Thesis no: BCS-2016-07

Performance Evaluation of A* Algorithms

Victor Martell
Aron Sandberg

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science of Computer Science. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Victor Martell
E-mail: vima13@student.bth.se
Aron Sandberg
E-mail: arsa13@student.bth.se

University advisor:
M.Sc Diego Navarro
Department of DIKR

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. There have been a lot of progress made in the field of pathfinding. One of the most used algorithms is A*, which over the years has had a lot of variations. There have been a number of papers written about the variations of A* and in what way they specifically improve A*. However, few papers have been written comparing A* with several different variations of A*.
Objectives. The objectives of this thesis is to find how Dijkstra's algorithm, IDA*, Theta* and HPA* compare against A* based on the variables computation time, number of opened nodes, path length as well as number of path nodes.
Methods. To find the answers to the question in Objectives, an experiment was set up where all the algorithms were implemented and tested over a number of maps with varying attributes.
Results. The experimental data is compiled in a table showing the result of the tested algorithms for computation time, number of opened nodes, path length and number of path nodes over a number of different maps as well as the average performance over all maps.
Conclusions. A* is shown to perform well overall, with Dijkstra's algorithm trailing shortly behind in computation time and expanded nodes. Theta* finds the best path, with overall good computation time marred by a few spikes on large, open maps. HPA* performs poorly overall when fully computed, but has by far the best computation time and node expansion when partially pre-computed. IDA* finds the same paths as A* and Dijkstra's algorithm but has a notably worse computation time than the other algorithms and should generally be avoided on octile grid maps.

Keywords: pathfinding, astar, performance evaluation

Contents

Chapter 1

Introduction

Pathfinding is any method which solves the problem of obtaining a path between two points in space. It is used in a lot of fields such as games, robotics, finding shortest routes to travel in aviation or by train and finding paths in construction sites [2] [12], just to name a few areas. Typically, pathfinding involves minimizing the path length from start to goal. However, restrictions in regards to computation time and memory requirements can make a minimum path length unfeasible. There are also often restrictions to how the navigation space is represented. Common methods of representation include waypoint graphs [16] and grid based maps [15], which restrict the pathfinding to only certain valid pathways. This means an optimal path length within the space representation is not necessarily optimal in Euclidean space.

One of the best known and widely used pathfinding algorithms is A* [6]. A* improved on previous algorithms, such as Dijkstra's algorithm [5], by introducing a heuristic that gives positive weighting to nodes closer to the goal. This reduces the number of nodes that needs to be explored before finding a correct path. In terms of path length, A* performs equal to Dijkstra's, providing an optimal path within the graph, assuming the heuristic used does not overestimate the distance to the goal.

A* has had numerous variations of it developed throughout the years, generally seeking to improve specific aspects of the algorithm, such as speeding up calculations in dynamic environments [14], or reducing memory requirements [7]. This could make it difficult for a programmer to get an overview of which algorithms are suitable when there are multiple variables to consider. While these A*-variations have been tested against other algorithms that focus on the same aspects, there have been few studies evaluating the performance of A* variations with differing specializations. There is also a lack of tests that account for variables where the algorithms in question are not expected to perform well.

This thesis aims to evaluate the performance of different A* variations in terms of computation time of the algorithm, the number of expanded nodes, the length of the path and the number of path nodes, as well as different conditions regarding map size and obstacle density. The evaluation is done on A*, Dijkstra's algorithm [5], Iterative Deepening A* (IDA*) [7], Theta* [9] and Hierarchical

Pathfinding A* (HPA*) [3]. Out of these, A* and Dijkstra's algorithm are used as control points. The remaining algorithms have been selected because they are each expected to perform well in regards to one specific variable or under one specific condition.

Chapter 2

Related Work

In this section, previous work with similar research questions will be discussed briefly. It will also be argued why a research gap exists that motivates the research question of this thesis. Furthermore, the theoretical background of the algorithms evaluated in this thesis will be discussed along with the psuedo code describing the implementations used for them.

2.1 Similar Research

A lot of the research done in the field compares a single algorithm to the same algorithm with an improvement done by the authors to a specific variable. There are however other papers that are similar to this thesis that compare several algorithms solving a specific existing problem. Andayesh, M. and Sadeghpour, F. as well as Soltani, A.R. et.al. are comparing algorithms to find the optimal path based on a number of variables to travel in construction sites [2] [12]. The paper written by Andayesh, M. and Sadeghpour, F. finds that: "The visibility graph is the fastest approach when the obstacles are less than 15 and it takes fairly reasonable time in compare to other approaches for up to 30 obstacles in the site." and "The Euclidean distance is more accurate than the grid based approach. Accordingly, for cases when the visibility graph is slow, the Euclidean distance should be used instead." Soltani, A.R. et.al. finds in their thesis that: Dijkstra's algorithm finds an optimal path but becomes "inefficient for large-scale problems", furthermore A* also finds the optimal and near to optimal solutions more efficiently thanks to its heuristics. In the case of genetic algorithms (GA) in their test cases, the authors find that "Probabilistic optimisation approach based on GA generates a set of feasible, optimal, and close-to-optimal solutions that captures globally optimal solutions." The GA become an algorithm that is optimal or close to optimal, the authors phrase it in a good way: "[...] good regions of the search space get exponentially more copies and get combined with each other by the action of GA operators and finally form the optimum or a near-optimum solution in substantially less time." The evolving of the GA to the final form takes a lot of time to fine tune, thus it would benefit from off-line learning.

Randria, I. et.al. compares several pathfinding algorithms with navigation

learning in an electric wheelchair. [10]. Randria, I. et.al. concludes that "[...] genetic algorithms applied to a continuous workspace let us reach an acceptable path and very near to the global optimal path, without being dependant on the necessary resolution to refine the trajectory."

Sathyaraj, B. et.al. compares different pathfinding algorithms for unmanned aerial vehicles to find the shortest paths between different nodes [11]. Sathyaraj, B. et.al. compares different algorithms and finds A* to be the algorithm best suited for their problem.

Apart from the papers that seeks to find a solution to an existing problem, there has also been literature reviews in the field of pathfinding. These papers compare pathfinding phenomena on an abstract level without an experimental approach. Abd Algfoor, Z et.al. summarize different pathfinding techniques in their paper [1]. They discuss different means of organizing data points in different data types.

Botea, A. et.al. comes to a conclusion that "research in pathfinding is not in a danger of fading." They continue with mentioning that thanks to a recent pathfinding competition they speculate that the interest in pathfinding still is high albeit be it in single-agent pathfinding [4].

Souissi, O. et.al. discuss two main fields of path planning methods. One method where the environment modeling is necessary and one where is not necessary. A few examples of cases where the algorithm needs environment modeling are the algorithms mentioned in this thesis. An example of an algorithm that do not need environment modeling of the entire map is potential fields. Potential fields have a weighting around the goal similar to a cone, the closer to the goal that the agent is, the more favorable the weighting of a node is. To avoid obstacles, the obstacles get an unfavorable weighting. Furthermore, Souissi, O et.al. suggest a classification of pathfinding algorithms, "Classic Dynamic" which includes algorithms such as D*, D* Lite and Lifelong Planning A*. In the next category, "Any-angle movement", they classify algorithms as Theta*, Field D* and Incremental Phi*. In the "Moving Target" category they categorize algorithms as Generalized Adaptive A*, Generalized Fringe-Retrieving A* and Tree-AA*. In the fourth and final category, they introduce "Sub optimal" algorithms. In this category they place algorithms as HPA*, Anytime D* and Partial Refinement A* [13].

2.2 Dijkstra's Algorithm

Created in 1956 and published in 1959, Dijkstra's algorithm is the direct predecessor to A* and by extension all the algorithms covered here. The basis for all of these algorithms, with the exception of IDA*, is that beginning with the start node each of its neighbors are added to a priority queue. The first node in the queue is then selected and all of it's neighbors are added, assuming a

better path to the node hasn't already been found. In the case of Dijkstra's algorithm, the queue is sorted according to which node is the closest distance to the start node. This value, often referred to as the g-value is defined for node n as $f(n) = g(p) + c(p, n)$, where p is the immediately preceding node and $c(p, n)$ is the cost to move from p to n. Using the g-value guarantees that Dijkstra's algorithm will find the shortest path, but as it expands every node which is a shorter distance from the start node than the goal node is, it can be quite slow.

```
Dijkstra.updateNode(node, NextNode)
    if g(node) + cost(node, nextNode) < g(nextNode)
        g(nextNode) := g(node) + cost(node)
        parent(nextNode) := node
        open.insert(nextNode, g(nextNode))
```

2.3 A*

A* expands on Dijkstra's algorithm by adding a heuristic value which estimates the remaining distance between the examined node and the goal. The next node n to expand is thus chosen through the lowest value $f(n) = g(n) + h(n)$, where $h(n)$ is the heuristic function. If the heuristic function does not overestimate the distance to the goal, it is said to be admissible. An example of such a heuristic would be the Euclidean distance $h(n) = \sqrt{(n_x - goal_x)^2 + (n_y - goal_y)^2}$, which is the shortest possible distance within a 2D plane. As long as the heuristic function is admissible, A* will find the shortest possible path. The use of a heuristic tends to direct A* towards the goal quicker than Dijkstra's. In the worst case, A* will expand the same amount of nodes as Dijkstra's.

```
AStar.findPath(start, goal)
    closed(start) := true
    node := start
    while node ≠ goal
        foreach nextNode ∈ neighbours(node)
            if ¬closed(nextNode) AND traversable(nextNode)
                updateNode(node, NextNode)
        while closed(node)
            if open = ∅
                return false
            node = open.pop()
        closed(node) := true
    while node ≠ start
        path.add(node)
        node := parent(node)
    return true
```

```
AStar.updateNode(node, NextNode)
    if g(node) + cost(node, nextNode) < g(nextnode)
        g(nextNode) := g(node) + cost(node)
        parent(nextNode) := node
        open.insert(nextNode, g(nextNode) + h(nextNode))
```

