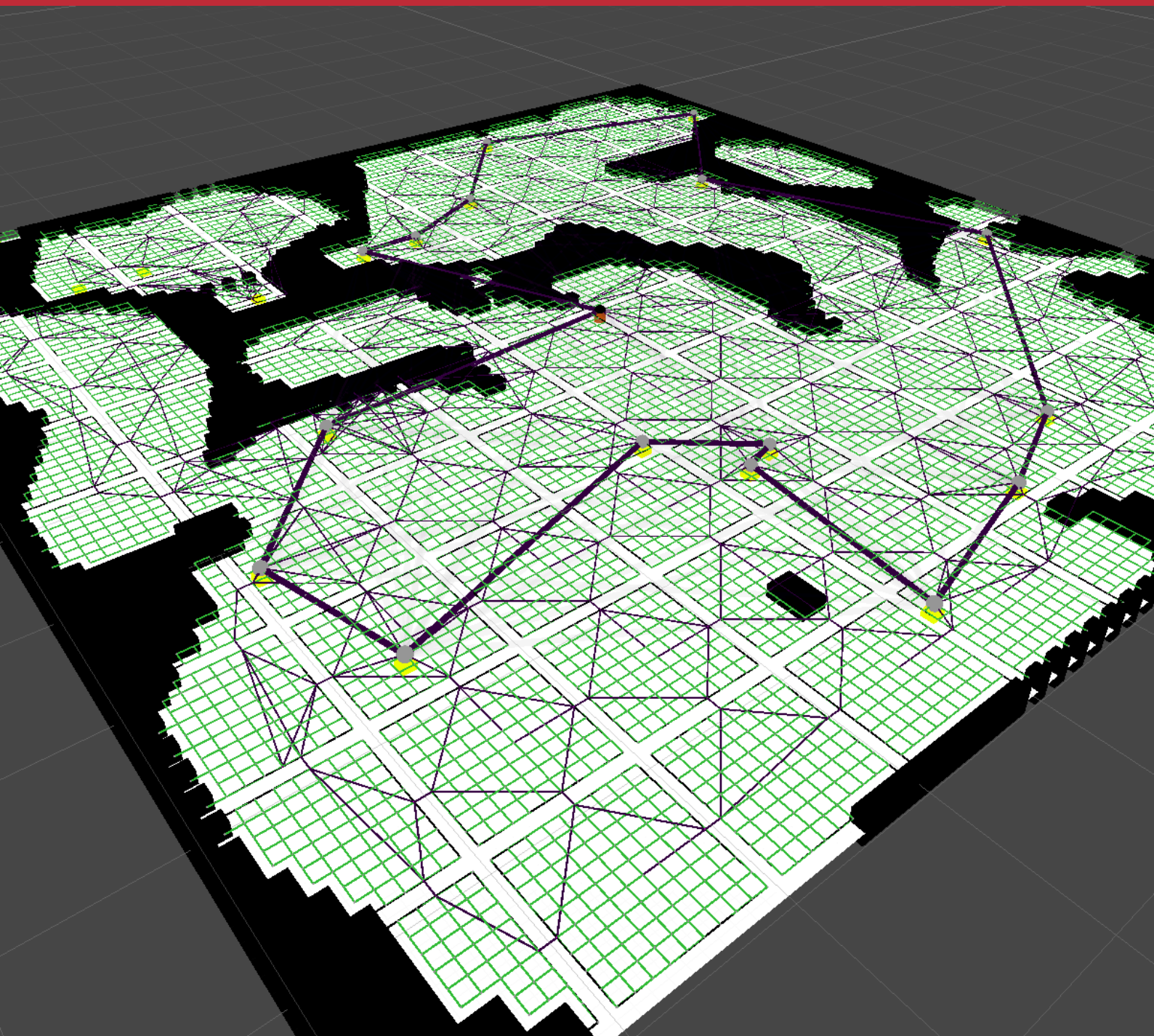
DTU

DTU Compute

Implementation and Evaluation of frontier based exploration and heuristic path finding in 2D Environments

Bachelor Project



Implementation and Evaluation of frontier based exploration and heuristic path finding in 2D Environments

Github repository:
https://github.com/Andreas-Sjogren-Furst/Pathfinding_With_Evolutionary_Agents

Final Release:
https://github.com/Andreas-Sjogren-Furst/Pathfinding_With_Evolutionary_Agents/releases/tag/publish

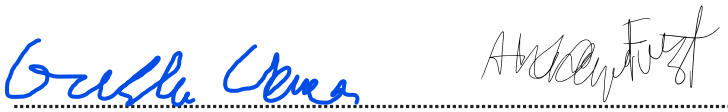
Bachelor Project
June,2024

By:
Gustav Clausen and Andreas Furst

Published by: DTU, 2800 Kgs. Lyngby Denmark

Approval

Gustav Clausen and Andreas Furst - s214940, s201189



Signature

7/6/2024

Date

Abstract

The purpose of this study is to implement and evaluate frontier based exploration and heuristic path finding in 2D Environments. We have developed a framework that implements Cellular Automata for map generation, Frontier Based Exploration, Shadow Casting, A Star (A*), Hierarchical Path finding with A* (HPA*) and the Max Min Ant System. Each algorithm is tested and visualized in our random maps, which are similar to grid based environments in computer games. Key studies are made on the algorithms ability to adapt to the maps. For this we evaluate DFS and BFS search strategies for frontier based exploration, and conclude DFS strategies performs best. Moreover, an simple implementation of multi-agents for exploration, shows that 4 agents are optimal. The shadow casting implementation shows that converting square walls to diamond shaped walls, results in significant improved field of view from each scan. For heuristic path finding, we examined both point to point search, and combinatorial optimization via a conversion of our problem to the symmetric travelling salesman problem (TSP) and Dynamic TSP. In point to point search, we evaluate the nodes explored and path accuracy with A* and HPA* on different abstraction levels, and our study shows that A* explores 5 times more nodes than HPA* on refined low level paths, while HPA* results in less accurate paths. The computation of path lengths are studied as heuristics in the MMAS between cities in the TSP, and our results shows that the HPA* abstraction level yields optimal best tour in terms of convergence and tour length compared to other heuristics. Further, we study techniques of adapting the MMAS to dynamic TSP instances, and we find that the complete reuse of the previous heuristics entirely without normalisation is best, and it performs significantly better than a static re-computation of the whole problem.

Contributions

Creators of this bachelor project.

[Gustav Clausen], [s214940], [Bsc. software technology, DTU]

[Andreas Furst], [s201189], [Bsc. software technology, DTU]

Section	Gustav Clausen	Andreas Furst
Abstract	X	X
Acknowledgements		
1 **Introduction**	X	X
2 **Background:**		
Cellular Automata	X	
A star (A*)	X	
Hierarchical Path Finding (HPA*)	X	
Ant Colony Optimization	X	
Symmetric Shadow Casting		X
Frontier Based Exploration		X
Design		
Introduction		X
Design		X
Scout Agent		X
User Interface		X
3 **Implementation**		
HPA* Implementation	X	
A* Algorithm	X	
Maze Generation	X	
Max Min Ant System	X	
Analysis:		
Optimization Impact on A* Complexity	X	
Approximated Time Complexity of MMAS	X	
Parameter Setting of MMAS	X	
Hierarchical Path Finding Complexity	X	
Evaluation and Results:		
Dynamic Changes in TSP Graph and MMAS	X	
Influence of Heuristics in MMAS	X	
HPA* Efficiency and Accuracy	X	
Frontier Based Exploration	X	
Conclusion	X	X
Product Implementation:		
Implementation: Frontier Based Exploration		X
Implementation: HPA* and visualisation	X	
Implementation: MMAS and visulisation	X	
Implementation: A* and visulisation	X	
Implementation: CA and visulisation	X	
Implementation: Symmetric Shadow Casting		X
Implementation		

Table 1: Contributions of Gustav and Andreas

Contents

1 Introduction

To make computer games feel truly alive the need for sophisticated non-playable characters (NPC) are often a necessity. The most important part of making a good NPC lies in the ability to perceive, navigate and understand the environment. To solve this problem a broad range of algorithms needs to play together to create and develop the complexity of these NPC's. In this bachelor thesis, we investigate the "Implementation and Evaluation of frontier based exploration and heuristic path finding in 2D Environments" Our primary aim is to develop and evaluate a complete framework that enables autonomous agents to explore and navigate two dimensional (2D) grid environments efficiently. This will be achieved by implementing and evaluating algorithms like A Star (A*) [1], Hierarchical Path Finding A* (HPA*) [2], and the Max-Min Ant System (MMAS)[3] to create agents capable of both frontier-based exploration and optimized pathfinding. Furthermore, complex routing problems that arise in the dynamic travelling salesman problem (TSP) will be solved by the use of Max-Min Ant System (MMAS) [4]. To test the capabilities of these agents a range of 2D grid based maps have been created using algorithms in the field of cellular automata. These maps differ in complexity from being open maps with few obstacles to high density maps like cave structures. [5]

This project features implementation and visualisation of the algorithms, allowing real time observation of how the algorithms react to the environment. The implementation of the algorithms is decoupled from the Unity Visualisation engine.

Our report is structured as follows: in the background section we give an overview of each problem, and algorithm we use to solve this problem. In addition, we give an comparative analysis to similar research and literature. The implementation includes optimizations, and specific adaptions we have made to the algorithms and visualisations. We then make an analysis of the selected algorithms, where we focus mainly on effectiveness and computational complexity. In the evaluation and results section, we examine, our specific research questions, which includes evaluating the nodes explored by HPA* compared to A*, the shortest paths. We then evaluate how different heuristics can be applied to the MMAS for optimal tours between checkpoints, same analogy as cities in the TSP, in our environment with A*, HPA* and geometric heuristics. We then modify our MMAS to adapt to a dynamic TSP, and evalute how it performs against a static recomputation, and normalization technique. [6] Two different exploration strategies will be compared, the first being a depth first search (DFS) and breath first search (BFS). Furthermore, an analysis of convergence rate will be made for single and multi agent environments. Finally, the optimal amount of agents will be found.

2 Background

2.1 Cellular Automata

2.1.1 introduction

The dynamic map generation is an important element of analysing the capabilities of our agents. We will introduce the concept of procedural map generation, of how to generate unpredictable maps. The automatic generation of maps allows us to evaluate the agents closer to real world unseen enviorments, and quickly tests different scenarios. The algorithm is mainly based on the Cellular Automata algorithm, which is very ”low computational cost, permitting real time content generation, and the proposed map representation provides sufficient flexibility with respect to level design.” [5]

2.1.2 Alternative algorithms

The main reason for choosng the Cellular Automata algorithmic design for our implementation is that it can generate both open maps with few obstacles, and highly dense map with many obstacles and ”hidden” caves. We did also consider implementing mazes as map generation, here one could use randomized DFS, or randomized minimum spanning tree algortihms such as Kruskal or Prim’s Algorithm [7] However, it is important to note that the solution of path finding in this project is independent of the underlying map, as long as it can be converted to a grid like structure.

The initial noise generations can be done in several ways, such as Random Noise, Perlin Noise, Simplex Noise each method will create different conditions, and will affect the structure of the Map. [8]

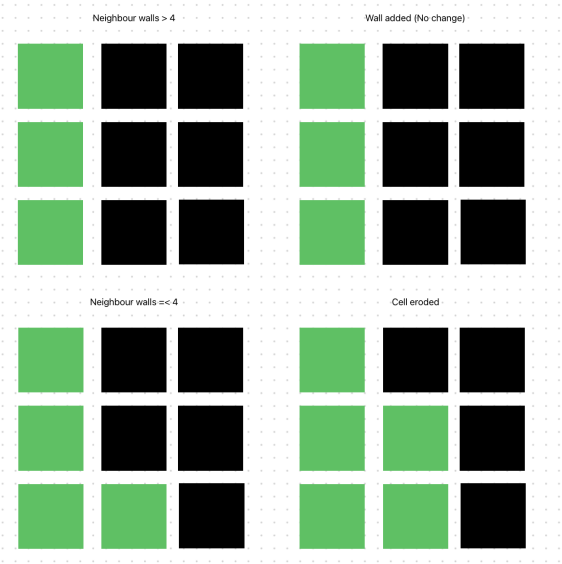


Figure 2.1: illustration of how our erosion works with threshold of >4 neighbours

2.1.3 how CA works

The automata for map generation consists of a grid of cells with two states walkable or non-walkable.

