CMSC 641, Sherman, Special Assignment 2
Shantanu Hirlekar, Alexander Spizler, Swati Verma
Online References: youtube.com (Algorithms: Memoization and Dynamic Programming), docs.python.org, docs.scipy.org, stackoverflow.com

Experimental Evaluation of Algorithms: The Number of Paths in a Matrix with Blocked Cells

The Problem and Algorithms:

The Number of Paths problem is to find the number of paths from the start of an m*n matrix, cell (0, 0), to end of the matrix, cell (m-1, n-1), with constraints on movement from each cell. From each cell one can either move right or down, and some of the cells are blocked. This is a common programming exercise often stated as the Robot Travel Problem, where a simple robot must navigate a grid given only two actuations, move left or move down. To compute the number of paths we have used dynamic programming to implement both iterative and recursive algorithms, shown below. In both methods, an m*n matrix, M, stores the total number of paths from M(i,j) to M(m-1, n-1). Additionally, the m*n matrix B contains whether or not each cell is blocked.

Pseudo code for iterative method:

```
NumPaths(B):
      For j from 0 to n-1: M(m-1,j) = 1
      For j from 0 to m-1: M(i,n-1) = 1
      For i from m-2 to 0:
            For j from n-2 to 0
                  Cost = 0
                  If B(i+1,j) == False
                        Cost +=M(i+1,j)
                  If B(i,j+1) == False
                        Cost += M(i,j+1)
                  M(i,j) = cost
      Return M(0,0)
```

Pseudo code for iterative method:

```
NumPaths(B,i,j):
      If i == m-1 and j == n-1: Return 1
      If B(i,j) == True or i >=m or i >= n: Return 0
      If M(i,j)>=0:
            Return M(i,j)
      Else
            M(i,j) = NumPaths(B,i+1,j)+NumPaths(B,i,j+1)
      Return M(i,j)
```

Methods:

We implemented both the iterative and recursive versions of the NumPaths algorithm in Python 2.7. All time and memory tests were run on a Dell Inspiron 7520 running Ubuntu 16.04 64-bit operating system with a Intel i7-3612QM CPU @ 2.10GHz and 8GB of RAM. We keep a fixed percentage of cells blocked for each run. For the experiments below, we block 10% of the matrix cells other than the start and end cells. In order to record the running time of one run of each method, we used Python's *time.time()* function before and after calling our NumPaths functions. For each value of n, where n is the total number of matrix cells, we ran 10 trials and recorded the average runtime. To calculate the total memory usage of each run of NumPaths, we used Python's *resource.getrusage()* function. Unlike recording runtime, we had to make sure we recorded total memory usage when the memory usage was at its maximum. For the iterative method, we only need one call at the end of the function while both block and cost matrices are still in memory. For the recursive method, we had to record memory usage at each recursion and maintain a global variable which held the maximum memory used. Like the runtime experiment, the average of 10 trials is recorded for each value of n.

Results:

n (matrix cells)	4	16	64	256	1024	4096	16384	65536
Time-Iter (s)	3E-05	0.0001	0.0005	0.0026	0.0038	0.0147	0.0575	0.244
Time-Rec (s)	2E-05	0.0001	0.0004	0.0023	0.0072	0.0194	0.0801	0.3159
Space-Iter (kB)	22192	22192	22192	22192	22192	22192	22192	22836
Space-Rec (kB)	21984	21984	21984	21984	21984	21984	22581	23727

Table 1: Time and space used by both implementations of NumPaths

The runtime experiments produced fairly similar results for both the iterative and recursive implementations. Both methods grew in linear time with the number of matrix cells, as seen in Figure 1, but at slightly different rates. The iterative method grew at a rate of

$$T_i(n) = 3.71E\text{-}6\ n - 1.53E\text{-}4,$$

while the recursive method grew at a rate of

$$T_r(n) = 4.81E\text{-}6\ n + 6.29E\text{-}4.$$

Ignoring the constant factor, the linear growth of the recursive method was almost 30% faster than the iterative method. For example, when the number of inputs was 65,536, the iterative method took 0.24 seconds while the recursive method took 0.32 seconds. For some of the smaller values of n, up to 256, the recursive method was usually slightly faster. However, this did not hold as n continued to increase. Additionally, the standard deviations are not shown since they are approximately one percent of the average times recorded for the iterative case, and approximately three percent for the recursive case.