2.4 IDA*

IDA* was made with the intent to lower memory usage, as it does not need to save an open queue. It is based on iterative depth first search, where a depth first search is performed until the search surpasses a certain depth. The search is then reset, but with the depth threshold increased. For iterative depth first search, the threshold usually starts at zero and is increased each time to the least depth that surpasses the previous threshold. IDA* instead, uses the f-value from A* as its threshold, starting with $f(start) = h(start)$ as the initial threshold and increasing with the lowest $f(n)$ that surpasses the previous threshold. Assuming the heuristic is admissible, the path length will be optimal, as the path length will always be greater than or equal to the threshold.

```
IDAStar.findPath(start, goal)
    node := start
    iteration := 0
    threshold = h(start)
    while node ≠ goal
        node, f := evaluateNode(start,0,threshold,threshold,i)
        threshold := f
        i := i + 1
    while node ≠ start
        path.add(node)
        node := parent(node)
    return true
```

```
IDAStar.evaluateNode(node, g, f, threshold, i)
    minf := ∞
    if g(node) + h(node) > threshold
        f = g(node) + h(node)
        return node, f
    foreach n' ∈ neighbours(node)
        if lastChecked(n') ≠ i
        OR g + c(node, n') < g(n')
            parent(n') := node
            g(n') := g + c(node, n')
            lastChecked(n') := i
            endNode,f := evaluateNode(n',g(n'),f,threshold,i)
            if endNode = goal
                return endNode, f
            if f < minf
                minf := f
    return position(minf), minf
```

2.5 Theta*

While the previously mentioned algorithms all provide an optimal solution, that optimality depends on the method used to represent the space that the algorithms traverse. The chosen representation in this thesis is an octile grid, which means the optimal path through the grid is not necessarily optimal in continuous 2D space. Theta* seeks to use grids for their space representation without having the resulting paths constrained to the grid, thereby providing shorter paths than regular A* as seen in figure 2.1 along with the shortest possible any-angle path. Functionally, Theta* is similar to A* except for how it determines the parent of a node. Whereas the parent of an expanded node in A* will always be the directly preceding node, in Theta*, the parent will instead be the parent of the preceding node, assuming there's a line of sight between the parent and the new node. This will typically cause Theta* to find a more direct path than would normally be possible when constrained to a grid, at the cost of some extra processing power to handle the line of sight calculations.

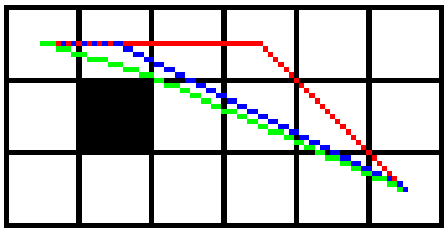

Figure 2.1: *Example of paths found by A* (red), Theta* (blue) and the optimal any-angle path (green).*

```
ThetaStar.findPath(start, goal)
// code identical to A*
    if ¬closed(nextNode) AND traversable(nextNode)
        if lineOfSight(parent(node), nextNode)
            updateNode(parent(node), NextNode)
        else
            updateNode(node, NextNode)
// code identical to A*
```

2.6 HPA*

HPA* has the opposite focus of Theta*, in that it sacrifices path length in return for improved processing speed. It does so by first drawing a rough graph of available paths, thereby reducing the number of nodes to examine compared to the initial grid. HPA* starts by dividing the map into rectangular clusters. For each open border to a neighboring cluster, nodes are added to a list of border nodes within each cluster, either one or two depending on the border's length. For borders with more than six consecutive traversable nodes, two nodes are added to the list for each cluster, one at each end of the consecutive path. For borders shorter than six nodes, one node in the middle of the border is added to the list for both clusters. The number six is arbitrary, however it is what the original paper used.

For every border node within a cluster, internal path lengths are calculated to every other border node in the same cluster. It would also be possible to store the full paths, but at the cost of extra memory. These paths form a high level graph of the map, which allows the pathfinding to be divided into several steps. First, the start and goal nodes are attached to the graph. This works similarly to how the border nodes are connected to each other. With the start node and goal nodes attached, a path is found on the high level graph. Finally, assuming the full paths weren't saved earlier, the detailed path can be found by backtracking along the high level path, finding the detailed path between each step in the high level path.

A big advantage of this algorithm is that the internal path lengths only has to be calculated once. The computation of the detailed path can also be delayed, as the detailed path only needs to be known between the current border node and the next. This means that real time pathfinding can be reduced to the relatively few high level nodes, as well as the detailed path from the start node which is limited by the size of the cluster.

```
HPAStar.findPath(start, goal)
    foreach c ∈ clusters
        foreach c' ∈ neighbours(c)
            findEdges(c, c')
        findInternalPaths(c)
    attachNodeToGraph(start)
    attachNodeToGraph(goal)
    //Uses the constructed graph instead of a tile grid
    //AStar.findPath(start, goal)
    node := start
    cluster := findCluster(start)
    while node ≠ goal
        foreach nextNode ∈ nodes(cluster)
            if pathLength(node, nextNode) ≠ ∞
                if ¬closed(nextNode) AND traversable(nextNode)
                    updateNode(node, NextNode)
        updateNode(node, edge(node))
        while closed(node)
            if open = ∅
                return false
            node = open.pop()
        closed(node) := true
    while node ≠ start
        AStar.findPath(node, parent(node))
        path.add(AStar.path)
```

```
HPAStar.findInternalPaths(cluster)
    foreach n, n' ∈ nodes(cluster)
        if n ≠ n' AND ∄ m ∈ nodes(cluster),
        (pathLength(n, m) = ∞ XOR pathLength(n', m) = ∞)
            AStar.findPath(n, n')
            pathLength(n, n') := AStar.pathLength
```

```
HPAStar.attachNodeToGraph(node)
    cluster := findCluster(node)
    foreach n ∈ nodes(cluster)
        AStar.findPath(node, n)
        pathLength(node, n) := AStar.pathLength
```

Chapter 3

Method

The thesis is based on an experimental evaluation of Dijkstra's, IDA*, Theta* and HPA* in relation to A*. This section will describe the tools needed to conduct the experiment, as well as the organization of the maps and the testing criteria.

3.1 Development tools

The code implemented for this thesis is written in C++ 14 using Visual Studio Enterprise 2015. The code was implemented to execute on the central processing unit (CPU) rather than the graphical processing unit (GPU). This is due to the experience of programming the algorithms on the CPU greatly exceeded the experience of programming on the GPU. To get graphical output, SFML version 2.3.2, ImGui version 1.48 and an SFML Backend for ImGui were used [8].

C++ is the language used to develop the algorithms as well as the graphical interface. The choice of C++ as programming language was due to the knowledge and experience accumulated by the authors of the thesis.

SFML was used to render the graphical elements. These include the graphical interface, the algorithm's path, the non-traversable tiles of the map, as well as the expanded nodes. It was chosen based on previous experience. ImGui builds on an SFML-environment to render its graphical output. ImGui was chosen as interface thanks to its simplicity and how quickly an interface could be up and running.

3.2 Maps

The maps that are used in the testing of the algorithms consists of traversable and non-traversable tiles arranged in a square octile grid. The maps consist of eight maps from Sturtevant's repository and 12 maps that are generated by the developed map randomizer, pictures of all the maps can be seen in Appendix B [15]. The maps that are used from Sturtevant's repository consist of two maps forming a grid of square rooms, two mazes and two maps each from the games Baldur's Gate 2 and StarCraft. The maps were selected from the repository with a few aspects in mind. Firstly, the maps should contain only traversable and non-traversable tiles, some maps in the repository contain tiles with further information. Secondly, the maps should offer the possibility of having a traversable path that is complex. Thirdly, the maps in each category had to be different in regards to map composition. And lastly, maps with longer path length were chosen over maps with shorter path length.

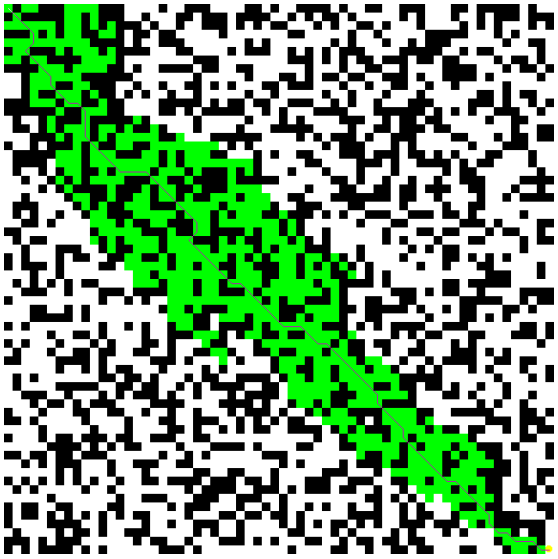

Figure 3.1: *A pathfinding of A* on the map "Randomized64x64-40-0". The map is 64 by 64 tiles. The green squares are A*'s expanded nodes, the line is the path A* finds.*

The generated maps uses a random distribution to place non-traversable tiles in the tile grid, an example of a generated map can be seen in figure 3.1. It exists twelve generated maps, eight of these maps have a given size (64 by 64) with different densities of obstacles, ranging from 10% up to 45% with increments of 5 percentage. The remaining four maps differ in size but have a fixed obstacle density (30%). The sizes of the maps start at 32 by 32 and both width and height are doubled up to 512 by 512, ignoring size 64 by 64 as it is included in the maps with incrementing obstacle density. When generated, the included maps are the first maps with a traversable path from $\{0, 0\}$ to $\{width - 1, height - 1\}$.

All the algorithms, apart from IDA*, are tested on all 20 maps. IDA* was not tested on the repository maps, as it is prohibitively slow on large maps.

3.3 Testing criteria

The testing is conducted on windows 8.1 x64, with an intel core i5 4690k @ 3.5GHz processor and 16GB RAM. A*, Theta* and HPA* uses Euclidean distance as a heuristic. IDA* uses octile distance as it runs too slowly with Euclidean distance to be viable on most of the maps. Naturally, Dijkstra's algorithm does not make use of any heuristic. Algorithms will only be tested once for each map, as the results are mostly deterministic. The computation time is not fully deterministic, however, its results are predictable enough that one test can be considered sufficiently accurate for the objectives of this thesis.

The tests measure the computation time of the algorithm, the number of expanded nodes, the length of the path and the number of path nodes. Computation time and path length are standard measurements when testing pathfinding. Computation time shows how quickly a path can be found while path length shows how good the resulting path is. Expanded nodes can, similarly to computation time, be used as a measure of efficiency, as it roughly simulates how much an algorithm needs to iterate before the goal is reached. It is also an indicator of memory usage as the expanded nodes will often have to be stored. Note that some algorithms such as IDA* and HPA* are calculated in several steps, where the expanded nodes do not need to be stored in between these steps. Finally path nodes can also be used to indicate memory, as fewer nodes stored in the path means that the path uses less memory.