The paper ”Cellular automata for real-time generation of infinite cave levels” by Lawrence Johnson, Georgios N. Yannakakis and Julian Togelius [5] leverages the concept of cellular automata to develop a method for procedurally generating level maps for an infinite cave game, focusing on creating a system that can self-organize to produce playable and well-designed tunnel-based maps.

The algorithm starts with a random noise grid, and then applies iterative CA rules to form the structure of so called caves. Cells in the grid are classified with properties such as floor or wall. By adjusting parameters such as the initial wall cell ratio, the number of CA iterations, the threshold for a cell to be considered a wall, and the size of the neighborhood around each cell, diverse map structures can be achieved dynamically, and due to randomness of the intial map before the rules are applied, it forms unpredictable structures. The eroision step of the algorithm is outlined in 2.1, as currently the only rule used.

An essential aspect of this approach is the ability to represent the map generation through parameters that can be easily manipulated, so one can easily choose between different map types.

Below is explained in pseudo code our adaption of the CA algorithm, it can also be seen as a erosion technique.

2.1.4 Our adaption of the CA algorithm

Algorithm 1 Our adaption of the Cellular Automata Algorithm for 2D Map Generation

```
1:  function GenerateNoise(map, density)
2:      for row ← 0 to height-1 do
3:          for col ← 0 to width-1 do
4:              randomValue ← Random(0, 101)
5:              map[row][col] ← (randomValue > density) ? 0 : 1
6:          end for
7:      end for
8:      return map
9:  end function
10: function CellularAutomaton(map, iterations, erosionLimit)
11:     for i ← 0 to iterations-1 do
12:         tempGrid ← Copy(map)
13:         for j ← 0 to height-1 do
14:             for k ← 0 to width-1 do
15:                 neighborWallCount ← 0
16:                 for y ← j-1 to j+1 do
17:                     for x ← k-1 to k+1 do
18:                         if (x ≥ 0 AND x < width) AND (y ≥ 0 AND y < height) then
19:                             if not (y = j AND x = k) then
20:                                 if tempGrid[y][x] = 1 then
21:                                     neighborWallCount ← neighborWallCount + 1
22:                                 end if
23:                             end if
24:                         else
25:                             neighborWallCount ← neighborWallCount + 1
26:                         end if
27:                     end for
28:                 end for
29:                 map[j][k] ← (neighborWallCount > erosionLimit) ? 1 : 0
30:             end for
31:         end for
32:     end for
33:     return map
34: end function
```

2.2 A star (A*)

2.2.1 introduction

To ensure the maps generated are navigable, and to evaluate the effectiveness of our agents ability to achieve its goals within these dynamic enviorment, we integrate the A* search algorithm. The A* algorithm was proposed by Peter Hart, Nils Nilsson and Bertram Raphael. The algorithm is widely used for shortest paths between two points in known graphs, and we will use it for validation of map navigability and optimizing and evaluating agent path finding strategies. [1] [9]

2.2.2 Comparative analysis of related research to path finding between two points in known graphs

We choose to implement A* since it has a good average time complexity, and in our project it will be useful for several use cases, such as using the algorithm on higher abstraction levels, and thus reducing the total nodes explored significantly.

We did also consider implementing other algorithms. The Dijkstra's algorithm will always explore all nodes and edges [10], and this will be very slow in our 2D enviorment with 4 edges per node, in a $N * N$ nodes grid.

Alternatively,the DFS and BFS are very simple to implement but DFS does not guran-tee the shortest path, and the BFS is too memory intensive [9]. Bidirectional runs two searches from the start and goal node and meet in the middle, which reduces the time complexity significantly due to the branching factor. [11], this bidirectional search could be good to reduce time complexity, however, it also increases the overall complexity of implementation.

Theta* seems as a good addition to the A* algorithm [12], as it uses line of sight when conencting nodes on the shortest path, to create a more smoothed and shorter path since the agent can only work in 4 directions, this would give a better visual motion, however, due to the increased time-complexity this is not considered for our project.

The Dynamic A (D*) could be good for our project since, we may have dynamic changes to the enviorment and thus recomputing the path could be very expensive [13], however in a later section, we will describe the higher abstraction levels, and then the recomputation will be quite minimal depending on the amount of changes in the map, since it is done at cluster level.

The Jump point search (JPS) [14] offers speed advantages in uniform grid-based envi-ronments by ”jumping” over nodes and thus reducing computational overhead, but lacks versatility.

Overall, the A* is a roboust algorithm well suited for our project, however, for very dynamic scenarios it lacks compared to D* .

2.2.3 How A* works

The A* introduces heuristics in Path finding. Compared to the well known Dijkstra's algorithm which always finds the shortest path, as long as there is no negative weights, since it examines all nodes, it is an exhaustive approach. The A* uses the Heuristic function and examines only the nodes that looks better, and therefore improves the computational efficiency significantly when a path exists. The A* guarantees to find the shortest path, and to be optimally efficient, if it is admissible and consistent. [15]

A* Algorithm implementation Overview In our implementation we used the Pseudo code proposed on the paper [15] as the overall structure, but we have clarified the algorithm below:

Algorithm 2 A* Algorithm

```
1:  procedure AStar(startNode, targetNode)
2:      add the startNode to the open list
3:      while the open list is not empty do
4:          current ← node from open list with the lowest f value
5:          move current from open to closed list
6:          for each node reachable from current do
7:              if node is in the closed list then
8:                  continue to next node
9:              end if
10:             if node is not in the open list then
11:                 add node to the open list
12:                 set current as parent of node
13:                 calculate f, g, and h values for node
14:             else
15:                 if current path to node through current is better then
16:                     change parent of node to current
17:                     recalculate f and g values for node
18:                 end if
19:             end if
20:         end for
21:         if current is targetNode then
22:             break from the loop
23:         end if
24:     end while
25:     if targetNode is in closed list then
26:         trace path back from targetNode to startNode
27:     else
28:         return failure
29:     end if
30: end procedure
```

The A* algorithm operates by maintaining two lists. An open list of nodes to be evaluated and a closed list of nodes already evaluated. Each node represents a possible state in the search for the shortest path, once all possible states is found for a node it is moved to the closed list. The open list is also called the frontier. For our implementation, nodes are defined with properties including position, cost from moving start to a node (gCost), the cost of moving from one node to another is 1, since the agent can only move in 4 directions. The estimated cost to the end node is often denoted as heurtics Cost (hCost),

and the total cost fCost is the sum of gCost and hCost for each node. [1]

Admissibility and Consistency is two very important notions when working with the A*. Since if the heuristics is admissible, it will always find the shortest path. Admissibility is formally defined as:

$$h(n) \leq h^*(n) \quad \text{for all nodes } n,$$

where $h^*(n)$ is the optimal cost to reach the goal from node n. This is also informally defined as the heurtics may never overestimate the cost of reaching the goal node.

If the heuristics are consistent, the algorithm is said to be optimally efficient, it will always explore the optimal set of nodes. Consistency is informally defined as the f-cost should be non-decreasing along any path. [16], and formally defined as, for every node n and every successor m of n generated by any action a,

$$h(n) \leq c(n, a, m) + h(m),$$

where $c(n, a, m)$ is the step-cost of moving from node n to node m via action a. It also important to that if the A* is consistent then it must also be admissible, but not vice-versa. [9]

2.3 hierarchical path finding (HPA*)

2.3.1 introduction

We considered several ways of reducing the amount of nodes and edges in the underlying graph representation of the Map. Firstly, our underlying graph representation was that each tile is a node, and from each tile there are 4 edges, allowing the agent to move in 4 directions from each tile. This creates a huge graph and search space to find the shortest path between two points. Since we are computing the optimal paths each time an agent moves, and also to determine the paths lengths between checkpoints for MMAS, then it is essential to reduce the search space to achieve near optimal results with less computational power. We considered the Navigation Meshes, Waypoint Graphs and Near optimal hierarchical path-finding (HPA*). [2] In our choice of graph construction we based our decision on implementation complexity, option for dynamically changing the graph without rebuilding everything, and the results from research.

Finally, we choose to implement HPA* as this has been experimentally evaluated to be very efficient. In the case of a dynamic environment with shared memory between agents, dynamically updating the graph is very efficient, as only certain requires re-computation, which can be matched to the agent's vision area. The experimental results from the paper [2] resulted in a 10 fold reduction in the expanded nodes. This will allow for significantly faster computation times when using A* for point to point search. Moreover, we can use the higher abstraction levels to compute estimated path lengths to the MMAS.

2.3.2 Releated research on Graph Abstraction Algorithms for point to point search

In this section, we will elaborate on our considerations of other algorithms for reducing the search space for A*, we primarily had the dynamic adaption capabilites, adaption for grid environments, implementation difficulity, preprocessing time, complexity in handling the nodes, and overall reduction of the search space with the algorithm. It is also important to keep in mind, that the graph should be efficient in collaboration with A*, and should in the end still maintain a real walkable path for the agent.

The contraction hierachies algorithm by Robert Geisberger et al, is said to be efficent in preprocessing, query time and generally good for static problems. It works by contracing nodes according to their heuristics, contracting nodes will remove them from the graph, and create a shortcut with an edge instead. This is very effiency for large static graphs such as road networks. It is not exactly designed for path refinement, but one could save the refined path in the shortcuts [17]

Another consideration was the Navgation Mesh (NavMesh) algorithm described by Tozour and Austin in "Building a near-optimnal navigation mesh" which covers walkable areas with convex polygons and builds a near optimal coverage faster by relaxing minimal polygon conditions. This is good in 2D maps and 3D maps in mainly static or semi-dynamic environments. [18]

An important consideration for the chosen HPA* algorithm, is its ability to handle clusers with internal obstacles, where other similar algorithms cannot. An example of a similiar algorithm is the Quadtrees, which is also way of doing hierachial map decomposition, however, if a cluster contains both obstacles and walkable tiles, then it is further decompositionen into smaller clusters. This will also make the agent walk to the middle of the cluster rather than the exact point desired. Another proposition to this is the framed quadtrees, which also uses clusters crossings as abstraction points, but HPA* does not consider all cluster crossings as abstraction points, which significantly reduces the search

space, this is called ”entrances” in HPA* algorithm . [2]

The difference between inter and intra edges in the HPA* algorithm ensures A star doesn't exhaust the whole search space, if there is no path. The HPA* algorithm also makes it possible to make change to a single cluster without it affecting other clusters. The implementation complexity seems more complex with HPA* than the other algorithms.

2.3.3 How the HPA* works

The strategy of the algorithm is to make A* find the optimal path first at a higher abstraction level thus reducing the search space. The Authors of the Paper ”Near optimal hierarchical path finding (HPA*)” [2] associates the algorithm with the real-world analogy of planning a car trip from one city to another. Rather than navigating the whole trip through every local street and road from start to finish, an human planner would intuitively plan the journey at a higher abstraction level by overall first assessing the major regions of going through, and then choosing the highways, and then local roads in between. As an computational example, one could imagine a 1000x1000 grid, then one could divide it into sub grids of 10x10 the it would reduce the search space to 100x100 grid, we will call these grids at a higher abstraction level for clusters. The A* algorithm can then recursively work through the abstraction levels (clusters) to a low level path of actual coordinates. This process involves pre-processing of the grid and then an on-line serach with HPA*. The-processing is the creation of the abstract problem graph in clusters, the clusters are then analyzed to determine entrances - obstacle free segments along the border that connect adjacent clusters. These entrances can also be seen as pathways that allow moving from one cluster to another. The HPA* algorithm then creates edges for each of these transitions and defines two types of edges $intraedges$ and $interedges$. $Intra-edges$ which is edges within the cluster between path ways, and inter-edges which is used when navigating between clusters. The weight of the inter-edges is the cost of moving inside the cluster, and is computed with optimal path search within the clusters' bounds. The online then consists of three-processes:

- Local pathfinding within start and goal clusters to find the optimal paths to the borders respectively.
- Abstract level search performs search at the abstract level to determine the best route between clusters containing the start and goal location.
- Path refinement which is refining the abstract path to a detailed lower level of abstraction, which ensures the agent navigates correctly in the environment without hitting walls etc.

