COEN352 G

Edge-matching Puzzle Solver:
Final Course Project

Submitted on: December 7, 2021

by Jamie Frendo-Cumbo (4XXXXXX1)

Prepared for Nawwaf Kharma, Concordia University, Fall 2021

Table of Contents

Part 1: Describe the Problem

Problem Statement 4
Some Observations 4
Strategy 5

Part 2: Model a Solver

Version 0 - Proof of Concept 6
Version 0.5 - Filling the First Row 8
Version 1.0: - Filling the Center Rows 10
Version 2.0/3.0 - The Final Solver 21
Version 3.xb - A Failed Experiment in Improved Backtracking . . . 26

Part 3: Present Solutions

The Code 27
Time Performance 27
Memory Performance 29

Part 4: Analyze the Results

Time Analysis 30
Space (Memory) Analysis 31
Possible Design Improvements 32

Appendix – Time Trial Data (Simple Backtracking) 33

Appendix – Breakdown of Solver Memory Requirements 37

Appendix – Abridged Source Code

Driver 41
Tile 42
GameBoard - Basic Gameplay Methods 43
GameBoard - solve() Method 45
GameBoard - assess and place Methods 49
GameBoard - Other Methods 53

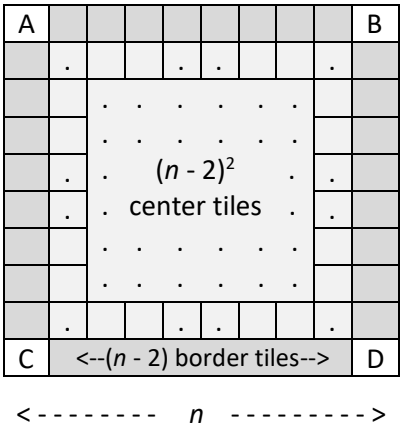
Part 1: Describe the Problem

Problem Statement

The task at hand is the creation of an *n* x *n* edge-matching puzzle solver using concepts learned in COEN 352. The domain of *n* is { $n \in [2, 16]$; $n \in \mathbb{Z}$ }, with *n* = 16 holding the additional constraint that the tileset must be that of the Eternity II puzzle.

Some Observations

1) For any *n*, one can define an *n* x *n* game board with parameters that directly depend on *n*:



An *n* x *n* board contains

- four corner tiles (tiles A, B, C, D)
- 4·(*n* - 2) border tiles (edge tiles that aren't corner tiles)
- $(n - 2)^2$ center tiles (all other tiles)

Summary

n	corners	borders	centers	sum (or *n* x *n)*
2	4	0	0	**4**
3	4	4	1	**9**
4	4	8	4	**16**
5	4	12	9	**25**
6	4	16	16	**36**
7	4	20	25	**49**
8	4	24	36	**64**
9	4	28	49	**81**
10	4	32	64	**100**
11	4	36	81	**121**
12	4	40	100	**144**
13	4	44	121	**169**
14	4	48	144	**196**
15	4	52	169	**225**
16	4	56	196	**256**

2) Any puzzle has only four corner pieces. The number of border pieces increases linearly with order n, and the number of center pieces increases with order n^2.

3) If $n = 2$, the puzzle consists only of corner pieces. Any puzzle with $n > 2$ contains at least one center piece. Any row of any puzzle with $n > 2$ contains at least one border piece.

4) Consider the tracing of a row from left to right. For any n, row 1 and row n consist of a *corner* piece, followed by (n-2) *border* pieces, followed by a second *corner* piece. For any row that is not row 1 or row n, the row consists of a *border* piece, followed by (n-2) *center* pieces, followed by a second *border* piece.

Strategy

A two-dimensional array will be implemented that satisfies the game board area. Each cell of the array will hold a single tile. Each tile will be an object with data to indicate orientation and color, and thus approximate placement on the board. A "solution", whether full or partial, will be represented by a board state with the highest number of tiles placed using valid moves.

The board will be filled row-by-row. As observed above, the top and bottom rows should be handled somewhat differently than every other (center) row due to the respective nature of their constituent tiles. This "forward" solve will be greedy (the first available tile that satisfies a placement attempt will be placed) and exhaustive (an attempt at placing a given tile will only be abandoned if there are no available tiles to place).

When the solver runs out of tiles to place it will "backtrack", removing the last k rows that it placed (including the current incomplete row). Single-row backtracking will attempt to mitigate the greedy selection of the current row's border piece: if there is another valid border piece available, it can be tried instead. Multiple-row backtracking will attempt to address the same issue with the center pieces, especially as n (and therefore $(n - 2)^2$) gets larger.

Given the approach of the solver, it will be convenient to design it in such a way that expects to produce infinite loops, while limiting their duration according to user preferences. When executing an entire solve routine, two parameters that are strong candidates for providing useful data are *time elapsed per solve* and *number of iterations per solve*, and indeed these will be the limiting factors imposed on a given routine.

The solver will be designed to be close to memory-neutral during execution. Required variables, data structures, and object instances will be defined up front and methods will use as few temporary locations or local variables as possible. This decision is intended to simplify solver analysis: implementing a "standardized" memory state across individual solves should afford more consistent batches of solves.

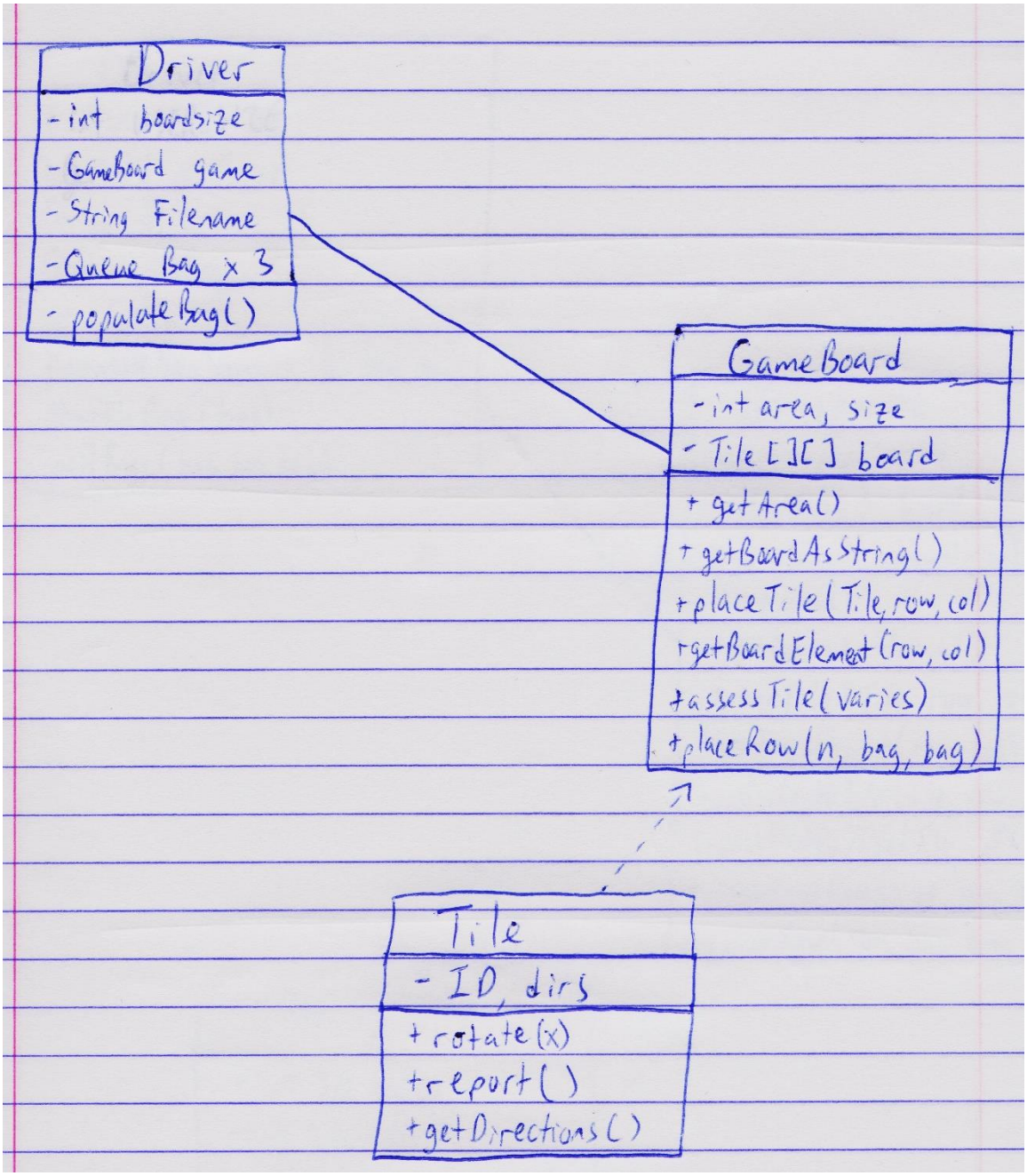
Part 2: Model a Solver

Version 0 - Proof of Concept: Tile objects will be created with an ID and four "colors": integer representations of color held in an array. Since the state of the colors represent an orientation of the Tile, a rotate method will be used to manipulate the color data and control the Tile's orientation. A report method will display essential information about a given Tile object instance, and will be useful for debugging. Finally, the method getDirections (in the code, a Tile's colors are also called its Directions) will return the Tile's color information in a useful way for any clients. Tiles will be "held" in three "bags": queues implemented with Linked Lists and treated as circular. One bag will hold the corner pieces, one the border pieces, and one the center pieces. Separating the tiles like this is analogous to what one might do when solving a jigsaw puzzle, and traversal through the lists will represent sifting through the pieces.

A GameBoard object is defined by the length of a row. It will be hardcoded during testing, with functionality for user-defined parameters planned for later versions. The row length is the game board's size, with area = $size^2$. A two-dimensional Tile object array is used to store game pieces while solving the puzzle. Other attributes may be added later; methods that will certainly be required are assessTile and placeTile, which will be used for moving any suitable candidate tiles into the next empty board cell. Derivative methods (like placeRow) are expected. Other important methods include getters for the full board state, the size of the board, and Tile objects located in specific cells.

The driver class currently contains or generates the source data for the size of the gameboard, and therefore the number of tiles used in a given solve. It handles the filename construction and queue population associated with loading the various tilesets. It also maintains the main GameBoard object, and any calls to GameBoard class methods will be made here. Helper methods (like populateBag) are expected.

Version 0 - Proof of Concept

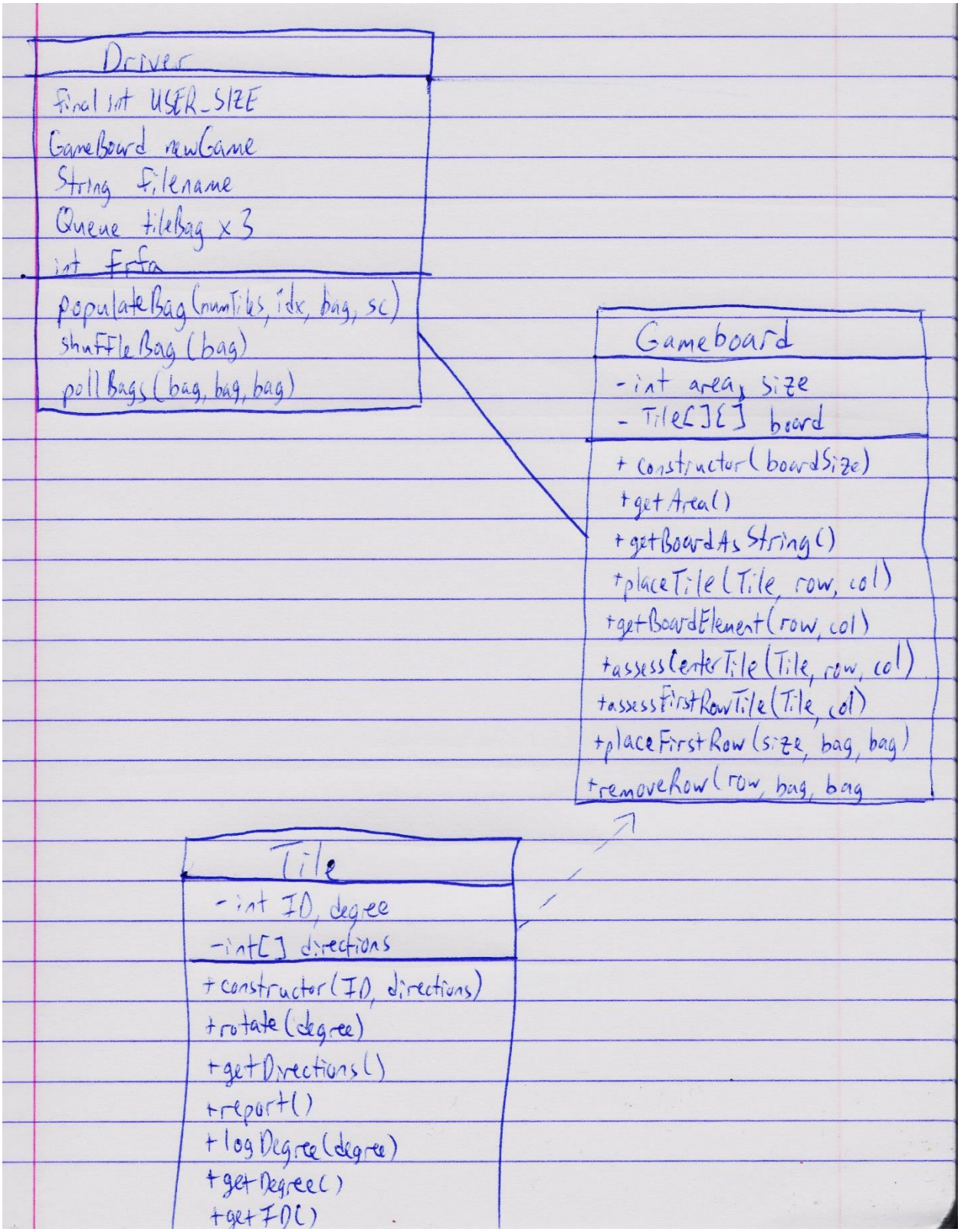


Version 0.5 - Filling the First Row: In the driver class, the board size has been changed to a final int called USER_SIZE that will most certainly be user-defined in future versions. A new integer flag FRFA ("first row flag assert") has been added, which is responsible for marking the state of the first row fill. Some methods have also been added to handle the tile bags: shuffleBag is a Knuth shuffle modified to randomize Linked Lists, and it is implemented to improve the uniformity of solve iterations; pollBags is used to check the state of all of the bags, and has proved useful for debugging.

The assessTile method in the GameBoard class has been split into assessCenterTile and assessFirstRowTile to avoid overly parameterized helper functions. This assess methodology will likely be split further in the future (to handle other board positions) since each type of position on the board examines the surrounding board cells somewhat differently; a single assess method would quickly become unwieldy. To this end, placeFirstRow has also become a distinct method, since the first row is placed somewhat by brute force and such an approach will not work for center rows (especially for large *n*). Finally, the removeRow method returns tiles on the board in a given row to their proper bags, an extremely useful method for freeing up a seized board. It will certainly be helpful when implementing any sort of iterating or backtracking.