The values of the variables are extracted via a metrics-class. The exception is the computation time, which is measured by placing breakpoints before and after the pathfinding function and using the Visual Studio diagnostic tools to track the time between the breakpoints. The measured numbers are saved manually from the metrics-class into a spreadsheet. The spreadsheet is compiled into table A.1. Memory is a variable that was originally planned to be included. However it proved difficult to properly measure, as no appropriate method was found to determine the exact memory usage by the algorithms themselves, separated from the memory used by the other parts of the program.

Chapter 4

Results

When analyzing the results from the test cases, A* is used as a control algorithm. The results below are organized according to algorithm. In the tables the algorithms are compared to A* since it is used as a control algorithm. The numbers in the tables are based on data from all the maps that they were tested on.

4.1 Dijkstra's Algorithm

Algorithm	Time	Expansion	Path Length	Path Nodes
A*	1.000	1.000	1.000	1.000
Dijkstra	1.802	3.078	1.000	1.000

Table 4.1: *Average performance of Dijkstra's algorithm compared to A*.*

Dijkstra's algorithm computes a path about 1.8 times slower on average. It is faster than A* on 3 maps, "AR0307SR", "Turbo" and "maze512-16-0". Of these maps, "maze512-16-0" is the fastest, with a computation time 76.4% of A*. Dijkstra's algorithm also takes an equal amount of time as A* on the map "maze512-1-1". Dijkstra's algorithm performed worst on the generated maps with a low percentage of obstacles. The maps "Randomized64x64-10-0" and "Randomized64x64-15-0" both took 3 times longer for Dijkstra's algorithm than A*, which is its worst relative performance.

Dijkstra's algorithm performs slightly worse in terms of expanded nodes than in computation time, on average about 3 times more nodes than A*. The number of expanded nodes is consistently higher than A* on every map, but as with computation time the performance is relatively worse on the sparse random distribution maps.

Dijkstra's algorithm is identical to A* in path length and number of path nodes for all tests. This is expected as both algorithms are theoretically optimal in terms of path length.

4.2 IDA*

Algorithm	Time	Expansion	Path Length	Path Nodes
A*	1.000	1.000	1.000	1.000
IDA*	25.718	40.659	1.000	1.000

Table 4.2: *Average IDA* performance compared to A*, on the maps which IDA* was tested on.*

Even with the octile heuristic, IDA* is the slowest of the tested algorithms, except on very small and open maps, such as "Randomized64x64-10-0". Testing on "Randomized64x64-45-0", the map with the highest obstacle density, it performs about 3 times slower than the second slowest, HPA* (with internal path calculation included) and about 50 times slower than regular A*. Likewise the time scales rapidly with increasing size, with IDA* taking more than 5 minutes to complete the largest of the generated maps, "Randomized512x512-30-0", almost 2500 times that of A*. Due to the time required to test IDA* for the larger maps, it was not tested on the other maps bigger than 511 by 511.

The number of expanded nodes scale at a similar rate to the processing time. However, as IDA* does not keep an open queue and does not need to to store information between iterations, the expanded nodes would not impact memory as much as it would with other algorithms.

In theory, IDA* should result in the same path length as with A*. However, there appears to have been some rounding error, most likely related to the implementation of the octile heuristic, which caused one extra node to be added to the path on the "Randomized128x128-30-0" map. For most maps, the path length is however equal to A*.

4.3 Theta*

Algorithm	Time	Expansion	Path Length	Path Nodes
A*	1.000	1.000	1.000	1.000
Theta*	9.049	0.856	0.971	0.372

Table 4.3: *Average Theta* performance compared to A*.*

The processing time of Theta* is longer than A*. Generally, Theta* is only slightly slower than A*, but on all four game maps, Theta* performed worse with a factor of ten. With the exception of map "AR0307SR", these maps are also where Theta* found the shortest path lengths relative to A*. The only other maps where Theta* took more than twice the time of A* was on "32room_008"

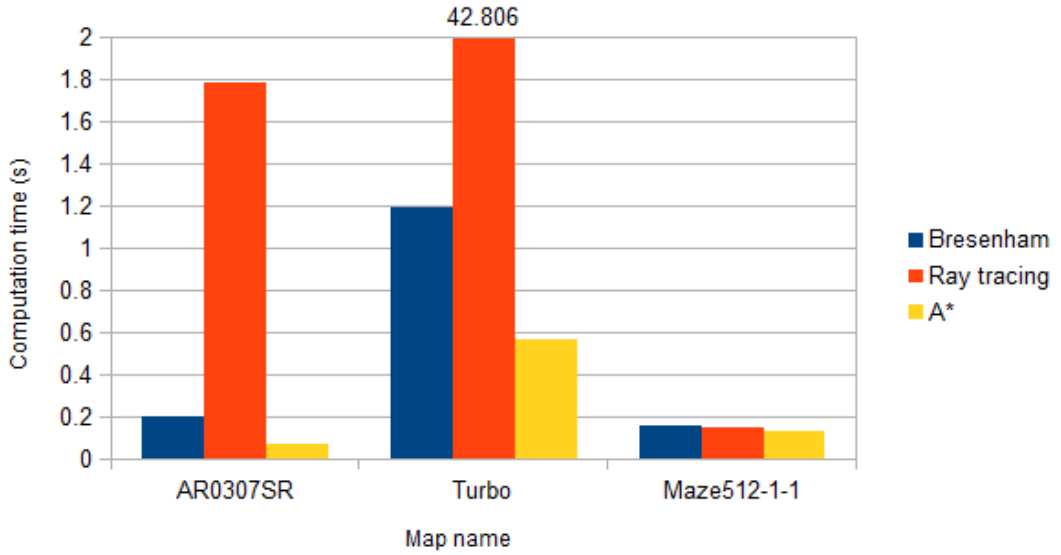

Figure 4.1: *Time comparison of Theta* with Bresenham (blue), Theta* with ray-tracing (red) and A* (yellow)*

and "maze512-16-0", where it performs 3.684 times slower and 9.316 times slower respectively.

A comparison with the Bresenham algorithm, as seen in figure 4.1, shows a decrease in computation time from 1.788s to 0.002s on "AR0307SR", a decrease from 42.806s to 1.200s on "Turbo" and a decrease from 1.130s to 0.156s on "maze-512-1-1". Unfortunately Bresenham lines will cause paths to clip through walls with the grid representation that was used for this experiment.

Theta* results in a shorter path length than A* in all but one map. On the map "maze512-1-1" the path length is instead equal to A*. The average length across all maps for Theta* was 97% of the A* path length.

The number of nodes needed to make up the path were consistently lower in Theta*. Even on the maze where path length was identical, the number of path nodes was only 59.3% of A*. On average, Theta* saved 37.2% of the path nodes compared to A*.

Theta* performed slightly better than A* in terms of the amount of expanded nodes. On average, Theta* expanded 85.6% as many nodes as A*. This was largely consistent across all maps with the lowest relative value being 71.7% of A* on the map "Randomized128x128-30-0", while the only value higher than A* happened on "Randomized64x64-10-0" where Theta* expanded 1.2% more nodes.

4.4 HPA*

Algorithm	Time	Expansion	Path Length	Path Nodes
A*	1.000	1.000	1.000	1.000
HPA*	8.532	12.671	1.030	1.043
offline HPA*	0.288	0.078	1.030	1.043

Table 4.4: *Average HPA* performance on maps of size 512x512 or greater compared to A*. Includes average time for HPA* assuming internal paths are pre-calculated.*

The complete HPA* algorithm performs worse on average than A* over all measured variables. However, 97% of the computation time consists of calculating the internal paths of each cluster, which only needs to be done once per map. Assuming the internal paths have been pre-computed, the average time on maps that are 512 by 512 or larger decreases from 853.2% of A* to 28.8% of A*.

The computation of internal paths can vary a lot in performance. The worst case was the generated random map of size 512 by 512 which had a total run time almost 50 times that of A* while expanding 82 times as many nodes. In contrast, on "Turbo" only 1.072 the time of A* was needed. Overall the maps with a random distribution of walls, were comparatively slower. The average computation time for the full HPA* was 31.276 that of A*.

Of the tested algorithms HPA* is the only one providing sub-optimal paths (along the grid tiles. All of them are sub-optimal in Euclidean space). Compared to A* the average path length was 4.3% longer with a maximum of 12.5% longer on "Randomized32x32-30-0" and a minimum of 2% longer on "Turbo".

Chapter 5

Analysis and Discussion

In this section, all the algorithms will be discussed in relation to A*. The results of the measurements will be analyzed and explained. This section will also take a closer look at specific interesting behavior exhibited by the different algorithms.

5.1 Dijkstra's Algorithm

In theory, Dijkstra's should perform like a worst case A*, but with a lower overhead due to not having to calculate the h-value. The tests seem to corroborate this, as Dijkstra's algorithm did relatively better in terms of computation time than it did with the amount of expanded nodes. This indicates that Dijkstra's algorithm needs to check a higher amount of nodes, but that each individual node takes less time to check.

Dijkstra's algorithm typically had better relative results on maps where the path included parts which were moving in a direction away from the goal. Under such circumstances the pathfinding algorithms that use a heuristic would be less useful as the heuristic would guide them the wrong way. Dijkstra's algorithm was also the only algorithm to become faster as obstacle density increased. As it is not directed by a heuristic the increased complexity of the path wouldn't affect it. Instead the lessened number of traversable nodes would mean that there are fewer nodes that needs to be expanded.

There is little to say in terms of path quality for Dijkstra's as it both experimentally and theoretically should find the same path as A* does.

5.2 IDA*

Due to IDA* frequently restarting its search, it was expected to be one of the slower algorithms, which has been confirmed by the tests. Due to the restarts, the speed of IDA* is also more affected by the choice of heuristic than the other algorithms. Euclidean distance, which was the heuristic originally used, has a greater variety of possible f-values than octile distance, which means the threshold increments at a much slower rate, leading to more search resets. As IDA* with

Euclidean distance is too slow to test within a reasonable time frame on the maps with sizes 512 by 512 and above, the tests for IDA* instead uses octile distance. Unfortunately there appears to have been some rounding errors when using octile distance as the map "Randomized128x128-30-0" shows an extra node added to the path of IDA*. As IDA* should theoretically find exactly the same path length as A* and Dijkstra's algorithm, this result is an error. A test was done with A* using octile distance as its heuristic, which would appear to confirm that the error lies with the implementation of the heuristic, rather than being inherent to the IDA* implementation.