2.3.4 Psuedo code of building the abstract graph

The abstraction levels in respective leveled clusters is called l-cluster, where l is the level number, each increased level leads to computing clusters of superior levels. So in $AbstractMaze(), c[1]$ is the set of 1-clusters, E is the set of all entrances defined for the map in that level:

The $buildGraph()$ creates the abstract graph by firstly creating the nodes and inter-edges, and then the intra-eges. The $newNode(e, c)$ creates a node contained in cluster c at the middle of the entrance e as we only assume one transition per entrance. The methods $getCluster(e, l)$ returns the adjacent l-clusters at same level by connected entrance e. The $addNode(n, l)$ add node n to graph l abstraction level, the $addEdge(n_1, n_2, w, l, t)$ adds an edge between node $n1$ and $n2$ with weight w , level l and $type$ of the edge which is defined as $t \in \{inter, intra\}$ The second $forloop$ of $BuildGraph$ adds intra-edges,

Algorithm 3 Abstract the maze into clusters and entrances

```
1: procedure AbstractMaze
2:     E ← ∅
3:     C[1] ← BuildClusters(1)
4:     for each c1, c2 ∈ C[1] do
5:         if Adjacent(c1, c2) then
6:             E ← E ∪ BuildEntrances(c1, c2)
7:         end if
8:     end for
9: end procedure
```

where the methods $SearchForDistance(n1, n2, c)$ searches for shortest path between two nodes and returns the cost.

Algorithm 4 Build the abstract graph for hierarchical search

```
1: procedure BuildGraph
2:     for each e ∈ E do
3:         c1 ← GetCluster1(e, 1)
4:         c2 ← GetCluster2(e, 1)
5:         n1 ← NewNode(e, c1)
6:         n2 ← NewNode(e, c2)
7:         AddNode(n1, 1)
8:         AddNode(n2, 1)
9:         AddEdge(n1, n2, 1, 1, INTER)
10:    end for
11:    for each c ∈ C[1] do
12:        for each n1, n2 ∈ N[c], n1 ≠ n2 do
13:            d ← SearchForDistance(n1, n2, c)
14:            if d < ∞ then
15:                AddEdge(n1, n2, 1, d, INTRA)
16:            end if
17:        end for
18:    end for
19: end procedure
```

The abstract levels to graph are added incrementally. Therefore an added level is an abstraction of the previous level $l-1$, and therefore the building of level l is only possible when $l-1$ is complete. In the $AddLevelToGraph(l)$ we group the clusters into new clusters into a set of l-clusters $C[l]$ and finally the new $INTRA$ edges are added to the graph at the level.

Finally we can define the pre-processing of the $HPA*$ algorithm where $PreProcessing$ builds the graph according to our previous description in section 4.1

Algorithm 5 Add a new level to the hierarchical graph

```
1:  procedure AddLevelToGraph(l)
2:      C[l] ← BuildClusters(l)
3:      for each c1, c2 ∈ C[l] do
4:          if not Adjacent(c1, c2) then
5:              continue
6:          end if
7:          for each e ∈ GetEntrances(c1, c2) do
8:              SetLevel(GetNode1(e), l)
9:              SetLevel(GetNode2(e), l)
10:             SetLevel(GetEdge(e), l)
11:         end for
12:     end for
13:     for each c ∈ C[l] do
14:         for each n1, n2 ∈ N[c], n1 ≠ n2 do
15:             d ← SearchForDistance(n1, n2, c)
16:             if d < ∞ then
17:                 AddEdge(n1, n2, l, d, INTRA)
18:             end if
19:         end for
20:     end for
21: end procedure
```

Algorithm 6 Preprocessing to initialize the hierarchical search

```
1: procedure Preprocessing(maxLevel)
2:     AbstractMaze
3:     BuildGraph
4:     for l ← 2 to maxLevel do
5:         AddLevelToGraph(l)
6:     end for
7: end procedure
```

2.4 Ant Colony Optimization

2.4.1 introduction

In our project we address the shortest path finding problem in our 2D enviorment with checkpoints, as to effienctly navigate an agent through all checkpoints in the least amount of steps. Hereby we can convert our problem to the same analogy of solving the Traveling Salesman Problem (TSP) where each checkpoints represents a town, our focus will be on the optimization of the route through each checkpoint, visited exactly once, making it a symmetric TSP scenario at firsthand. The paths between the checkpoints will be estimated using heuristics. The effectiveness of different heuristics will be investigated in section 6.2. In addition to visiting each checkpoint, we will investigate how checkpoints can be added and removed from the graph, while utilizing heuristics from previous runs, and thus investigating a dynamic TSP scenario in section 6.1. We have decided to use the Max Min Ant System (MMAS) to solve approximated optimal tours between several checkpoints.

Given the computational complexity of solving the TSP through brute-force methods, which escalates factorially with the number of checkpoints ($N!$), using meta-heuristic such as the the Max-Min Ant System (MMAS) becomes a good approach to find a near-optimal solution within reasonable time constraints.

Literature review

We could also consider other algorithms to solve combinatorial optimization problems such as the travelling sales problem. Another algorithm is the Simulated Annealing, which is simple to implement, but mainly made for static problems. [19]

Genetic Algorithms can also solve combinatorial optimization problems, and would work better than simulated annealing for dynamic scearnios, since the geneomes could be modified will still keep the previous state. Directly comparing genetic algorithms and MMAS is difficult, often the MMAS will find a better shortest path than the genetic algorithms. [20]

We considered 3 different ACO algorithms choose to implement MMAS proposed by Hootz [3]. Other than the MMAS it is worth considering the Ant System (AS) originally proposed by Dorigo [21], this further described below, but in short MMAS perform better than AS. Dorigo made an description of this in 2004. The Ant Colony System (ACS) differs mainly from MMAS by applying pheromone on the last edge, and has other update rules. [22] The performance of the MMAS vs ACS is investigated in 2017 by Jangra, and Kait [23], where they conclude ACS is best with their parameter setting tested, however, also conclude it is problem specific, and the MMAS performs well in dynamic environments according to the case study by Mavrovouniotis et al in 2020 [4]

ACO (Ant Colony Optimization) introduction

Ant colony optimization was originally introduced by Marco Dorigo in 1992 [21]. It is an bio-inspired evolutionary algorithm by the behavior of real ants. [24] The main idea with the algorithm is the indirect communication between ants based on the chemical substance, pheromones, which ants lay on their paths, this is also known as stigmergic information. The stigmergic information ensures, that even though the ants have limited individual conginitve capabillities, they excel at finding shortest paths to food sources, through the collective intelligence in mainly depositing and detecting pheromones. The main idea is that, the ants which find food faster, will be able to return to the colony faster, and thus depositing more pheromones on their paths, then the next ants starting in the colony, will base their decision on these pheromone levels. A higher level will increase the likelihood of ants choosing that path, which over time the system will statically find a good solution.

[25]

This metaheurtic is used to find near optimal solutions to particular problems such as path finding, or other NP-hard combinatorial optimization problems. The Ant System, introduced by Dorigo was the first attempt to use this indirect communication of ants to construct solutions for NP-hard problems.

2.4.2 Ant system and TSP

Given a set of checkpoints, the travelling sales problem, is the path finding solution to find the shortest path between all checkpoints, while only visiting each checkpoint once and return the initial position, where the weight of the edges is often the length between towns i and j, which in our case is the length of the shortest path, calculated by a heuristic, the TSP is represented as a construction graph $G_n = (N, E)$, where N is the set of checkpoints and E is the set of edges between these checkpoints which forms a fully connected graph, we use the same representation of the TSP as in the paper by Dorigo.[25]

The total number of ants (m) is given by: $m = \sum_{i=1}^{n} b_i(t)$, where $b_i(t)$ is the number of ants at checkpoint i at time t.Then each ant's behavior is given by:

- The next checkpoint is selected based on a probability which considers both the distance to the checkpoint, and the amount of pheromone present on the connecting edge.
- To ensure ants only visit each checkpoint once per tour, the previous transitions are stored by a "tabu list"
- When a tour is completed, it deposits pheromone on the each edge visited.

The intesity of pheromone on edge (i, j) at time t is defined as $\tau_{ij}(t)$. Each at time $t + 1$ decide which checkpoint to go to. We can then define one iteration of the AS to be the in the time interval $(t, t + 1)$ with $m\ ants$ then every n is a cycle of the algorithm, when each ant has a completed a whole tour, the trail intensity of pheromone on edge (i,j) is then updated according to:

$$\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{2.1}$$

where ρ represents the trail evaporation for each cycle in the interval $rho \in [0, 1]$, and with $\Delta\tau_{ij}^{k}$ being the amount of pheromone substance laid on edge (i, j) by the k-th ant, calculated as:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{Q}{L_k}, & \text{if } k\text{-th ant uses edge } (i,j) \text{ in its tour} \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

where Q is a constant and L_k is the length of the k-th ant's tour.

For each ant a tabu list is kept, which contains checkpoints already visited upto time $t)$ and forbids the respective ants to visit them again before a tour has been completed. When a tour has been completed the tabu list is used to compute the ant's current solution, and the tabu list is cleared, for the ant to choose again. The list of all tabu lists is defined as a set $tabu_k$ where it contains the tabu list of each $k - th$ ant, and the $tabu_k(s)$ is defined as the $s - th$ element of the list, which is the checkpoint visited by the k-th ant.

The transition probability for the k-th ant from checkpoint i to checkpoint j at time t is given by:

$$p_{ij}^{k}(t) = \begin{cases} \frac{\tau_{ij}^{\alpha}(t) \cdot \eta_{ij}^{\beta}}{\sum_{k \in \text{allowed}_k} \tau_{ik}^{\alpha}(t) \cdot \eta_{ik}^{\beta}}, & \text{if } j \in \text{allowed}_k \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

where $\text{allowed}_k = \{N - \text{tabu}_k\}$, and α and β are parameters controlling the relative importance of trail intensity versus visibility. The η_{ij} is defined as the visbility, but also known as the heuristics for the ant. Then we can derive that that the probability of a path chosen is a trade-off between heurtics and pheromone.

2.4.3 MAX–MIN Ant System (MMAS)

The Max-Min ant system (MMAS) was proposed by Thomas Stützle and Holger H. Hoos [3], and it signifcantly improves the performance by more effectively exploiting the best solutions found during the search. In the following main areas, the MMAS make a larger difference:

- MMAS has an optimized pheromone update method, it utilizes the best solutions by allowing only one ant to add pheromone after each iteration. This ant could be either the one that found the best solution in the current iteration (iteration-best ant) or the one that has found the best solution since the start of the trial (global-best ant).
- To prevent search stagnation, MMAS has strict limits on the range of possible pheromone trails on each solution component, limiting it to an interval $[\tau_{\text{min}}, \tau_{\text{max}}]$. This interval is also further discussed in our implementation section.
- The pheromones trails initially set to the τ_{max} which encourages exploration of the solution space at the start of the algorithm.

These strategies are geared towards balancing the exploitation of known good solutions with the exploration necessary to discover potentially better solutions. In MMAS, pheromone trails are updated using the solution cost $f(s_{\text{best}})$, where $f(s_{\text{best}})$ denotes the solution cost of either the iteration-best or the global-best solution. The updated pheromone trail rule is:

$$\tau_{ij}(t+1) \leftarrow (1-\rho) \cdot \tau_{ij}(t) + \frac{1}{f(s_{\text{best}})}, \tag{2.4}$$

where ρ is the pheromone evaporation coefficient, and $f(s_{\text{best}})$ represents the solution cost of the best solution. The trail limits particularly prevents the risk of stagnation. Note that the Paper by Hoos et Al [3] defines Convergence in the MMAS on TSP as each of the selected edges part of the best solution has τ_{max} pheromone level, while the others has τ_{min}, where stagnation is where all ants keep following the same path. The paper describes that the τ_{max} should be updated dynamically when new best solutions are found, and the τ_{min} should be determined by experimental investigation. This is further described in our implementation section of the MAMS.

Parameter optimization

The scope of this project is not focused on setting the optimal parameters of MMAS for our problem instance, though we did still spent time on researching the choice of parameters, and later in the analysis section, we will describe on our own conducted experiments on

the Berlin52. We read parts of another DTU bachelor thesis by Emil Lundt Larsen and the paper by Dorgio [22], and where the the optimal parameters are described to be in following range for various tsp instances:

Algorithm	α	β	ρ	m	τ_0
AS	1	2 to 5	0.5	n	$\frac{m}{C^{nn}}$
MMAS	1	2 to 5	0.02	n	$\frac{1}{\rho C^{nn}}$

Table 2.1: Optimal parameter settings for AS and MMAS for the TSP (from Emil Lundt Larsen bachelor thesis [26])

where m is the amount of ants, and τ_0 is the initial pheromone and C^{nn} is a heuristic, which is a greedy strategy, which is often simply choosing the nearest neighbour greedily for each node, and is often a good initial estimate of the tour length.