In the Tile class, a new integer attribute called degree tracks the degree of the Tile's rotation. This will make reconstructing solutions easier when the solver is complete. The report method now returns orientation info, ID, and degree in one line, and can display on screen or return a string depending on the circumstances. Attribute getters were also added.

Version 0.5 - Filling the First Row



Version 1.0 - Filling the Center Rows: This version is very close to a complete solver. Repeated invocation of the fillCenterRow method is proving efficacious, and a planned fillLastRow method is the only component yet to be implemented. As it stands, the routine is capable of filling (*n* - 1) rows assuming low *n* and/or ideal tile placement. At this point, the main solving routine will clearly need to be encapsulated and looped: this will allow iterative solves and facilitate time analysis of the solver. This will also necessitate a clearBoard method that wipes the board state and returns all tiles to their bags.

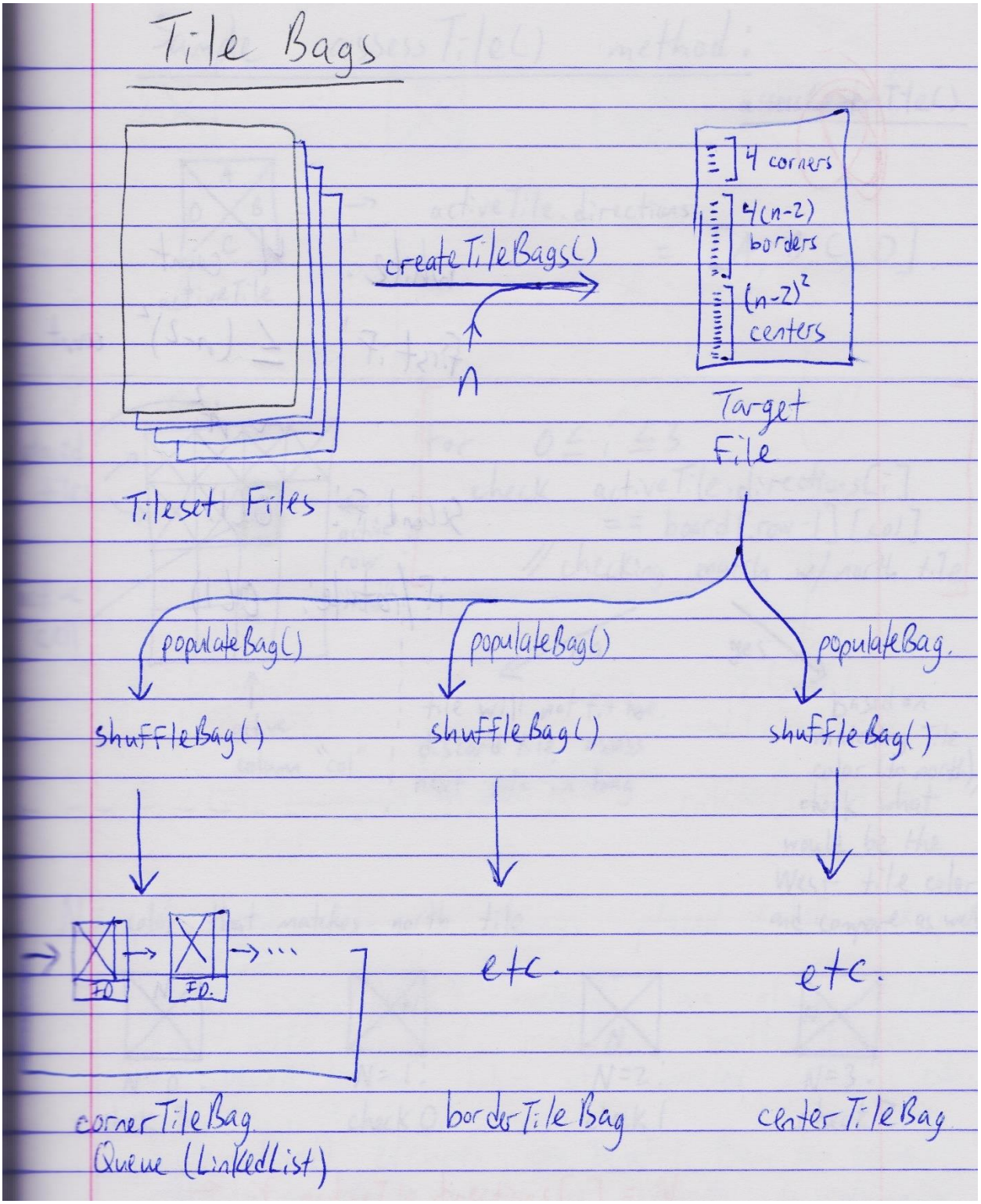
This naïve solver is surprisingly efficient: for *n* = 16, the first 7 rows can be placed comfortably, and the 8^{th} or 9^{th} row placed reliably. The solver currently employs a very basic sort of backtracking that uses the removeRow method on failed row placement attempts to test all available border pieces that could start the row. This mitigates the greedy Tile selection employed by the solver, explained in Part 1.

The first row is now placed as a "seed". That is, after the first corner piece is placed, the solver assumes that the rest of the row will be placed properly, and does not check the suitability of any of the three remaining corner pieces until one needs to be placed. If the placement fails, the row is cleared, and the solver does not stop trying to replace the row until it is placed successfully. This is a decision designed to further modularize first row placement, and to simplify it as much as possible. The center rows hold more importance when considering backtracking, especially for increasing *n*. In addition, when future improvements to backtracking are made, they will not be applied to the first row.

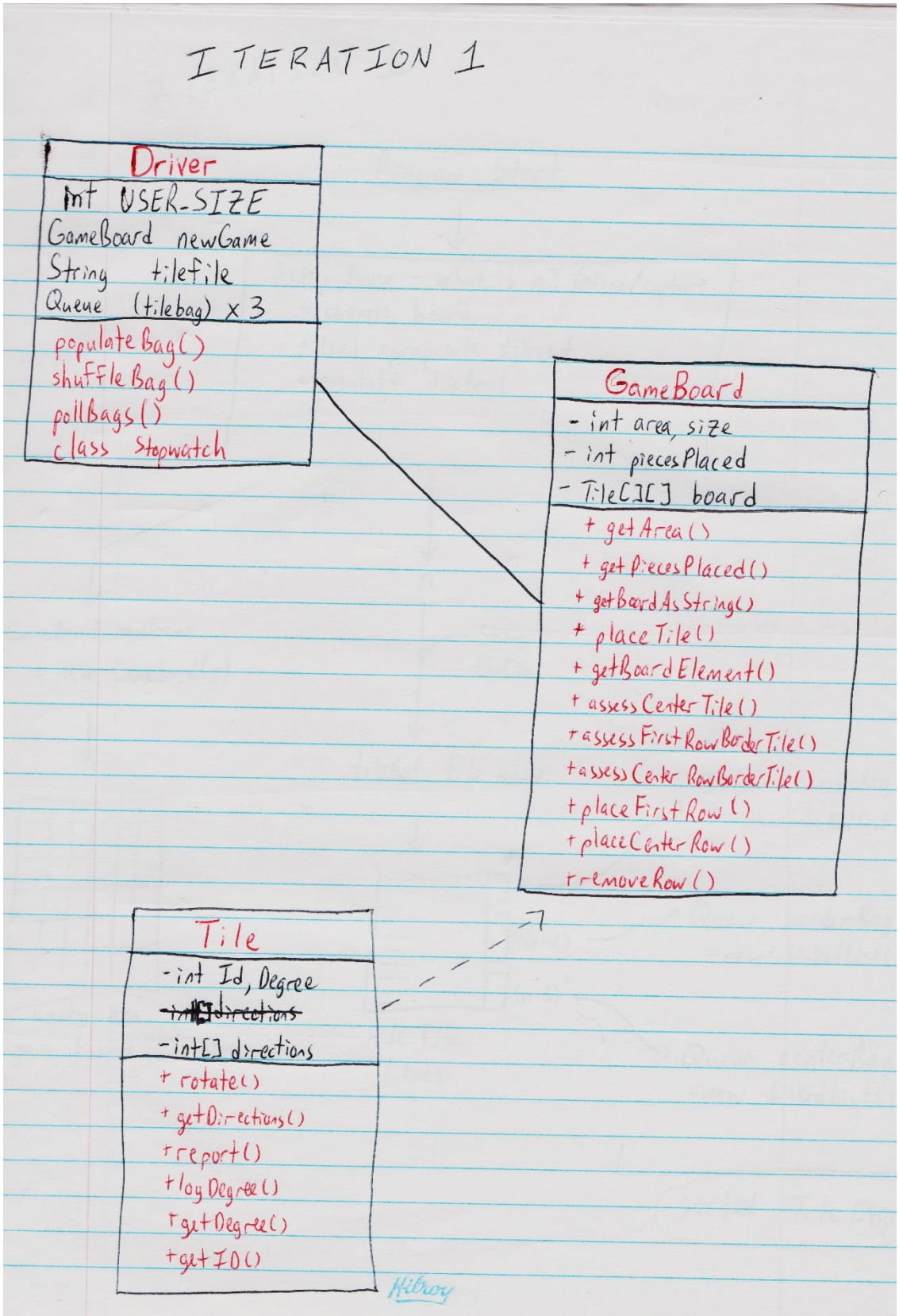
There are now three assessTile methods in the GameBoard class that assist in placing tiles in the first (*n* - 1) rows, depending on their destination. Other changes include cleaner code, more getter methods, and the addition of Sedgewick's Stopwatch class for testing solve times.

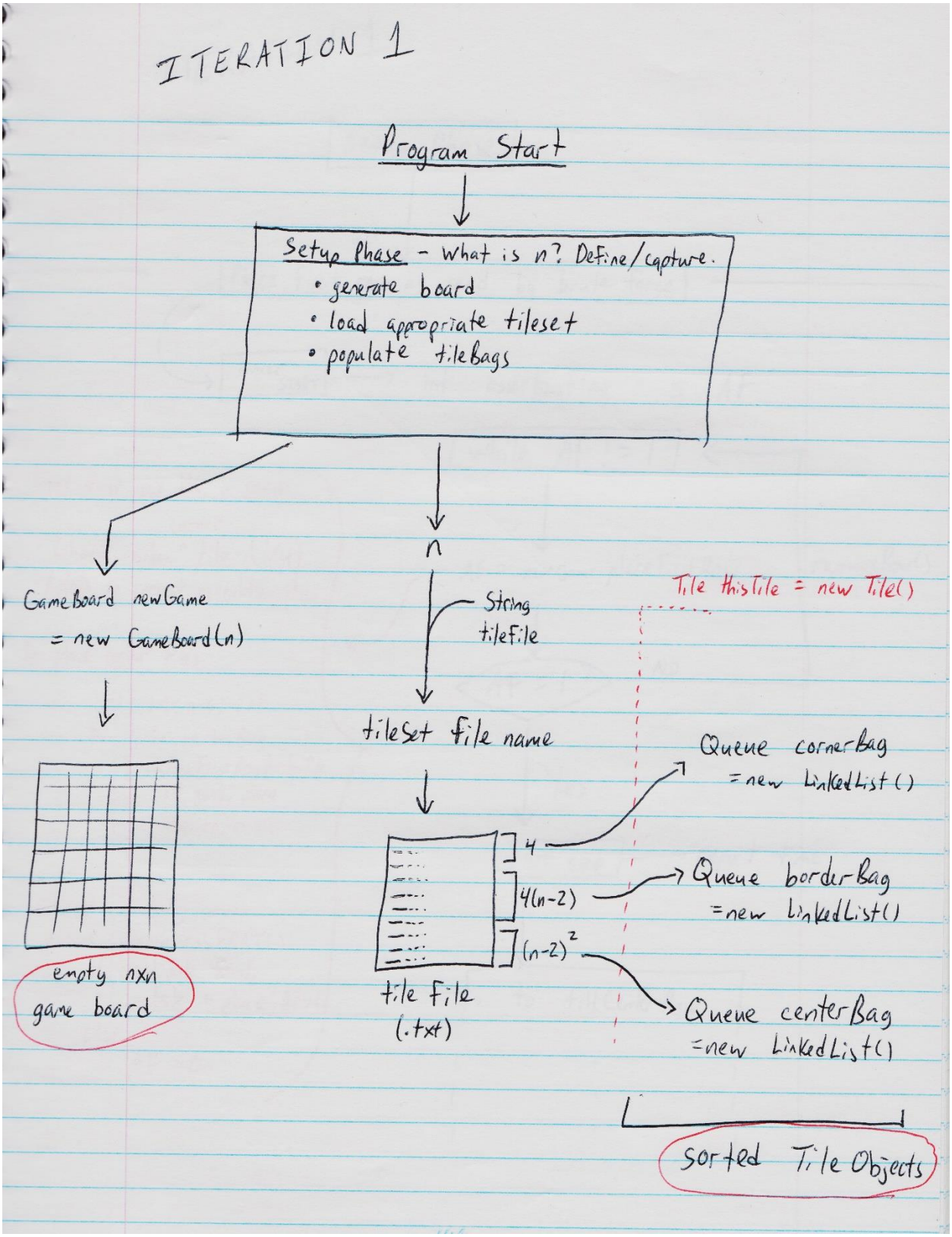
Version ("Iteration") 1.0 - Filling *k* rows ($1 \le k \le n - 1$)