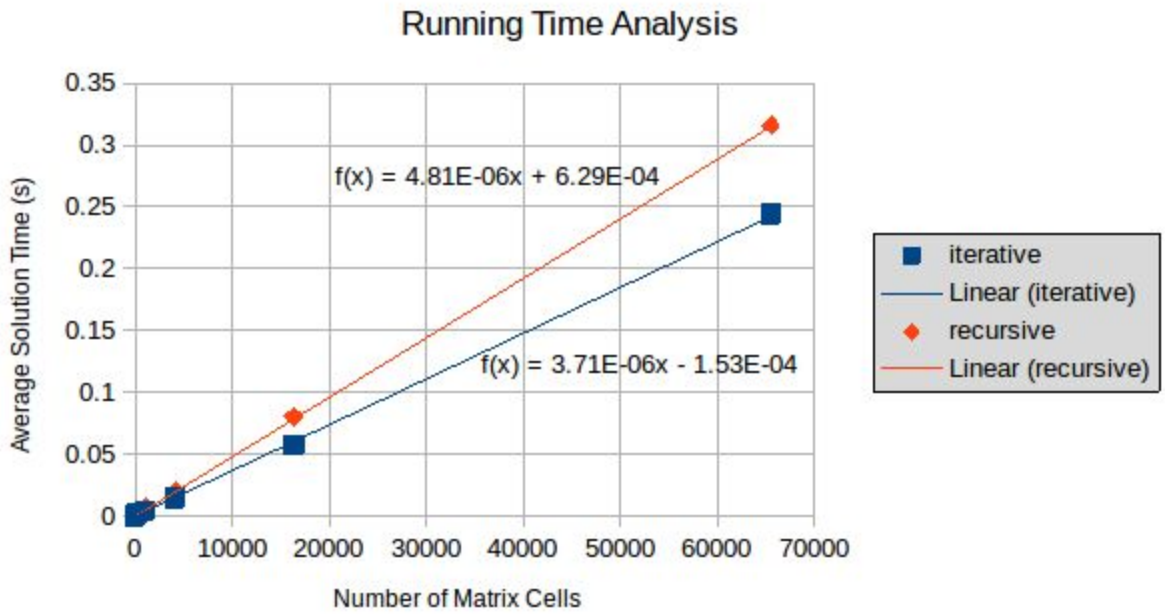

Figure 1: Running time of the iterative and recursive implementations of NumPaths

The method we used to record total memory usage is not precise enough on small inputs, as there appears to be a constant amount of memory allocated to programs up to a certain size. Once the number of matrix cells increased above 4096, increases in memory use are noticeable. In fact, the memory usage for the recursive method increases so much that we ran out of space for input matrices with 262,144 cells. For both iterative and recursive implementations, the minimum memory usage was a constant, at 22,192 kB and 21,984 kB respectively. Because we expect memory use to grow linearly in both cases, we fit a linear function to our data, which is shown in Figure 2. The iterative method grew as

$$S_i(n) = 9.70\text{E-}3\ n + 2.22\text{E}4,$$

while the recursive method grew as

$$S_r(n) = 2.71\text{E-}2\ n + 2.20\text{E}4.$$

We would have gathered more data for both methods, but the recursive implementation was unable to process matrices with 262,144 cells or larger, due to space limitations on the machine used. Again, the standard deviations are not shown because they are less than one percent of the average in both the iterative and recursive cases.

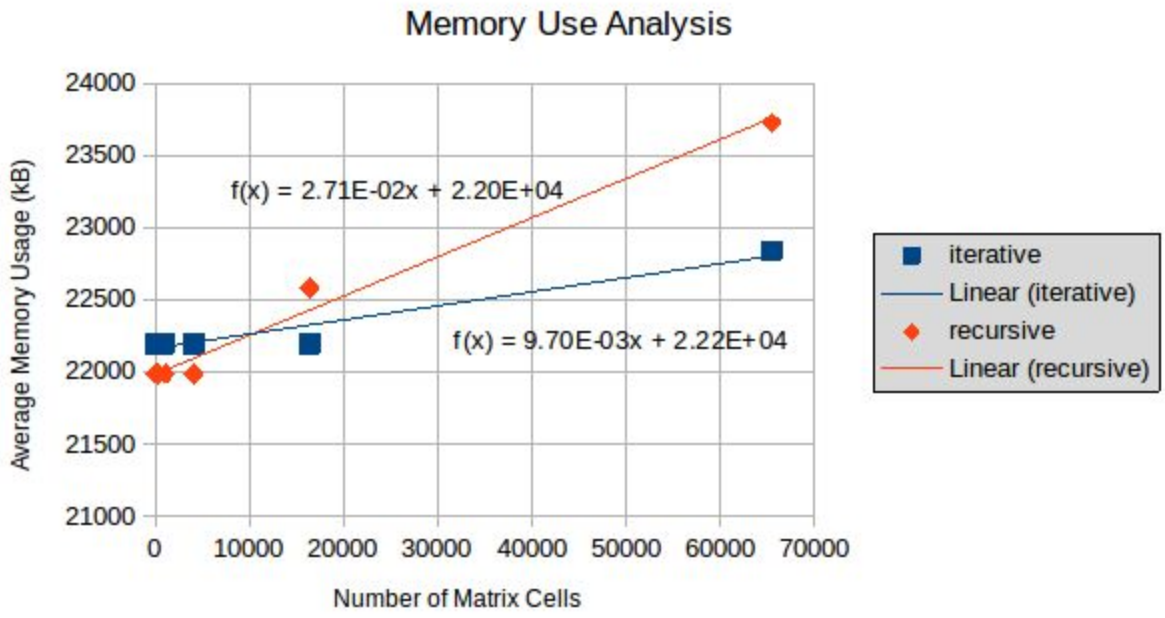

Figure 2: Memory usage of the iterative and recursive implementations of NumPaths

Discussion:

Our results show a clear distinction in runtime and space used by the iterative and recursive implementations of the Number Of Paths problem. As seen in Table 1, the recursive algorithm runs faster when n is small. As the value of n increases, the iterative method performs better. Similar results can be found when it comes to the space used by each implementation. For the first few initial values of n, the recursive algorithm performs better for the space usage. However, as n increases the iterative algorithm outperforms the recursive algorithm. The recursive algorithm also grows quickly enough to run out of space before the iterative method. The recursive implementation is slightly better for smaller datasets, However, when larger matrices are considered, the iterative method outperforms the recursive method. This is likely due to the growing number of stack frames for each recursive call. While this was not tested, it is likely that the recursive method will perform better when a large percentage of cells are blocked, since the recursive method will only explore cells possible paths, while the iterative method explores all cells.

The people who wish to use our algorithm should use the recursive approach in the case where the matrix is small. If the matrix is large, they should stick to the iterative method. Keeping this in mind, if one comes up with an efficient memoization solution, the recursive method can be preferred over the iterative one.