Number of Non-Intersecting Rook Paths in a MxN Grid from M,0 to M,N

ASHIR BORAH
Dickinson College
Comp 332: Analysis of Algorithms
Professor Skalak

Dec 8, 2016

Abstract

Even with techniques of combinatorics, there is no known systematic way of calculating the number of non-intersecting rook paths in a MxN Grid from M,0 to M,N. The solutions for M=1,2 and 3 can be calculated with combinatorics but the number of paths for M=4 and higher does not seem to follow any formula. In this paper, we try to investigate the series and find the number of paths for M>3.

Introduction

This paper investigates the number of non-intersecting rook paths in a square grid of size MxN. The total number of paths in the rectangular lattice cannot be determined without any constraints as paths can indefinitely transition between the same set of nodes leading to infinite possibilities. Hence the number of paths have been examined with constraints.

When this problem is examined with the restriction that only upward or eastward moves are allowed from 0,0 to M,N, the total number of paths is $\binom{m+n}{m}$. This is because we have a total of M right moves and N up moves. To generate all the possibilities, we just need to choose which of the m+n moves are right moves or up moves. We get the two formulas $\binom{m+n}{m}$ and $\binom{m+n}{n}$ which are equal [Math 311 (Tesman), 2016].

When the restriction that the path can never exceed the diagonal x=y is imposed with the above restriction, we get the Catalan numbers. As the path cannot go over x=y, there are at least as many right moves as there are up moves in any given sub sequence which starts at move 1[Math 311 (Tesman), 2016].

But with no directional constraints are places, the possible number of paths from a given exhaustive paths grow exponentially and no systematic way to count them has been discovered yet. This paper explores various methods of calculating all self-avoiding walks from M,0 to M,N with visit every node only once and discusses the necessity for optimized algorithms for calculating the same.

Non-intersecting paths or Self Avoiding Paths (SAW) are defined as “a sequence of moves on a lattice (a lattice path) that does not visit the same point more than once” [Wikipedia].

The term rook path is derived from the game of Chess and is used in the paper to define the moves allowed from any lattice point to the other. Only horizontal and vertical moves are allowed with the restriction that the start and end lattice points need to be valid and there is not wrapping from one boundary to the other.

Background

The project started as a question which was proposed in our Algorithms class and we could not find the solution to the specific case of N=3. The program was implemented to generate the answers and then find a pattern and arrive at a solution.

When the code was used to generate solutions for higher numbers, the program turned out to be slow and higher efficiency algorithms were required. We came across the paper “ZDD-Based Computation of the Number of Paths in a Graph” which is believed to be the state of the art algorithm in calculating the paths on an NxN grid. It is based on an exercise from Donald Knuth’s book: The Art of Computer Programming and is called SIMPATH [Hiroaki, 2012].

But the number of paths is not available for rectangular paths (no results in Online Encyclopedia of Integer Sequences(OEIS) for M>10). This paper tries to compute the number for the general case (MXN grid).

____

Notation

A grid of size MxN has M*N points. In this paper, the lattice points are numbered in the follow manner:

2	5	8	11
1	4	7	10
0	3	6	9

Figure 1

This is a 3x4 grid. According to the problem, we always start at point 0 and need to go to point (N-1)M which in this case is 9. This number helps to make referring to points much easier than the 2-D coordinate system.

Methods and Experimentations

As we tried to code the ideas and generate the number of paths, it became evident that the number of evaluations needed to calculate the number of paths increases exponentially and optimizations are required to be able to calculate the result for higher numbers.

For testing, each pair of M,N was allotted 60 seconds on a Surface Book (2015) with Core i5 6300U having 8 GB of DDR3 RAM running Windows 10.

Detailed experiment results can be found in the appendix.

The Base code:

Two essential methods in the code are *generatePath()* and *validCells()*. The implementation and parameters of these functions change as the experiments progress but their functionality remain the same.

The function *validCells()* takes the current cell as parameter and returns the possible cells that can be the next lattice point in the path. It does not take into consideration which cells have been visited. So, if the input point is 4, the function would return 1,3,5,7 while input 0 would return 1 and 3.

The function *generatePath()* takes the input from *validCells()* and builds the path after checking that the cells function is trying to add has not already been visited. This is a recursive method and it calls itself on all the valid paths that were generated.

The validity of the results was tested by checking the number of paths generated with the sequence A271592 from OEIS which calculates the same result for M<10, N<10.

Experiment 1

The first method used to calculate the path is very similar to a brute force exhaustive search approach with the addition of a data structure to keep track of the node that have been visited in the current walk. Because of the number of paths and evaluations increase exponentially, this turned out to be very slow as numbers got larger. The paths were built using strings where the sequence of vertex that was visited in the current call was appended to the argument *Path* and then passed into the recursive function. Because Strings in Java are immutable, this program was highly inefficient. The same string was also used to check which matrix points have been visited. It was done using the contains method from the String library which is a O(n) operation.

Experiment 2

After a conversation with Professor Skalak[2016], the program was modified to use ArrayList to construct the sequence of paths. If the path was valid, the final solution was converted to a string and then stored into a ArrayList so that the actual path can be used to debug and check the validity of the solution. These are not actually required as we are concerned with the number of paths and not the actual paths themselves. This feature is disabled once confidence on the code is high to save space and computation time.

To check which lattice points have been visited, a HashSet (Java.util) is used to store the visited points. This allows us to have a constant time contains check instead of O(n). This implementation was significantly faster than experiment 1.