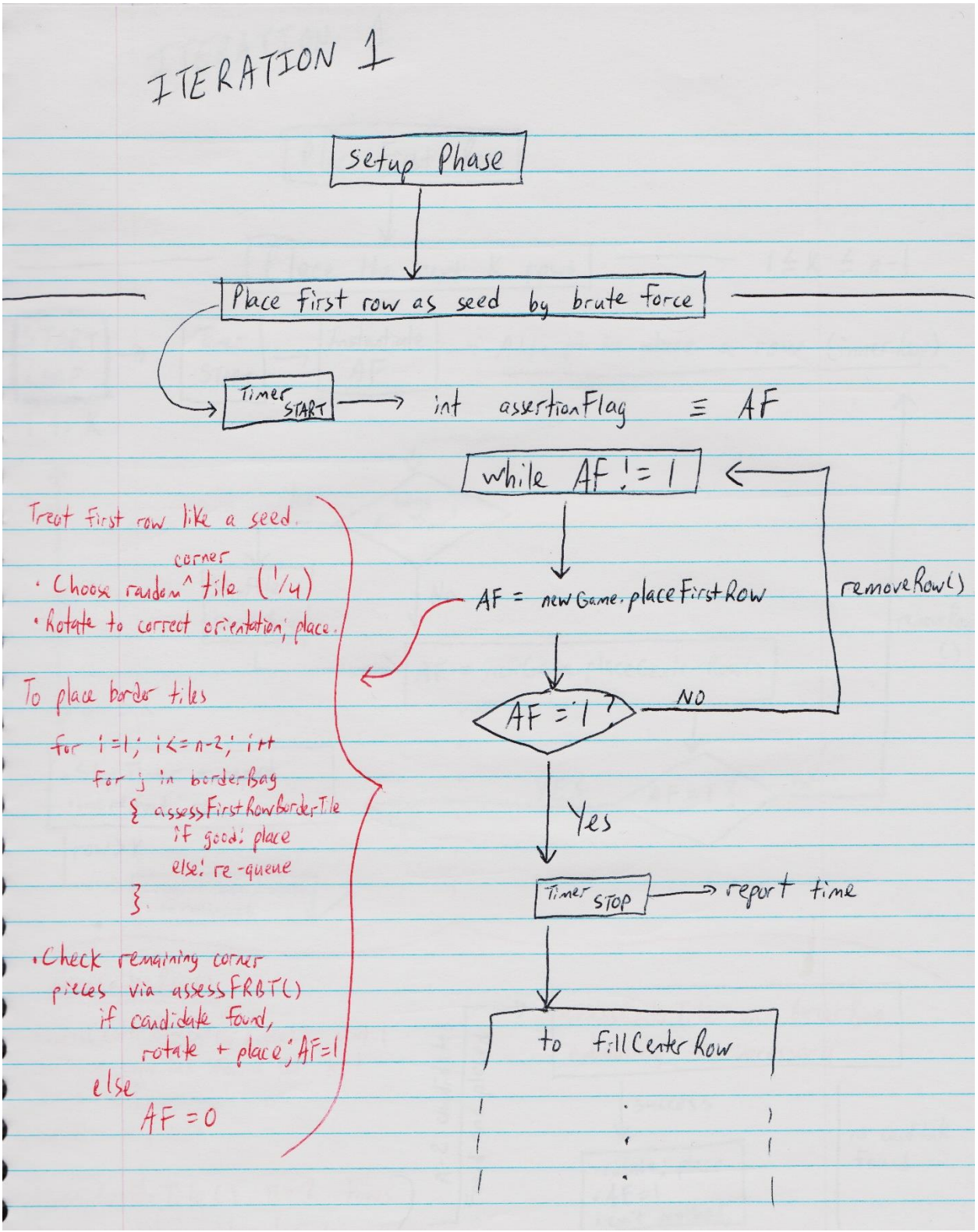
Implementing tileBags

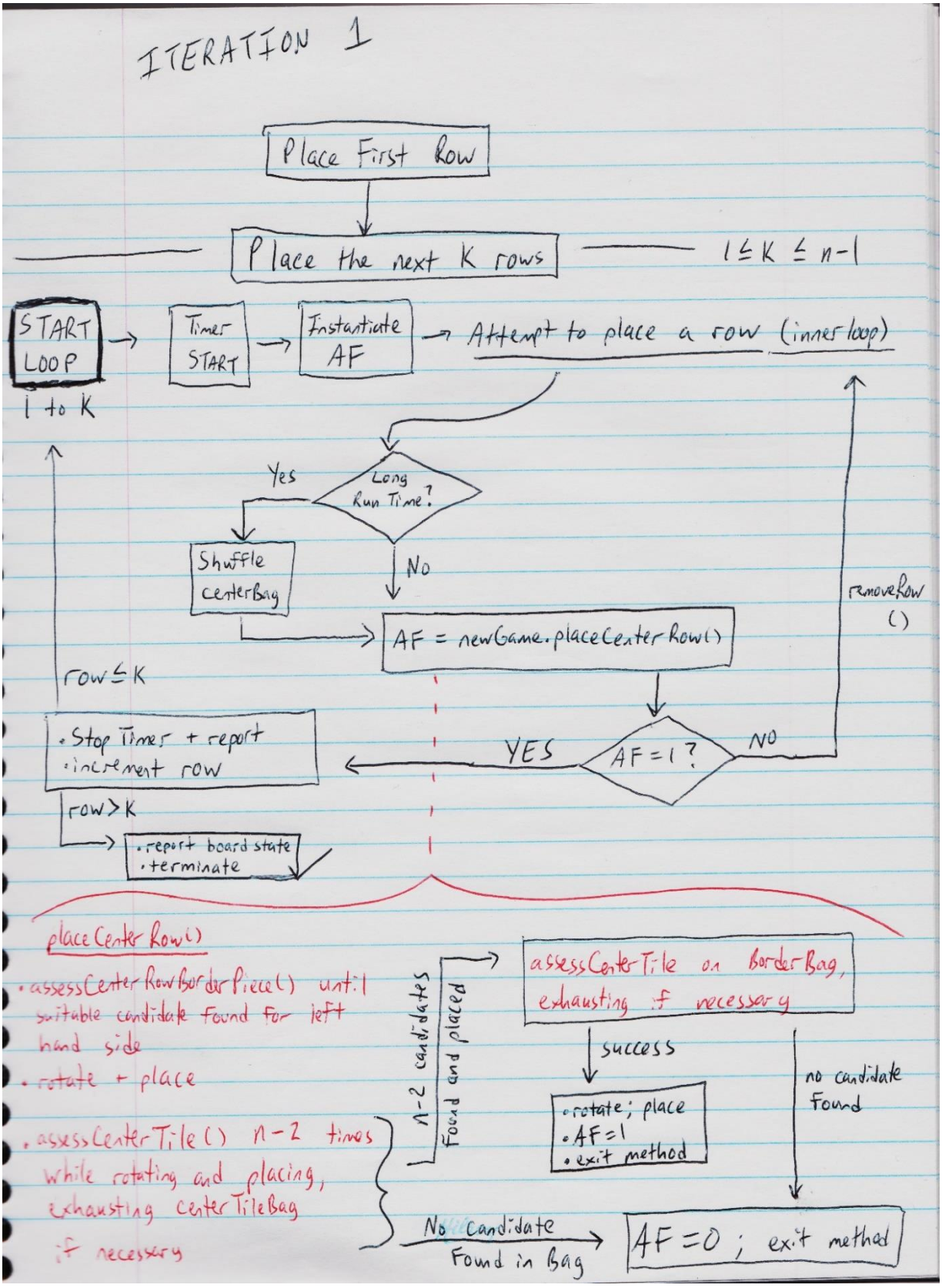


Class Dependency

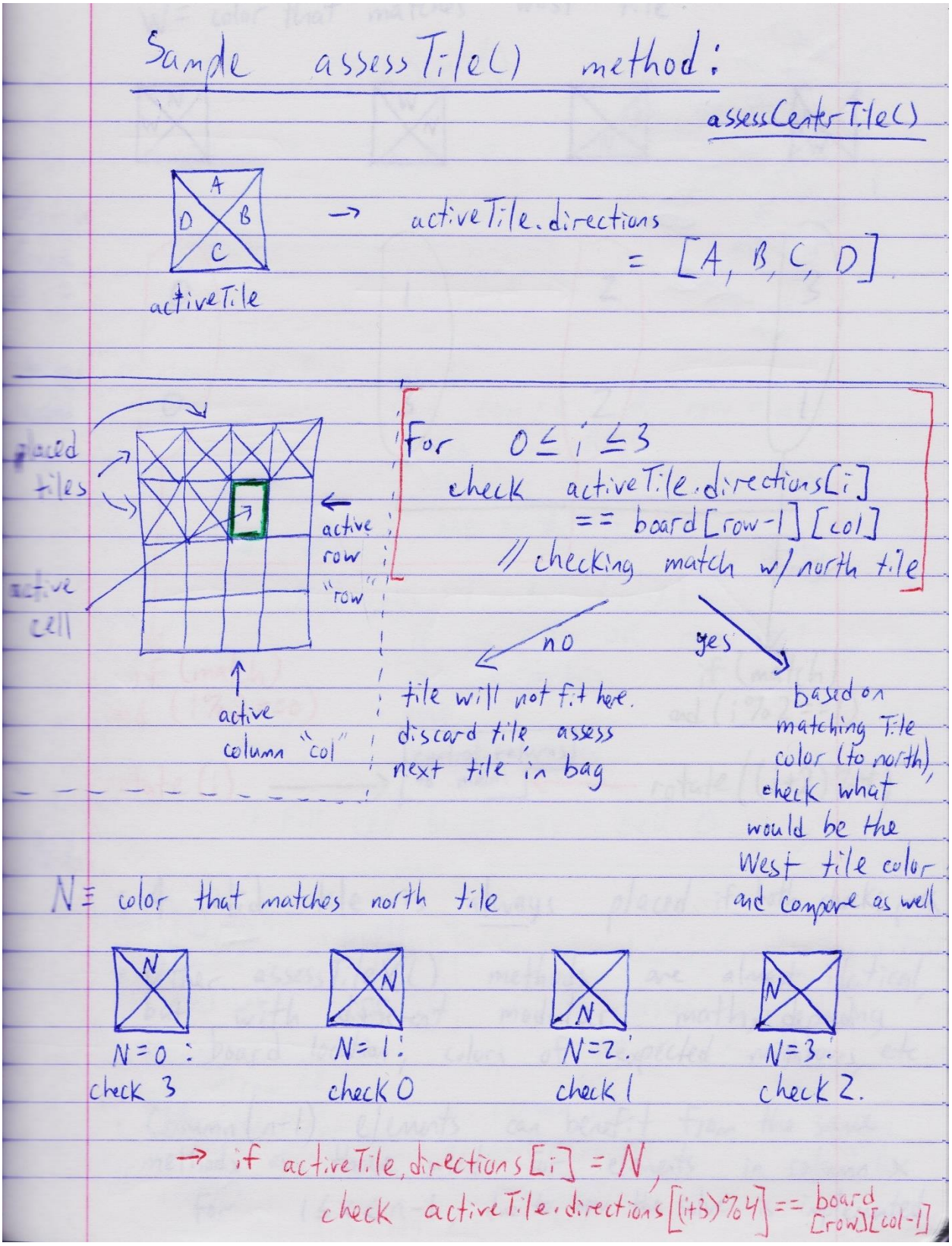



ITERATION 1
Program Start
Setup Phase - What is n? Define/capture.
• generate board
• load appropriate tileset
• populate tileBags
n
String tileFile
GameBoard newGame = new GameBoard(n)
Tile thisTile = new Tile()
tileSet file name
Queue cornerBag =new LinkedList()
4
Queue borderBag =new LinkedList()
4(n-2)
(n-2)²
Queue centerBag =new LinkedList()
empty nxn game board
tile File (.txt)
sorted Tile Objects

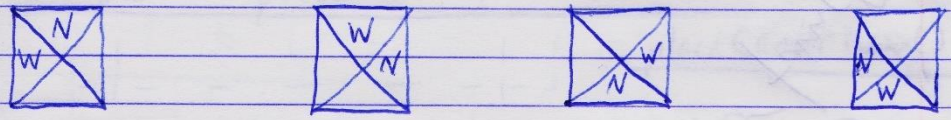

ITERATION 1
Setup Phase
Place First row as seed by brute force
Timer START
int assertionFlag ≡ AF
while AF != 1
AF = newGame.placeFirstRow
removeRow()
AF = 1 ?
NO
Yes
Timer STOP
report time
to fillCenterRow
Treat First row like a seed.
• Choose random corner tile (1/4)
• Rotate to correct orientation; place.
To place border tiles
for i=1; i<=n-2; i++
For j in borderBag
{ assessFirstRowBorderTile
if good: place
else: re-queue
}
•Check remaining corner pieces via assessFRBT()
if candidate found,
rotate + place; AF=1
else
AF=0


ITERATION 1
Place First Row
Place the next K rows
1 ≤ K ≤ n-1
START LOOP
1 to K
Timer START
Instantiate AF
Attempt to place a row (inner loop)
Long Run Time?
Yes
No
Shuffle centerBag
AF = newGame.placeCenterRow()
removeRow()
row ≤ K
• Stop Timer + report
• increment row
YES
AF = 1 ?
NO
row > K
• report board state
• terminate
placeCenterRow()
• assessCenterRowBorderPiece() until suitable candidate found for left hand side
• rotate + place
• assessCenterTile() n-2 times while rotating and placing, exhausting centerTileBag if necessary
n-2 candidates found and placed
assessCenterTile on borderBag, exhausting if necessary
success
no candidate Found
• rotate; place
• AF = 1
• exit method
No candidate Found in Bag
AF = 0 ; exit method

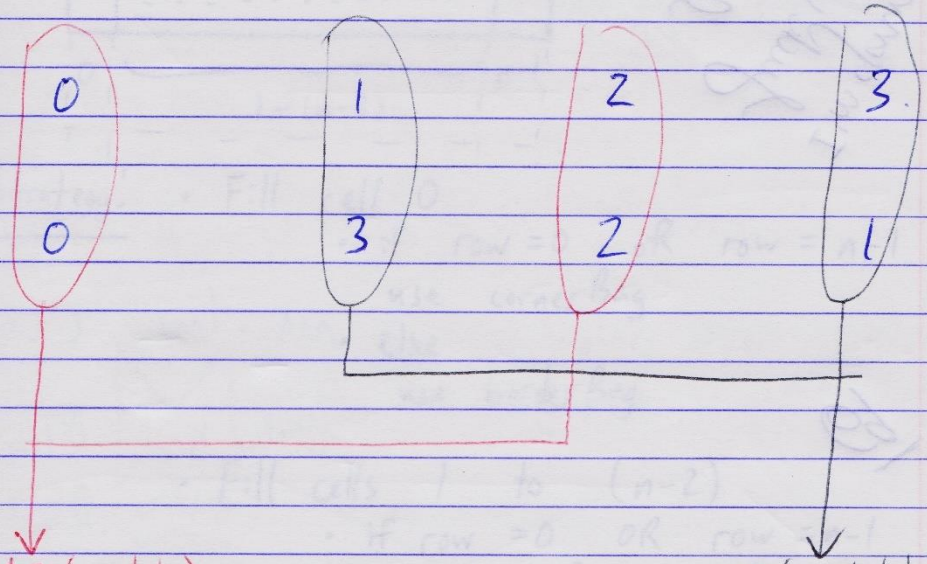
Sample assessTile() Method



W = color that matches west tile



Match found @ i =



if (match) and (i%2 == 0)

rotate(i)

control returns to main.

