Analysis of Maze Generating Algorithms

Gabrovšek, Peter

Abstract: ***In this paper our main goal is to rank different maze generating algorithms according to the difficulty of the generated mazes. Following our main goal we implement and analyse six algorithms. For the purpose of evaluating and ranking maze generating algorithms we devise four agents which solve mazes and report the results. To assess the level of difficulty of a maze we inspect several features such as number of visited intersections, dead ends, and overall steps of the agents. According to agents performances we rank maze generating algorithms. The best performing algorithms are derived from algorithms for finding uniform spanning trees in graphs.***

Index Terms: ***assessment***, ***difficulty***, ***generator***, ***labyrinth***, ***maze***.

1. Introduction

Mazes are closely related to labyrinths which have been known since ancient times. Usually they were build with naturally occurring materials. Originally they have had spiritual connotation [8]. Their later purpose was mainly amusement. In the modern times mazes became intriguing for scientists, especially for mathematicians.

Mazes are used in various fields. Besides entertainment purposes, mazes are also used in psychology studies [9, 13] of human and animal behaviour to determine space awareness and also intelligence. Among others, mazes can be used in physics, for example in study of crystal structures [4].

There are many maze generating algorithms [2, 7] but hardly anyone considered checking their level of difficulty and complexity.

We describe and analyse six maze generating algorithms. Our algorithms are originally used in graph theory [14, 15]. By using specialisation method [3] we implemented them to generate mazes. We generate mazes which are represented as trees only. Our focus is on 2D and planar mazes that do not contain overpasses.

We devise four different agents that walk across the mazes to help us analyse the difficulty of each maze type. They tell us the number of steps they make from the beginning to the end as well as the number of visited intersections and dead ends.

Finally, we analyse the relation between properties of mazes and attributes that agents provide us with. Final goal is to determine which algorithms are giving us the most difficult mazes.

In entertainment as well as psychology the levels of difficulty can be used for generating mazes for different user groups.

In the next section we describe six maze generating algorithms. In section 3 we establish the maze solving agents and their properties. In section 4 we analyse mazes and present the results. In section 5 we conclude the paper.

2. Generating Mazes

Let us first explain the data structure that we are using to represent maze. We start with the square[1] grid graph. Initially, all the edges (connections) represent walls. The algorithms convert specific walls into passages. After the algorithms do their job, the subgraph made out of passages represent a tree-like structured maze. In general mazes can contain loops, overpasses, etc., but we focused on simple ones.

Manuscript received Dec, 2017.

The author is with the Faculty of Computer and Information Science, University of Ljubljana, Slovenia (e-mail: peter.gabrovsek@fri.uni-lj.si).

[1] All analysed mazes that are square, but for the space saving and aesthetic reasons figures show rectangular mazes.

Every maze has properties which we use in the analysis, such as size, number of intersections, number of branches, average branch length and length of the dead ends (branches that do not split further).

2.1 Recursive Backtracking (RB)

Firstly, we implement depth-first search (DFS) algorithm also called backtracking [12]. Main idea of DFS is to go forward as much as possible, then backtrack to the first branch that has unvisited paths and repeat until everything is searched. We use randomised DFS to obtain random (non-trivial) mazes. Basic implementation of DFS uses recursion. However, we implemented it using an explicit stack to avoid stack overflow error caused by large mazes having long paths.

To generate the maze we mark the starting vertex v as visited, push it on the stack and then repeat the following until the stack is not empty:

1. Pop the vertex v from the stack.
2. Choose random unvisited neighbour u of v.
3. Visit u.
4. Connect u and v.
5. Push u on the stack.

The maze generated with RB is shown in Figure 1.

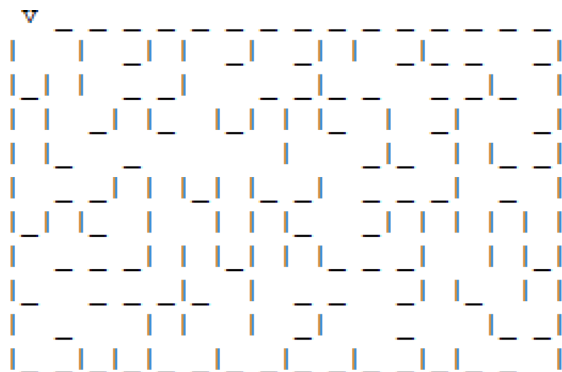

Figure 1: A maze generated with Recursive backtracking algorithm.

