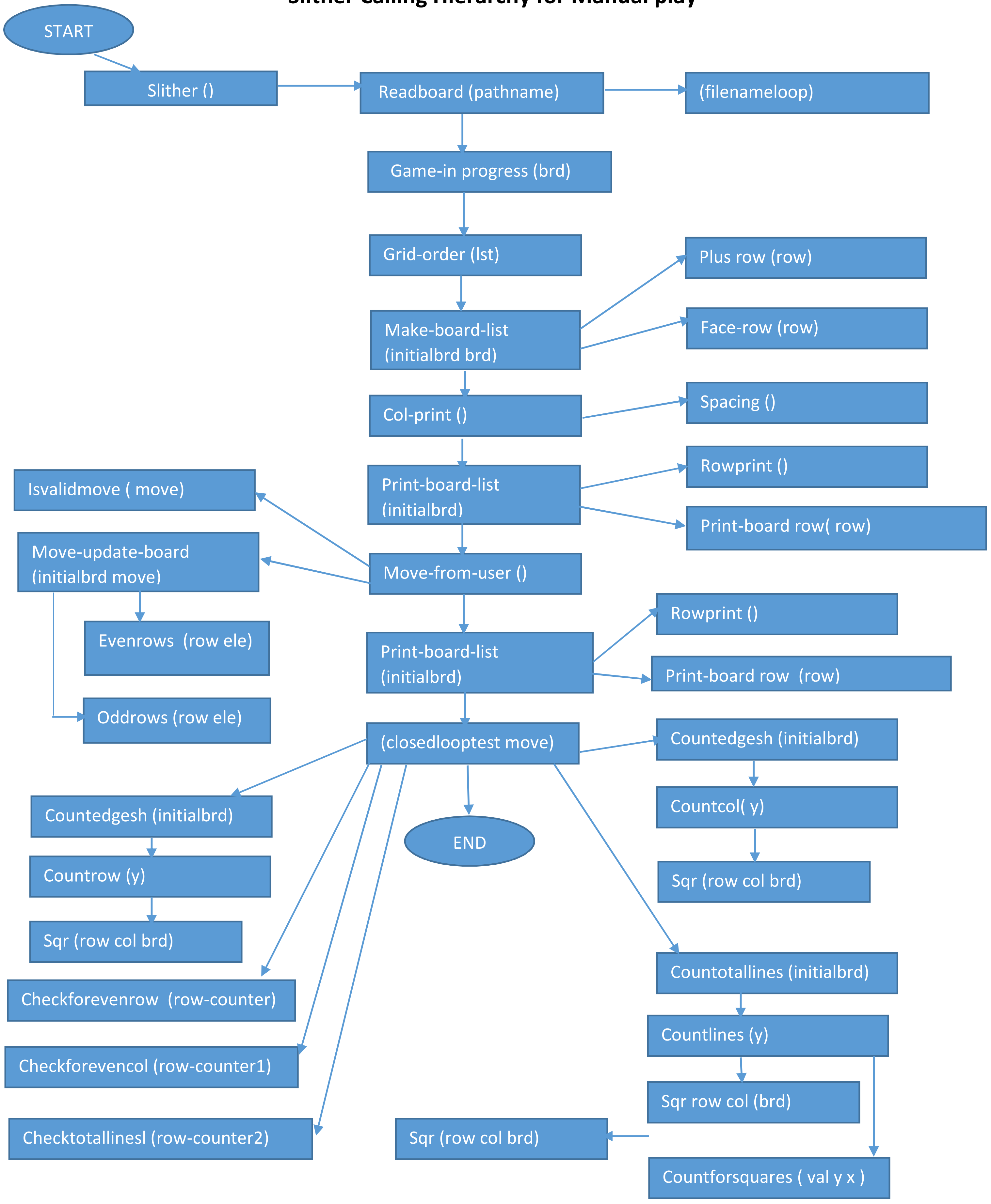

START
Is manual mode selected?
YES
NO
Enter move into the board
Is the closed loop condition satisfied?
No
Yes
Player wins, List of moves is displayed
STOP
Select starting point by scanning for 3 or 2 or 1 in the board
Set starting point
Is Processnode si sj forming a valid loop?
No
Yes
STOP

Flow-chart for process node function (depth first search) shown in next page.