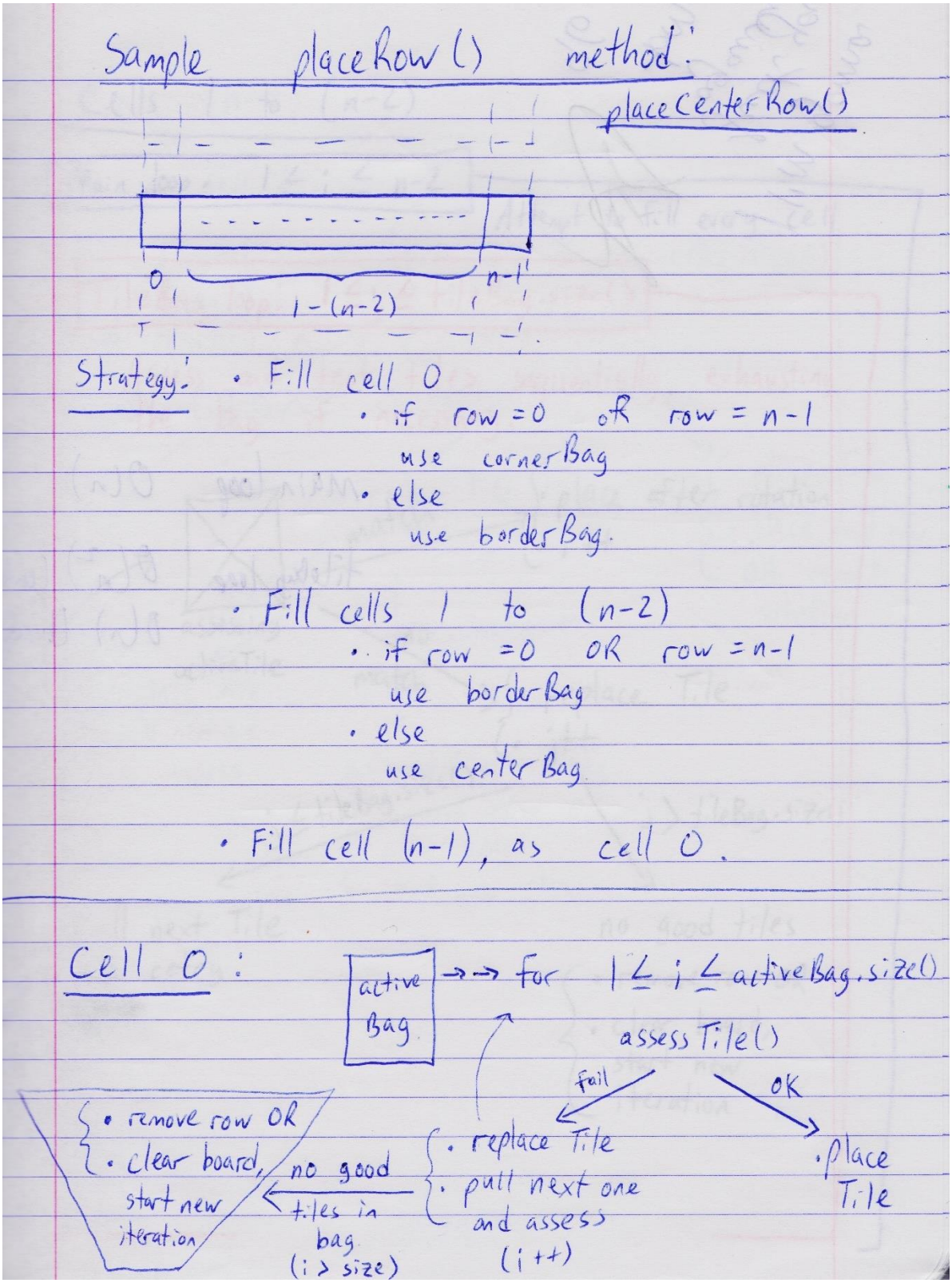
if (match) and (i%2 == 1)

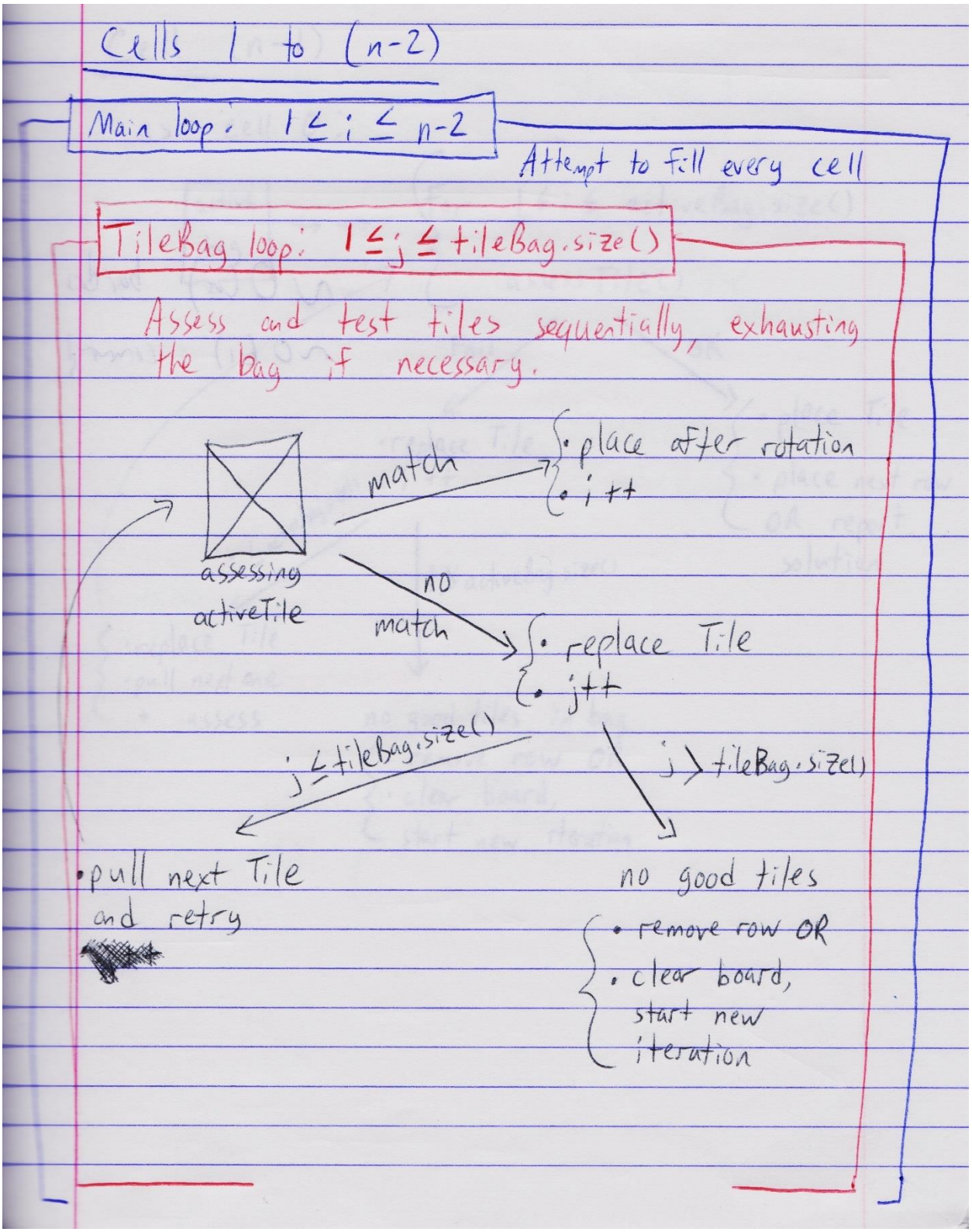
rotate((i+2)%4)

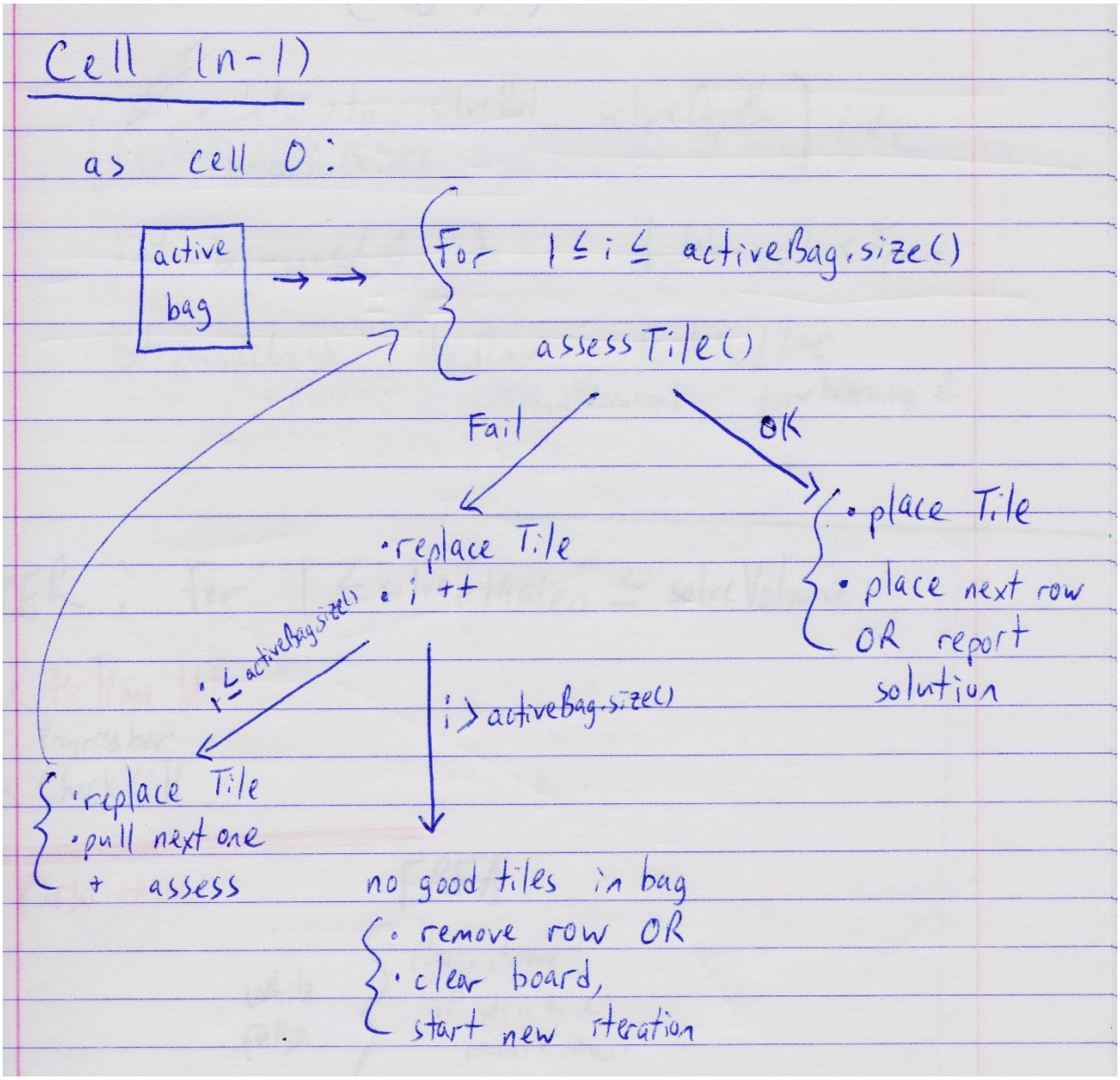
N.B.

- A valid tile is always placed if both checks pass.
- Other assessTileX() methods are almost identical, but with different modular math, depending on board location, colors of expected neighbours, etc.
- Column (n-1) elements can benefit from the same methods as those used for elements in column x for $1 \leq x < n-1$, due to how the solve is implemented.

Sample placeRow() Method




Cells 1 to (n-2)
Main loop: 1 ≤ i ≤ n-2
Attempt to fill every cell
TileBag loop: 1 ≤ j ≤ tileBag.size()
Assess and test tiles sequentially, exhausting the bag if necessary.
match
• place after rotation
• i ++
assessing activeTile
no match
• replace Tile
• j++
j ≤ tileBag.size()
j > tileBag.size()
• pull next Tile and retry
no good tiles
• remove row OR
• clear board, start new iteration


Cell (n-1)
as cell 0:
active bag
For 1 ≤ i ≤ activeBag.size()
assessTile()
Fail
OK
• replace Tile
• i ++
• place Tile
• place next row OR report solution
i ≤ activeBag.size()
i > activeBag.size()
• replace Tile
• pull next one + assess
no good tiles in bag
• remove row OR
• clear board, start new iteration

Versions 2.0/3.0 - Notes Taken in a 72hr Period While Building the Final Solver:

- After adding the fillLastRow method, puzzles can be solved for $n = 2$ to $n = 4$ fairly easily by just starting over if the last row fails at any point ($n > 5$ is infeasible without encapsulating the solver). At this point, better backtracking implementation is becoming a high priority.

- The solver has been encapsulated (Version 1.5) and can be run thousands of times. There are problems with the timers but once they are fixed, each solve iteration will be timed, and the time of the first (or best) solve will also be recorded. Timers will also be useful in terminating a solver that is looping excessively.

- [After extensive code cleanup:] The solver (Version 2.0) can now solve up to $n = 6$ by brute force (high iterations). Iterations are now monitored based on the number of attempts to fill a row, and how long the current iteration has been running; if either are too high, the board is cleared and the solve begins again with a new iteration. Backtracking is still minimal but the problems with the timers persist, and these should be addressed before attempting to push the solving capability of the routine any further.

- The solver is working well but I'm no programmer, and it's only once I've stepped back that I've realized the obvious: there is way too much data being passed back and forth between the driver and GameBoard classes. To remedy the inefficiencies, I've moved large chunks of code over to GameBoard since that's where all the action is anyhow. The driver class was meant to implement user interaction anyways, so it's definitely the smart move in terms of maintaining well-defined class functionality.

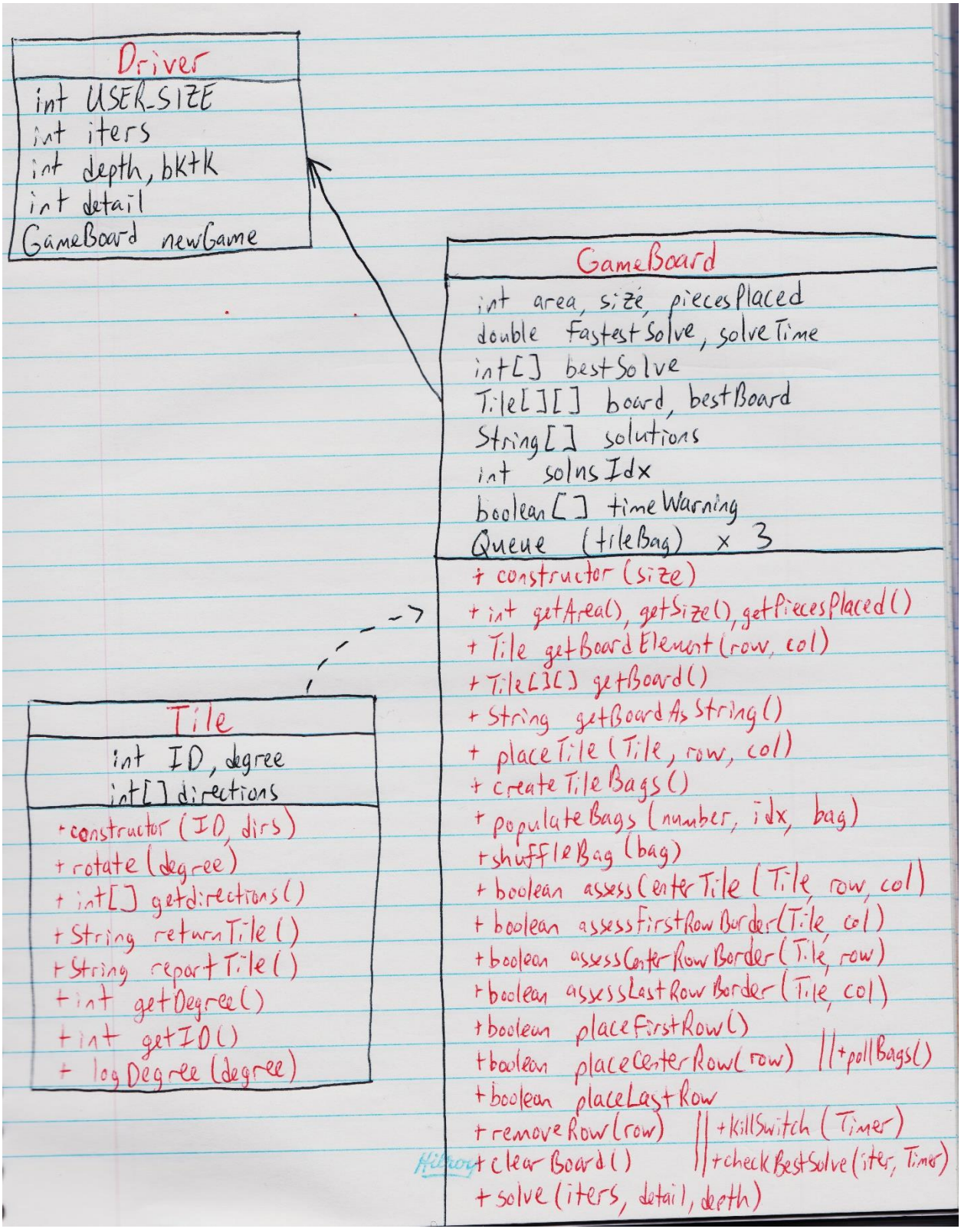
- The solver is now functionally complete (Version 3.0). After extensive code reworking, the solver is relatively clean and efficient. The driver class now functions primarily as a user menu: all solver activity is now contained in the GameBoard class, and a solve is performed with a single method call. Solve quality is good for $n = 7$ to about $n = 10$, and acceptable for higher n. Solve time is on the order of milliseconds or seconds.