2.2 Aldous-Broder Algorithm (AB)

Aldous-Broder [14] uses random walk until all vertices are visited. During the walk, suitable connections between the vertices are created (under certain criteria). This algorithm is originally used to find uniform spanning tree [1] in the graph.

To implement AB we mark starting vertex v as visited and repeat the following steps:

1. Choose a random neighbour u of v (not necessarily unvisited).
2. If u is not visited visit it, and connect it with v.
3. Set v as u.

The Aldous-Broder algorithm is very simple to implement. The simplicity contributes to its relatively high running-time performance, at least for the sizes of the mazes in our analysis.

The maze generated with AB is shown in Figure 2.

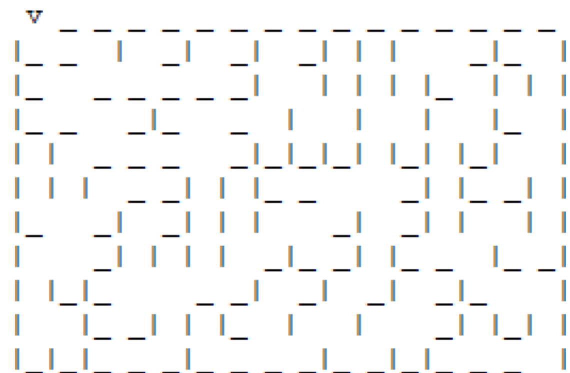

Figure 2: A maze generated with Aldous-Broder algorithm.

2.3 Wilson's Algorithm (W)

Wilson's algorithm is originally used to find a uniform spanning tree in the graph [14]. The algorithm is very similar to Aldous-Broder with slightly better asymptotic time-complexity in theory. In practice Wilson turns out to be slower because it uses dictionaries unlike the Aldous-Broder, which uses only arrays.

When implementing W, we firstly initialise dictionary d, choose random vertex v_0, visit it, and repeat the following steps:

1. Choose random unvisited vertex w
2. Repeat until we find a visited vertex: Choose a random neighbour u of w and put the following entry in the dictionary: $d[w] = u$. Remember u as w ($w = u$). NOTE: Do not visit any vertex yet!
3. With the help of d, we visit and connect vertices from w to v_0 (repeat: visit w, connect w and $d[w]$, $w = d[w]$).

The last step of the procedure may look confusing, but the dictionary d automatically takes care of finding the path from w to v_0.

The maze generated with W is shown in Figure 3.

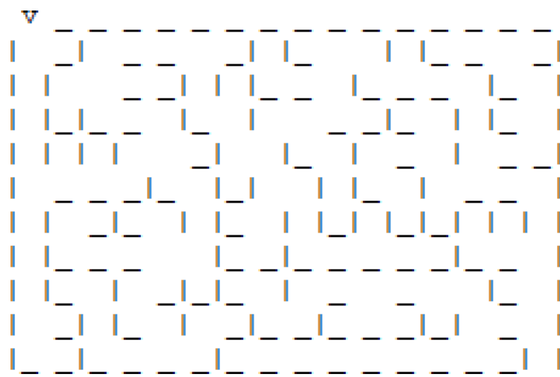

Figure 3: A maze generated with Wilson's algorithm.

2.4 Prim's Algorithm (P)

Prim's algorithm is derived from randomised breadth-first search (BFS). The basic algorithm is used to search for a minimum spanning tree in a graph [5]. This algorithm creates a lot of short dead ends, which leads to high miss rate of agents but not for humans.

To implement P we first initialise the set f (frontier), add a random vertex v from f, and mark v as visited. While f is not empty, repeat:

1. Choose random vertex v from f and remove it from f.
2. Connect v with random visited neighbour.
3. Add unvisited neighbours of v to f.