2.5 Symmetric Shadow Casting

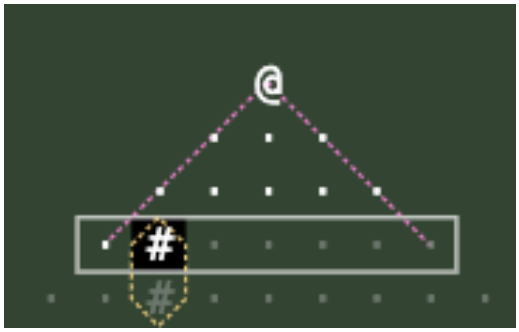
2.5.1 introduction

To solve the problem of exploring an unknown area an implementation of field of view is needed for the agents. There is several useful algorithms which could solve the problem such as Ray casting, Shadow casting, Permissive field of view and Digital field of view. In this implementation a modification of the regular shadow casting algorithm have been used. [27] The main reason being computational efficiency compared to other solutions, since we are using multiple agents and need to calculate field of view in real time. This will be explained in greater details in the following sections.

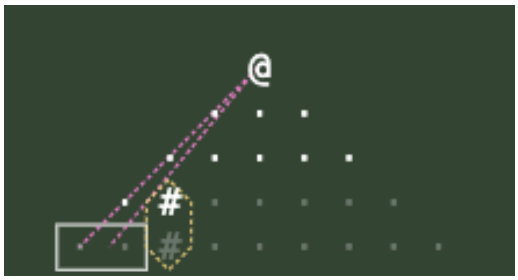
2.5.2 Background

In most cases field of view algorithms works by casting sectors or rays of light from a given light source or position. Then it detects what regions of the area around it that are hit by the light. In our scenario the light source is the position of the scout agent and the area is defined as a 2-dimensional grid with square shaped tiles and walls.

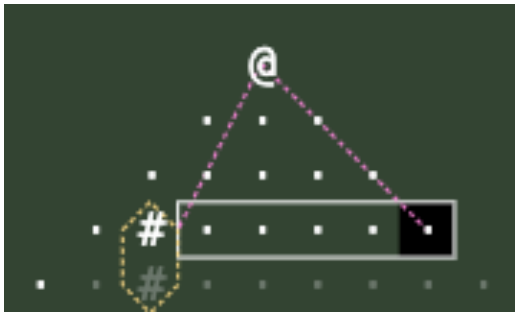
Shadow casting algorithm works by casting circular sectors of light outward from the agent. This is done by dividing the area around the light source in equally sized sub sections, mostly 4 or 8. The lines dividing the sub sections will span out the area that is visible. When the light hits an obstacle the section is then reduced in angle or split into two sectors, which are then processed independently. This will make sure that each tile is only visited once in the original section. In the figures below the initial field of view can be seen adjusted when a wall occurs in the field of view. [27]



(a) Initial field of view



(b) new end slope



(c) new start slope

2.6 Frontier Based Exploration

2.6.1 Introduction

Frontier based exploration is about finding the most efficient way to explore unknown environments.[28] Since environments can have many different shapes and complexities from computer generated to real world environments a wide range of sophisticated exploring strategies has been developed. In this section we will look into how to explore 2 dimensional maps with varying obstacle density. These maps goes from completely open maps without obstacles to dense environments like cave structures. To make the problem more exciting a simple implementation of a multi agent system have been developed. Where multiple agents are working together to explore the environment. The goal is to minimize the total amount of steps the agents has to take to fully explore the map.[29] This will be explained in greater details in the following sections.

2.6.2 Background

The algorithm implemented follows the idea described by Brian Yamauchi in his paper called a ”A Frontier-Based Approach for Autonomous Exploration”.[28] In the paper and in the field of antonymous robots a uncertainty occurs when detecting the area around it.[29] In this implementation we are not taking into consideration this uncertainty since the scanning algorithm implemented in this solution is deterministic and with certainty knows the type of the tile. As described earlier the central idea behind frontier based exploration is to gain the most amount of information about the environment in shortest amount of steps. Where each step is defined as one iteration of the agent. The length of each step is defined as the length of a tile. The agent can only move in four possible directions either north, east, south or west.

The first question to answer is to decide where the agent should place itself to maximize the amount of discovered area from scanning the environment. Since there is a boundary between unknown area and known area the most reasonable strategy will be to place the agent on the border of these areas. The border is defined by the frontier region where a visible tile lies next to a unknown tile. These visible tiles are defined as frontier points and they outline the boundary. Since they outline the boundary a centroid is found from grouping close frontier points together. The purpose of the centroid is to find the location that can gather the most amount of information from the unknown area around the boundary. The agent then navigates to the centroid using path finding algorithm like A* and Hierarchical Pathfinding A*. When there is no more centroids left the environment is fully explored.

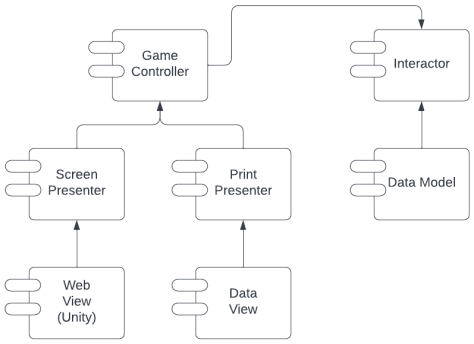
3 Design

3.0.1 Introduction

In the field of software development the use of well designed architectures can increase and maintain the system being developed. [30] This is done by using abstract diagrams and implementation techniques to help developers with preserving their overview. This is especially important when multiple people are working on the same project. During our implementation the use of well known design principles have been used for this purpose. The principles followed are developed by Robert Cecil Martin also known as Uncle Bob which has more than 50 years of industry experience with developing software systems.[30]

3.0.2 Design

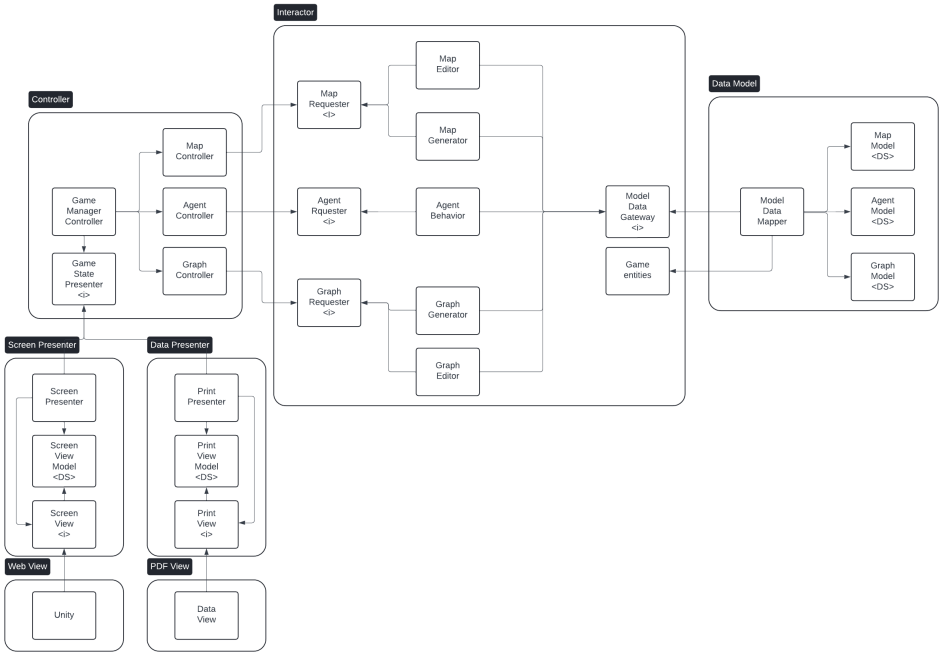
In this section the design and architecture of the software system is explained. The system consist of five main components being Screen Presenter, Game Manager, Data View, Interactor and Data Model. The reason for using components is to increase and maintain development speed. This is done by separating the systems functionalities into stand alone components, which means that each component can be executed and tested by itself. This maintains the flexibility of the system in terms of development and maintenance. Since both new and old components can be attached or detached without worrying about if the system is functioning. The overall design can be seen in the diagram below:



(a) Component diagram

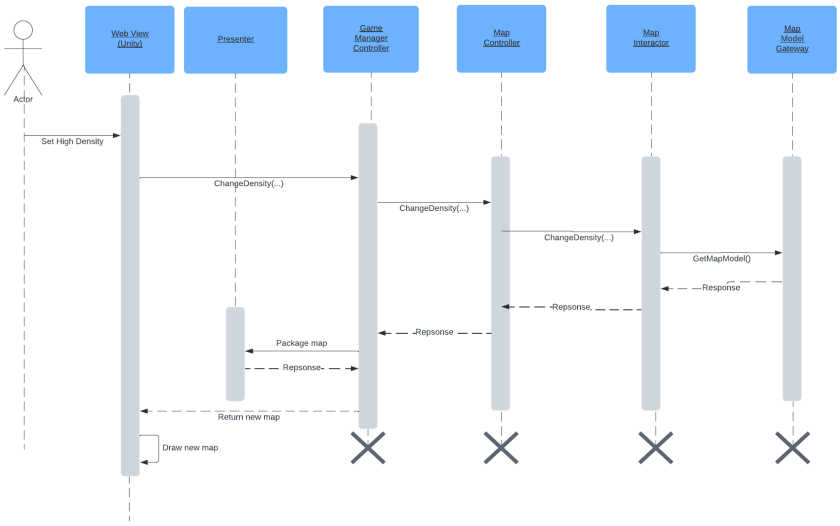
The arrows between each component can be understood as a relationship, meaning that if component A is pointing to component B then A knows about B, but B doesn't know about A. This means that component A is using a implementation from B, without B using any implementations from A. This relationship is important since components that change a lot should not be pointing to many other components. The main functionality of our implementation lies in being able to render the different simulation states, meaning that the main component is the interactor. The interactor functions as a bridge between the Web View being unity in this case and the data models being the state of the game. The goal of the interactor is to always have the overview of the simulation meaning that being able to gather and change data as requested from other components. This is also where the entities lies, since it functions as a gatekeeper for what is coming in and out, then invalid requests can easily be checked. For instance if we know that a wall cannot be placed outside of the map a failed response will be sent back to the user. Which also

makes error handling easier. The component called game controller sends request to the interactor about changing the state or variables of the map. The detailed component diagram can be seen below.



(a) Component relationship diagram

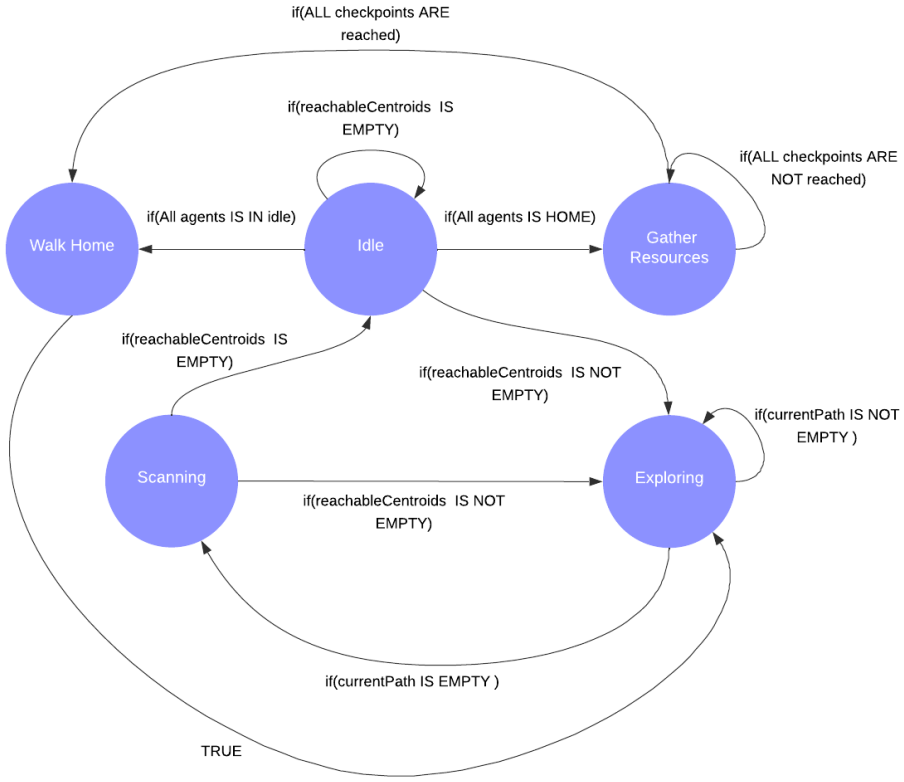
In the diagram above a detailed infrastructure of each component can be seen. This doesn't change the relationship and the idea behind each component. The component called data presenter functions as a data extractor for analysing and model the data from the different simulation states. Below is a use-case diagram showing how the system is working when a user changes the map density.