5.3 Theta*

Theta* being slower than A* is not surprising, since Theta* is near identical to A*, but with an added line of sight calculation. The large difference in performance can likely be attributed to there being more open areas, where Theta* has to calculate more costly line of sight checks. This is further strengthened by the fact that poor computation times being linked to a relative reduction of path nodes for Theta*. The difference in performance between the two line of sight algorithms emphasizes the importance of optimizing the line of sight algorithms. The original article makes use of the Bresenham algorithm along with using grid edges rather than grid centers, which is likely to be a more fitting implementation for Theta*.

The lack of improvement in path length on "maze512-1-1" is likely due to the tight constraints of the map, preventing any corners to be cut by Theta*. It still had fewer path nodes, which suggests that some successful line of sight checks were made, but that they were along the same angles as the octile grid. While Theta* is saving path nodes along straight paths, it doesn't help the path length, it would be valuable in cases where storing the path would cause memory concerns. An example of where it might be useful would be storing the internal paths of HPA*.

Theta* generally needing to expand less path nodes is likely caused by Euclidean distance working better as a heuristic distance for Theta*. Generally, if a heuristic doesn't underestimate the distance towards the goal too much, the search will be more directly guided towards the node. As Theta* has a shorter path length than A* the Euclidean distance would be less of an underestimation when used with Theta*. The difference in expanded nodes would likely be reduced if A* instead used octile distance, which is inadmissible for Theta* as it could potentially overestimate its distance.

5.4 HPA*

When using the full computation time of HPA* it would seem to not be worth using over A*. However, using pre-calculated internal paths, HPA* starts seeing performance gains compared to A* after roughly ten separate pathfinding uses. Needing to use pathfinding multiple times on the same map is not an uncommon scenario, suggesting that HPA* would be quite useful for time sensitive applications.

The internal path calculation seemed to have the most difficulties with the random distribution maps. These maps are highly fragmented compared to the larger and more coherent obstacle segments in many of the repository maps. Having more fragmented obstacle segments would cause the cluster borders to be divided into many smaller borders, which would result in far more nodes to connect in the high level graph.

In terms of path length HPA* had the worst performance. This is not surprising as its hierarchical nature causes many valid paths to not be considered.

Chapter 6

Conclusions

In this section, the conclusions that can be drawn from the experiments are stated. The discussed cases are the algorithms that performed the best in a certain aspect as well as the algorithms that performed the worst.

This thesis has tested five different pathfinding algorithms in regards to path length, node expansion, computation time and number of path nodes, the relative performance of which can be seen in figure 6.1. It has been shown that of the tested algorithms Theta* will produce the highest quality path, showing consistently the best performance in terms of path length and number of path nodes needed in return for an above average computation time, which can be mitigated with a good line of sight algorithm.

For computation time, A* showed the best performance when starting from no prior knowledge of the map. However HPA* can with a high initial computation cost become more effective over subsequent searches. The path quality of HPA* is however the worst of the tested algorithms.

It can also be concluded that IDA* is not a suitable search algorithms for anything but very small search spaces when represented by an octile map, as the algorithms scale too quickly in computation time as the distances increase.

	Time	**Expansion**	**Path Length**	**Path Nodes**
A*	Good	Good	Average	Average
Theta*	Average	Good	Good	Great
HPA* online	Bad	Bad	Bad	Bad
HPA* offline	Great	Great	Bad	Bad
IDA*	Very Bad	Very Bad	Average	Average
Dijkstra	Average	Average	Average	Average

Figure 6.1: *Summary of algorithm performance.*

Chapter 7

Future Work

In this section, the future work is discussed. There are a number of different fields suggested that can stem out of the work done in this thesis.

A possible future work is to do a more extensive experiment with more algorithms. The algorithms that are tested in this thesis are a sample of the most common algorithms. Out of these common ones, the chosen algorithms are algorithms that differ in expected results. Another related future work could be to compare algorithms which focus in a single aspect such as comparing an improved A* focusing on memory efficiency and Theta* with an improved memory efficiency.

The only tested heuristic in this thesis is Euclidean for all algorithms apart from IDA* since the computation time increased drastically with this heuristic. Instead, IDA* used octile to calculate its path. A future work could be to expand this and test how different algorithms are affected by changing heuristics. In the case of HPA*, only a single cluster size (16 by 16) was used. If the cluster size is bigger, the measured results will most likely differ, especially if HPA* is pre-calculated offline.

One more thing not being discussed in this thesis is dynamic environments with moving targets or moving obstacles. It was originally discussed to be included but due to time constraints it was dropped.

Another field worth looking into is map representation which is in three dimensions instead of the two which were evaluated in this thesis. The grid used in the maps in this thesis only use an octile grid. With more map representations the calculated results may differ.

In this thesis, multi-threading was not used when computing the algorithms. There are algorithms that could greatly benefit from threaded computation. HPA* would benefit with the calculation of its pathfinding in the clusters.

The algorithms can be implemented both on the CPU and the GPU. Due to previous experience of programming on the CPU and lack of experience in programming on the GPU, this thesis only focus on the CPU implementation. One thing that could be further investigated is implementing the algorithms on both the CPU and the GPU to compare how they fare.

References

[1] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. *International Journal of Computer Games Technology*, 2015:1–11, 2015.

[2] Mohsen Andayesh and Farnaz Sadeghpour. A Comparative Study of Different Approaches for Finding the Shortest Path on Construction Sites. *Procedia Engineering*, 85:33–41, 2014.

[3] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1:7–28, 2004.

[4] Botea, Adi, Bouzy, Bruno, Buro, Michael, Bauckhage, Christian, and Nau, Dana. Pathfinding in Games. 2013.

[5] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[6] Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[7] Richard E. Korf. Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.

[8] Alff Mischa. Imgui backend. `https://github.com/Mischa-Alff/imgui-backends`, 2015.

[9] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Feiner. Theta*: Any-angle Path Planning on Grids. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pages 1177–1183, Vancouver, British Columbia, Canada, 2007. AAAI Press.

[10] I. Randria, M. M. B. Khelifa, M. Bouchouicha, and P. Abellard. A Comparative Study of Six Basic Approaches for Path Planning Towards an Autonomous Navigation. In *33rd Annual Conference of the IEEE Industrial Electronics Society, 2007. IECON 2007*, pages 2730–2735, November 2007.

[11] B. Moses Sathyaraj, L. C. Jain, A. Finn, and S. Drake. Multiple UAVs path planning algorithms: a comparative study. *Fuzzy Optimization and Decision Making*, 7(3):257–267, June 2008.

[12] A.R. Soltani, H. Tawfik, J.Y. Goulermas, and T. Fernando. Path planning in construction sites: performance evaluation of the Dijkstra, A*, and GA search algorithms. *Advanced Engineering Informatics*, 16(4):291–303, October 2002.

[13] O. Souissi, R. Benatitallah, D. Duvivier, A. Artiba, N. Belanger, and P. Feyzeau. Path planning: A 2013 survey. In *Proceedings of 2013 International Conference on Industrial Engineering and Systems Management (IESM)*, pages 1–8, October 2013.

[14] A. Stentz. Optimal and efficient path planning for partially-known environments. pages 3310–3317. IEEE Comput. Soc. Press, 1994.

[15] N. R. Sturtevant. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, June 2012.

[16] Weiping Zhu, Daoyuan Jia, Hongyu Wan, Tuo Yang, Cheng Hu, Kechen Qin, and Xiaohui Cui. Waypoint Graph Based Fast Pathfinding in Dynamic Environment. *International Journal of Distributed Sensor Networks*, 2015:1–12, 2015.

Appendix A

Metrics

The time is measured in seconds. The cluster size of HPA* is 16 by 16 tiles. Euclidean heuristic was used for all the tests apart from IDA* which used octile heuristic. If a cell is empty, the data in the cell is the same as the cell above. The results in IDA* written as N/A were judged to be too slow based on the computation of map "Randomized512x512-30-0". Note that the computation time of A* and Theta* in the map "Randomized32x32-30-0" is below 0.001 s but is written as 0.001. The start- and goal-nodes are marked as x- and y-positions on the map. Expansion measures the number of nodes the algorithm evaluates.

The naming convention of generated maps is Randomized[width]x[height]-[obstacle density]-0. The remaining maps are from Sturtevant's repository and use the same name as in the repository [15].

Map	Start	Goal	Algorithm	Time	Expansion	Path Length	Path Nodes
AR0307SR	{350, 54}	{467, 387}	A*	0.073	38583	1115.306	873
			Theta*	1.788	37614	1070.537	52
			Online HPA*	0.217	109401	1162.320	933
			Offline HPA*	0.024	5477	1162.320	933
			IDA*	N/A	N/A	N/A	N/A
			Dijkstra	0.060	43139	1115.306	873
AR0700SR	{343, 5}	{188, 462}	A*	0.106	53656	637.126	546
			Theta*	1.437	46600	602.826	45
			Online HPA*	0.457	257361	656.156	555
			Offline HPA*	0.033	5771	656.156	555
			IDA*	N/A	N/A	N/A	N/A
			Dijkstra	0.169	117149	637.126	546
FloodedPlains	{0, 0}	{767, 759}	A*	0.396	166860	1244.560	989
			Theta*	11.539	128420	1182.508	44
			Online HPA*	0.820	397946	1280.202	1040
			Offline HPA*	0.059	7526	1280.202	1040
			IDA*	N/A	N/A	N/A	N/A
			Dijkstra	0.532	363096	1244.560	989
Turbo	{534, 125}	{225, 638}	A*	0.566	254520	1470.719	1228
			Theta*	42.806	248304	1396.230	41
			Online HPA*	0.607	268738	1499.817	1221
			Offline HPA*	0.055	18914	1499.817	1221
			IDA*	N/A	N/A	N/A	N/A
			Dijkstra	0.515	342522	1470.719	1228

