The Effects of Parallelization on Common Maze Generation Algorithms

Peter Freedman, Gary Zoppetti
Millersville University of Pennsylvania
pwfreedm@millersville.edu, Gary.Zoppetti@millersville.edu

ABSTRACT

This paper covers implementation details for an effective benchmarking of the four following maze generation algorithms: Wilson's, Hunt-and-Kill, Kruskal's, and Cellular Automation in serial and parallel C++. The binaries will then be imported to Python, from which a maze building application and a benchmarking application will be built. Quantitative and qualitative test data will be gathered in hopes of defining the best maze generation algorithm from a computational standpoint, a standpoint of generating 'natural' mazes, and finding an intersection of the two to informally define a best algorithm to use for novice implementations.

KEY WORDS

Parallel, Maze, Pathfinding, C++, Python

1. Introduction

Delving through the records of human history, one begins to see labyrinths showing up in the Middle East as early as 3000 years ago. Despite appearing to be mazes, labyrinths were single paths; walking a labyrinth was an emotional and/or spiritual journey with outcomes being decided by the designer, then conveyed through the winding paths and ambiance of the structure itself. These labyrinths could house terrifying Minotaur, as the ones in Crete, or turn action into prayers of wellbeing as those in ancient Nordic cultures [1,3]. In the Middle Ages, labyrinths would be reinterpreted into the mazes we know today – amusing pastimes often carved out of the gardens of royalty for fun. This influence has stuck around, with most physical mazes being carved out of nature to this day [1,3].

Making the distinction of physical mazes, of course, implies that there is much more to the picture than meets the eye. Indeed, in the modern era, computers present a much bigger purpose for mazes. Fundamentally, mazes on computers are nothing more than a graph theory problem waiting to be solved, but the applications of solving that problem are far less finite. Pathfinding of non-player characters in video games, terrain and level generation in video games, navigation applications (such as Google Maps) can all be boiled down to mazes that need to either be created or solved [1]. One possible implementation for maze generation could be mapping out a room, which would in turn allow a robot to navigate it independently of human input. These applications exist among many others and have predicated a great deal of research into maze finding algorithms.

The purpose of this research, however, is not to push the boundaries of maze generation with a new cutting-edge technology or a revolution to the established system. Instead, this research will serve as a profile of existing algorithms. It will seek to generate comparative benchmarks, both in serial and parallel, of algorithms that take a variety of approaches to maze finding with a variety of different characteristics.

2. Background and Terminology

The process of completing this research includes algorithm selection, implementation, verification, and benchmarking twice apiece: once for a set of five serial algorithms and again for the same set of algorithms in parallel. In the case of the parallel implementations, all algorithms may not be suited for parallelization. Those which are deemed unfit for parallelization but fit to include in the test suite will, instead of parallel benchmarking times, have some combination of past research, mathematical proofs, and data forming an argument against their parallelization.

To best express these algorithms, some terminology must be laid out first. The cover time($\bar{G}$) is the time it takes for a random walk to hit every vertex in a graph [2]. From this, it follows that the hitting time, denoted as either average(τ) or maximum(h) is the expected time to hit every vertex in the graph; both have hard limits at $\bar{G}$ and, per [2] never reach $\bar{G}$.

2.1 Program Specifications

To build a test suite for any application, as many pieces as possible need to be held constant between trials. As such, the interface through which all algorithms will be run has been set. A Maze will be defined as a one-dimensional vector of cells, alongside unsigned integers for length and width and a bool to make sure that the maze has an end (this is currently being used primarily as a sanity check on algorithm implementations and may be removed later if it is deemed unnecessary). A cell is a half byte in which each bit corresponds to each cardinal direction. Any bit set to 0 represents a direction in which the maze cannot be traversed, while a bit set to 1 means that one could leave

the current cell in that direction. Given this information, the maze itself can be deemed to be relatively memory efficient, taking up a total of (N x M) / 2 + 33 bytes total, where N and M are the length and width of the maze, respectively and in cells. The additional 33 bytes come from the following: 24 for the vector that stores the maze, 8 for two integers, the length and width of the maze, and one for a bool that gets padded out to a byte for the sake of proper alignment.