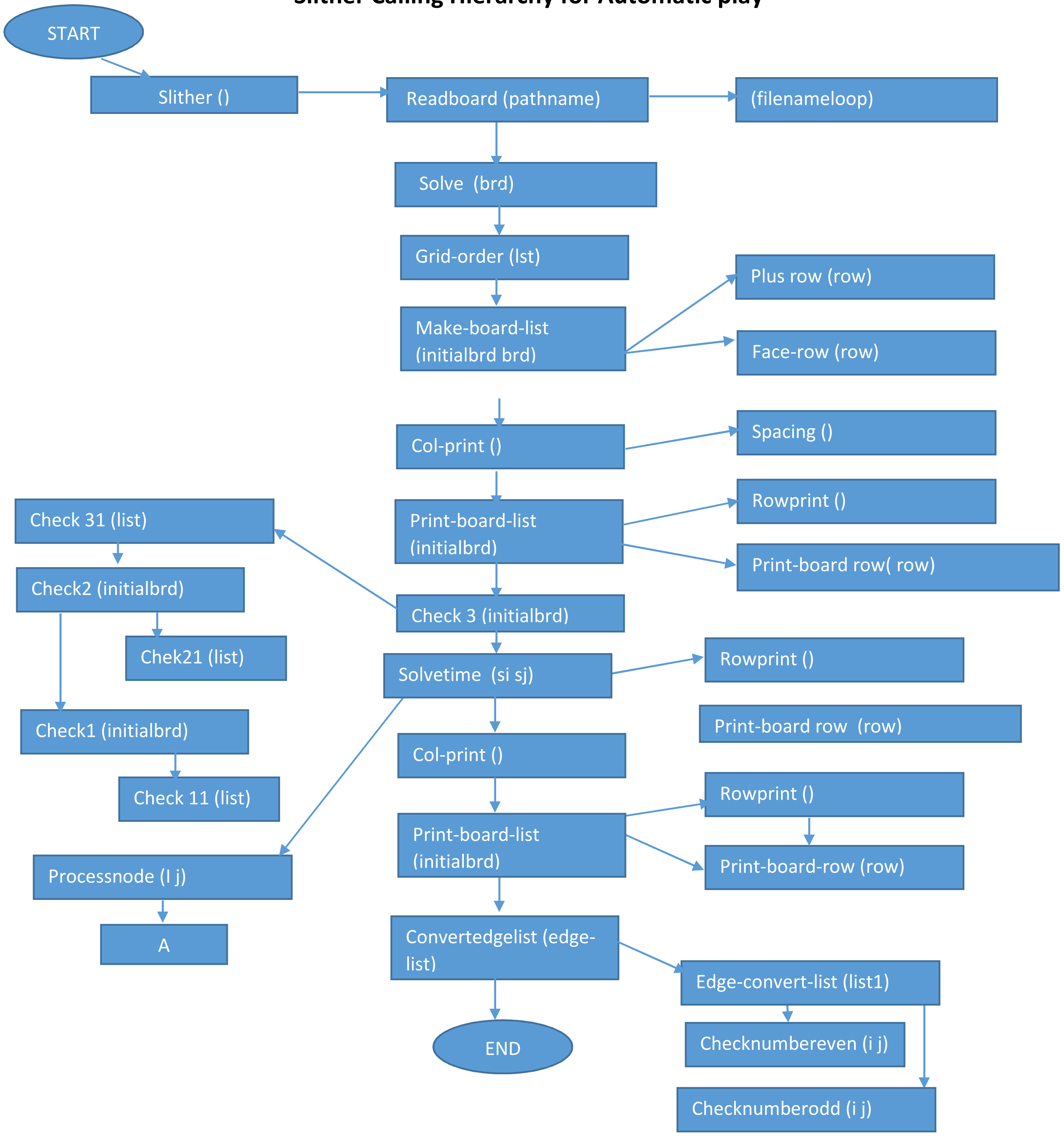

Processnode i j NIL
Is there a closed loop?
Yes
Is the closed loop valid?
No
Return 0
Yes
Return 1
No
Is node (i j) already in path?
No
Place node (i J) in node-path
Place 4 neighbors of node (i j) in neighbor-list
Set, K = 0
Return 0
No
Remove node (i j) from node-path
while K < no. of neighbors
Return 0
Yes
K ++
