Jeffrey Wood
Final Project Report

The game that was implemented for this project is a puzzle game called Numbrix. At the start of a game, you are given a square grid with width *n* and some seemingly random numbers filled in already. The possible values that can be placed on the grid range from 1 - n^2 but they must be placed on the grid in such a way that a path forms on the grid from the minimum value to the maximum value by only moving vertically or horizontally. Some game boards may have multiple solutions but they all have at least one solution.

In order to solve this game as efficiently as possible, we need to establish a way of finding the easiest moves and performing them first. To do this, we need to keep track of which moves have been made using a Boolean array and from this array, we can build a list of endpoints to work with. These endpoints we are looking for can be any value played on the board that is missing at least one of the needed values around it. In the list of endpoints, we of course store the value of the endpoint as well as its location on the grid. From this list, we can check each endpoint to see if it is completely surrounded by values except for one space. If this condition is true, we can place a new value on the board and then update the list of endpoints. If we make it through the entire list of endpoints without making one of these trivial moves, then we need to move on to a more complex approach.

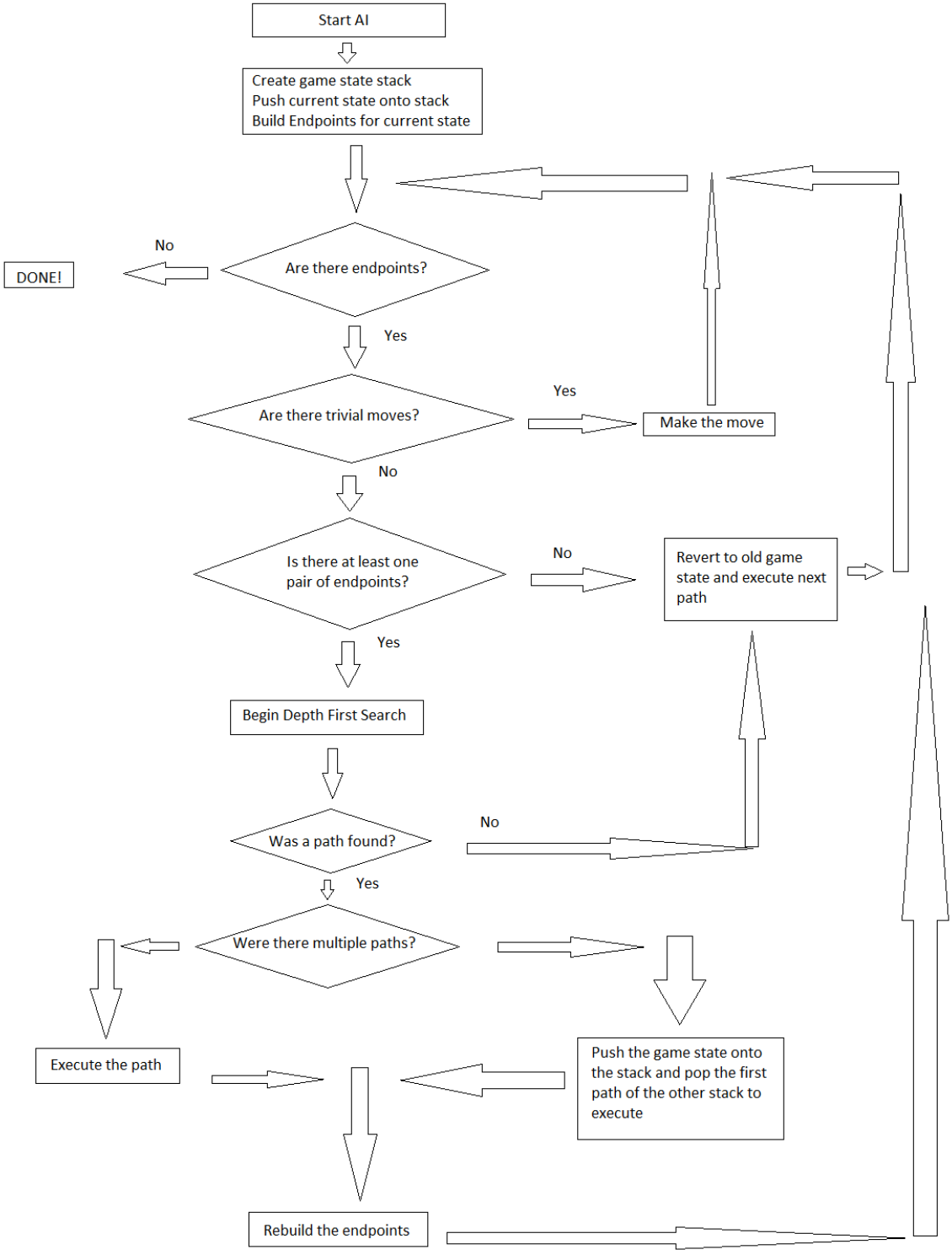
Since we cannot make any trivial moves, we need to make pairs out of all of the endpoints and calculate the distance between them. We then make a new list of these endpoint pairs and order them by increasing distance between them. From here, we can take the endpoint pair with the shortest distance between them and perform a depth first search to find all possible paths of the correct distance. If there are multiple paths for an endpoint pair, we will store all of the paths in a stack and save them with the current game state, which will also be stored in a stack. To reiterate, we will keep a stack of game states where a search was done and multiple paths were found. These paths will be saved with the game states in a stack so that after we execute the path, we do not try it again if we need to come back to this state. Once we execute the path that we found, we can update the endpoint list and begin checking for more trivial solutions. If after we execute a path, we get in a position in which we cannot make a move, we can pop the current game state off the stack and try working on a previous game state with a different path. This method has proven to be incredibly efficient for all of the tournament boards, with the longest time taken to solve a board being around fifteen seconds.