Map	Start	Goal	Algorithm	Time	Expansion	Path Length	Path Nodes
16room_000	{1, 1}	{511, 511}	A*	0.262	105024	820.971	676
			Theta*	0.448	87958	789.731	119
			Online HPA*	0.317	148549	838.205	704
			Offline HPA*	0.028	3510	838.205	704
			IDA*	N/A	N/A	N/A	N/A
			Dijkstra	0.351	231807	820.971	676
32room_008	{1, 1}	{511, 511}	A*	0.250	107601	839.296	679
			Theta*	0.921	87932	802.156	59
			Online HPA*	0.341	153868	867.560	716
			Offline HPA*	0.033	5179	867.560	716
			IDA*	N/A	N/A	N/A	N/A
			Dijkstra	0.370	246431	839.296	679
maze512-1-1	{1, 1}	{511, 511}	A*	0.131	88761	3601.258	3076
			Theta*	0.148	88761	3601.258	1823
			Online HPA*	1.507	1204317	3683.230	3214
			Offline HPA*	0.092	16664	3683.230	3214
			IDA*	N/A	N/A	N/A	N/A
			Dijkstra	0.131	102801	3601.258	3076
maze512-16-0	{1, 1}	{511, 511}	A*	0.373	202382	2789.263	2537
			Theta*	3.475	201579	2689.041	141
			Online HPA*	1.227	872317	2884.593	2605
			Offline HPA*	0.013	11577	2884.593	2605
			IDA*	N/A	N/A	N/A	N/A
			Dijkstra	0.285	211151	2789.263	2537

Map	Start	Goal	Algorithm	Time	Expansion	Path Length	Path Nodes
Randomized32x32-30-0	{0, 0}	{31, 31}	A*	0.001	309	48.184	37
			Theta*	0.001	262	46.802	20
			Online HPA*	0.007	3548	54.184	43
			IDA*	0.009	8551	48.184	37
			Dijkstra	0.002	715	48.184	37
Randomized128x128-30-0	{0, 0}	{127, 127}	A*	0.008	3590	190.149	145
			Theta*	0.009	2575	185.413	87
			Online HPA*	0.354	233884	206.492	161
			IDA*	0.881	732875	190.735	146
			Dijkstra	0.017	11459	190.149	145
Randomized256x256-30-0	{0, 0}	{255, 255}	A*	0.030	12575	379.956	288
			Theta*	0.033	10035	371.762	159
			Online HPA*	1.657	1004748	396.600	315
			IDA*	11.717	9413154	379.955	288
			Dijkstra	0.071	45870	379.956	288
Randomized512x512-30-0	{0, 0}	{511, 511}	A*	0.132	50885	758.978	573
			Theta*	0.138	40006	741.999	299
			Online HPA*	6.467	4185105	788.857	624
			Offline HPA*	0.095	8897	788.857	624
			IDA*	326.000	255333536	758.982	573
			Dijkstra	0.330	183481	758.978	573

Map	Start	Goal	Algorithm	Time	Expansion	Path Length	Path Nodes
Randomized64x64-10-0	{0, 0}	{63, 63}	A*	0.002	407	90.853	66
			Theta*	0.003	412	89.659	18
			Online HPA*	0.051	27919	93.782	70
			IDA*	0.002	1250	90.853	66
			Dijkstra	0.006	3686	90.853	66
Randomized64x64-15-0	{0, 0}	{63, 63}	A*	0.002	565	91.439	67
			Theta*	0.002	425	89.788	27
			Online HPA*	0.069	39019	93.782	71
			IDA*	0.003	2232	91.439	67
			Dijkstra	0.006	3481	91.439	67
Randomized64x64-20-0	{0, 0}	{63, 63}	A*	0.002	859	93.196	70
			Theta*	0.003	687	90.899	29
			Online HPA*	0.090	41855	96.125	75
			IDA*	0.015	14304	93.196	70
			Dijkstra	0.005	3275	93.196	70
Randomized64x64-25-0	{0, 0}	{63, 63}	A*	0.002	848	93.782	71
			Theta*	0.003	726	91.771	44
			Online HPA*	0.072	45531	96.125	75
			IDA*	0.015	14654	93.782	71
			Dijkstra	0.005	3068	93.782	71

Map	Start	Goal	Algorithm	Time	Expansion	Path Length	Path Nodes
Randomized64x64-30-0	{0, 0}	{63, 63}	A*	0.003	1020	95.539	74
			Theta*	0.003	811	92.977	40
			Online HPA*	0.052	32838	98.468	79
			IDA*	0.055	51350	95.539	74
			Dijkstra	0.005	2863	95.539	74
Randomized64x64-35-0	{0, 0}	{63, 63}	A*	0.003	935	96.711	76
			Theta*	0.003	715	94.443	46
			Online HPA*	0.054	33634	100.811	83
			IDA*	0.045	42328	96.711	76
			Dijkstra	0.004	2859	96.711	76
Randomized64x64-40-0	{0, 0}	{63, 63}	A*	0.002	625	98.368	76
			Theta*	0.003	547	96.201	48
			Online HPA*	0.049	33232	107.640	89
			IDA*	0.047	42617	98.368	76
			Dijkstra	0.004	2435	98.368	76
Randomized64x64-45-0	{0, 0}	{63, 63}	A*	0.002	719	99.882	80
			Theta*	0.002	629	97.716	49
			Online HPA*	0.038	22750	105.640	87
			IDA*	0.094	86652	99.882	80
			Dijkstra	0.004	2222	99.882	80

Appendix B

Maps

Image set of all maps used. Maps smaller than 256x256 have been scaled up. Maps larger than 512x512 have been scaled down. Cyan is the start position and blue is the goal position.