The algorithms, once implemented serially, will be tested by first converting their output into a format that is accepted by an existing maze solver[11]. To ensure that the algorithm generates consistently solvable mazes, this process will be repeated 1,000 times, varying the size of the maze every 10 runs using a seeded random generation in the range [10,11]. Once all algorithms have been tested and implemented serially, this process will be repeated – with each previously implemented and tested algorithm being reimplemented in parallel. The technologies used for parallelization will vary based on what either makes the most sense or has the most straightforward implementation (with a strong bias placed on the most sensible option, only deviating where a certain technology adds unreasonable complexity). Due to the performance sensitive nature of benchmarking algorithms, all implementation to this point will be done in C++. The parallel implementations will be tested with the thread sanitizer active as well. Once all methods are implemented and tested, classes encapsulating algorithm behavior will be created and exported as binaries for use in a Python front end.

There will be two different Python front ends built, each to serve a different purpose. The first will be a benchmarking application, whose purposes will be discussed in §2.3. The second will be a basic GUI allowing users to generate a maze, save it to a PDF, and (time permitting) solve it with one of the algorithms mentioned in §2.2, three of which can be easily modified to solve mazes instead of generating them.

2.2 Algorithm Selection

The first algorithm selected is Wilson's Algorithm. Not only is it a more recent addition to the world of maze generation algorithms, but it can also be easily adapted to solve mazes and has applications outside the maze generation world. It is described as "easy to code up, with small running time constants" [2]. Wison guarantees that this method of generating random spanning trees will always be faster, often by a factor of at least two, than the standard algorithm that generates random spanning trees in maze creation. The algorithm chooses a starting point and destination, then attempts to connect them, partially restarting itself if it creates a loop, then once it has found a path adds that to the existing tree and picks a new point to start from, connecting that point to the existing branch until all possible connections are made. The algorithm also promises to make evenly distributed mazes [2].

Wilson's Algorithm generates its random spanning trees in $O(\tau)$ time on undirected graphs, with worst cases being at $O(h)$ time for Eulerian graphs and something between the two as the general case time complexity for any graph [2]. Given the size of an N x M maze, per §2.1, as S_M, this algorithm should take $2S_M$ space in memory. The memory requirement for this algorithm is twice that of the size of the maze being generated because it needs to maintain a visited and unvisited list, each of which are separate from the maze itself but effectively share a copy of the maze, moving cells from one list to the other as they are visited and walked back into the maze.

Next on the list is the Hunt-and-Kill (HK) Algorithm. This algorithm is, in essence, a primitive recursive backtracker [5]. A random starting cell is chosen, then a random walk is performed from it. A current cell is tracked, choosing a random unvisited neighbor until it reaches a point at which no neighbors are unvisited. Once this dead end is reached, instead of backtracking to look for an empty cell (as a recursive backtracker would), it simply starts iterating from one corner of the maze in search of an unvisited cell with a visited neighbor to start from [5]. Upon finding such a cell, another random walk is started from that cell, and the process repeats until the iteration reaches the last cell. Unlike Wilson's, this algorithm makes no promises about loops and tends to generate longer straight segments, resulting in mazes with less dead ends overall [5].

HK has much less formal documentation than Wilson's, making it rather difficult to pull precise numbers for time complexity and memory usage. The algorithm does share some characteristics with existing algorithms, namely the traditional recursive backtracker mentioned earlier. This method of generating mazes is nothing more than a randomized depth first search (DFS) that expands all possibilities instead of stopping when it finds one path [6,7]. As a result of these properties, HK's time complexity must be at least O(V x E) – the time complexity of DFS, with V representing the number of vertices and E the number of edges in a given graph [6]. Given this implementation of maze generation is not using vertices and edges but instead cells, and every cell must be visited, the substitution O(N x M). This can then be translated to, in the absolute best case (a square maze) O(N x N) or $O(N^2)$ runtime. Keeping in mind that this is a best-case prediction made based on a traditional depth first search instead of a randomized one, however, the time complexity will, on average, be significantly worse than this. The memory complexity is looking good, however. Given the fact that there is no need to verify whether loops exist, a full second copy of the maze does not need to exist. Instead, all that should be needed are two references – one to the current cell and one to the first row that has an empty cell, to be used as a starting point for the hunt for a new starting point when the current recursive walk expires. This makes the total memory complexity of the algorithm $S_m + 16$.

Third, we will explore Kruskal's Algorithm. At the start of this algorithm, each cell is treated as a member of its own set. The walls between two cells are then picked at random. If the cells adjacent to the walls are not in the same set, they become members of the same set and the walls between them are destroyed [8]. This process is then repeated until every cell in the maze is part of a single set. This algorithm also ends up creating a spanning tree [8]. Notably, unlike the other algorithms on this list Kruskal's, even when randomized, biases towards making mazes with many short paths that are always fully connected [9]. These traits are reported to make mazes generated with Kruskal's easier to complete.