The maze generated with P is shown in Figure 4.

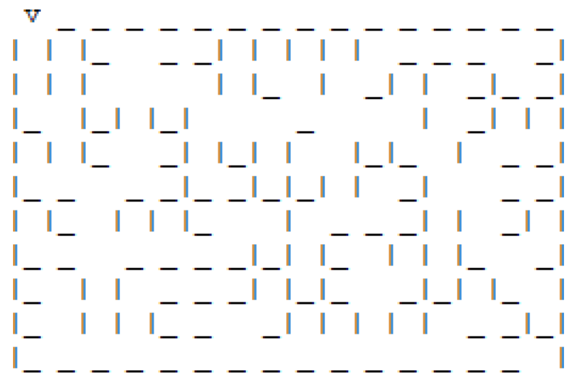

Figure 4: A maze generated with Prim's algorithm.

2.5 Hunt and Kill (HK)

Hunt and Kill algorithm uses the idea of the recursive backtrack but it starts from a random unvisited vertex whenever hits the dead end. It does not backtrack to the last vertex with unvisited neighbours.

To implement HK we choose random vertex v, mark it as visited, and repeat the following steps until all vertices are visited:

1. If v has unvisited neighbours, choose one (u), visit it and connect v and u.
2. Find and visit unvisited vertex v which has a visited neighbour u, connect u and v, and remember u as v ($v = u$).

The maze generated with HK is shown in Figure 5.

2.6 Kruskal's Algorithm (K)

The Kruskal's algorithm is originally used to find minimum spanning tree in a graph [15]. This is the most complex algorithm to implement out of all considered algorithms in this paper thus it leads to high time complexity. With algorithm optimisation and tuning techniques

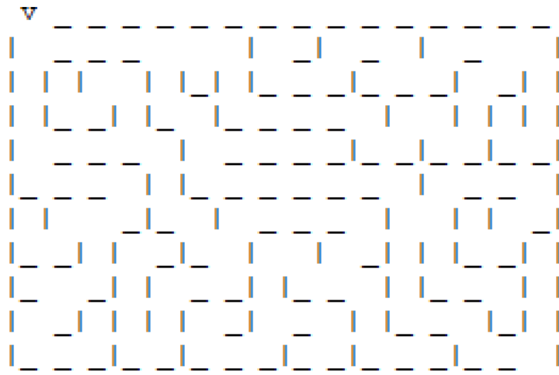

Figure 5: A maze generated with Hunt and Kill algorithm.

we could improve the time-performance of this algorithm.

To implement K we set unique label to every vertex. We get a set E of all possible connections (edges) among neighbouring vertices. Repeat until E is not empty:

1. Choose random edge e and remove it from E.
2. If vertices $u, v \in e$ have different labels (l_u, l_v), connect u and v, and give label l_u to all vertices with label l_v.

The maze generated with K is shown in Figure 6.

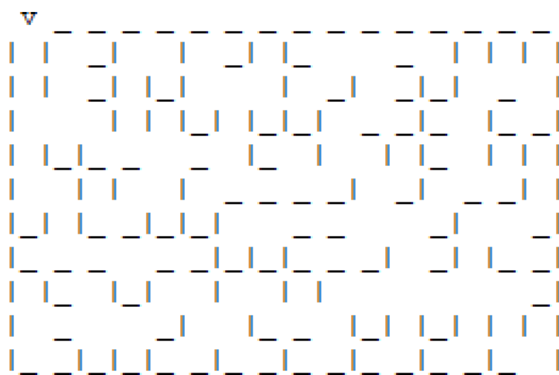

Figure 6: A maze generated with Kruskal's algorithm.

3. Solving Agents

Agents help us understand how difficult a particular maze is to solve. Agents produce various attributes of mazes with which we will later analyse the difficulty of the mazes. These attributes are:

- number of steps needed from the beginning to the end of the maze,
- number of visited cells,
- number of visited intersections,
- number of visited dead ends.

3.1 Random Walk (RW) Agent

The agent walks randomly from a vertex to its random neighbour until it gets to the end of the maze.

In particular, when located in a node, an agent selects a neighbouring node uniformly at random and moves into it. It repeats this procedure until it finds the end.