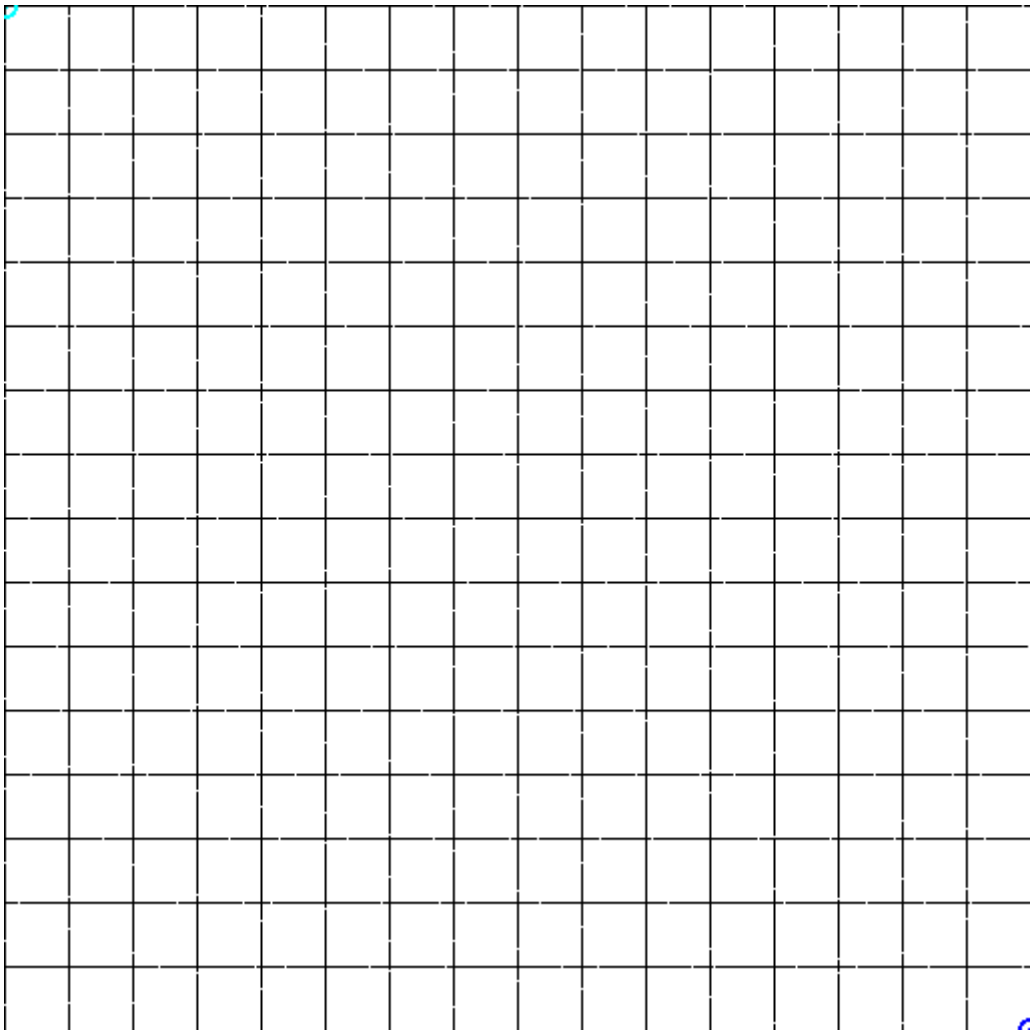

Figure B.1: *32room_ 008*

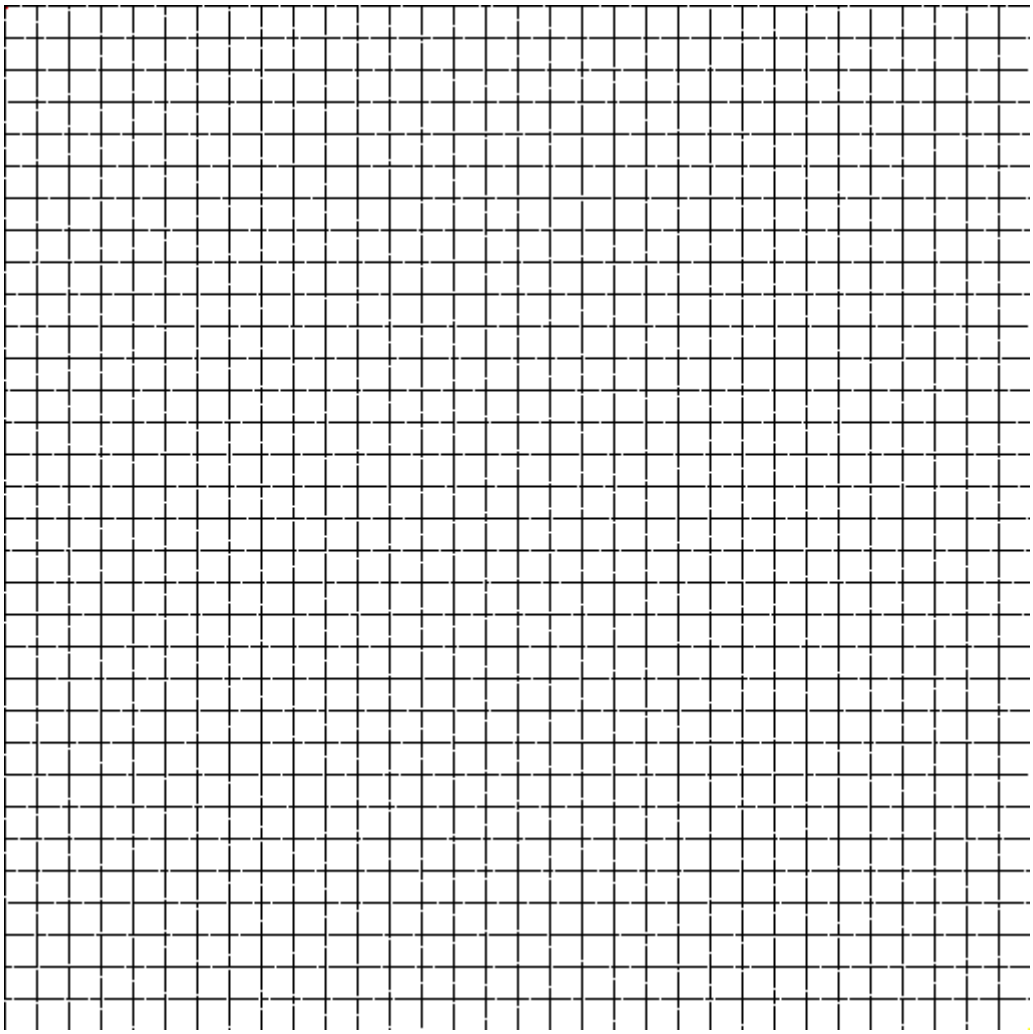

Figure B.2: *16room_000*

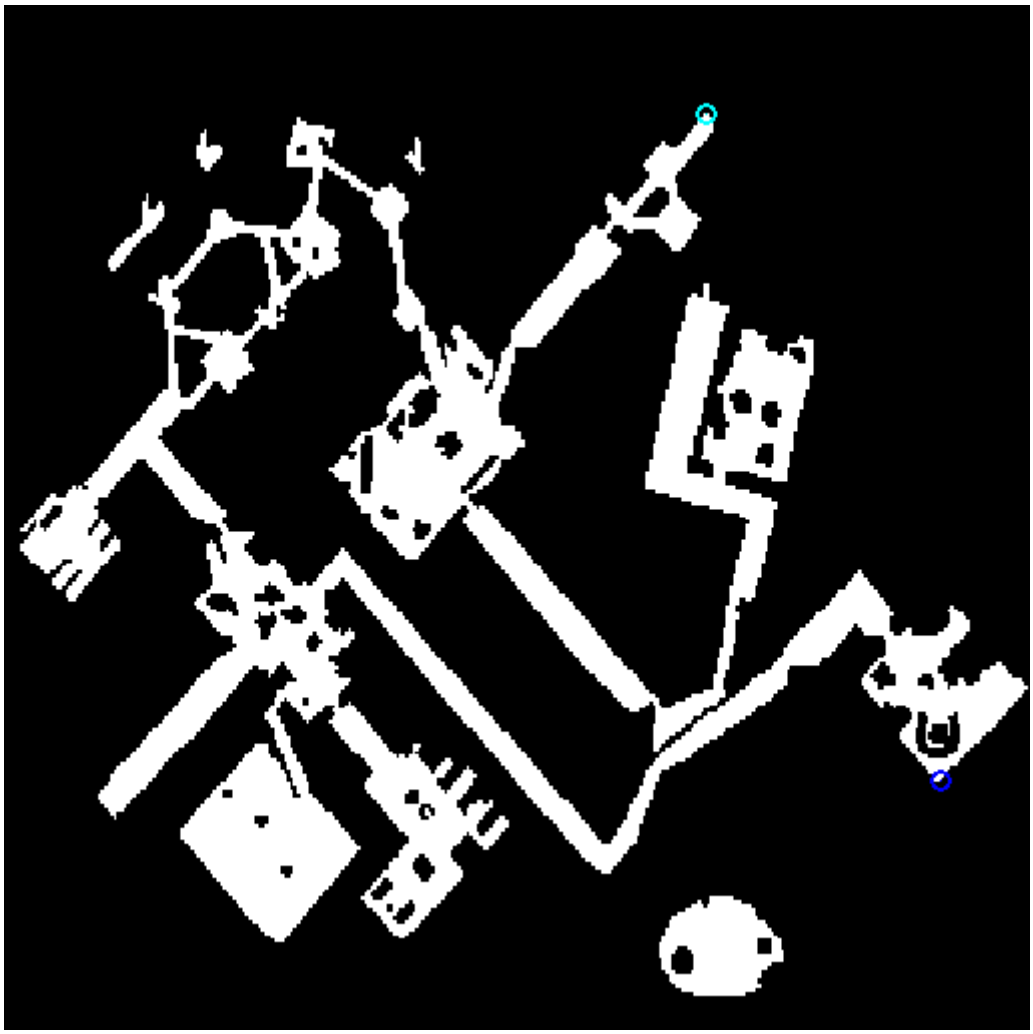

Figure B.3: *AR0307SR*

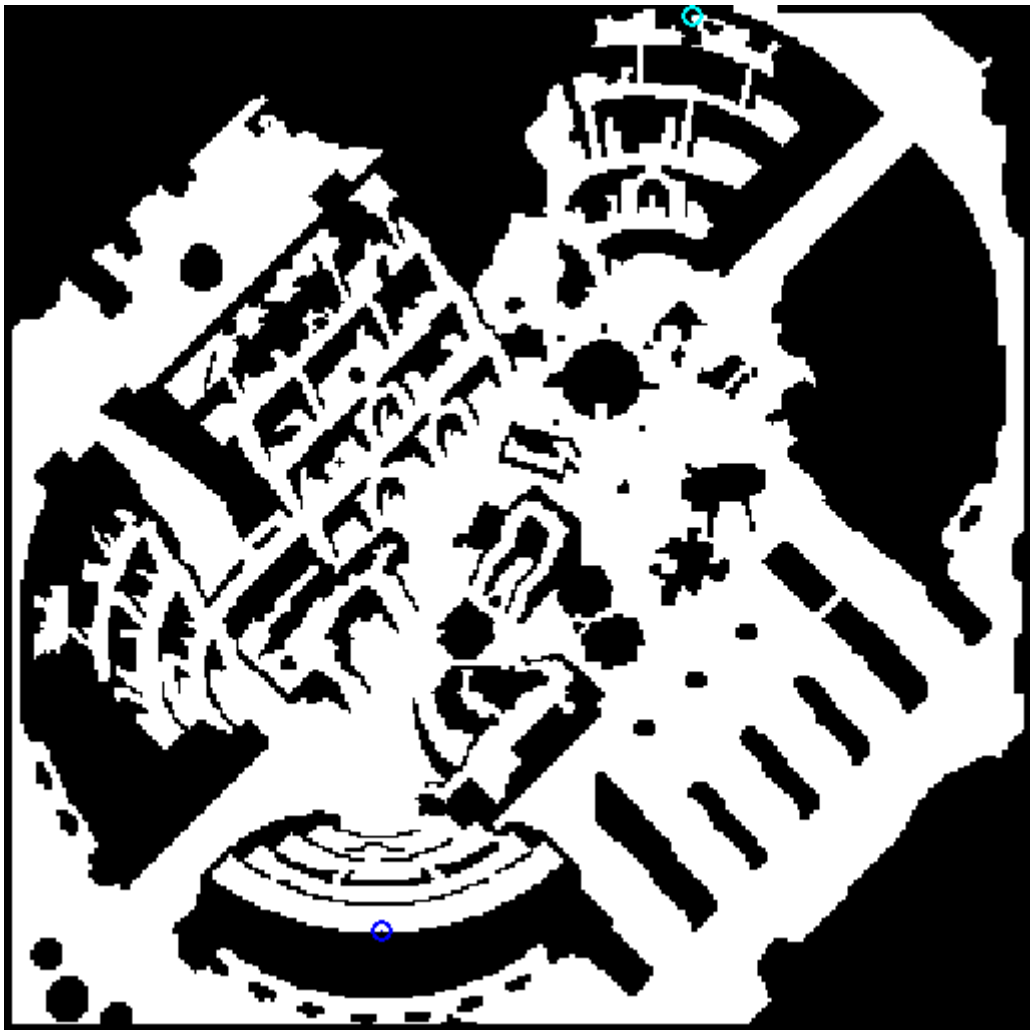

Figure B.4: *AR0700SR*

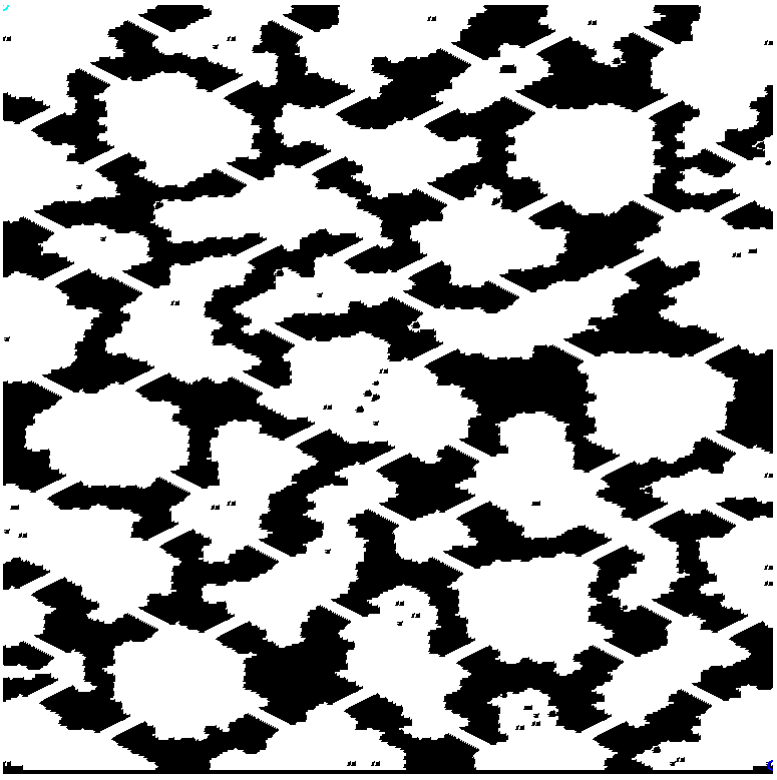

Figure B.5: *Flooded Plains*

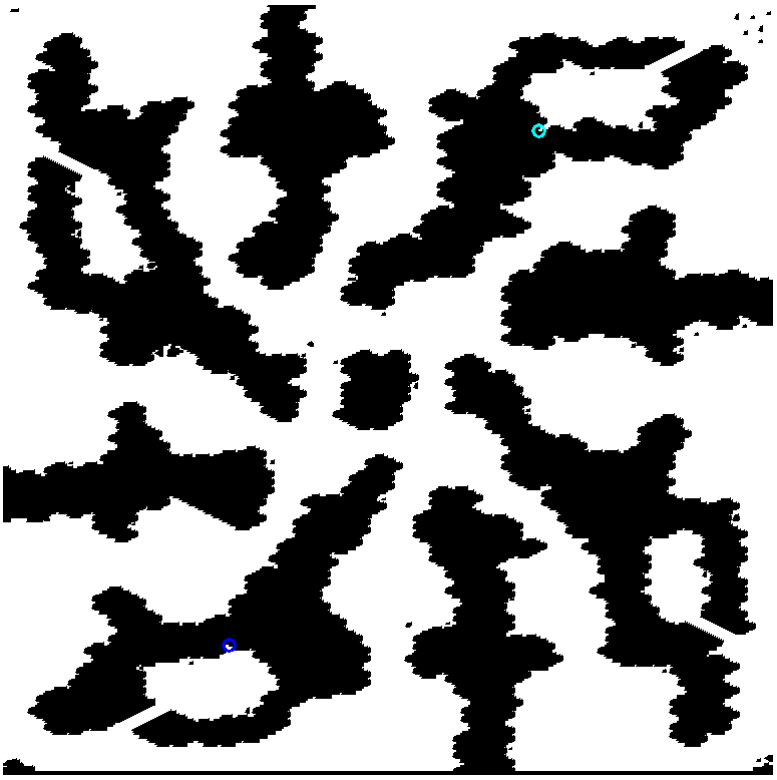

Figure B.6: *Turbo*

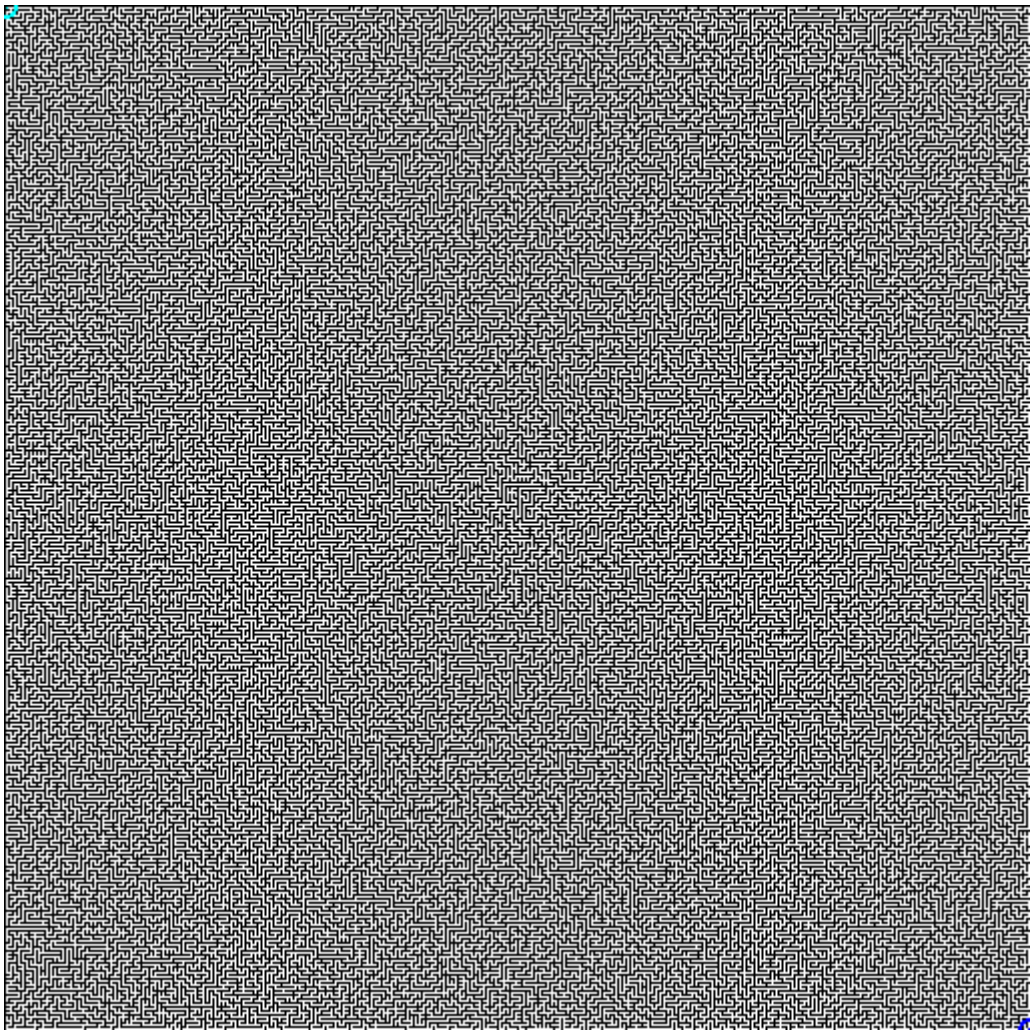