The last algorithm being implemented is not actually an algorithm – more of an idea. Cellular Automata have been employed for many different purposes, notably Conway's Game of Life. The idea is simple – create a state-based machine that decides whether a cell is disconnected, is seeded for connection, or is connected. These states, in order, are equivalent to being dead, born, and alive in Conway's Game of Life; the key difference being that connected cells cannot die, they just also do not expand any more [10]. The implementation will keep a bit vector representing the maze with each state initially set to 0. One cell will be seeded (its state will be changed to 1). All seeded cells will check their neighbors and pick one unvisited neighbor to seed based on a percentage chance to turn. The previously seeded cell will now turn to state 2, regardless of the state of the newly seeded cell. At this point, the walls between these two cells in the actual maze will be removed (connecting the maze cells). This process will start over in the newly seeded neighbor. Based on a hardcoded probability, the cell in state 2 will either become a seed again or move to state 3, connected. State 3 cells are essentially finalized, however as a failsafe if all seeds die all state 3 cells will have a chance to return to state 1 based on the same external probability which governs the change from state 2 [10]. Note that the higher this branching probability is, the more branches the maze will have, and thus the more difficult it will likely feel to solve. The mazes resulting from cellular automation can be very easily tuned. Changing either the probability to reseed (for a stage 2 to return to stage 1) and the chance for the cell to choose to turn both impact the maze structure, with too much or too little homogenizing the overall structure of the maze, as Justin Parr demonstrated [10].

The time complexity of this implementation heavily depends on the values of the two constants defined in the previous section. Parr notes two cases as opposite extremes: 0% chance to branch with no collisions (best case) and 100% chance to branch started in a corner. The former yields a time complexity of $2N^2 - 1$, where the latter reaches $4N^2$ in the case of a square maze, substituting N for N x M for any non-square maze [10]. Experimental data compiled in Parr's research shows that the worst observed case was closer to $1.1N^2$, with the best case being an impressive 4N [10]. The memory requirement for this implementation should be exactly $1.5S_m + 8$ bytes of memory. This number can be quickly derived from the fact that the maze is needed alongside capacity to store four additional states for each cell, which can be a bit vector in which each element is two bits. This is half the size of the original maze exactly, then the 8 extra bytes of memory are the two constants used, stored as integers.

2.3 Benchmarking Process

Before mentioning the actual data that is being gathered, it is important to attempt to control the environment the data is being gathered on. As such, all data for this experiment will be gathered on a CLI only install of Arch Linux on a machine with no other processes running. Specifics about the machine's memory and CPU will be gathered and disclosed prior to data presentation.

The quantitative benchmarks gathered across the various algorithms will begin with seeding the random number generator that picks the starting position in each maze. This should not result in the same mazes being generated by each algorithm. It will result in each of them having the same starting point; see the time complexity of solving mazes with cellular automata in §2.2 for a detailed explanation of the importance of starting position.

With the random generation held constant, trial specifics can begin to be outlined. Each algorithm will be tested with a set of 10,000 mazes generated, with the first starting size in each case being a 25 x 25 cell maze. The first 5,000 mazes will see both length and width increase by 25 every 100 runs until all 5000 mazes are complete. This will result in the last 100 mazes requiring that 1,562,500 cells be solved. Once the 5000 mark is reached, the size will reset to 25x25, but now the length will increase by 25 while the width increases by 35 between each 100 runs. This will ensure that all methods are benchmarked on both square and rectangular mazes. If time permits, additional testing may be done with more extreme rectangles to see if that biases the data any.

The data recorded for each trial will be memory usage, time to complete, and maze size. Python's data analysis tools will be used to aggregate the data gathered based on maze size. For both time and memory usage at a given maze size, the minimum, median, maximum, and mean will be recorded. In the case of the serial implementations, the graphs built from this data will be compared to the graphs of the time and memory complexities calculated in §2.2 to prove or disprove the estimates made there. The data for the parallel implementation of a given algorithm will be compared to the data for the serial implementation to calculate speedups, with standard deviations and min/max times being used as general measurements of the consistency of the algorithm's performance. Bandwidth data will also be gathered and, if either memory or cache become bottlenecks based on the calculated values, those will be released alongside theoretical times with the

bottleneck removed. Parallel implementations will be compared to theoretical perfect speedups, from which overhead costs will be extrapolated, and commentary will be given on the point at which it is worth switching to the parallel implementations provided, if that point is reachable.