3.2 Depth First Search (DFS) Agent

This agent walks as far as it can until it hits a dead end. The agent then backtracks to the first node with unvisited neighbours. It keeps repeating the walk, until it comes to the end of the maze. The precedence of agent's turns at intersections are manually predefined: east, south, west, north.

3.3 Heuristic Depth First Search (HDFS) Agent

Similar to the DFS agent, but selects the preferred directions with a simple heuristic. In particular neighbours with lower Manhattan distance to the end of the maze are preferred.

3.4 Breadth First Search (BFS) Agent

This agent uses the idea of BFS [6] to solve the maze but instead of the queue to visit the nodes the agent first visits the nodes closest to the end of the maze. This agent resembles a human solver which can freely jump from one path to another (at least when solving printed mazes).

4. Analysis and Results

In this section we analyse the difficulty of the mazes constructed by the above six generating algorithms. We analyse properties of mazes and results of solving agents to determine difficulty of mazes. For the sake of completeness we also experimentally analyse the time performance of algorithms.

4.1 Maze Generating Time

We did not make a formal analysis of time complexity of algorithms but an experimental one. Our goal was not the perfect implementation of algorithms but the quality of the results that they produce. Hence the lack of focus on the analysis of time complexity.

We analyse the execution time of maze generating algorithms with respect to the maze size. We did not bother to consider the number of nodes of the maze which would be a true measure. We used only the length of the side of the square grid graph because we are interested in relations among performances of different algorithms. The result is shown in Figure 7.

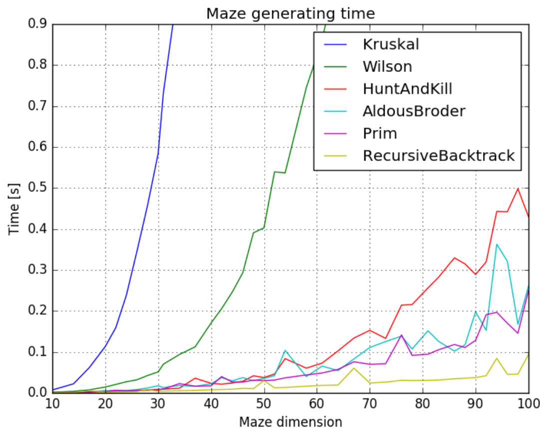

Figure 7: Running time of the maze generating algorithms with respect to the maze size.

Some algorithms stand out performance-wise. They are either exceptionally slower or faster than we would expect:

- **Kruskal** is the slowest.
- **Wilson** is very naive with random behaviour which makes it slow, again.
- On the other hand we have **Aldous-Broder** algorithm which is surprisingly fast. It is simple and uses primitives instead of higher data structures.

4.2 Maze Properties

To analyse the difficulty of a maze, we consider the following properties:

- size s,
- number of intersections n_i; intersections are vertices with more than two neighbours,
- number of dead ends n_{de}; dead ends are vertices with only one neighbour.

The bigger n_i the more difficult is the maze. The same goes for dead ends.

We generated 1000 mazes (of size 100×100) of each type, calculated the number of intersections and dead ends. The average results are shown in Table 1.

algorithm	n_i	n_{de}	rank
Prim	2946	3559	1
Kruskal	2654	3058	2
Aldous-Broder	2577	2933	3
Wilson	2576	2932	4
Hunt and Kill	920	939	5
Recursive Backtrack	869	898	6

Table 1: Average number of intersections and dead ends of the mazes.

Table 1 indicates that certain mazes generated with similar algorithms behave similarly. In particular Aldous-Broder and Wilson have practically the same number of intersections and dead ends. They both originate from algorithms for finding uniform spanning tree.

Hunt and Kill and Recursive backtrack also stand out, they have notably smaller values of n_i and n_{de} than other mazes. These algorithms originate from DFS.

Another pair of algorithms is Kruskal and Prim. They originate from minimum spanning tree algorithms but do not have such distinct property values as other two groups.

Larger number of intersections and dead ends means more difficult maze one can deviate

from a correct path easily. Prim and Kruskal perform best in this case.