Figure B.7: *Maze512-1-1*

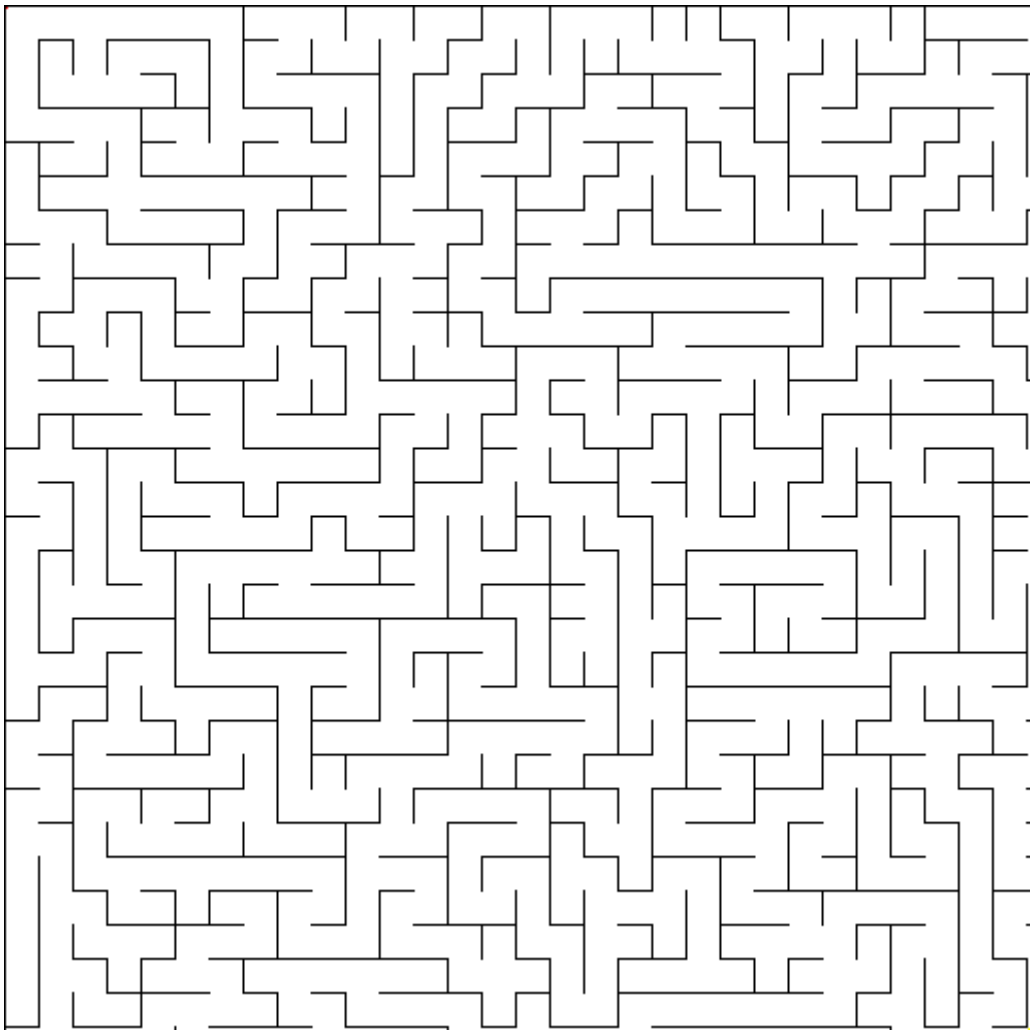

Figure B.8: *maze512-16-0*

Figure B.9: *Randomized32x32-30-0*

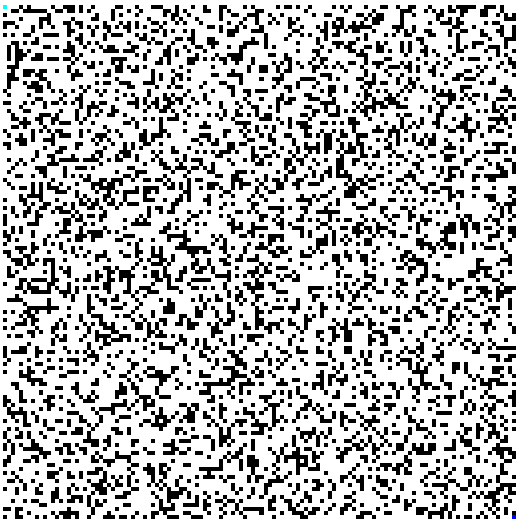

Figure B.10: *Randomized128x128-30-0*

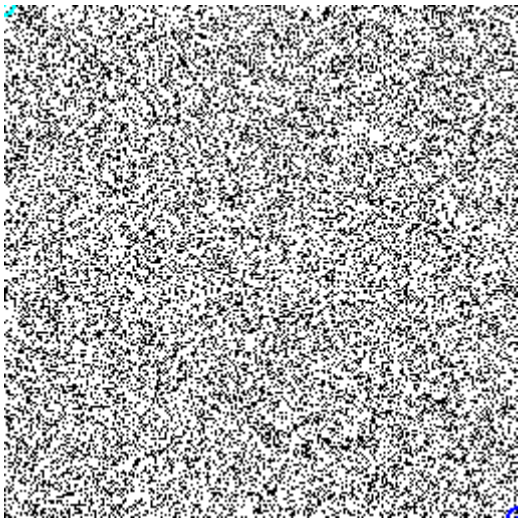

Figure B.11: *Randomized256x256-30-0*

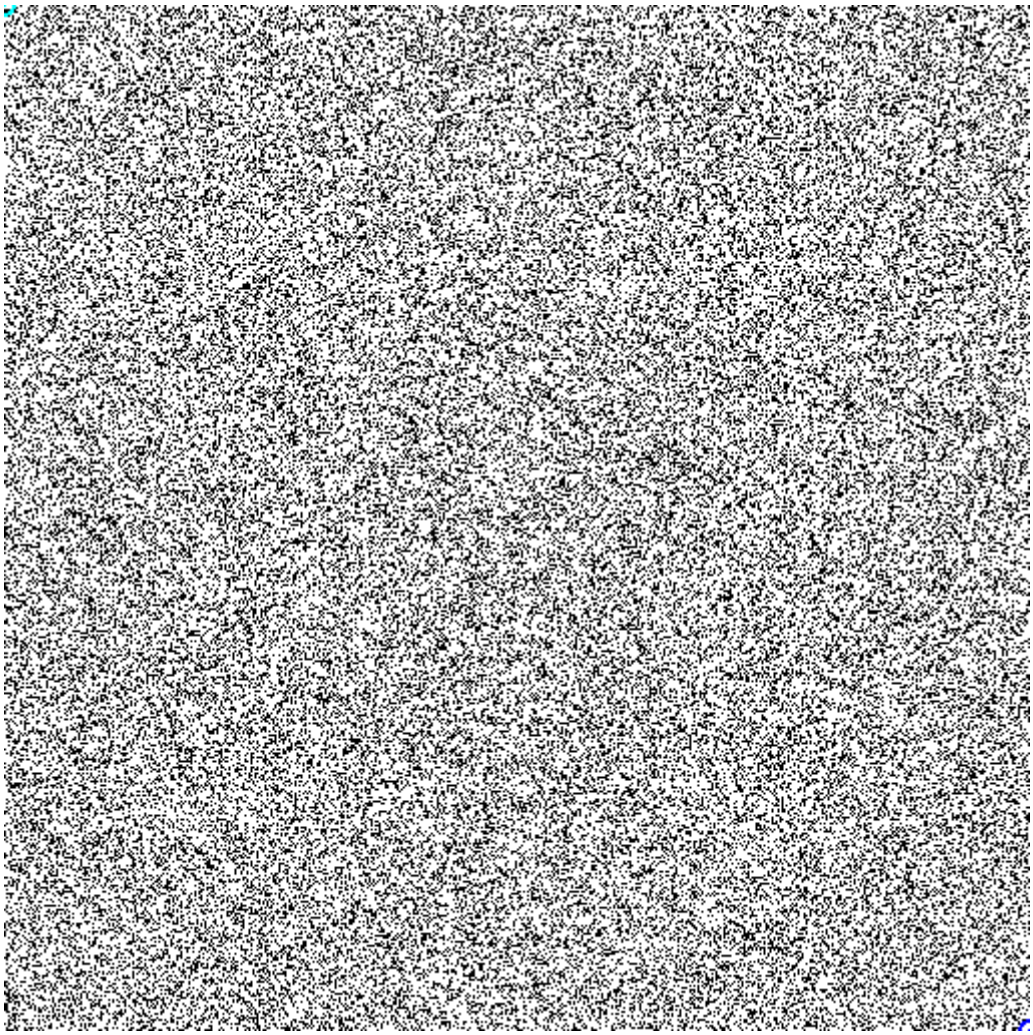

Figure B.12: *Randomized512x512-30-0*

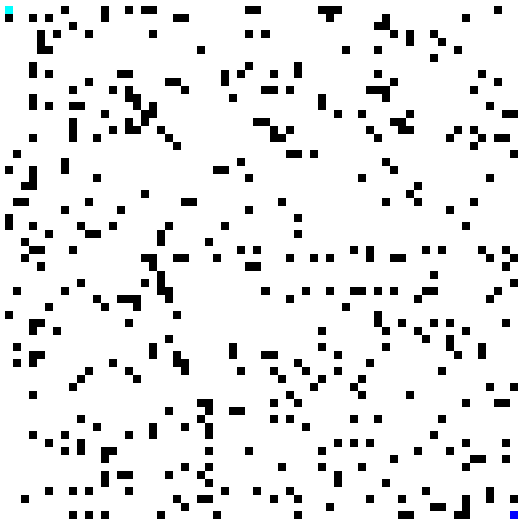

Figure B.13: *Randomized64x64-10-0*