(a) Sequence diagram for set density

3.0.3 Scout Agent

In this section the design and decision making process is explained for the scout agent. The behavior of the agent is develop using a state diagram. The agent can be in one of five states being Idle, Gather Resources, Exploring, Walk Home or Scanning. In the Idle state the agent is waiting for a transition, meaning that it doesn't do anything productive. First it checks if there is any available centroids, if this is true, then it begins exploring the environment. When all the agents are done exploring the environment, meaning there is no centroids left on the map, then they begin returning home. When all the agents has arrived to their home position being the spawn point a transition to Gather Resources happens. They will keep gathering resources until all checkpoints have been visited. When they are done gathering resources meaning that all the checkpoints have been visited a transition to Walk Home occurs. Then they stay in Idle, since all resources have been gathered and the whole area is explored. This can be seen in the diagram below.



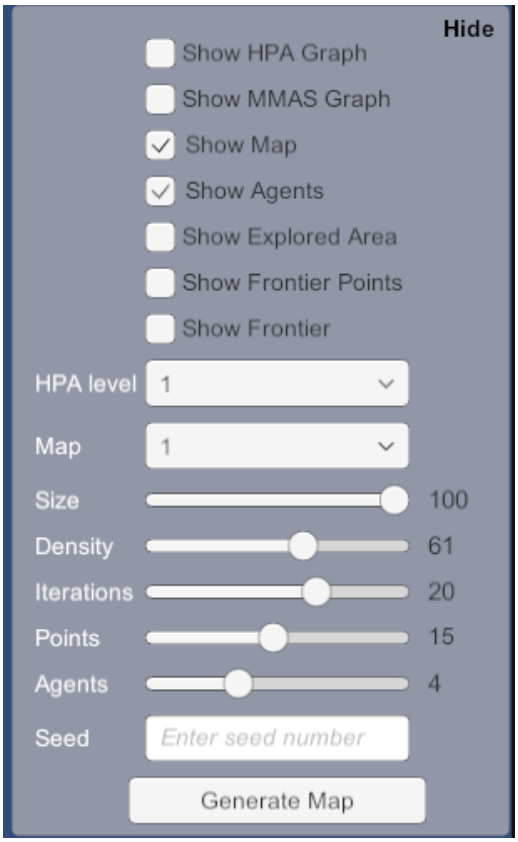
(a) State Diagram

The communication between agents takes place by the agent-controller, which means that the agents do not communicate directly. The controller makes sure that these states are complied with. When specific scenarios occur, like when all agents are in idle state, then the controller manipulates the state of each agent individually.

3.0.4 User Interface

In this section the performance, design and implementation of unity will be described and discussed. The user interface (UI) consists of two main panels being located in the upper left corner of the screen and to the right. The small panel shows the amount of area explored by the agents. The panel to the right is a intractable menu for generating custom maps by the user. Furthermore, it contains a combination of different toggles,

which can show and hide components on the map. These components are useful for showing how the different algorithms are working. But they doesn't have any effect on how the algorithms are computed. The dropdown menu called HPA level can set the different levels to be rendered on the screen by the toggle button called "Show HPA Graph". The other dropdown menu called Map will suggest custom maps to the user. To render these maps the button called "Generate Map" has to be clicked. The input field called Seed is useful for creating the same maps with different amount of agents etc. A picture of the panel can be seen below.



(a) UI panel

We are only using unity for rendering the different states of the simulation, since everything is computed backend. When using unity a performance of the different components can be measured. The performance will be measured using frame per seconds (FPS) while the application is running. It is important to note that this is done on a 2 GHz Quad-Core Intel Core i5 processor with the graphics card being Intel Iris Plus Graphics 1536 MB. This off course has a huge effect on the FPS being measured. The measurements can be used to identify the components that should be optimised. The components being rendered are HPA, MMAS and the map. In the table below the FPS can be seen for each component being rendered by itself.

Component	FPS
Map	180
MMAS	350
HPA	50

Table 3.1: Measurement of FPS for each component

4 Implementation

4.1 HPA* implementation

The implementation of HPA* was signicantly more comprehensive than firstly anticipated, this is mainly due to the actual alignment of clusters, entrances, nodes and edges to the ”real world” enviorment, thus requires careful handling of the 2D space. The positions is known of each node by its position, similiar to a real world GPS signal. This resulted in approximately 1000 lines of code. The implementation follows the SOLID principles to keep a clean architecture, moreover, the actual agent relies on a interface of HPA* which decoupels algorithm from the ui. This allows for different future implementations, of each sub component respectively.

In general, the simplified pseudo code in Near optimal hierarchical path-finding (HPA*) by Adi Botea and Martin Müller [2] differs from our implementation in several key areas, which include the management of graph abstraction, dynamic handling of nodes and edges. These differences enhance the algorithm’s robustness and adaptability to complex scenarios. In the below section, the ”pseudo code”, is referred to as the explanation and pseudo code proposed by Adi Botea and Martin Müller in the Paper [2].

Graph Abstraction and Management
`AbstractMaze()` is responsible for initializing clusters and entrances, however, in the implementation the Entraces are saved in a HashMap in `GraphModel`, which allows to efficiently retrieve entrances.

Dynamic Handling of Nodes and Edges
The implementation extends the algorithm by allowing dynamic addition and removal of nodes, while still ensuring the validity of the underlying graph model, moreover, it only recalculates specific clusters and entrances that are required to update, and thus minimizing re-computation, also handling edge case scenarios such as the entrances of the cluster changes. When a cluster is fully explored, it can be rebuild, and in some cases also the neighbour clusters if the entrances change.

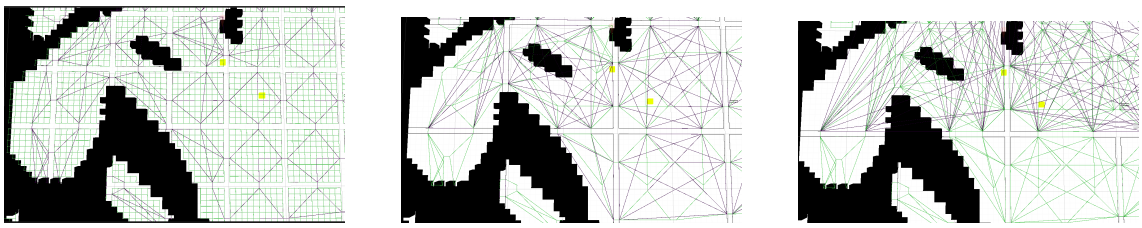
Error Handling and Cluster Merging
The merging of clusters in higher abstraction levels also handles irregular scenarios, such as, when it is not possible to merge all clusters evenly in irregular maps, it will calculate the remainder, and keep these clusters, at the previous abstraction level without recomputing them. We did also introduce a function, that automatically calculates a good cluster size for the Map, however, this function could be further optimized in future work for cluster size and entrances per cluster.

Conversion of edges across levels of abstraction
We also convert the previous inter-edges to intra-edges in the new cluster, this will minimize re-computation, while still keeping the underlying graph structure, moreover, it also ensure when computing inter-edges in the new abstraction level, the search space for A* will be very small compared to an complete open map, or complete open cluster, since edges are minimized to only allow the necessary paths to navigate around the cluster to specific entrances.

If an agent wishes to navigate to a specific point in a cluster, we place a node on this point, and computes the actual path by connecting it to the entrances of the cluster, and thus still ensures accuracy. This can then be cached.

4.1.1 Visualisation of HPA* in the GUI

The implementation of the visualisation is made with Unity's built in game-objects. The green color is Intra-edges, and the the Purple color is the inter-edges. Cluster size at $abstractionlevel1$ is 10. The black area is walls, and the white area is walkable area. There is generally one entrance per side, unless there is a wall on the border of the cluster, then an additional entrance is created. The entrances on higher abstraction levels are preserved, however it is optional to reduce entrances on higher abstraction levels, at the cost of accuracy, as it would be more difficult to refine a path between abstraction points (entrances).



(a) Abstraction level 1 (b) Abstraction level 2 (c) Abstraction level 3

Figure 4.1: Close-up screenshot of HPA* graph on different abstraction levels in the Unity environment

4.1.2 Testing of HPA* and A*

We created automated test for HPA* for dynamic map changes, and static path finding in the Map. We used blackbox testing, by setting tests on a known map generated map with pseudo random numbers, and two reachable points in the map. In fact, it is the same as in the 4.1. Moreover, we created automated tests with data extraction option, where one can specify a range on map parameters, and a number of iterations, and the function will generate the data for you on different abstraction levels, and compare it to the A*. This data generation is also used in evaluation section of HPA*.

4.2 A* Algorithm

In our implementation we used the Pseudo code proposed on the paper [15] as the overall structure.

In our implementation the heuristic cost is calculated using the Manhattan distance, considering the grid-based layout of our 2D maps, this allows for an accurate estimation of the distance to the goal.

We made two implementations of A* one original implementation which can run directly on the grid, and another implementation which runs on the HPA* nodes. The second implementation differs between only running on Intra, or Inter Edges, and thus reducing the exhaustive search of A* when there is no path. Both implementations utilizes the openlist as a priority queue, to get the node with the lowest f cost, or if equal then the lowest h-cost. This Node is then transferred to the closed list, and the algorithm examines all the neighboring nodes, by checking if the node is traversable and if it is not already in the closed list, if not then the neighbour is added to the frontier (queue) if not already present, and the neighbours' costs are calculated or updated. This logic is found in the FindPath in the AStar class. We use the PositionIsValid method to determine of the neighbouring nodes are walls or out-of-bounds areas and in that case not considered valid. We also use the GetDistance, or if HPA*, the edge weight between nodes to calculate the cost of moving from one node to another. In the same way, the GetDistance is used as an heuristic to calculate the hCost as the Manhatten distance.

4.2.1 Optimizations to the A* Algorithm

The main optimization made to the A* was the introduction of a fibonacci heap and Hash Map, and thus reducing the time complexity of the algorithm, how this affects the algorithm will be described in the analysis section. The C# framework available in Unity does not contain Fibonacci Heap as a datastructure so we used the advanced-algorithms repository which has an implementation of fibonacci heap in C# [31]. The affect of these optimizations on the time complexity of the whole algorithm is further discussed in the analysis section. The fibonacci heap for the openlist is the main optimization of the algoirthm, since this data structure offers amortized $O(1)$ time complexity for node insertion (enqueue), and only $logn$ for node extraction (dequeue). The HashMaps allows for optimizing the search process, if the node is already present in the frontier or closed list, which allows for $O(1)$ average-time complexity as well as for insertions.

4.3 Map generation

Our Cellular Automata algorithm consists mainly of two parts: makeNoiseGeneration and applyCellularAutomata. The algorithm takes following parameters: map size, density, iterations and erosion limit. The implementation follows the pseudo code, illustrated in the background section 2.1.4

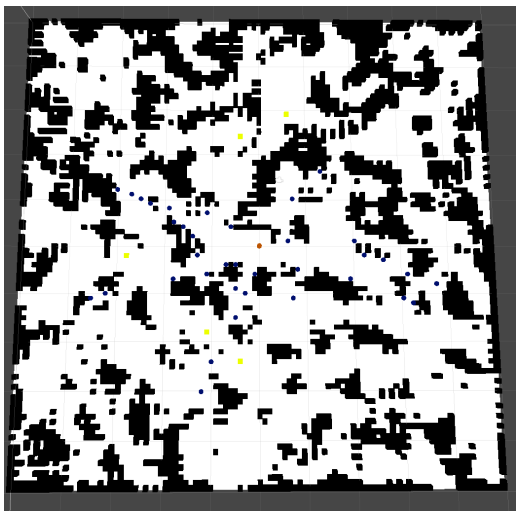
We follow the pseudo code strictly, and keeps a very simple implementation, where the The makeNoiseGeneration iterates over all cells in the 2D array, and if generates a random value between 0-100%, if the random value is greater than the density, then a wall should be placed at this cell. This function is used once initially in the algorithm.