Is neighbor of (i j) valid?
No
Yes
Remove edge between (i j) node and current neighbor
No
Is neighbor of node (i j) in path?
Yes
Yes
Is result 1?
Return 1
No
Place edge between neighbor and (i j) node
Set result to process node neighbor of node (i j)

Topic 5 Calling hierarchy of functions

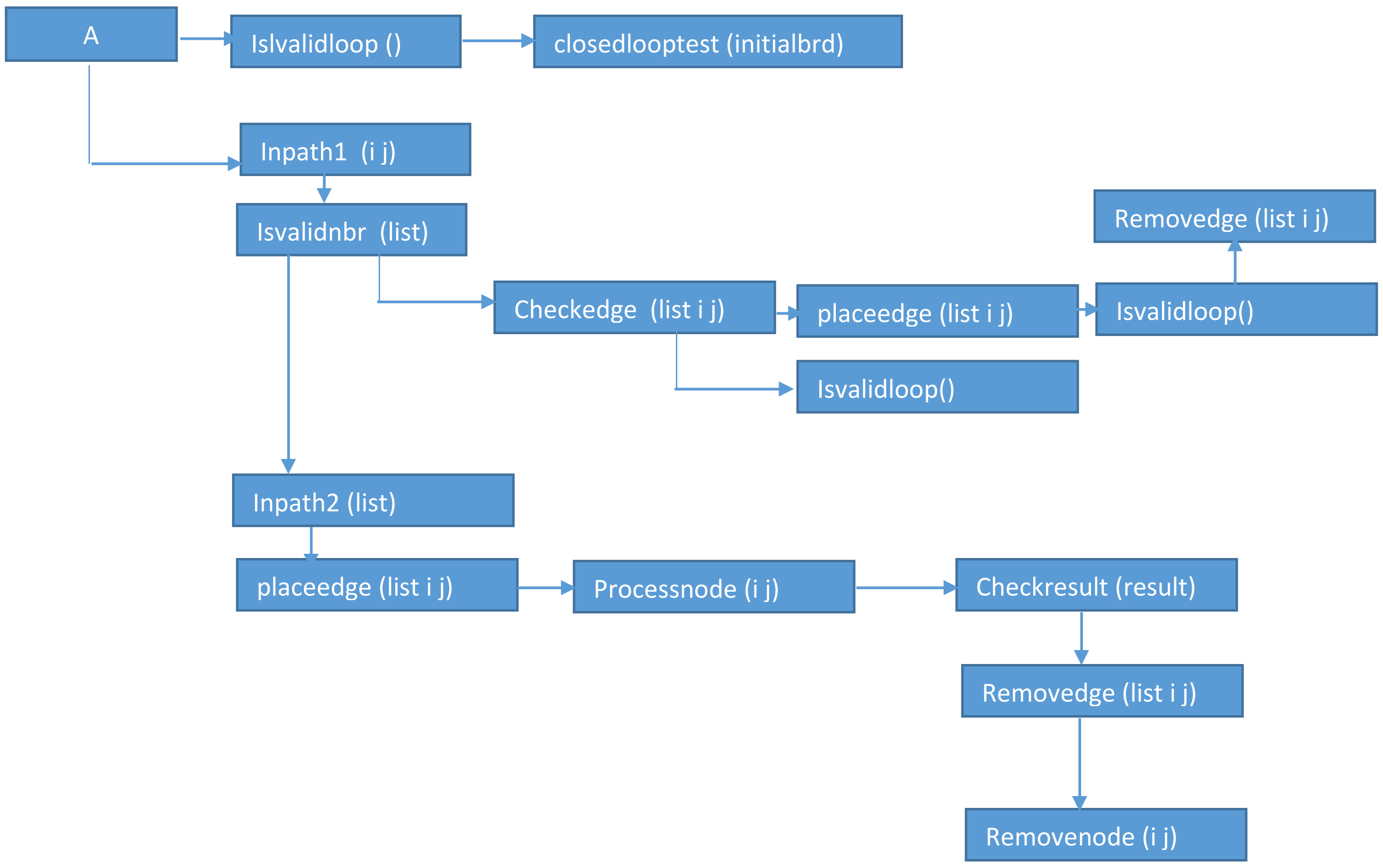
Slither Calling Hierarchy for Manual play