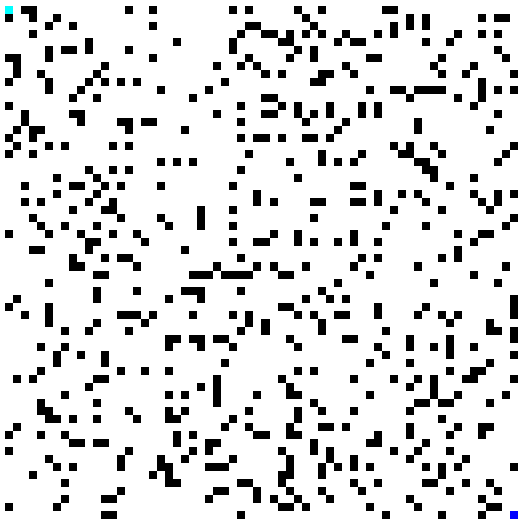

Figure B.14: *Randomized64x64-15-0*

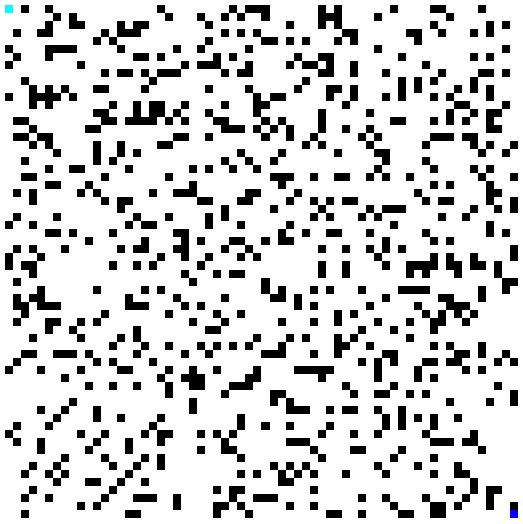

Figure B.15: *Randomized64x64-20-0*

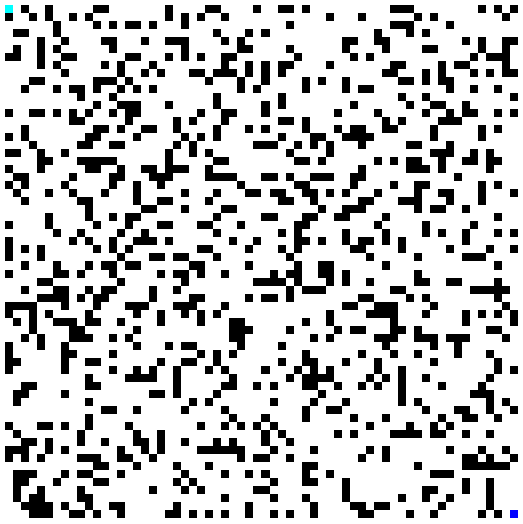

Figure B.16: *Randomized64x64-25-0*

Figure B.17: *Randomized64x64-30-0*

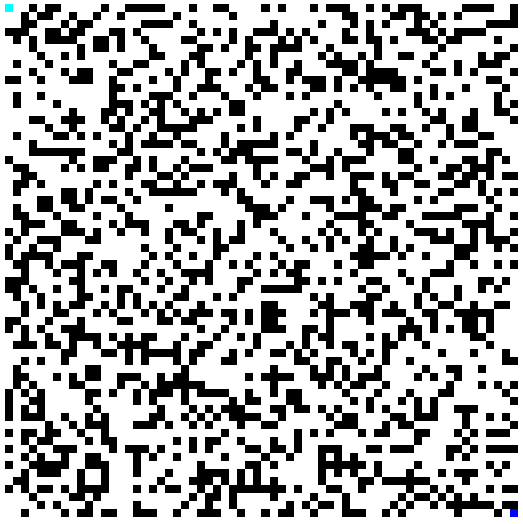

Figure B.18: *Randomized64x64-35-0*

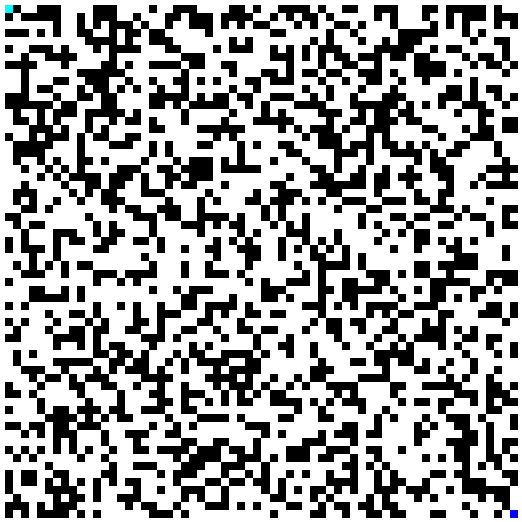

Figure B.19: *Randomized64x64-40-0*

Figure B.20: *Randomized64x64-45-0*