The applyCellularAutomaton takes the iterations and erosion limit as parameters. It uses the erosion technique to create larger holes in the map. The overall complexity of the Map is in general adjusted by the iterations and noise, by increasing noise we create more walls, and thus by increasing the iterations we generally decrease the walls.

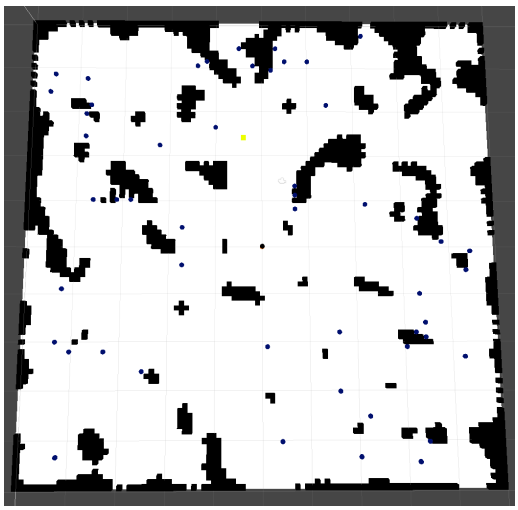
To keep consistency in our tests, we use a set of different pseudo random numbers to generate the maps.

4.3.1 Visualisation of Map Generation in Unity

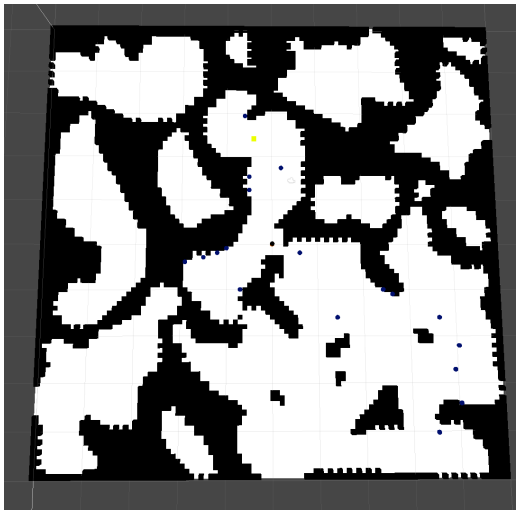
Below is an illustration of adjusting the different parameters of our cellular automata .



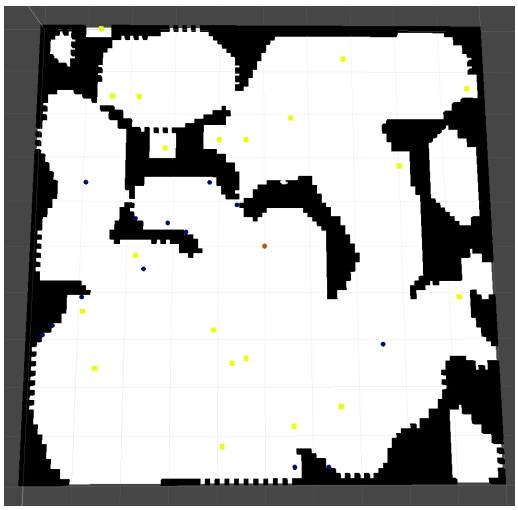
(a) Density: 50%, Iterations: 2



(b) Density: 50%, Iterations: 10



(c) Density: 60%, Iterations: 10



(d) Density: 60%, Iterations: 20

Figure 4.2: Comparison of four different random maps generation with different parameters

As illustrated in the Figure 4.2 complexity can be increased in the maps by:

- Higher percentage density
- Few Iterations
- More checkpoints
- Spacing around checkpoints

Variation can be achieved with random or pseudo random numbers in the respective complexity levels.

4.4 Max Min Ant System

4.4.1 implementation details and optimizations

In our implementation of the MMAS, several key optimizations have significantly improved the algorithm's efficiency and reduced computational times. First, we implemented an adjacency matrix to store edge weights between checkpoints. This change allows for constant time $O(1)$ lookup of edge weights, which drastically cuts down on the computation time needed for evaluating routes. *Initially, processing times were around 88 seconds; with the adjacency matrix, these were reduced to 7.1 seconds.* However, this was an obvious change.

Further enhancements were achieved by optimizing the management of the "tabu list," which tracks nodes already visited by each ant. By transitioning from a traditional list to a hash set, we capitalized on the average $O(1)$ time complexity for checks and updates, *reducing the computation time by an additional 2 seconds.*

The most substantial optimization came from parallelizing the tour construction process for the ants. By leveraging concurrent computing, we managed to significantly reduce the overall time complexity, allowing the system to solve the Berlin52 benchmark from the TSPLIB—a standard test for TSP algorithms—in just 2.7 seconds on an M1 Pro processor at 500 iterations.

The convergence check is implemented by calculating the sum of pheromone, and comparing it to the previous iteration with a threshold. If the convergence check is successful 10 times in a row, then we stop the algorithm.

It is possible to use the iteration best, global best or a combination of the two for updating pheromones and trail limits. In our implementation, we use the global best ant.

In the paper by Hooltz they also suggest different settings for the trail limits, we use the suggested dynamic max limit, and also set the minimum limit by:

$$\tau_{\text{max}} = \frac{1}{(1-\rho) \cdot f(s_{\text{best}})}$$

$$\tau_{\text{min}} = \frac{\tau_{\text{max}}}{2 \cdot n}$$

where s_{best} is the length of current global best tour.

Additionally, we implemented the option to add and remove nodes from the Max Min Ant System, where we then reuse heuristic information from previous iterations in the next. When we add a node, we connect it to all other nodes, and set the pheromone levels on the edges to τ_{max}, and the distance is set to a heuristic. We reset the global best tour when adding or removing a node. Later in section 6.1 , we will evaluate this approach, to a normalization technique, and to re computing the whole graph.

When adapting the mmas to handling addition and removal of nodes, we switched from using an adjency matrix for edges to a Dictionary in C#, which is also known as a HashMap to save memory and recomputing the length of the 2D array.

4.4.2 Visualisation of MMAS in Unity

In the illustration is shown a Best Tour on a random generated map in Unity while using the HPA* Abstraction Level 1 as a heuristic for the path lengths. Notice, how the yellow checkpoints in the upper left corner, are not part of the tour, since the heuristic is HPA* paths. Notice, the edges is passing through the walls, and this is due to the visulisation of a higher abstraction level, though all checkpoints are reachable within the Best Tour computed by MMAS. The reader, can also notice, there is slight visible edges in the picture, illustrating edges with low pheromone.

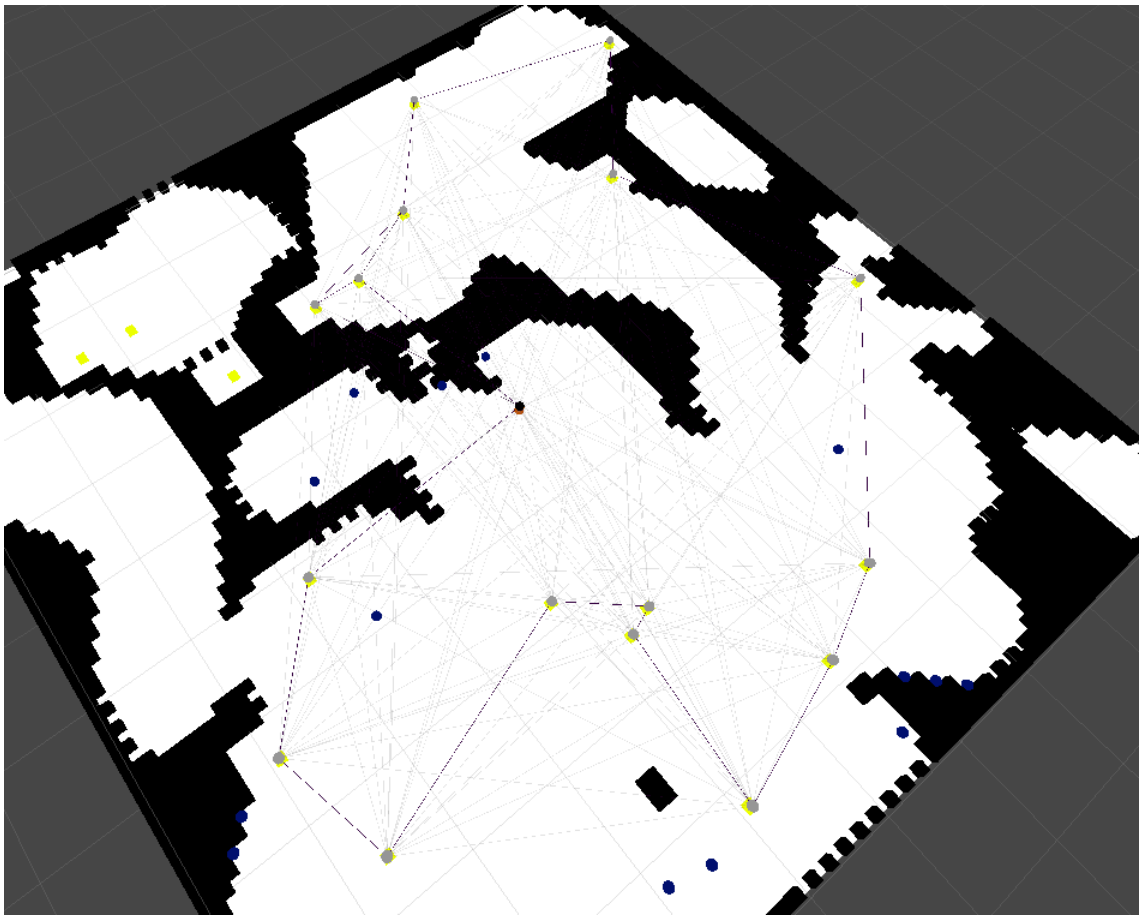


Figure 4.3: Visualisation of pheromones levels between checkpoints in random map

4.4.3 Testing of MMAS

We generated automated blackbox testing of MMAS on Berlin52, which results in success count, and number of iterations. This is further described in 5.3 . This allowed us to quickly iterative, and try different implementations. Moreover, we also generated automated test, to run the MMAS on different map generations both dynamic TSP instances, and static maps. This also allowed for easy data extraction, and comparison of different heuristics, and implementations.

4.5 Shadow Casting

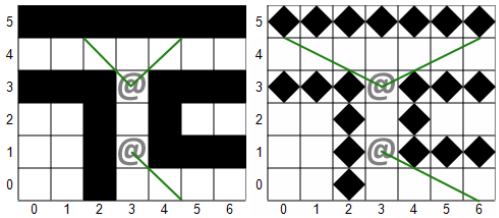
4.5.1 Implementation

In our implementation 4 sections have been used where the middle of each section is either pointing north, east, south or west. We define each section as a quadrant. The initial lines outlining each quadrant are called slopes, where the line to left and right are defined as start and end slope respectively. The angle for each slope is stored using fractions to minimize floating-point errors. There is two scenarios to take into consideration when adjusting the angle. As explained earlier, when a wall is encountered the angle of either the start slope or the end slope has to be adjusted. The start slope is adjusted when the previous tile is a wall and the current tile is a floor. The end slope is adjusted when the previous tile is a floor and the current tile is a wall. This is because the start slope should always be tangent to right side of the wall and end slope to the left side of the wall. This continues until there is no more visible tiles left or the left slope is bigger or equal to the right slope. The slope angle are calculated using the following function.

```
1    public Fraction Slope(Point tile){
2        int rowDepth = tile.x;
3        int col = tile.y;
4        return new Fraction(2 * col - 1, 2  * rowDepth);
5    }
```

The function takes a tile with a x and y position relative to the position of the agent. The reason we only need one function to calculate both start and end slope lies in the fact that the start slope is equal to the tangent of the floor tile next to it. Since we know that the slope can be calculated by the following formula $slope = \delta y/\delta x = y2 - y1/x2 - x1$, with the position of the agent being the origin, the equation can be simplified to $slope = y2/x2$. The term $y2$ needs to be adjusted so the slope is tangent to the side of tile and not the middle. This can be done by subtracting $1/2$ from $y2$, but since we don't want to evaluate the fractions to avoid errors, we multiply both $y2$ and $x2$ by 2. This will keep both the numerator and denominator as integers and preserve the relation between them. Then 1 is subtracted from $y2$ to make it tangent to the tile.