4.3 Agent Performance

Maze solving agents give us another set of maze properties:

- number of steps s that agent needed to get from the beginning to the end,
- number of visited intersections i_v,
- number of visited dead ends de_v.

4.3.1 Number of Steps

The basic measure is the number of steps an agent makes from the start to the end. A step is defined as a transition from a node to the adjacent node.

	RW	DFS	HDFS	BFS	rank
HK	7.3M	5.3k	13.3k	3.5k	1
AB	4.3M	7.1k	12.6k	3.1k	2
W	4.3M	7.2k	12.4k	3.0k	3
RB	9.1M	2.2k	14.7k	2.7k	4
K	4.0M	6.8k	12.5k	2.8k	5
P	2.3M	5.6k	15.4k	1.7k	6

Table 2: Average number of steps needed from the start to the end.

We ranked maze generating algorithms according to the performances of the maze solving agents. To rank maze generating algorithms we devised a simple method. For every algorithm i we calculated score s_i:

$$s_i = \sum_{A \in agents} \frac{A_i}{\max(A)}$$

where A_i represents value of score of an agent A for algorithm i, and $max(A)$ represents the maximum value that the agent scored among all generating algorithms. According to the score s_i we ranked generating algorithms. Algorithm with the biggest (best) score has rank 1, etc.

According to this scoring Hunt and Kill performs best, as listed in Table 2. It is interesting that Recursive Backtrack performed the worst although it is similar to Hunt and Kill.

4.3.2 Visited Intersections

Next, we analyse how many intersections agents visit. The more intersections that an agent visits the better chance to miss the right path. Hence, the generating algorithm is more difficult.

	RW	DFS	HDFS	BFS	rank
AB	1.8M	2.9k	5.1k	850	1
W	1.7M	2.9k	5.0k	844	2
K	1.7M	2.9k	5.2k	800	3
P	1.1M	2.7k	7.3k	567	4
HK	1.0M	750	1.8k	337	5
RB	1.2M	232	2.0k	211	6

Table 3: Average number of visited intersections of each agent.

We used the same ranking technique as in section 4.3.1. Unlike the number of steps, here Hunt and Kill performs badly. The best performing algorithms are Aldous-Broder and Wilson, which original idea is finding uniform spanning trees in graphs. They are followed by Kruskal and Prim, etc.

4.3.3 Visited Dead Ends

The last property that we analyse is the number of dead ends that agents visit on average.

	RW	DFS	HDFS	BFS	rank
AB	0.6M	1023	1863	823	1
W	0.6M	1035	1840	816	2
K	0.6M	1030	1927	769	3
P	0.4M	974	2759	535	4
RB	0.4M	75	681	190	5
HK	0.4M	249	626	307	6

Table 4: Average number of visited dead ends of each agent.

The ranking in Table 4 is roughly similar to ranking in Table 3.

5. Discussion

Our goal was to rank the maze generating algorithms from those that generate the most difficult mazes to those that generate the least difficult mazes. We did that with the help of sev-

eral criteria: maze properties and solving agents performances.

We ranked the algorithms according to the criteria. The final ranking of difficulty level: For every measure we ranked algorithms. Finally we calculated average of all the ranks which gives us the final order in Table 5.

rank	algorithm
1	Aldous-Broder
2	Wilson
3	Kruskal
4	Prim
5	Hunt and Kill
6	Recursive Backtracking

Table 5: Ranking of algorithms by the level of the difficulty.

Having established the ranking of the algorithms we can now find the properties that distinguish among the various levels of difficulty of the algorithms.

The number of intersections is correlated to the difficulty of mazes. More intersections means that the maze is more difficult, and that there is more chance to miss the correct path.