- The code housekeeping undertaken to produce Version 3.0 has revealed why the timers were not working, and the implementation bugs are numerous: Stopwatch instances implemented incorrectly within (or outside of) loops, excessive Stopwatch instances (remnants of the patchwork of snippets that became the solve() method) collecting redundant data or sometimes not even used, incorrect execution timeframes, and wrong output variables producing erroneous output in the display. The timer system has been rebuilt from the ground up; it has been reduced to a single solver routine timer (killSolve, used with the killSwitch method), and a second timer that now properly tracks the execution time of a single solve iteration (iterTime).

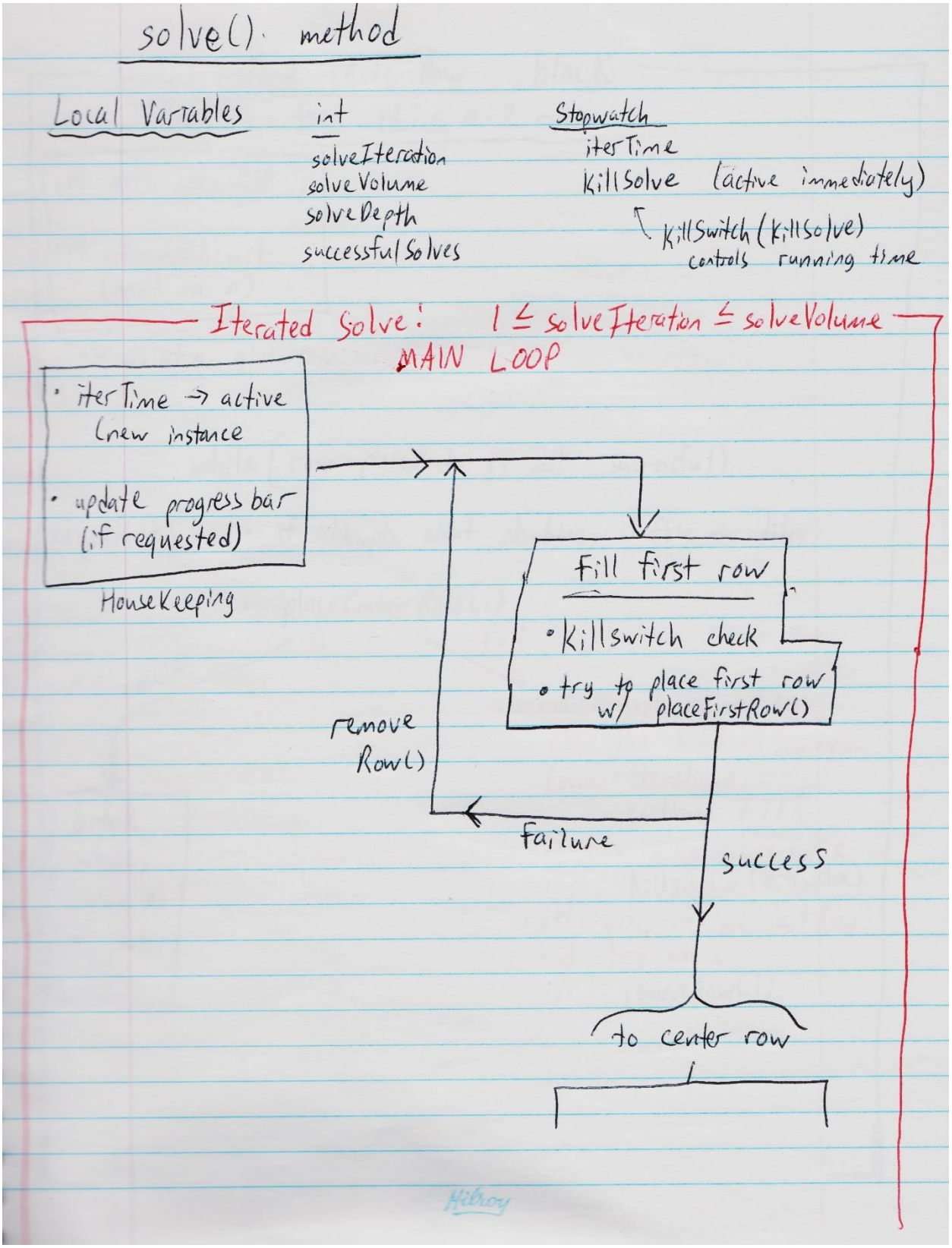
- I've implemented an additional loop in the menu to allow batch solving: the solver is now in a state where data collection can be performed reasonably. This feature will be removed for the final version as it's cumbersome, and further bloats the user menu, and is unlikely to be the sort of thing the COEN 352 assessment team (i.e. target user) needs.

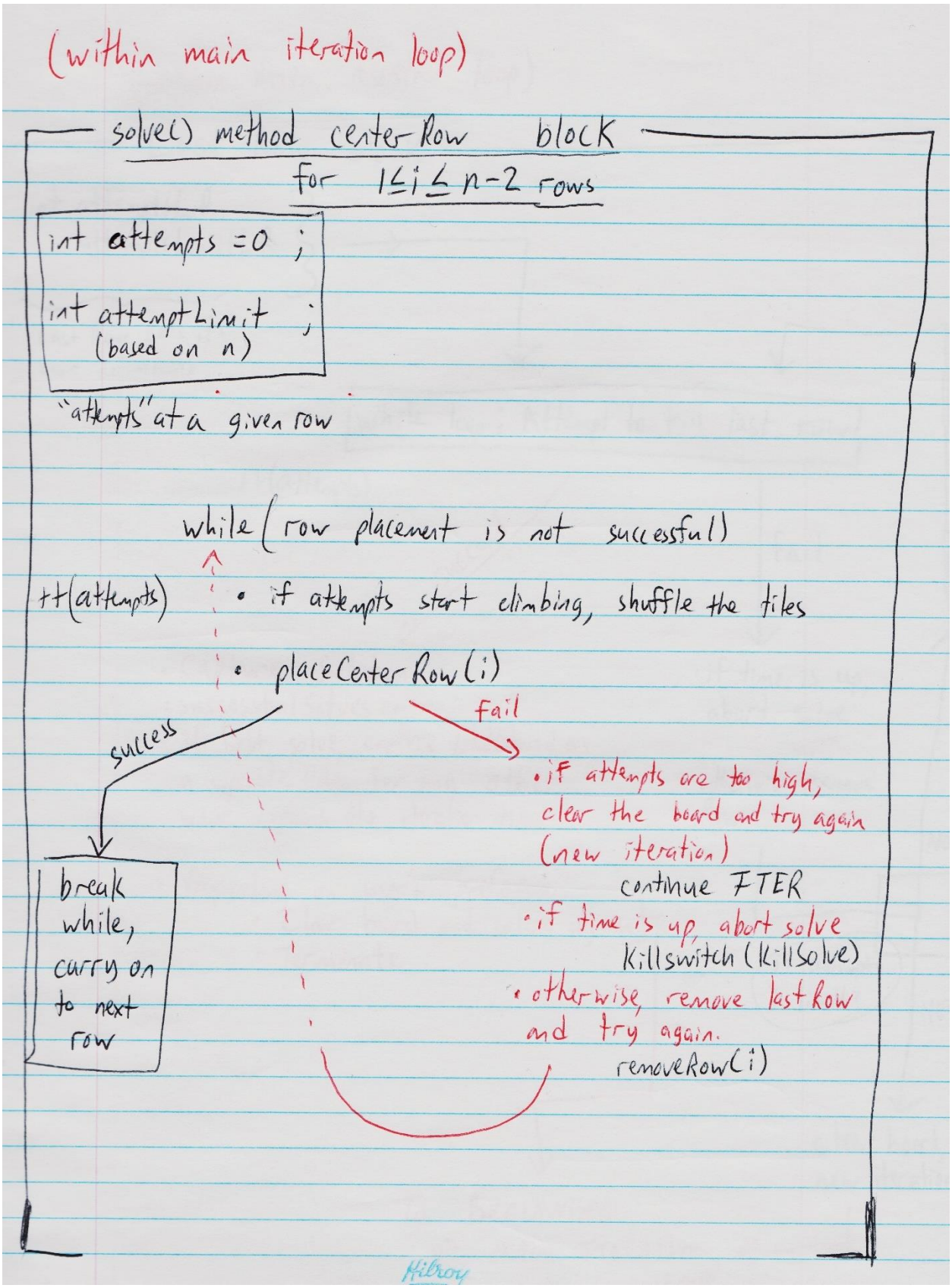
Version 3.0 - Full Solver

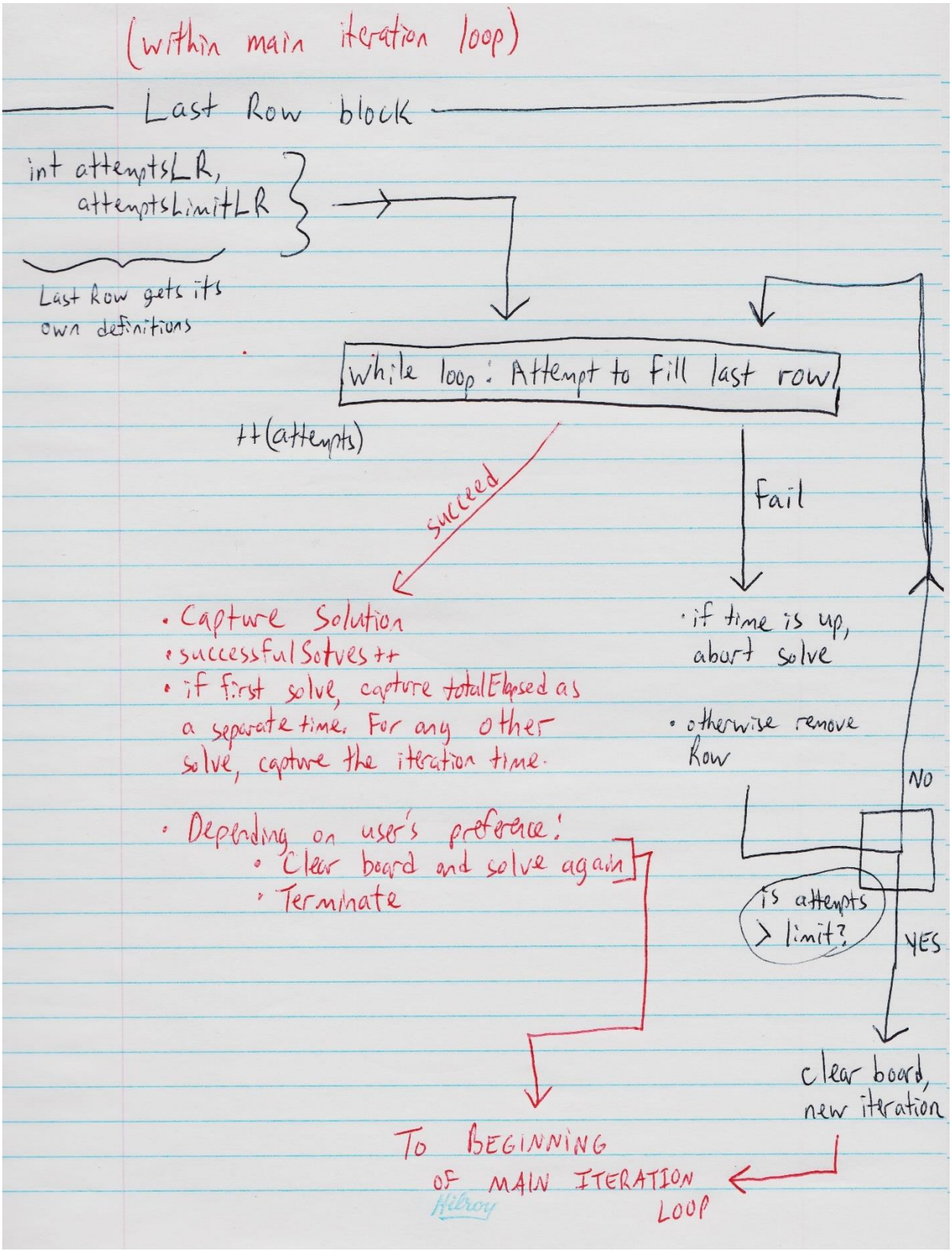
Class Dependency



solve() Method – Full Definition




(within main iteration loop)
solve() method centerRow block
for 1≤i≤n-2 rows
int attempts = 0 ;
int attemptLimit ; (based on n)
"attempts" at a given row
while (row placement is not successful)
++(attempts)
• if attempts start climbing, shuffle the tiles
• placeCenterRow(i)
fail
success
• if attempts are too high, clear the board and try again (new iteration)
continue ITER
• if time is up, abort solve
killswitch (killSolve)
• otherwise, remove last row and try again.
removeRow(i)
break while, carry on to next row


(within main iteration loop)
Last Row block
int attemptsLR,
attemptsLimitLR
Last Row gets its own definitions
while loop: Attempt to fill last row
++(attempts)
succeed
fail
• Capture Solution
• successfulSolves ++
• if first solve, capture totalElapsed as a separate time. For any other solve, capture the iteration time.
• Depending on user's preference:
• Clear board and solve again
• Terminate
• if time is up, abort solve
• otherwise remove Row
NO
is attempts > limit?
YES
clear board, new iteration
To BEGINNING OF MAIN ITERATION LOOP
Hilroy

Version 3.xb - A Failed Experiment in Improved Backtracking: Data collection using simple backtracking (see Part 3) indicated that the solver was never able to solve a puzzle with size $n \geq 7$. To address this deficiency, plans were made to grant the solver "improved backtracking" capabilities: instead of removing one row, the solver would be permitted to remove multiple rows under certain circumstances.

In theory, this is the next logical step for a greedy row-by-row solver: if multiple failed attempts to place the same row suggests that there is only one "best" valid placement pattern, removing many rows permits the possibility of a deeper "retry" of a partially completed puzzle, while still guaranteeing that the original "best" solution can be reconstructed.

To this end, several changes were implemented to the solver to enable use of this feature:

- Created removeMultipleRows method, which facilitates multiple invocations of the removeRow method, along with a centerBag shuffle.
- Each iteration is granted an integer array of "removeTokens", which control how many times a given iteration can take advantage of each improved backtracking block (next point)
- Several nested if-branches were constructed to check the active row alongside the number of placement attempts within a given iteration. Each block invokes the removeMultipleRows method, eliminates one removeToken, fixes centerRowIdx, and passes control back to the beginning of the "Center Row Fill" loop. The fillCenterRow method then continues working from the beginning of the newly cleared block of rows.
- Version 3.2b introduced replenishment, allowing boards with higher *n* to replenish tokens for previous removeMultipleRow opportunities, and thus deepen the scope of the solve even further.