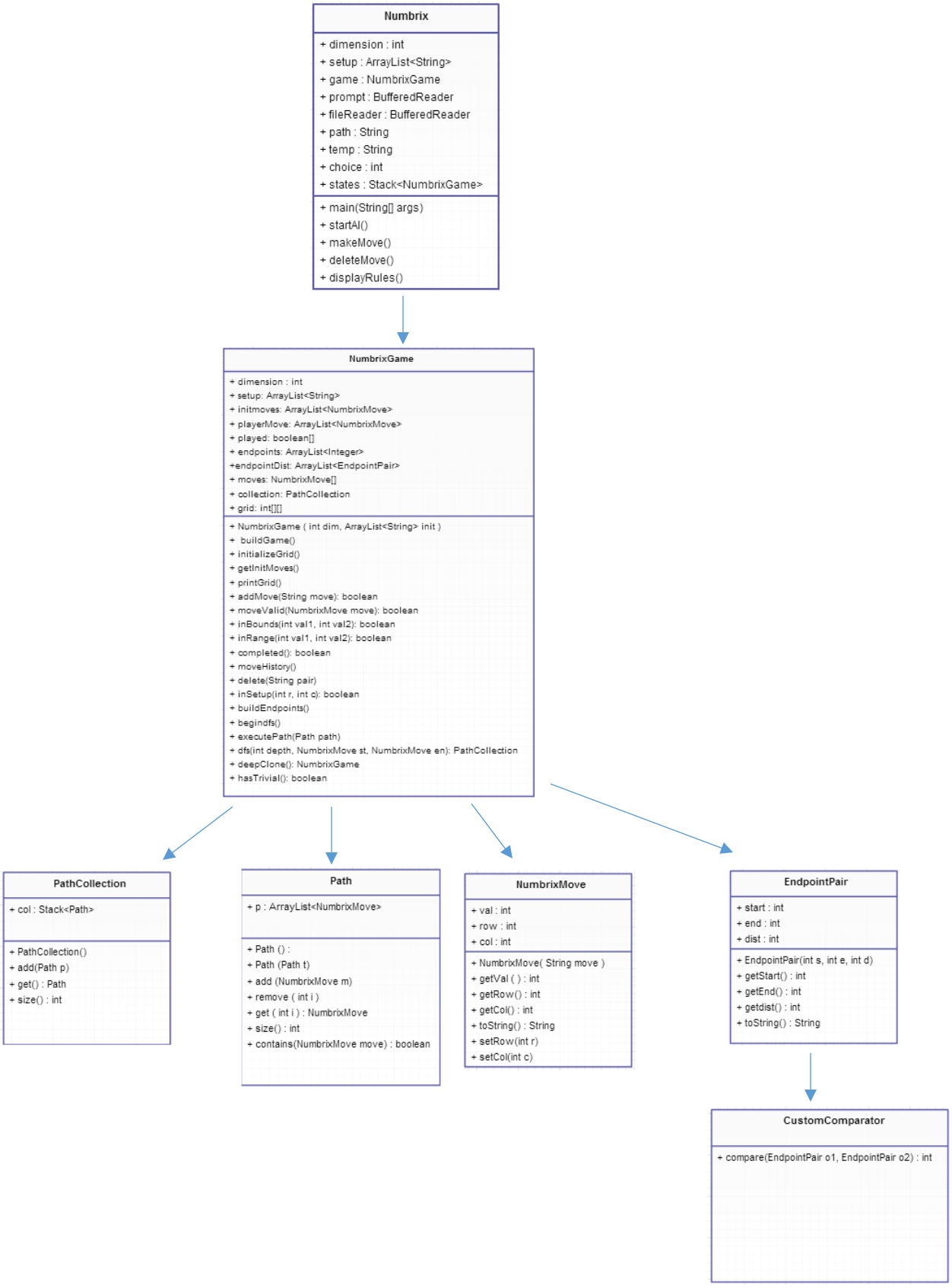
The intelligence used in this method would be the depth first search. This search would not search past the maximum depth, being the distance between the endpoint pair, and would find all possible paths between the pair. To do this, a list would be created with the first entry being the first endpoint and the start of our path. We then check the surrounding spaces of our current move to see if there are any possible places to move. If there are, we add that space to our path and then add that path to our list. Doing the search this way, we can add a maximum of four new paths to our list of paths through