	Columns									
	1	2	3	4	5	6	7	8	9	10
Experiment 1	0	0.0043	0.0356	1.1041	1.369	2.43	1.579	0.495	1.903	10.059
Experiment 2	0	0.0012	0.0108	0.2993	3.0916	0.83	0.603	8.113	0.7462	4.5765

Table 1: Running time for Experiment 1 and Experiment 2 averaged over the total values that were calculated before Time Out

Experiment 3

After experiment 2, we started to improve the program by improving basic ideas to determine which paths were invalid without completely exploring them. As this is an SAW, there are certain paths that do not need to be fully explored to conclude that no valid path can result from them. We can see that a path like the follow will never have a solution:

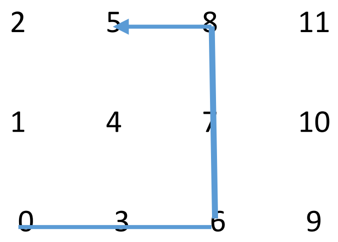

Figure 2

To minimize unnecessary searches, a function named *colGreaterFilled()* was created while took the column number as parameter and would return true if there was any column on the left than the input column which was completely filled. This condition should never be true and so search is terminated for that set of nodes and we start search other instances which might lead to a valid solution.

		Columns					
		1	2	3	4	5	6
Row	1	1	1	1	1	1	1
	2	1	1	1	1	1	1
	3	1	0.94737	0.93548	0.93333	0.93994	0.94789
	4	1	0.89796	0.85992	0.85714	0.87114	0.88748
	5	1	0.85965	0.79567	0.78749	0.80615	0.8282
	6	1	0.832	0.74634	0.72781	0.74813	0.77349
	7	1	0.81285	0.71035	0.6784	0.69782	
	8	1	0.8	0.68461	0.63805		
	9	1	0.79159	0.66631	0.60518		

Table 2: The ratio of number of evaluations needed to compute the same MxN grid (Experiment 3 vs Experiment 2)

Experiment 4

Because of the success of Experiment 3, the same idea was implement on the rows. The new method takes the row number as the input and checks if there are any rows below the specified row that are filled

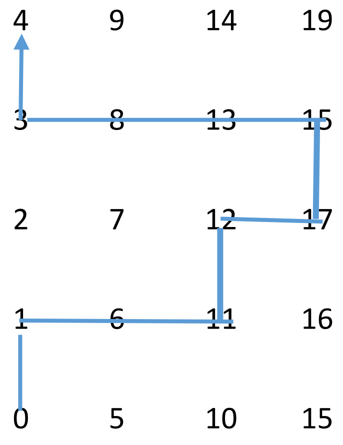

Figure 3

As we can see, no search at this point can lead to a valid solution. This implementation almost cut the search space by half for larger numbers of M and N.

		Columns					
		1	2	3	4	5	6
Row	1	1	1	0.83333	0.625	0.4375	0.29167
	2	1	0.94737	0.83871	0.72381	0.61732	0.52072
	3	1	0.89796	0.78988	0.70014	0.61969	0.54684
	4	1	0.85965	0.74303	0.66966	0.61451	0.56384
	5	1	0.832	0.7056	0.63699	0.59855	0.5646
	6	1	0.81285	0.67828	0.60796	0.58103	
	7	1	0.8	0.65904	0.58298		
	8	1	0.79159	0.64572	0.56183		

Table 3: The ratio of number of evaluations needed to compute the same MxN grid (Experiment 4 vs Experiment 2)

Experiment 5

During a discussion with Jon Evans(Dickinson'19), we came up with the idea that search can be cut off on the node that leads to the enclosed section. For example, in figure 3, the moment node 3 is reached, it can be concluded that no valid solution can be reached as there will be a section of nodes that can never be reached for this will following the definition of SAW.

When this idea was implemented for both rows and columns, it led to a large speed up because of reduction in the search space. The performance increased as the gird got larger.

		Columns					
		1	2	3	4	5	6
Row	1	1	1	0.83333	0.625	0.4375	0.29167
	2	1	0.94737	0.83871	0.72381	0.61732	0.52072
	3	1	0.83673	0.75097	0.6597	0.58881	0.52229
	4	1	0.69298	0.63674	0.56038	0.52801	0.4945
	5	1	0.544	0.51908	0.44776	0.4468	0.4409
	6	1	0.40832	0.41258	0.34635	0.36796	
	7	1	0.29498	0.32216	0.26167		
	8	1	0.20617	0.2484	0.19476		
	9						

Table 4: The ratio of number of evaluations needed to compute the same MxN grid (Experiment 5 vs Experiment 2)

Conclusion and Future Work

There are many real work applications for counting the number of SAW in a grid and advances in this problem would also lead to techniques for exploring graphs and reduction of search space that might be applied to different problems. More work is required to improve the algorithm discussed on this paper. A dynamic programming seems promising to prevent the same calculations but a memory efficient representation is required.

A topic of future study would be to explore the use of ZDDs as the paper that was referenced used and try to generate data for the more general case.

Bibliography

Self-avoiding walk. https://en.wikipedia.org/wiki/Self-avoiding_walk. Accessed on: 11-09-2016.

Hiroaki Iwashita, Jun Kawahara, and Shin-ichi Minato. 2012. ZDD-Based Computation of the Number of Paths in a Graph. (September, 2012). Retrieved November 2, 2016 from https://www-alg.ist.hokudai.ac.jp/~thomas/TCSTR/tcstr_12_60/tcstr_12_60.pdf

OEIS Sequence A271592. https://oeis.org/A271592. Accessed on: 11-09-2016.