The type of the algorithm contributes to the level of the difficulty:

- Best results are achieved by Aldous-Broder and Wilson. They originate from algorithms for finding uniform spanning trees in graphs. We speculate that agents have difficulty navigating through the maze because the paths are unbiased in any direction.
- Next pair by ranking are Kruskal and Prim. They originate from algorithms for finding minimum spanning trees in graphs. In comparison to the first group the paths here are not so evenly distributed which makes the mazes less difficult.
- The worst performing pair is Recursive Backtracking, and Hunt and Kill algorithms. They originate from the graph search algorithms. On the other hand most solving agents use the same approach which enables them to solve the maze easily. Therefore the mazes are generated in a way that suit the solving agents. To substantiate this statement we implemented Recursive Backtracking maze generator that generates mazes from the end to the beginning, the opposite of the original. In this case the agents performed considerably worse. The number of steps, visited intersections, and visited dead ends were 3 to 7 times higher than at the original Recursive Backtracking mazes. An exception was HDFS agent which improved its performance.

6. Conclusion

In our paper we studied and analysed three different approaches of generating mazes and were able to rank them by levels of difficulty. Nevertheless all three considered types were somehow kindred since they are used for finding trees in graphs. In the future it would be useful to take into the consideration algorithms with completely different approach [10, 11] and then compare the results.

As a continuation of our work it would be worthwhile looking into more complex mazes such as spatial, braided, overlapping, etc.

References

[1] David J Aldous. The random walk construction of uniform spanning trees and uniform labelled trees. *SIAM Journal on Discrete Mathematics*, 3(4):450–465, 1990.

[2] Daniel Ashlock, Colin Lee, and Cameron McGuinness. Search-based procedural generation of maze-like levels. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):260–273, 2011.

[3] V. Blagojević, D. Bojić, M. Bojović, M. Cvetanović, J. Đorđević, Đ. Đurđević, B. Furlan, S. Gajin, Z. Jovanović, D. Milićev, V. Milutinović, B. Nikolić, J. Protić, M. Punt, Z. Radivojević, Ž. Stanisavljević, S. Stojanović, I. Tartalja, M. Tomašević, and P. Vuletić. Chapter one - a systematic approach to generation of new ideas for phd research in computing. In Ali R. Hurson and Veljko Milutinović, editors, *Creativity in Computing and DataFlow SuperComputing*, volume 104 of *Advances in Computers*, pages 1 – 31. Elsevier, 2017.

[4] Simon R Broadbent and John M Hammersley. Percolation processes: I. crystals and mazes. In *Math-*

ematical Proceedings of the Cambridge Philosophical Society, volume 53, pages 629–641. Cambridge University Press, 1957.

[5] John C Gower and Gavin JS Ross. Minimum spanning trees and single linkage cluster analysis. *Applied statistics*, pages 54–64, 1969.

[6] M Tim Jones. *Artificial Intelligence: A Systems Approach: A Systems Approach*. Jones & Bartlett Learning, 2015.

[7] Aliona Kozlova, Joseph Alexander Brown, and Elizabeth Reading. Examination of representational expression in maze generation algorithms. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 532–533. IEEE, 2015.

[8] Gailand Macqueen. *The spirituality of mazes and labyrinths*. Wood Lake Publishing Inc., 2005.

[9] David S Olton. Mazes, maps, and memory. *American psychologist*, 34(7):583, 1979.

[10] Andrew Pech, Philip Hingston, Martin Masek, and Chiou Peng Lam. Evolving cellular automata for maze generation. In *Australasian Conference on Artificial Life and Computational Intelligence*, pages 112–124. Springer, 2015.

[11] AM Reynolds. Maze-solving by chemotaxis. *Physical Review E*, 81(6):062901, 2010.

[12] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[13] Ian Q Whishaw and Jo-Anne Tomie. Of mice and mazes: similarities between mice and rats on dry land but not water mazes. *Physiology & behavior*, 60(5):1191–1197, 1996.

[14] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 296–303. ACM, 1996.

[15] Hande Yaman, Oya Ekin KaraşAn, and Mustafa Ç PıNar. The robust spanning tree problem with interval data. *Operations research letters*, 29(1):31–40, 2001.

Peter Gabrovšek received his master degree in Computer Science from the University of Ljubljana in 2017. Currently, he is with the Laboratory of Algorithmics, Faculty of Computer and Information Science, University of Ljubljana, Slovenia, as an assistant and PhD student. His research interests include algorithmics, mathematics, and neural networks.