Unfortunately, this feature proved fruitless (though it has been retained in the source code for demonstrative purposes). Even with generous backtrack depth and significantly more solve time allotted per iteration, the "improved backtracking" could not improve solution quality. In fact, the lower bound of quality seems to *decrease* for higher *n* and deeper backtracking (without affecting the upper bound), suggesting that the so-called "improved backtracking" actually *lowers* mean solution quality.

Part 3: Present Solutions

The Code

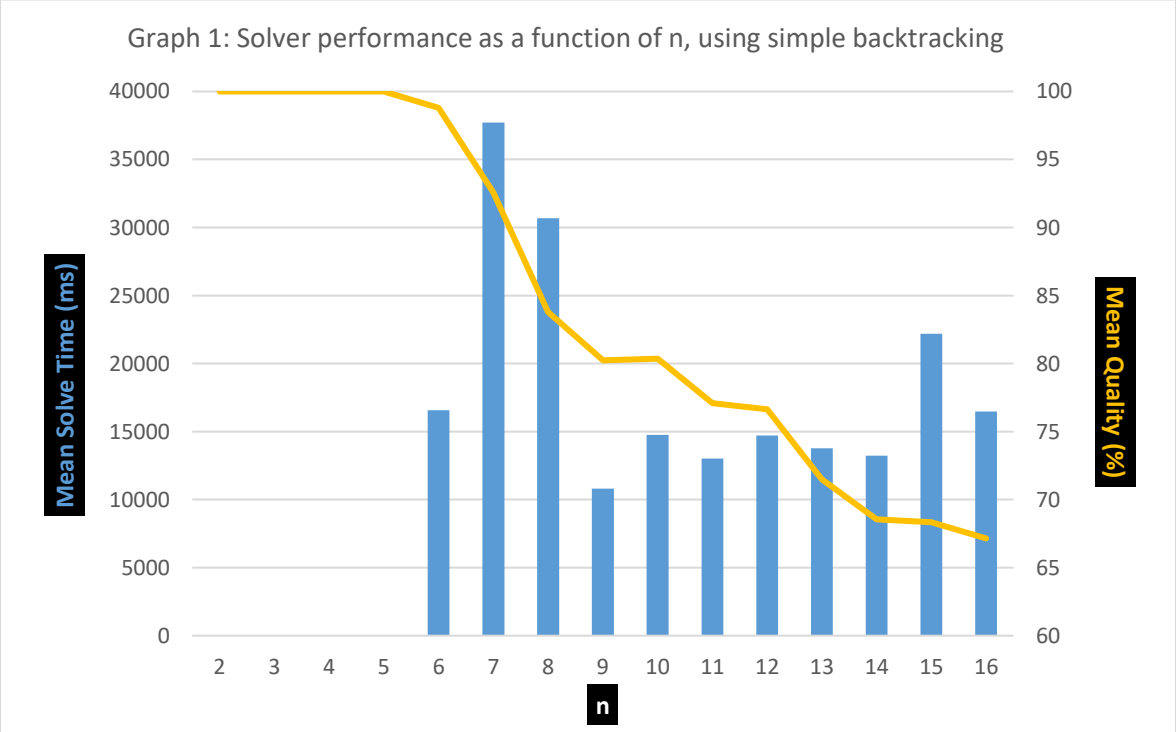
The final solver has three class files comprising about thirty methods (including things like getters) and up to forty class variables (including things like loop counters). There is also a single static class used for implementing the Stopwatches. The driver and Tile classes are relatively stable in terms of time and memory requirements with increasing *n*; as expected it is within the GameBoard class that the most variation is observed, with respect to system resource requirements.

The solver implementation closely follows the model description laid out in Part 2 for Version 3.0, with functionality added for improved backtracking. However, since "improved" backtracking did not actually improve the quality of solutions, all data and calculations are taken using "simple" backtracking (i.e. single row removal), since this mode uses the fewest instructions per solve.

For the abridged source code, please refer to the Appendix.

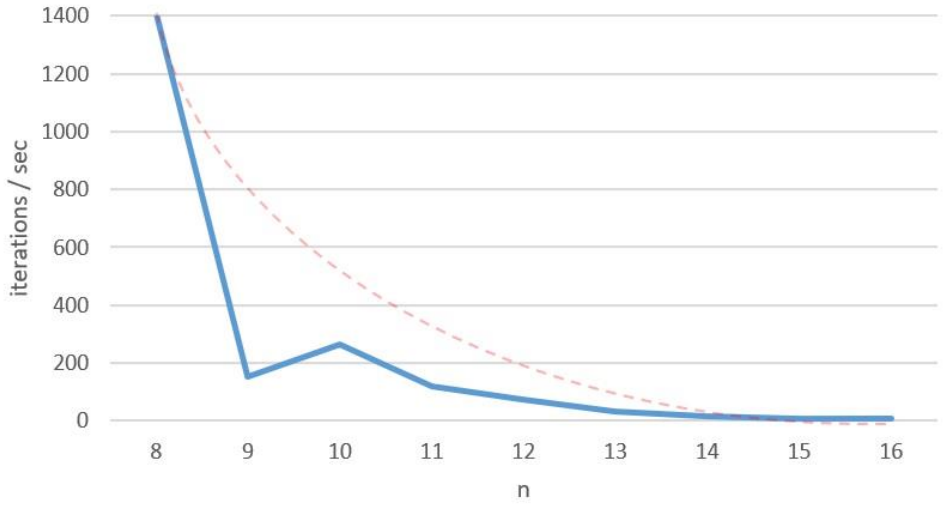
Time Performance

Graph 1 demonstrates the efficiency of the solver as *n* increases. Solutions can reliably be found for boards with size *n* = 2 to *n* = 6, while solutions for *n* > 6 remain elusive. The *mean solve time* of the solver appears to increase sharply for boards it is able to solve, before plateauing for unsolvable boards at a solve time (i.e. of the best solution) around 15 sec/solve. *Mean solve quality* decreases steadily for large *n*, falling from 100 after *n* > 5 and settling in between 65 and 70% for the Eternity II tileset at *n* = 16. Given these two observations, and the polynomial growth of *n*, it appears that the solve quality is also decreasing with polynomial order.



For $n > 7$ it was also found useful to observe the throughput of solve attempts: as the number of rows increases, the number of center tiles increases with polynomial order, and thus increases the amount of work the solver must accomplish. Graph 2 illustrates the expected result: as n increases, the amount of time spent per solve iteration also increases, and the solver is not able to process as many iterations per second. The throughput decreases accordingly, again with (seemingly) polynomial order

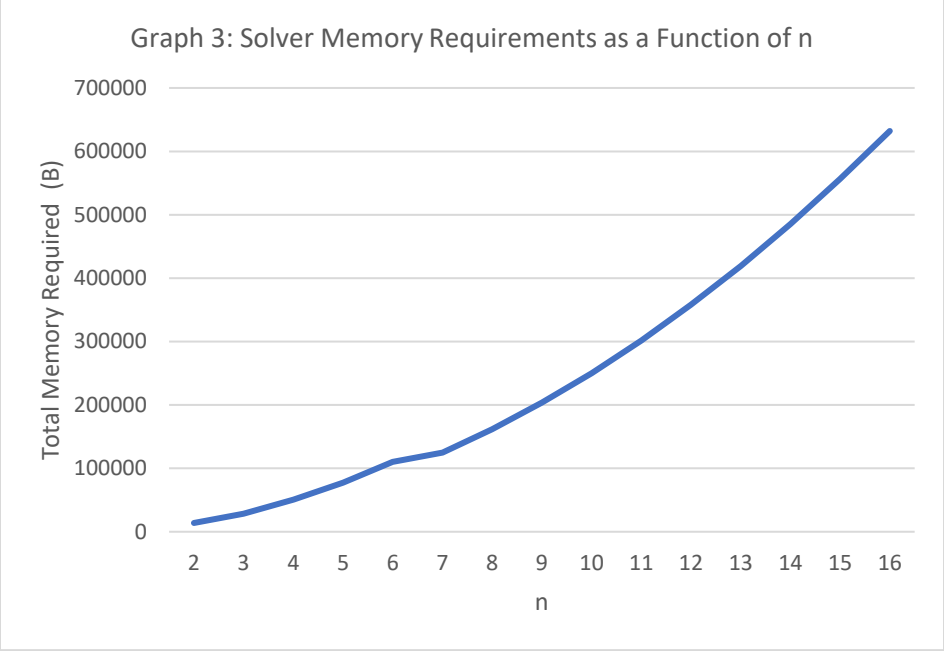
Graph 2: Iteration completion efficiency as a function of n



For the full time trial dataset, please refer to the Appendix.

Memory Performance

Even with a self-imposed mandate to design the solver as memory-neutral during execution, the size of the program was still expected to increase steadily with *n*. Indeed, Graph 3 expresses the trend succinctly: as *n* increases, the area of the board n^2 also increases. Therefore, the size of any data structure that handles either the game board or the body of tiles must also increase with polynomial order.



n	TOTAL MEMORY* in Bytes
2	13 854
3	28 614
4	50 622
5	77 578
6	110 382
7	124 614
8	161 614
9	203 462
10	250 158
11	301 702
12	358 094
13	419 334
14	485 422
15	556 358
16	632 142

* Sum of (calculated compile time) and (expected runtime) memories required

Main sources of increasing memory demand include the gameboard (multidimensional array of increasing size that holds custom objects) and the tileset file Scanner object (high overhead with a similarly increasing number of integers processed for increasing *n*).

For a complete breakdown of the solver's memory requirements in terms of both compile time and runtime components, please refer to the Appendix.

Part 4: Analyze the Results

Time Analysis

As shown in Graph 1, solver performance seems to decrease with polynomial order. This suggests not only polynomial growth in the data set (confirmed by inspection) but also in the number of instructions performed by the solver. Analyzing the behaviour of a typical solve can test this hypothesis.

Big O analysis of GameBoard.solve() :

// BLOCK	**// LOOPS AND SUBROUTINES (Nesting shown w/ indentation)**					**O(x) upper limit**	
// first		while(flag, constant m_1)				$O(m_1)$	
// row			placeFirstRow()				
			while(q < 4)			O(4)	
			while(q < 4)			O(4)	
			for(i < n-2)			O(n)	
				for(j ≤ 4(n-2))		O(n)	$O(n^2)$
			while(q < 3)			O(3)	
//center							
// rows	for (n - 2)					O(n)	
		while(flag, constant m_2)				$O(m_2)$	
			placeCenterRow()				
			while(q<borderBag)			O(n)	
			for(i<n-2)			O(n)	
				for(j<centerBag)		$O(n^2)$	$O(n^3)$
			while(q<borderBag)			O(n)	
// last		while(flag, constant m_3)				$O(m_3)$	
// row			placeLastRow()				
			for(1, 2)			O(2)	
			for(i<n-2)			O(n)	
				for(j<borderBag)		O(n)	O(n^2)

For details, please refer to Appendix – Abridged Source Code

Summary:

Place First Row:	$O(m_1) \cdot (O(11) + O(n^2))$	$\approx O(n^2 m_1)$
Place Center Rows:	$O(n) \cdot (O(m_2) \cdot (O(n^3) + O(n)))$	$\approx O(n^4 m_2)$
Place Last Row:	$O(m_3) \cdot (O(2) + O(n^2))$	$\approx O(n^2 m_3)$

Therefore, the total order of a single solve is approximately **$O(n^2 m_1) + O(n^4 m_2) + O(n^2 m_3)$** . Current implementation of the solver performs *k* iterations per routine, which would multiply the above expression by **an additional O(*k*)**. If the number of iterations is ignored, and the constant m_2 is allowed to depend on *n*, the solver can in this way reach a final upper limit of $O(n^5)$ per solve. Considering that input only scales with n^2 - and that n^2 was thereby the estimated order of decrease of solution efficiency - a polynomial of fifth order is a remarkable magnitude, suggesting fundamental inefficiencies in the solver.

This deficiency partially explains the lack of solves for higher *n*: the solver is operating three polynomial orders lower than intended. This greatly lowers the solve throughput and, in the absence of very high levels of iterations or computation time, limits the ability of the solver to operate according to expectations. Correcting this deficiency would require a redesign of the current solver with less dependence on loop-nesting and more precise choices of the "attempt limit" constants m_1, m_2, and m_3.

Space (Memory) Analysis

Program memory requirements range from about 14KB for $n = 2$ up to almost 618 KB for $n = 16$. As previously mentioned, even a solver design focused on memory-neutral execution cannot escape the mathematics of the situation: with increasing *n*, the creation of n^2 user-defined objects (as well as the ever-growing space in which to place them) quickly inflates memory requirements. Such issues could likely be addressed by a cleaner Tile object class definition, or by changing the way data is managed within the program (see Possible Design Improvements, below).

It is unlikely that the memory requirements were as impactful on solution quality as were the time requirements. Modern memory is cheap and fast, and even 618 KB is not a large amount of memory for a typical student's laptop to manage. The fact that space requirements increase with polynomial order for increasing *n*, however, suggests that such a solver would not be easily implemented on smaller (e.g. microcontroller) devices.

Possible Design Improvements

These are some features that I would love to have implemented, given more time. Each of these features would likely improve the solve by either reducing time/memory requirements or improving solve quality. Unfortunately, most would require a complete re-design of the current solver, or big ugly pieces added at the last minute that might affect the solve in unknown ways without proper time for testing.

Wildcard Tiles – if the solver runs out of attempts for a given iteration, or performs a certain number of iterations, reconstruct the best solve and insert a Wildcard Tile in the next "unfillable" array cell. A Wildcard can be placed anywhere on the board and always matches with every neighbouring tile, but does not increase piecesPlaced for solution quality assessment purposes. Wildcards would permit the solver to continue deeper into some solves that might have otherwise required surrender.

Tile Mining – when tiles or available moves are running low, the solver can access previously placed tiles among its candidates. If removing k placed tiles allows the forward solve to place even $k + 1$ additional tiles, the resulting solution will have a higher quality than otherwise. The solver would need a way to recognize which placed tiles are best candidates for replacement, and the best time to invoke the strategy.

Better Use of Mathematical Modelling – Some features (notably the menu, the saved solutions array, and the number of row placement attempts per iteration) were implemented using "good enough" arithmetic: a domain of values that satisfy the requirement of the feature but are not optimized. This is not important for things like choice of data type for menu options, but for large data structures like String arrays with dozens of lines per entry, the result is some of the memory bloat observed above. Improved mathematical modelling would allow more exact domains of operation to be determined, streamlining the solver and improving throughput (and possibly the operation of other features like backtracking, which depends on the number of attempts per iteration, i.e. the constants m_1, m_2, m_3 used in the Big O analysis of solve()).

Improved Backtracking 2.0 – Improved backtracking that includes the first and last rows, allowing better placeCenterRow attempts by freeing up the border tiles "locked" in these rows. Redesigning the solver around improved backtracking would also allow for smoother execution than currently implemented.

Ditching Linked Lists and Scanner Objects – For increasing n, a significant amount of overhead memory could be saved by using a custom data structure (or even an ArrayList) instead of Linked Lists. Similar gains could be made by using Java StdIn instead of Scanner objects, to avoid overhead costs that scale with n^2.