Qualitative testing will be a lot less strictly formed. Two sample groups will be collected. The first will be given a set of only mazes generated by cellular automata, each at different calibrations of the two constants discussed in §2.2 and asked to rank how natural the mazes look. Natural will be defined as how willing you would be to believe they were made by a human on a scale of 1-10. From there, results will be tallied, and that data will be used to calibrate the cellular automata mazes for the second group, who will now be ranking the various implementations – both serial and parallel implementations – based on the same criteria presented to the cellular automata group. This data will be presented as an informal answer to the question of which algorithm generates the 'best' mazes, though should by no means be taken as an absolute answer to the question because it is a matter of opinion that has been very informally polled.

3. Foreseeable Problems

The problems that are easily spotted going into this can be split into two main categories: implementation problems and testing problems. The first of these groups arises from the fact that in our searching we found no record of all these algorithms being implemented in parallel. While we have loose ideas of the ways that they could be done, there is no guarantee going in that they will all be parallelizable in a way that makes mathematical and logical sense. At a bare minimum, they should all be parallelizable by chopping up the maze and having different threads each run the serial method on different parts of the maze, however the synchronization costs to facilitate that would likely take a large enough chunk out of the observable speedup that it is not worth doing unless the mazes are massive. If the parallel implementations are slower or near the same speed at the bounds of the current testing, some tests may be redone on mazes orders of magnitude larger to give parallelism a chance to shine.

The second major category of problems that could be encountered are testing problems. Relying on a human sample size, beyond introducing the uncontrollable human element, relies on humans to show up to the tests. Asking this once is hard enough, but twice has a huge risk of not generating enough participants to gather interesting data. We are debating dropping the first test and subjecting my thesis advisors and panel to being informal test dummies for the calibration of the cellular automata as a partial solution, but unfortunately, we do not have a contingency plan if not enough people show up – the qualitative tests will just need to be cut from the final results.

4. References

[1] CMU Class Notes, https://www.cs.cmu.edu/~112/notes/student-tp-guides/Mazes.pdf (accessed Feb. 22, 2024).

[2] B. Wilson, "Generating Random Spanning Trees More Quickly Than Cover Time," CMU University Website, https://www.cs.cmu.edu/~15859n/RelatedWork/RandomTrees-Wilson.pdf (accessed Feb. 22, 2024).

[3] S. Magazine, "The winding history of the maze," Smithsonian.com, https://www.smithsonianmag.com/travel/winding-history-maze-180951998/ (accessed Feb. 22, 2024).

[4] "Markov Chains," Markov chains, https://nlp.stanford.edu/IR-book/html/htmledition/markov-chains-1.html (accessed Feb. 22, 2024).

[5] J. Buck, The Buckblog, https://weblog.jamisbuck.org/2011/1/24/maze-generation-hunt-and-kill-algorithm (accessed Feb. 22, 2024).

[6] S. H. Shah, J. M. Mohite, A. G. Musale, and J. L. Borade, "Survey Paper on Maze Generation Algorithms for Puzzle Solving Games," *International Journal of Scientific & Engineering Research*, vol. 08, no. 02, Feb. 2017. doi:10.14299/ijser.2017.02

[7] L. Peachey, Parameterized maze generation algorithm for specific ..., https://portfolios.cs.earlham.edu/wp-content/uploads/2022/05/Parameterized-Maze-Generation-Algorithm-for-Specific-Difficulty-Maze-Generation.pdf (accessed Feb. 22, 2024).

[8] P. H. Kim, Intelligent maze generation, https://etd.ohiolink.edu/acprod/odb_etd/ws/send_file/send?accession=osu1563286393237089&disposition=inline (accessed Feb. 22, 2024).

[9] L. Williams, "Kruskal algorithm Maze Generation," Kruskal Maze Generating, https://www.integral-domain.org/lwilliams/Applets/algorithms/kruskalmaze.php (accessed Feb. 22, 2024).

[10] Generating mazes using cellular automata | Justin A. Parr, http://justinparrtech.com/JustinParr-Tech/wp-content/uploads/Creating Mazes Using Cellular Automata_v2.pdf (accessed Feb. 22, 2024).

[11] J. D'Silva, "A generic C++ implementation of a maze data structure along with maze solving algorithms using graphs.," GitHub, https://github.com/JeremyDsilva/MazeSolver (accessed Feb. 22, 2024).