each iteration and we will check every path to see if it has the correct depth and ends in the correct location. If we ever have a path that matches both of these conditions, we add it to a stack of paths to be saved to our game state. Performing the search in this manner, we obtain every possible path of the correct depth between the endpoints so that we may use the extra paths later if necessary.

The intelligence that was not implemented in this solution but I included in my initial report involved a way to detect “holes.” These holes would be any cluster of positions of the grid that were entirely surrounded by values. I originally thought that by being able to detect holes, I would save time by not executing certain paths that would create holes that could not be filled. While it may be slightly beneficial to do this, I decided that actually implementing this feature could potentially cost me more computation time as I would need to run it before every path execution. Perhaps if I were to begin designing this system again, I could integrate the “hole” detection into the search method but I feel it still may be a waste of time.

While on this subject, I would like to continue discussing what I might do differently if I were to start this project over entirely. My code could of course be cleaned up considerably as well as tweaked to be made more efficient. Now that I know exactly what I need to use to solve the problem, some of the data structures I had already committed to could be trashed for more time and space efficient structures or be thrown out entirely. All things considered, I thoroughly enjoyed working on this project and I would love to see how others tackled this problem.


Start AI
Create game state stack
Push current state onto stack
Build Endpoints for current state
Are there endpoints?
No
DONE!
Yes
Are there trivial moves?
Yes
Make the move
No
Is there at least one pair of endpoints?
No
Revert to old game state and execute next path
Yes
Begin Depth First Search
Was a path found?
No
Yes
Were there multiple paths?
Execute the path
Push the game state onto the stack and pop the first path of the other stack to execute
Rebuild the endpoints


Numbrix
+ dimension : int
+ setup : ArrayList<String>
+ game : NumbrixGame
+ prompt : BufferedReader
+ fileReader : BufferedReader
+ path : String
+ temp : String
+ choice : int
+ states : Stack<NumbrixGame>
+ main(String[] args)
+ startAI()
+ makeMove()
+ deleteMove()
+ displayRules()
NumbrixGame
+ dimension : int
+ setup: ArrayList<String>
+ initmoves: ArrayList<NumbrixMove>
+ playerMove: ArrayList<NumbrixMove>
+ played: boolean[]
+ endpoints: ArrayList<Integer>
+endpointDist: ArrayList<EndpointPair>
+ moves: NumbrixMove[]
+ collection: PathCollection
+ grid: int[][]
+ NumbrixGame (int dim, ArrayList<String> init)
+ buildGame()
+ initializeGrid()
+ getInitMoves()
+ printGrid()
+ addMove(String move): boolean
+ moveValid(NumbrixMove move): boolean
+ inBounds(int val1, int val2): boolean
+ inRange(int val1, int val2): boolean
+ completed(): boolean
+ moveHistory()
+ delete(String pair)
+ inSetup(int r, int c): boolean
+ buildEndpoints()
+ begindfs()
+ executePath(Path path)
+ dfs(int depth, NumbrixMove st, NumbrixMove en): PathCollection
+ deepClone(): NumbrixGame
+ hasTrivial(): boolean
PathCollection
+ col : Stack<Path>
+ PathCollection()
+ add(Path p)
+ get() : Path
+ size() : int
Path
+ p : ArrayList<NumbrixMove>
+ Path () :
+ Path (Path t)
+ add (NumbrixMove m)
+ remove (int i)
+ get (int i) : NumbrixMove
+ size() : int
+ contains(NumbrixMove move) : boolean
NumbrixMove
+ val : int
+ row : int
+ col : int
+ NumbrixMove(String move)
+ getVal ( ) : int
+ getRow() : int
+ getCol() : int
+ toString() : String
+ setRow(int r)
+ setCol(int c)
EndpointPair
+ start : int
+ end : int
+ dist : int
+ EndpointPair(int s, int e, int d)
+ getStart() : int
+ getEnd() : int
+ getdist() : int
+ toString() : String
CustomComparator
+ compare(EndpointPair o1, EndpointPair o2) : int