Appendix – Time Trial Data (Simple Backtracking)

	N = 2		N = 3		N = 4		N = 5		N = 6		N = 7	
Trial #	Solve time (ms)	Quality	Solve time (ms)	Quality	Solve time (ms)	Quality	Solve time (ms)	Quality	Solve time (ms)	Quality	Solve time (ms)	Quality
1	< 1	100	< 1	100	1	100	111	100	548	100	59 688	92
2	1	100	< 1	100	1	100	36	100	9 739	97	2 015	92
3	< 1	100	< 1	100	< 1	100	150	100	58 834	97	59 765	92
4	< 1	100	< 1	100	6	100	98	100	33 846	94	3 573	94
5	< 1	100	< 1	100	1	100	47	100	2 802	100	26 341	92
6	< 1	100	< 1	100	1	100	1	100	44 670	97	41 355	92
7	< 1	100	< 1	100	2	100	33	100	2 908	100	10 614	92
8	< 1	100	< 1	100	< 1	100	3	100	4 392	100	21 991	94
9	< 1	100	< 1	100	3	100	45	100	1 683	100	52 049	94
10	< 1	100	< 1	100	2	100	41	100	1 075	100	10 001	90
11	< 1	100	< 1	100	8	100	25	100	1 024	100	34 483	92
12	< 1	100	< 1	100	2	100	27	100	183	100	14 846	94
13	< 1	100	< 1	100	1	100	107	100	1 067	100	65 290	92
14	< 1	100	< 1	100	2	100	49	100	3 870	100	54 707	94
15	< 1	100	< 1	100	2	100	9	100	1 223	100	45 058	92
16	< 1	100	< 1	100	1	100	13	100	467	100	94 801	96
17	< 1	100	< 1	100	< 1	100	18	100	95 763	94	100 745	92
18	< 1	100	< 1	100	2	100	21	100	317	100	9 462	90
19	< 1	100	< 1	100	6	100	10	100	1 325	100	1 707	92
20	< 1	100	< 1	100	< 1	100	14	100	65 478	97	45 775	94
Mean	**< 1**	**100**	**< 1**	**100**	**<2.25**	**100**	**42.9**	**100**	**16 560.7**	**98.8**	**37 713.3**	**92.6**

N.B.: data for $n \in [2,7]$ obtained using 100,000 solve iterations per trial

	N = 8			N = 9			N = 10		
Trial #	**Solve time (s)**	**Quality**	**iters***	**Solve time (s)**	**Quality**	**iters***	**Solve time (s)**	**Quality**	**iters***
1	21	86	84.8 K	1.2	80	9.3 K	19.7	77	15.2 K
2	46	84	84.6 K	24.5	80	9.3 K	4.5	79	15.1 K
3	9.6	83	84.2 K	15.4	81	9.3 K	2	79	15.3 K
4	15.5	84	84.7 K	0.08	77	9.1 K	10.2	79	15.2 K
5	0.3	83	84.4 K	2.8	80	9.4 K	0.9	79	16.7 K
6	48	83	84.4 K	6.6	81	9.3 K	7	78	16.6 K
7	35	84	84.8 K	0.7	80	9.2 K	13.9	84	16.4 K
8	44	83	83.9 K	5.3	81	9.3 K	13	83	16.4 K
9	30	84	84.2 K	8	79	9.2 K	1.2	77	16.7 K
10	28	83	84.9 K	7.9	83	9.3 K	13.6	82	16.7 K
11	41	86	84.1 K	5.8	80	9.2 K	30.1	83	16 K
12	12.4	84	84.2 K	0.6	81	9.1 K	65.8	84	16.2 K
13	30	84	84.1 K	28.1	80	9.3 K	4.5	82	16.1 K
14	51	84	84.3 K	19.9	83	9.2 K	23	82	15.9 K
15	46	83	84.6 K	8.7	81	9.2 K	22.5	78	15.7 K
16	34.1	83	84.2 K	20.1	77	9.3 K	15	79	15.6 K
17	58.6	84	84.6 K	33.2	80	9.1 K	27.8	82	16 K
18	17.5	83	84.4 K	3.6	81	9.4 K	0.5	79	16 K
19	11.1	84	84.4 K	5.5	81	9.3 K	12.9	79	16 K
20	34.5	84	84.3 K	18.1	79	9.2 K	7.3	82	15.8 K
Mean	**30.68**	**83.8**	**84.41 K**	**10.8**	**80.3**	**9.25 K**	**14.8**	**80.4**	**15.98 K**
iters/sec			**1407**			**154**			**266**

***** batch run time = 60 seconds

	N = 11			N = 12			N = 13		
Trial #	**Solve time (s)**	**Quality**	**iters***	**Solve time (s)**	**Quality**	**iters***	**Solve time (s)**	**Quality**	**iters***
1	11.7	76	7 K	7.5	76	4.5 K	21.2	73	1868
2	14.5	78	7.5 K	27.1	76	4.5 K	0.3	72	1877
3	3.9	76	7.4 K	19.5	76	4.5 K	6.9	72	1875
4	2.7	77	7.4 K	20	74	4.5 K	16.2	70	1887
5	47.4	77	7.3 K	13.1	76	4.6 K	1.6	74	1871
6	1.4	78	7.4 K	23.6	79	4.5 K	0.8	70	1878
7	20.2	78	7.4 K	12.8	76	4.5 K	21.2	72	1854
8	0.7	77	7.4 K	30.4	78	4.6 K	4.5	72	1864
9	4.7	79	7.3 K	18.1	77	4.5 K	0.9	68	1901
10	15.6	78	7.1 K	0.1	76	4.5 K	15.7	72	1866
11	23.8	77	7.5 K	15.6	76	4.5 K	12.6	71	1850
12	36.4	78	7.3 K	16.7	81	4.5 K	12.3	73	1878
13	28.5	78	7.4 K	2.8	79	4.5 K	2.2	71	1870
14	10.8	77	7.4 K	0.9	74	4.5 K	18.5	72	1876
15	17.5	78	7.3 K	48.9	78	4.5 K	3.7	72	1860
16	13.1	76	7.5 K	2.9	74	4.6 K	38.5	71	1872
17	0.6	75	6.7 K	1.8	78	4.5 K	50.4	72	1871
18	2.3	77	6.7 K	8.9	78	4.5 K	8.3	72	1898
19	1.9	76	6.7 K	10.1	74	4.6 K	13	69	1856
20	2.8	76	6.7 K	13.4	77	4.5 K	26.6	72	1841
Mean	**13.0**	**77.1**	**7.22 K**	**14.7**	**76.7**	**4.52 K**	**13.8**	**71.5**	**1870.7**
iters/sec			**120**			**75**			**31**

***** batch run time = 60 seconds

	N = 14			N = 15			N = 16		
Trial #	**Solve time (s)**	**Quality**	**iters***	**Solve time (s)**	**Quality**	**iters***	**Solve time (s)**	**Quality**	**iters***
1	15.8	67	847	47.6	69	532	24.5	67	337
2	15.7	71	863	4.6	68	536	16.6	67	347
3	10.5	68	873	27.5	68	552	29	66	345
4	0.1	70	859	7	72	527	8.7	68	348
5	5	69	869	26.4	69	536	37.4	68	348
6	46.3	71	850	10.8	69	533	42.5	68	343
7	15	68	870	39.1	70	535	11.5	65	341
8	0.2	66	846	36.6	66	531	22.3	66	339
9	35.7	67	831	17.6	66	537	12.1	67	349
10	19.4	68	829	11.3	66	549	17.2	68	352
11	10.9	68	854	29	69	531	11.3	70	341
12	34.2	69	845	11.3	68	543	1	67	357
13	4.8	68	856	66.5	69	541	13	67	346
14	11.3	69	822	3.2	68	536	11.5	67	350
15	1.4	68	851	15.5	68	543	9.4	68	341
16	5.1	69	844	10.3	71	544	22.1	66	338
17	1.4	70	863	26.7	68	542	3.4	68	351
18	15.3	71	845	2.6	69	550	16.1	68	342
19	13.1	70	849	8.9	68	525	13.3	65	357
20	3.4	64	824	41.3	66	535	6.7	67	345
Mean	**13.2**	**68.6**	**849.5**	**22.2**	**68.4**	**537.9**	**16.5**	**67.2**	**345.9**
iters/sec			**14**			**9**			**6**

***** batch run time = 60 seconds

Appendix – Breakdown of Solver Memory Requirements

N.B.: All sizes are in bytes **N.B.: String sizes calculated via 8·(int)(((# of chars)·2)+45)/8 †**

Driver - pre-solve overhead			
Item	**Quantity**	**Size**	**Total**
Scanner[1]	1	1	< 100
int	6	4	24
GameBoard	1	9 112 - 440 520	9 112 - 440 520
		Grand Total	9 236 - 440 644

Single Tile object – cost of existence			
Item	**Quantity**	**Size**	**Total**
overhead	1	8	8
int	2	4	8
int[]			
o.head	1	8	8
ints	4	4	16
		Grand Total	40

Tile class methods – reserved memory			
Item	**Quantity**	**Size**	**Total**
rotate()			
int	2	4	8
int[]			
o.head	1	16	16
int	4	4	16
returnTile()			
int	1	4	4
String	1	≤ 58	58
		Grand Total	98

GameBoard instantiation			
Item	**Quantity**	**Size**	**Total**
overhead	1	8	8
int	4	4	16
double	2	8	16
boolean[]			
o.head/pad	1	16	16
values	3	?	~ 0
int[]			
o.head/pad	1	16	16
int	2	4	8
Tile[][] board			
o.head/pad	1	16	16
size of data	1	7 680 - 419 840	7 680 - 419 840
String[] solutions			
o.head/pad	1	16	16
size of data	1	1 216 - 14 416	1 216 - 14 416
LinkedLists (sum of three)	3	varies	120 - 6 168
		Grand Total	9 112 - 440 520

† Source: https://stackoverflow.com/questions/4385623/bytes-of-a-string-in-java/4385653

[1] Source: https://stackoverflow.com/questions/8135903/java-memory-usage-for-scanner

GameBoard class methods – reserved memory			
Item	**Quantity**	**Size**	**Total**
solve()			
Stopwatch	2	24	48
int	12	4	48
double	1	8	8
long	2	8	16
(double) cast	3	8	24
boolean flags	3	?	~ 0
Strings (debug)	2	48	96
String (sol'n)*	1	0 - 3 600	0 - 3 600
int[]			
o.head/pad	1	16	16
int	10	4	40
		solve Subtotal	296 - 3 896
placeFirstRow()			
Random	1	8	8
int	4	4	16
Tile	2	40	80
placeCenterRow()			
int	4	4	16
Tile	3	40	120
placeLastRow()			
boolean	1	?	~ 0
int	3	4	12
Tile	3	40	120
		place Subtotal	372

*In practice. Larger solutions would have larger strings.

Item	**Quantity**	**Size**	**Total**
removeRow()			
int	1	4	4
removeMultiple()			
int	1	4	4
int from Single	1	4	4
clearBoard()			
int	1	4	4
shuffleBag()			
Random	1	8	8
int	4	4	16
Tile	1	40	40
populateBag()			
int	2	4	8
int[]			
o.head/pad	1	16	16
int	4	4	16
String[]			
o.head/pad	1	16	16
data	1	≤ 22	22
Tile	1	40	40
createTileBags()			
String	1	≤ 54	54
Scanner[1]	1	3 200 - 204 800	3 200 - 204 800
Exception	1	8	8
	other methods Sub.		3 260 - 205 060

Grand Total, Gameboard Methods	3 928 - 209 318

Detail: tileFile Scanner object (GameBoard) (four integers per tile @ 200B per integer)[1]			
n	**area**	**integers**	**bytes**
2	4	16	3200
3	9	36	7200
4	16	64	12800
5	25	100	20000
6	36	144	28800
7	49	196	39200
8	64	256	51200
9	81	324	64800
10	100	400	80000
11	121	484	96800
12	144	576	115200
13	169	676	135200
14	196	784	156800
15	225	900	180000
16	256	1024	204800

Detail: String[] solutions (within solve())
(one Tile.report() produces 88 to 104 bytes;
for simplicity, each Tile is assumed to produce a 100 B report)

			Size of one solution[3]		
n	**# solutions[2]**	**Size of one row of solution**	**Theoretical**	**Practical**	**Max size of data in array**
2	3	200	400	400	1200
3	3	300	900	900	2700
4	4	400	1600	1600	6400
5	4	500	2500	2500	10000
6	4	600	3600	3600	14400
7	4	700	4900	40	40
8	5	800	6400	40	40
9	5	900	8100	40	40
10	5	1000	10000	40	40
11	5	1100	12100	40	40
12	6	1200	14400	40	40
13	6	1300	16900	40	40
14	6	1400	19600	40	40
15	6	1500	22500	40	40
16	7	1600	25600	40	40

[2] Hardcoded into solver [3] Solution strings for n > 6 are always empty

Detail: tileBags (LinkedLists) (@ 24B per element)[4]							
n	**corners**	**borders**	**centers**	**size of cornerBag**	**size of borderBag**	**size of CenterBag**	**total size**
2	4	0	0	96	0	0	96
3	4	4	1	96	96	24	216
4	4	8	4	96	192	96	384
5	4	12	9	96	288	216	600
6	4	16	16	96	384	384	864
7	4	20	25	96	480	600	1176
8	4	24	36	96	576	864	1536
9	4	28	49	96	672	1176	1944
10	4	32	64	96	768	1536	2400
11	4	36	81	96	864	1944	2904
12	4	40	100	96	960	2400	3456
13	4	44	121	96	1056	2904	4056
14	4	48	144	96	1152	3456	4704
15	4	52	169	96	1248	4056	5400
16	4	56	196	96	1344	4704	6144

[4] Source: https://stackoverflow.com/questions/11564352/arraylist-vs-linkedlist-from-memory-allocation-perspective

Detail: board Tile[][] (@ 40 B per Tile)				
n	**size of n Tiles**	**[n2] unpadded**	**padded**	**Size of Tile[][], padded[5]**
2	80	92	96	7680
3	120	132	136	16320
4	160	172	176	28160
5	200	212	216	43200
6	240	252	256	61440
7	280	292	296	82880
8	320	332	336	107520
9	360	372	376	135360
10	400	412	416	166400
11	440	452	456	200640
12	480	492	496	238080
13	520	532	536	278720
14	560	572	576	322560
15	600	612	616	369600
16	640	652	656	419840

[5]Multidimensional arrays:

Size of Type[n] ≡ K

= n x sizeOf.(Type), padded

Size of Type[n][n]

= { o.head[n] + n x sizeOf.(Type) }

* K , padded

Appendix – Abridged Source Code

// DRIVER CLASS

```
import java.util.Scanner;

public class E_II_Driver
{
	public static void main(String[] args)
	{
		// program intro: println statements omitted

		Scanner userIn = new Scanner(System.in);

		// user menu
		while(true)
		{
			// PARAMETER DEFINITION BLOCK
			// several prompts and parameter instantiations omitted

			// MAIN SOLVE ROUTINE BLOCK
			GameBoard newGame = new GameBoard(USER_SIZE);
			newGame.solve(iters, detail, depth, bktk);    // params defined
                                                          // in menu block by user
			// Re-run? prompt
			if (prompt)
				continue;
			else
				break;
		}	// end menu

		userIn.close();
	}	// end main
}		// end driver class
```