Slither Calling Hierarchy for Automatic play



Calling Hierarchy for Process node function



Topic 6 INTELLIGENCE IMPLEMENTED

- Search for 3 in the board, all the four corners of cell in which 3 is present will be in closed loop. So start DFS from any one of these corners.
- If 3, is not present in the board then the search for 2. For a 2, 3 corners of the cell are in closed loop, so if one of these doesn't work then we just have to try once more from any of the other ends.
- If 2 is not present in the board then the search for 1. For a 1, 2 corners of the cell are in closed loop, so if one of these corners doesn't work then we just have to try once and if that also doesn't work try once more from any of the other ends.
- Depth for search starts with a node which has been found from steps described above.
- Out of the four neighbors, they are explored in sequence of up, down, left and right.
- If a path with a particular node leads to a dead end or closed loop, we discard that neighbor and move on to the next one.
- If all the four neighbors are discarded, then we backtrack and explore other neighbors of previous node.
- In this way we proceed with depth for search and explore the potential solutions.
- Depth for search has been optimized in following ways:
  1. If the nodes been explored are already in path, they are discarded and they are not explored further. This is similar to the pruning process we learnt in the AI class.
  2. Instead of checking whether a valid closed loop is formed for each edge placed, I have placed a logic to test if there is a closed loop. If there is a closed loop formed, then only check for valid closed loop is carried out. This increases the efficiency.
  3. Node path list is used to keep track of the nodes being explored and being removed. A single list to keep track provides efficiency with any complexity of adding colors for nodes visited etc.

Topic 7 INTELLIGENCE NOT IMPLEMENTED

- **Places certain edges on board before DFS is called on it:** I wanted to implement intelligence wherein I place the moves on board which are known from board configuration for example 3 on corners or 2 on corners.
- **Set priority for neighbors being explored**: Before depth first search is carried out, the neighbors which are placed in final list of moves are explored first, so that we can further improve the efficiency of program by exploring those nodes first which we know are going to be for sure in final set of moves.

  Reason for not implementing these Intelligence mentioned above:
  Depth first search and the time taken to implement backtracking and improving its efficiency took a considerable amount of time and was of high difficulty level for me. So even though I wanted to implement further intelligence which I thought of, I could not find sufficient time for implementing it. I am certain that logic I that I thought for improving efficiency would have significantly improved efficiency of my code.

Topic 8 Retrospection of things to be done if given a chance to start over

- I would like to see how depth first search can be carried out provided I had chosen some other representations.
- Compare whether representations would affect the efficiency of depth first search and check with my existing solution
- Get the search implemented early enough to have sufficient time left for implementing search.