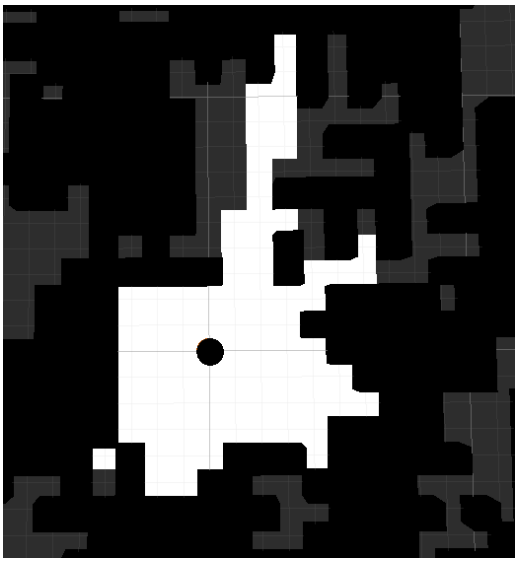
To increase the efficiency of the algorithm a slight adjustment has been made from the original implementation.[27] The difference lies in how the walls are interpreted in terms of their shape. In the original implementation the shape of the walls are viewed as regular squares, where in this implementation they are viewed as diamond shaped. This greatly increases the field of view around corners and narrow spaces, which significantly decreases the amount of scans made by the agents. This can be seen in the figures below. [32]



(a) Comparison for field of view

There is a possibility that tiles are detected through walls, which needs to be taken into

consideration when using frontier based strategies. This can be seen on the figure below, where the tile located button left are detected between two walls.



(a) Tile detected between two walls

Since the slope is only calculated from using the agent's position a function called transform is implemented to return the global position of the map tiles.

```
1    public Point Transform(Point tile){
2        int row = tile.x;
3        int col = tile.y;
4        if(cardinal == south) return new Point(origin.x + col, origin.y - row)
             ;
5        if(cardinal == north) return new Point(origin.x + col, origin.y + row)
             ;
6        if(cardinal == east) return new Point(origin.x + row, origin.y + col);
7        if(cardinal == west) return new Point(origin.x - row, origin.y + col);
8    }
```

The class Quadrant has the method transform to return the global position of each tile, so it can be evaluated correctly. This is a clever method since we don't have to take into consideration the orientation between the agent and tile when calculating the slope. The pseudo code for computing field of view using symmetric shadow casting can be seen below.

Algorithm 7 compute field of view

```
1: procedure computeFOV(origin)
2:     markVisibleTiles := SET containing origin
3:     markVisibleWalls := EMPTY SET
4:     for i FROM 0 TO 3 do
5:         quadrant := NEW Quadrant(i, origin)
6:         firstRow := NEW Row(1, startSlope, endSlope)
7:         scan(firstRow, quadrant)
8:     end for
9: end procedure
```

Algorithm 8 scan environment

```
1: procedure scan(row, quadrant)
2:     rows := EMPTY STACK containing row
3:     while rows IS NOT EMPTY do
4:         row := POP rows
5:         prevTile := NONE
6:         for each tile IN ROW do
7:             if IsWall(tile, quadrant) OR IsSymmetric(row, tile) then
8:                 Reveal(tile, quadrant)
9:             end if
10:            if IsWall(prevTile, quadrant) AND IsFloor(tile, quadrant) then
11:                row.startSlope := Slope(tile)
12:            end if
13:            if IsFloor(prevTile, quadrant) AND IsWall(tile, quadrant) then
14:                nextRow := row.next()
15:                nextRow.endSlope := Slope(tile)
16:                PUSH nextRow TO rows
17:            end if
18:            prevTile := tile
19:        end for
20:        if IsFloor(prevTile, quadrant) then
21:            PUSH row.Next() to rows
22:        end if
23:    end while
24: end procedure
```

The computeFOV(...) function iterates over each quadrant starting with the first row and then working outwards. The IsSymmetric function checks if the tile is inside the area spanned by the start and end slope. If the start or end slope is touching the tile it is included. The reveal function adds the global position of tile to either the lists called visible tiles or visible walls, depending on the tile type.

4.5.2 Time Complexity Analysis

In this section a time and space complexity analysis is made of the shadow casting algorithm. In each scan, the algorithm goes through each row starting at the position of the agent. The next row increases the current amount of tiles by 2. This creates a pyramid shape with a maximum depth of the maps height, if the agent is located at left, top, right or button of the map. Since the transform function can access the global position of the tiles in constant time $O(1)$, the overall running time is $O(N)$ where N is the total amount of

nodes in each quadrant. We can find a expression for the total amount of nodes defined by the height. We know at each level the amount of nodes is $(2k+1)$, then the following expression must be true for the total amount of nodes.

$$N = \sum_{k=0}^{h-1}(2k+1) \tag{4.1}$$

We can simplify the sum as follows:

$$\sum_{k=0}^{h-1}(2k+1) = 2\sum_{k=0}^{h-1}k + \sum_{k=0}^{h-1}1 \tag{4.2}$$

The first part can be written as the sum of the first h natural numbers:

$$\sum_{k=0}^{h-1}k = \frac{(h-1)h}{2} \tag{4.3}$$

Where the second part is simply the sum of h ones:

$$\sum_{k=0}^{h-1}1 = h \tag{4.4}$$

Combining these:

$$N = 2\frac{(h-1)h}{2} + h = (h-1)h + h = h^2 \tag{4.5}$$

Therefore, the total number of Nodes with respect to the height is equal to $N = h^2$. The space complexity is equal to the total amount of tiles on the map, which is the height times the width of the map. Since if there is no walls on the map the total amount of visible walls would be equal to the total amount of tiles. The running time for comuteFOV is just four times the running time of scan, since it's always four it can be calculated to just $O(N)$.

4.5.3 Conclusion

Symmetric shadow casting algorithm is an effective choice for large open maps because of the computational efficiency. It can scan large areas of the map in real time both in single and multi agent environments. When interpreting the environment using diamond shaped walls (instead of square shape) the field of view is greatly increased with no drawbacks. This decreases the overall amount of scans made by each agent. There is a possibility that tiles are detected through walls, which needs to be taken into consideration when using frontier based strategies.

4.6 Frontier Based Exploration

4.6.1 Implementation

First we need to find and locate frontier points on the map. Visible walls and tiles have been found from using symmetric shadow casting. As described earlier the frontier points is located on the frontier region, which in fact is a sub set of the visible tiles. Therefore, we can simply loop through the list of visible tiles and check if any adjacent tiles are not already in visible tiles or walls. This can be seen in the following function.

```
public HashSet<Point> FindFrontierPoints(HashSet<Point> visibleTiles, HashSet<
    Point> visibleWalls) {
        HashSet<Point> frontierPoints = new();
        foreach(Point point in visibleTiles){
            List<Point> neighBoringPoints = FindNeighboringPoints(point);
            foreach(Point neighBoringPoint in neighBoringPoints){
                if(!visibleTiles.Contains(neighBoringPoint) && !visibleWalls.
                    Contains(neighBoringPoint)){
                    frontierPoints.Add(point);
                }
            }
        } return frontierPoints;
    }
```

The frontier points are then added to a hashSet containing objects of points. The Point object is a simple class containing integer values for x and y. The next step is to group or cluster these frontier points together. This is done by taking the first element out of the set of frontier points and then run breath first search (BFS) starting from the position of that point. The BFS algorithm implemented can go in 8 directions, but 4 directions can be used as well. The reason for using 8 directions lies in grouping bigger areas of frontier points together. This creates better results as the precision of the centroids stays the same but the overall amount of centroids created greatly decreases. Which also decreases the amount of steps taken by the agents. When BFS is done running the cluster is returned. The psuedo code for the BFS implementation can be seen below.

Algorithm 9 Cluster Frontier Points

```
1:  procedure ClusterFrontierPoints(frontierPoints, frontierPoint, map)
2:      frontierPoints ← QUEUE containing frontierPoint
3:      visited ← List containing frontierPoint
4:      while queue IS NOT EMPTY do
5:          for each adjacent tile do
6:              frontierPoint ← queue.dequeue()
7:              clusteredFrontierPoints.add(frontierPoint)
8:              if each tile IS NOT NONE AND NOT VISITED then
9:                  if frontierPoints CONTAINS tile then
10:                     queue.enqueue(tile)
11:                     visited[tile] := true
12:                 end if
13:             end if
14:         end for
15:     end while
16: end procedure
```

The set returned by BFS is a sub set of all the frontier points. Therefore, we can extract the sub set from the original set of frontier points and then run BFS on the extracted set. The algorithm continues doing this until the set of frontier points are empty. By doing this we guarantee to group all frontier points in their respective clusters. The function can be seen below.

```
1   public HashSet<Point> FindFrontier(HashSet<Point> visibleTiles, HashSet<
        Point> visibleWalls){
2       HashSet<Point> frontier = new();
3       HashSet<Point> frontierPoints = FindFrontierPoints(visibleTiles,
            visibleWalls);
4       while(frontierPoints.Count != 0){
5           List<Point> temp = frontierPoints.ToList();
6           Point point = temp[0];
7           HashSet<Point> frontierCluster = GridExplorer.
                ClusterFrontierPoints(frontierPoints,point,map);
8           frontier.Add(FindClosestToCentroid(frontierCluster));
9           frontierPoints.ExceptWith(frontierCluster);
10      } return frontier;
11  }
```

The dictionary named frontier holds the centroids calculated from the function called findClosestToCentroid(...). The function takes in a frontier cluster and returns the closest frontier point to that centroid. The reason for adjusting the position of the original calculated centroid is to avoid the possibility that the centroid is placed on a wall. There is several techniques to account for this, but in this implementation the centroid is moved to the closest frontier point in that cluster. The euclidean distance is used as the measurement for closest distance. The function iterates over all the frontier clusters, adds each of the calculated centroids to the dictionary with a unique key. When the loop is over the dictionary is returned. The reason for using a dictionary as a data structure is to make the agents able to access the centroids in constant time. This is done by first finding the centroid closest to the agent and then remove this element from the dictionary. Since each agent has to loop through the whole dictionary to find the closest centroid, the key has to be stored, to remove the element from the dictionary afterwards. This could seem inefficient, but the reason being is that multiple agents have to access the same dictionary. So it wouldn't work if the dictionary is sorted by an arbitrary agent because the relative position varies between them. A more detailed description of the agents can be found in the design section.

4.6.2 Testing and Validation

One of the biggest problem using shadow casting algorithms for scanning the environment lies in the probability of detecting tiles through walls. This is explained in greater details in the implementation section about shadow casting. This leads to a problem where centroids are created inside non reachable areas on the map. To solve this problem the path finding algorithm A* have been used to check if the centroid is reachable. If is not reachable then the centroid is removed from the set. This is off course not optimal since other methods has to be implemented to check these edge cases. But taking into consideration the computational efficiency of using shadow casting compared to other field of view algorithms, then it can be justified. Black box testing was used to validate the algorithm in the form of user interface (UI) testing.

4.6.3 Time Complexity Analysis

To define the performance of the frontier based algorithm a time complexity analysis is made. The main functionality lies in the function called FindFrontrier(...) which includes three smaller functions being FindFrontierPoints(), ClusterFrontierPoints() and FindClosestToCentroid(). The time complexity of the algorithm lies in finding the slowest algorithm in terms of running time of these three functions. The function FindFrontierPoints(...) loops through visible tiles and checks the adjacent tiles. The worst case scenario lies in a open map with very few walls. In this scenario the algorithm has to nearly loop trough all the tiles on the map. Since the amount of adjacent tiles is constant, the running time can be defined as $O(N)$, where N is the total amount of tiles on the map. This can be written as $N = m * n$ where m and n is the height and width of the map. ClusterFrontierPoints(...) uses Breath First Search (BFS) to locate clusters between frontier points. The running time for BFS is $O(V + E)$, where V is vertices and E is edges. Since there is always 8 edges for all vertices the running time can be simplified to $O(V + 8 * V)$, which is in big O notation simplified to $O(V)$. V is the total amount of tiles on the map, which can also be written as $O(N)$. The last function FindClosestToCentroid(...) iterates over all the frontier points in that cluster and then finds the closest one to the centroid using euclidean distance. Since the frontier points is in a cluster it's only a sub set of all frontier points. This takes linear time O(k) where k is number of frontier points in that cluster. Then the running time of these combined functions become $O(N) + O(N) + O(k)$, which can be simplified to the overall running time of the frontier based algorithm being $O(N)$.