// TILE CLASS

```
public class Tile
{
	private int ID;
	private int[] directions = new int[4];
	private int degree = 0;

	public Tile(int newID, int[] directionNumbers)
	{
		ID = newID;
		for (int i = 0; i < 4; i++)
			directions[i] = directionNumbers[i];
	}

	public void rotate(int degree)   // 0, 1, 2, or 3. Number of ROR-style ops
	{
		int[] temp = new int[4];

		for (int i = 0; i < 4; i++)
			temp[(i+degree)%4] = this.directions[i];

		for (int i = 0; i < 4; i++)
			this.directions[i] = temp[i];

		this.logDegree(degree);
	}

	public int[] getDirections()     // returns current orientation, int array
	{	return directions;        }


	public String returnTile()       // returns current orientation, string
	{
		String s = "";

		for (int i = 0; i < 4; i++)
			s = s + this.directions[i] + " ";

		s = s + "  ID: " + this.getID() + "  Deg: " + this.getDegree();
		return s;
	}

	public void reportTile()         // reports current orientation, for debugging
	{	System.out.println(this.returnTile();  }

	public void logDegree(int rotDeg)
	{	degree = (degree + rotDeg) % 4;  }

	public int getDegree()
	{ return this.degree;     }

	public int getID()
	{ return ID; }
}
```

// GAMEBOARD CLASS
// Basic Gameplay Methods

```
// imports omitted

public class GameBoard
{
	// instance data declarations omitted

	// Constructor
	public GameBoard(int boardSize)
	{
		// basic instantiation operations omitted
		createTileBags();

		//Shuffling bags
		shuffleBag(cornerBag);
		shuffleBag(borderBag);
		shuffleBag(centerBag);
	}

	// Getters omitted

	public void placeTile(Tile tile, int row, int col)
	{
		board[row][col] = tile;
		piecesPlaced++;
	}

	// Helper method to constructor: populates bags with Tiles
	public void createTileBags()
	{
		// Build tileFile string and load file
		String tileFile = "./src/tilesets/";	// hardcoded tileset folder!

		if (size >=2 & size <= 15)
			tileFile = tileFile + size + "x" + size + ".txt";
		else if (size == 16)
			tileFile = tileFile + "E2_Tiles.txt";
		else
			tileFile = tileFile + "badInput.txt";

		try
		{
			Scanner tileFileScanner = new Scanner(new File(tileFile));

			int w = size - 2;
			populateBag(4, 0, cornerBag, tileFileScanner);
			populateBag(4 * w, 4, borderBag, tileFileScanner);
			populateBag(w * w, 4 + (4*w), centerBag, tileFileScanner);
		}

		catch (FileNotFoundException e)
		{	e.printStackTrace();	}
	}
```

```
// Helper helper
public static void populateBag(int numberOfTiles, int startIDx, Queue<Tile>
                                activeBag, Scanner sc)
{
	for (int i = 1; i <=numberOfTiles; i++)
	{
		String[] nextLineInFile = sc.nextLine().split(" ");
		int[] newDirs = new int[4];
		for (int j = 0; j < nextLineInFile.length; j++)
			newDirs[j] = Integer.parseInt(nextLineInFile[j]);

		int newID = startIDx + i;
		Tile nextTile = new Tile(newID, newDirs);
		activeBag.add(nextTile);
	}
}

// Modified Knuth shuffle for a tile bag
public static void shuffleBag(Queue<Tile> bag)
{
	int size = bag.size();

	for (int i = 1; i < size; i ++)
	{
		Random r = new Random();
		int q = r.nextInt(i) + 1;

		Tile heldTile = bag.poll();
		for (int j = 0; j < q; j ++)
			bag.add(bag.poll());
		bag.add(heldTile);
	}
}
```

// GAMEBOARD CLASS
// solve() method

```
	public void solve(int numberOfIterations, int detail, int depth, int bktk)
	{
		// Parameter definitions
		int solveIteration;
		// "fine gain" on iterations; can also change in driver
		int solveVolume = numberOfIterations * 5 + 0;
		int solveDepth = 1 + depth;

		// Solution tracking - works with bestSolve and solveTime
		int successfulSolves = 0;

		// Stopwatches
		Stopwatch iterTime;
		Stopwatch killSolve = new Stopwatch();

		// Main solve loop: each loop is one solve attempt on an empty board
		System.out.println("Solve in progress...");
		ITER: for (solveIteration = 1; solveIteration <= solveVolume;
								solveIteration++)
		{

			// Depth of the multiple backtracking protocol.
			int[] removeTokens = {3, 3, 2, 2, 2, 2, 1, 1, 1, 1};

			// Time elapsed for this iteration
			iterTime = new Stopwatch();

			// Progress bar implementation omitted

			// Place First Row of puzzle by brute force

			boolean firstRowFlagAssert = false;
			while (!firstRowFlagAssert)
			{
				// timeCheck but no bestSolve check (first row)
				if (killSwitch(killSolve, solveDepth, ""))
					break ITER;

				firstRowFlagAssert = this.placeFirstRow();
				if (!firstRowFlagAssert)
					this.removeRow(0);
			}

			// Fill the next (n-2) rows
			CENTER: for (int centerRowIdx = 1; centerRowIdx <= size-2;
								centerRowIdx++)
			{
				// attempt to place a valid row
				String s = "";
				boolean centerRowFlagAssert = false;
				int attempts = 0;
```

```
// define iteration attempt limits for this row.
// arithmetic designed to normalize attempts wrt. n
int attemptLimit = size*centerBag.size();
if (size > 8)
	attemptLimit = attemptLimit –
			(3*centerBag.size()) + (25*size);

// "trap" escaped by placing a valid row
while (!centerRowFlagAssert)
{
	++attempts;

	// Shuffle a stale bag
	if ((attempts*attempts) > centerBag.size())
		shuffleBag(centerBag);

	// Try and place a center row
	centerRowFlagAssert =
			this.placeCenterRow(centerRowIdx);

	if (!centerRowFlagAssert)  // row placement fail
	{
		checkForBestSolve(solveIteration, iterTime);

		// iteration attempt limit exceeded
		if (attempts > attemptLimit)
		{
			//System.out.println("Jammed up!");
			this.clearBoard();
			continue ITER;
		}

		// Time limit exceeded
		if (killSwitch(killSolve, solveDepth, s))
			break ITER;

		// Otherwise, backtracking protocol

		// multiple row backtracking
		if(bktk > 0)
		{

		// here is an example block
		if (centerRowIdx >= 4 && attempts > 100)
			if(removeTokens[0] > 0)
			{
			   removeMultipleRows(centerRowIdx, 3);
			   centerRowIdx = centerRowIdx - 3;
			   --removeTokens[0];
			   continue CENTER;
			}

		// Nine more such blocks omitted for varying
		// centerRowIdx, attempts, removeTokens[x]
```

```
                    }

                    // else, vanilla backtracking
                    this.removeRow(centerRowIdx);

              }       // end if (row placement unsuccessful)
        }             // end while (trap to place a valid row)
}                     // end for (done filling center rows)

// Place Last Row

// Definition of string, flag, attempts and limit omitted;
// see center row subroutine

while (lastRowFlagAssert == false)
{
       attemptsLR++;

       if (attemptsLR > attemptsLimitLR)
       {
              checkForBestSolve(solveIteration, iterTime);
              this.clearBoard();
              continue ITER;
       }

       // Shuffle omitted; occurs sooner than centerRow block

       lastRowFlagAssert = this.placeLastRow();

       //Successful solve
       if (lastRowFlagAssert == true)
       {

              if(detail>0)
              // save stuff for display output

              // Creation of string containing solution omitted

              // Deal with successful solves: break if depth = 0
              // and examine redundant solves if not
              if (successfulSolves == 1)
              {
                     solutions[0] = soln;
                     solnsIdx++;
                     fastestSolve = killSolve.elapsedTime("m");
                     if(depth == 0)
                     // Break after first successful solve
              }

              // subroutine to eliminate redundant solutions
              // in solutions array omitted (it's broken anyway

       }      // end successful solve
```

```
                        else   // unsuccessful solve
                        {
                               // check every time now, since it's last row
                               checkForBestSolve(solveIteration, iterTime);

                               if (killSwitch(killSolve, solveDepth, s))
                                      break ITER;

                               // Remove last row
                               this.removeRow(size - 1);
                        }

                 }      // end place last row

                 // Clear board and start again
                 this.clearBoard();

          } // end ITER block (all iterations are done, or time has run out)

          // Displaying: data summary subroutine omitted
          // Displaying: solutions data subroutine omitted

}      // end main solve() method
```

// GAMEBOARD CLASS
// assess and place Methods

```
	// There are four assess_____Tile() methods. They are functionally the same:
	// assess whether or not a given tile fits in a target cell. The math depends
	// on the destination, and to avoid unwieldly parameter passing in a single
	// assess method, four separate methods are defined. One is included here
	// as an example.
	public boolean assessCenterTile(Tile activeTile, int targetRow, int targetCol)
	{
		int i = 0;
		while (i < 4)
		{
			// check north board tile
			if (activeTile.getDirections()[i] ==
					board[targetRow-1][targetCol].getDirections()[2])
			{
				// check west board tile
				if (activeTile.getDirections()[(i+3)%4] ==
					board[targetRow][targetCol-1].getDirections()[1])
				{
					// a match has been found. rotate and flag up
					if (i%2 == 0)
						activeTile.rotate(i);
					else  // i = 1 or 3
						activeTile.rotate((i+2)%4);
					return true;
				}
			}

			i++;
		}

		// One or more matches failed; tile is discarded
		return false;
	}

	public boolean assessFirstRowBorderTile(Tile activeTile, int targetCol)
	{	// omitted   }

	public boolean assessCenterRowBorderTile(Tile activeTile, int targetRow)
	{	// omitted   }

	public boolean assessLastRowBorderTile(Tile activeTile, int targetCol)
	{	// omitted   }
```

```
// The three place_____Row methods are another case of similar functionality
// being difficult to parameterize in a streamlined way. Essentially,
// placeCenterRow grew out of the first placeRow method created in
// development, and the other two grew out of it in turn.
// This means placeFirstRow and placeLastRow are essentially placeCenterRow
// with customizations; these customizations are shown but the rest of the
// method bodies similar to placeCenterRow are omitted.

public boolean placeFirstRow()
{
	// Choose a random corner tile for the first tile
	Random r = new Random();
	int q = r.nextInt(4) + 1;
	while (q > 0)
	{
		cornerBag.add(cornerBag.poll());
		--q;
	}

	Tile first = cornerBag.poll();

	// continues like placeCenterRow() using cornerBag and borderBag
}

public boolean placeCenterRow(int thisRow)
{
	// Place left border piece
	int k = 1;
	while(k <= borderBag.size())
	{
		Tile activeTile = borderBag.poll();

		if (assessCenterRowBorderTile(activeTile, thisRow))
			// place tile and break
		else
		{
			borderBag.add(activeTile);
			if(++k == borderBag.size())
				return false;
		}
	}

	for (int i = 1; i <= size-2; i++)  // Place the next n - 2 tiles
	{
		// Exhaust tile bag if necessary
		for(int j = 1; j <= centerBag.size(); j++)
		{
			Tile activeTile = centerBag.poll();

			if (assessCenterTile(activeTile, thisRow, i))
				// place tile and break
```

```
                else
                {
                        centerBag.add(activeTile);
                        if (j == centerBag.size())
                                return false;
                }
        }
    }

    // place second border piece
    int k2 = 1;
    while(k2 <= borderBag.size())
    {
        Tile activeTile = borderBag.poll();

                if (assessCenterTile(activeTile, thisRow, size - 1))
                        // place tile and return true

                else
                {
                        borderBag.add(activeTile);
                        ++k2;
                }

    }

    // no valid border piece
    return false;

}


public boolean placeLastRow()
{
    // place first corner tile
    boolean ncFlag = true;
    for (int i = 1; i <= 2; i++)
    {
        Tile activeTile = cornerBag.poll();
        if(assessCenterRowBorderTile(activeTile, size - 1))
        {
                this.placeTile(activeTile, size - 1, 0);
                ncFlag = false;
                break;
        }

        else
                cornerBag.add(activeTile);
    }

    if(ncFlag)
        return false;

    // Place the next n - 2 tiles like placeCenterRow, but with borderBag
```

```
        // place last puzzle piece

        Tile activeTile = cornerBag.poll();

        if (assessLastRowBorderTile(size, activeTile, size - 1))
                // place tile and return true

        else
                // replace tile in bag, and return false

}               // end placeLastRow
```

// GAMEBOARD CLASS
// other Methods

```
	public void removeRow(int row)
	{
		int col = 0;

		if (row == 0 || row == size - 1)	// Row is the top or bottom row
		{
			if (board[row][col] != null)
			{
				cornerBag.add(board[row][col]);
				board[row][col++] = null;
				piecesPlaced--;
			}

			while(board[row][col] != null)
			{
				if(col == size - 1)
					cornerBag.add(board[row][col]);
				else
					borderBag.add(board[row][col]);

				board[row][col++] = null;
				piecesPlaced--;
				if(col == size)
					break;
			}
		}

		else	// Row does not contain corner pieces
		{
			if (board[row][col] != null)
			{
				borderBag.add(board[row][col]);
				board[row][col++] = null;
				piecesPlaced--;
			}

			while(board[row][col] != null)
			{
				if(col == size - 1)
					borderBag.add(board[row][col]);
				else
					centerBag.add(board[row][col]);

				board[row][col++] = null;
				piecesPlaced--;
				if(col == size)
					break;
			}
		}	// end else
	}		// end remove row
```

```
// for improved backtracking
public void removeMultipleRows(int startIdx, int numberOfRows)
{
	for (int i = 0; i <= numberOfRows - 1; i++)
		this.removeRow(startIdx - i);

	shuffleBag(centerBag);
}

public void clearBoard()
{
	int i = 0;

	while (board[i][0] != null)
	{
		if (i == size - 1)
		{
			removeRow(i);
			break;
		}

		else
			removeRow(i);
		i++;
	}
}

public boolean killSwitch(Stopwatch killTimer, double timeCoeff, String debug)
{
	if (killTimer.elapsedTime("m") == (double)(10000 * timeCoeff) &&
							timeWarning[0] == false)
		// Alert user and flip bit in timeWarning

	// two more such blocks omitted

	else if (killTimer.elapsedTime("m") > 30000 * timeCoeff)
	{
		System.out.println(debug + "Time has run out.");
		return true;
	}

	return false;
}

public void checkForBestSolve(int iter, Stopwatch Timer)
{
	if (this.getPiecesPlaced() > bestSolve[1])
	{
		bestSolve[0] = iter;
		bestSolve[1] = this.getPiecesPlaced();
		solveTime = Timer.elapsedTime("m");
		bestBoard = this.getBoard();
	}
}
```

```
	public void pollBags(int cornbag, int bordbag, int centbag)
	{
		if (cornbag == 1)
		{
			System.out.println("Corners: " + cornerBag.size());
			for (int i = 1; i <= cornerBag.size(); i++)
			{
				System.out.print(i + " ");
				cornerBag.peek().reportTile();
				cornerBag.add(cornerBag.poll());
			}
		}

		if (bordbag == 1)
		{	// similar   }

		if (centbag == 1)
		{	// similar   }

	}


	public static class Stopwatch
	{	// as in Sedgewick  }

}		// End GameBoard class
```