Aldous Broder: Sequential Implementation

Aldous Broder has a simple sequential implementation to form a uniform spanning true. This will serve as background to build off in the concurrent implementation section. The following is an overview of the algorithm. The algorithm first chooses a random vertex, or cell in this case, to be its starting point. From here it chooses a random direction, within the grid, to travel. If this cell hadn’t been visited yet, a path will be carved from the previous cell to this one. The random walk will occur until all cells has been visited.

Sequentially, this algorithm has an issue with the singular random walk missing cells. As the number of unvisited cells dwindle, the singular random walk can miss lone cells. This greatly affects the efficiency of the algorithm as the number of cells becomes less. The concurrent implementation of this algorithm seeks to remedy this flaw.

Aldous-Broder runs in time worse case, with n being the number of vertices (or cells) in the graph. tables below show the aggregated data of 300 runs with different Ns. As the N increases the execution time gets significantly worse. The great separation between the mins and maxes are also shown, with this separation showing the volatility of the algorithm. This is due to the random nature of the algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| (in ms) | N = 10 | N = 50 | N = 100 |
| Mean | 0.28 | 4.7 | 24 |
| STDEV | 0.15 | 1.6 | 7.05 |
| LOW | 0.096 | 2.23 | 13.13 |
| TOP | 0.9 | 12.51 | 52.40 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| (in ms) | N = 200 | N = 300 | N = 400 | N= 500 |
| Mean | 109.18 | 276.95 | 524.28 | 886.51 |
| STDEV | 30.37 | 68.91 | 122.83 | 195.58 |
| LOW | 61.28 | 167.36 | 330.34 | 561.55 |
| TOP | 282.94 | 588.85 | 1408.27 | 1768.06 |

Aldous Broder: Concurrent Implementation

The concurrent version of this algorithm will see many threads executing many random walks to cover the whole square grid. With more random walks within the grid, there’s an increased chance the lone cells will be covered as the number of remaining cells decreases. There’s a looming problem that this concurrent implementation presents, however. Contention between threads needs to be solved when two threads both visit a cell at the same time.

There are two cases of cell contention, one in which two threads move to a cell that’s already been visited and when both threads move to an unvisited cell at the same time. If a cell’s already been visited it’s okay for both threads to occupy the same space as no global data manipulation is occurring. When both threads visit an unvisited cell however, some contention may occur between threads as both try to access the cell and set the bit map to their desired value.

Each cell will have an AtomicBoolean flag that denotes whether its currently being manipulated or not. If a thread reads true on the flag it may enter it, set the flag to false, then manipulate it. If a thread reads false on the flag two options were theorized. Option one has the thread wait till the cell it wanted to enter is available. This approach implements some sort of lock. Option two sees the thread turned away. Once turned away it’ll be forced go into a new direction until it finds a cell it can enter. Locks are not an option here, option two is better with the reasoning explained below.

Option one sees potential deadlocking between two threads. If two threads want to move into each other’s square both will be waiting for the other to move. This results in both of them being frozen and unable to move, resulting in the algorithm stalling forever. Option two avoids this as the threads in contention will pick a different cell to move once the contention occurs. Each thread will be contributing to the universal counter of remaining cells to visit. Once that counter hits 0, all threads will end and the program will be complete.