5 Analysis

In the following sections when describing time complexity we consider the worst-case Big O notation unless other specified, and we focus on specific essential algorithms of the project, while still keeping a high abstraction level. For the concrete implementation, we refer to the code attached to the project.

5.1 Optimization Impact on A* Complexity

In general, when studying the A* running time in academic research it is often metioned as the a b^d [9] where B is the branching factor and d is the number of nodes on the shortest path, however, the A* algorithm can be expressed in polynomial time in a similar finite environment as ours. We recall from section 2.2.3, If heurtics is admissible it guarantees to find the shortest path. The A* algorithm is also optimally efficient if the heurtic is consistent. Hart, Nilsson, and Raphael (1968), who first formalized A*, showed that A* is ”optimally efficient” among all admissible heuristics under the assumption of a consistent heuristic. This means no other algorithm using the same heuristic can expand fewer nodes than A* [1]

Admissiblity of Manhatten distance.
The Manhatten distance is used as our heurtic and is calculated as: $h(n) = |x_2 - x_1| + |y_2 - y_1|$

This measure the total number of horizontal and vertical moves needed to reach a point without obstacles. The cost of moving is 1. This means that the manhatten distance is admissible.

Monoticiy of our Manhatten distance
To confirm monotocity the following must be true: $h(n) <= c(n, a, m) + h(m)$ where $h(n)$ is the heurtics cost from node n to the $goalnode$, $c(n, a, m)$ is the cost of moving from node n to m this is the $G - cost$ and the $h(m)$ is the heurtics cost from the m node to the $goalnode$. This is informally defined as the f-cost must be non-decreasing along any path. [9] Since the manhatten distance matches the step cost it will always align with $c(n, a, m)$ therefore the Manhatten distance is monotonic.

The admissibility and monoticity on higher abstraction levels in HPA* should be valid since cost of moving from entrance to another will be the A* cost on the lowest level, and since we keep the same coordinate of entrances on higher abstraction levels, however, one should be very careful when working with heuristics on abstract graphs, since you might happen to introduce a non-admissible heuristic.

5.1.1 time complexity of A* on our grid-based map

Our optimizations have a significant impact on the time complexity of the A* algorithm. The introduction of the priority queue implementation with Fibocacci Heap reduced the time complexity in amortized cost of Enqueue to (1) and dequeue to $(Log(n))$, then we instead of only using a fibnacci heap our $openlistLookUp$ works as a pointer to the nodes in the Priority Queue, and thus also gurantee (1) for $closeListLookUp$ look ups. Once we have reached the target node, we retrace the path using the parent pointers. This is done in $O(N)$ time relative to the length of the path.

Since our Map is a $i * j$ grid the total number of nodes is denoted by $i * j$. The map will often consist of open and closed areas where a closed area is an unreacble area

for the A* algorithm. The worst-case time complexity of the A* algorithm will then be in the case of no viable path to the checkpoint, and thus an exhaustion of all nodes in the openarea. We define N nodes as part of the $openarea$ nodes and thus reachable. In this worst-case scearnio the algorithm would remove all nodes from the fibonacci heap, which would result in $N * O(LogN)$. Also there is 4 edges per node, in the lowest level, and this would then require each node to be processed. $N * 4 * O(1)$ as the $fcost$ is calculated in constant time. Thus the overall worst-case running time will be $O(N * Log(n))$

The best case running time will be if the goal node is an neightbour and thus $O(1)$. However, analyzing the average case, we can see this a A* walking on the diagonal from one corner to another with few obstacles, $k * N * logN$, where K is a factor determining the difficiulty of the Map. If it is completely open K would be $K = \frac{1}{\sqrt{N}}$ since the A* would simply walk directly the shortest path along the sides of the Map, thus we can summarize this analysis in a table:

Table 5.1: Our Time Complexity Analysis of A* Algorithm on reachable $N \times N$ Grid

Scenario	Description	Time Complexity
Best-Case	Goal at or next to start node	$O(1)$
Average-Case	Effective heuristic, mixed grid obstacles	$O(k \cdot N \log N)$
Worst-Case	Ineffective heuristic, no path to goal	$O(N \log N)$

5.2 Approximated Time complexity of the MMAS and building the graph

We have N nodes (Checkpoints) since we have converted the problem to a classical TSP with a fully connected undirected graph without self-loops. Then if you have N verticies, then you can pair each vertex with every other vertex resulting in a $(N * (N-1))$ pairings, but then since the graph is undirected, then the total number of edges can be described as: $E = \frac{N \cdot (N-1)}{2}$, we observe that the complexity grows quadratically with the number of vertices N. Thus, in Big O notation, this complexity is denoted as $O(N^2)$ So for each edge we need to calculate the weight of this Edge which is given by a heuristic, which can be expressed as $(E * O(Heuristic))$, an heuristic could be A*, which we earlier discussed that the worst-case running time of A* on our graph Then the total running time of building the Graph will then be:

$$O(N^2 * \text{A*}) \tag{5.1}$$

The MMAS algoirthm is running maximum I iterations, and for each iteration an ant Constructs a tour which is refered as $BuildTour$ in our implementation. This function has a running time of $O(N^2)$. There is an ant for each Node, which then results in a running time of N^3 . However, we have optimized the algorithm by each ant constructing their tours in parallel , then it is assumed that constructing tours for all ants is N^2 . Pheromone updating and limiting is N^2 thus the total running time can be derived to: : $(O(I * N^2))$

However, it is very important to have in mind that the actual time complexity of the MMAS should consider the actual convergence I, optimal solution and the stochastic behavior of the algorithm, and the maximum capacity of managing N threads.

5.3 Parameter setting of MMAS

We implemented functionality to test the TSP from the TSPLIB. We have conducted the following optimizations and tests on the Berlin52 to verify our implementation of the MMAS, and test different parameters. [33]

	Alpha	Beta	Rho	Success Count	Average Iterations
Without dynamic limits	1.5	4.5	0.1	59%	94
With dynamic limits	1.5	4.5	0.1	62%	96
With dynamic limits	1	4.5	0.1	91%	242
With dynamic limits	1	5	0.1	95%	234
With dynamic limits	1	5	0.05	0%	500
With dynamic limits	1	5	0.3	42%	45

Table 5.2: Summary of Parameters and Results on Berlin52

A run is considered a success in the table, when the Best Tour is the optimal tour of Berlin52. Additionally we also tried with Alpha = 1, and Beta = 2, as originally used in the paper, but this performed poorly, and thus we changed to a higher range of beta values. Each experiment is conducted on 100 runs on the Berlin52. Dynamic limits is referring to trail limits. It is important to note that when choosing the parameters one has to notice, that it also affects the algorithm's convergence, and thus also the success. A higher success is associated with higher iterations as well. For our project, we have many computationally processes running in the background, in particular as Unity is expensive to run. We then choose to sacrafice on solution quality, and get a higher speed at the cost of success. This means, that in our simulation, we choose Alpha 1.5, Beta 4.5, and Rho 0.9. We a note, that these parameters can be optimized even more with algorithms such as CMA-ES. [34]

5.4 Hierachial path finding time complexity

The paper on Near optimal hierarchical path-finding [2] doesn't not conduct a time complexity analysis of the graph construction, but we will try to do this in this section of our implementation, we have named our implemented functions the same as in the Pseudo code in section 2.3.4. When doing this it is important distinguish between below definitions:

- **Nodes**: N
- **Clusters**: C
- **Intra Edges**: E_{intra}
- **Inter Edges**: E_{inter}
- **Entrances locally in Cluster C**: E_C
- **The time complexity of the A* algorithm:** $O(\text{A*})$:

The $AbstractMaze$ function runing time in Big O notation can be reduced to: $O(N^2)$ since the buildClusters function is $O(N^2)$, we try to build entrances between Clusters which is $O(C^2 * E_C)$ which combined as sequential operations yields to: $O(N^2) + O(C^2 * E_C)$

Given that the number of clusters C is significantly less than the number of nodes N, and assuming E_C is relatively small compared to N, we can simplify the expression.

Since C is proportional to $\frac{N}{\text{clusterSize}}$, the total time complexity can be approximated for the `AbstractMaze` to:

$$O(N^2)$$

The $BuildGraph$'s time complexity, can be determined by, the fact that why need to calculate A* between each entrance. However, we decided in average to try match 1 entrace every 10 nodes, there will only be 1 entrance per side, if we assume a cluster size of 10 on a $100x100$ graph, the number of entrances depends on the actual complexity of the map, but it will usually be 4 entrances. The time complexity of $BuildGraph$ can then be derived to:

$$O(C * E_C^2 * \text{A*})$$

We then analyze the $AddLevelToGraph$, then merging of the previous clusters can be reduced to $O(C * M)$. This function mainly consits of 4 parts, merge full clusters, handle remainders for rows and columns, convert previous inter edges to intra edges, and add new inter edges between entrances. Since our finding is that the adding of new edges using A* is the bottleneck, we will quickly summarize below operations.

- **Merge Full Blocks**:

$$O\left(\frac{\sqrt{C}}{\text{clustersToMergePerSide}}\right) \times O\left(\frac{\sqrt{C}}{\text{clustersToMergePerSide}}\right) \times O(\text{clustersToMergePerSide}^2) \quad \times O(M)$$

  Since the ClustersToMergePerSide is set to 2 in our implementation, and O(M) is the cost of merging clusters, which can be estimated to $O(4 * E_C)$, it can be reduced to:

$$O(C * E_C)$$

- **Handle Remainders for Rows and Columns**:

$$O(C * I)$$

- **Convert Previous Inter Edges to Intra Edges**:

$$O(E_{\text{inter}})$$

- **Add New Inter Edges Between Entrances, where E_C is the entrances in each specific cluster**:

$$O(C * E_C^2 * \text{A*})$$

We can summarize this to :

$$O(C \cdot E_C) + O(C \cdot I) + O(E_{\text{inter}}) + O(C \cdot E_C^2 \cdot O(\text{A*}))$$

We see that the creation of new inter edges, is by far the dominant factor in $AddLevelToGraph$, however, we should keep in mind that this is done at higher abstraction level, than when we used A* earlier.

$$O(C * E_C^2 * \text{A*})$$

We can then conclude on the total preprocessing time of HPA* to:

$$2 * O(C * E_C^2 * \text{A*}) + O(N^2)$$

By converting the previous inter edges to intra edges, we save significant compute, and on higher abstraction levels the cost of actually computing the path between entrances is low, since on abstraction level 2 as an would typically consist of 25 nodes, 8 entraces and 100 edges, which is signifcantly less when computing a star on A* on a full open cluster grid every time, that consists, of of an example $20x20$ grid of 400 nodes, or even worse, using A* directly on 10000 nodes and 40000 edges due to the branching factor described earlier of A*.

The handling of prime numbers in the implementation is specially challenging since as per the definition, there is no divisor other than the number itself and 1, which leads to incomplete coverage of the map, and thus out of bounds errors, however, one simple solution to this, is to adding padding of walls around the map to convert the map-size into a non-prime number, or calculating a optimal cluster size, that accounts for this.

6 Evaluation and Results

6.1 Dynamic changes in the TSP Graph and the MMAS

Related research on using the max min ant system for dynamic TSP, has been made, particularly by Leguizamón and Alba in 2013 in their paper ”Ant colony based algorithms for dynamic optimization problems” [35], in this paper they outline the importance of dealing with two major concepts in dynamic optimization problems, which is the the mechanism to avoid stagnation, and the capacity of the algorithm to react to changes. They highlight that the most important aspect of ACO is the pheromones of the algorithm, since this represents the memory of the algorithm. They suggest to reduce the pheromone levels, to lower the learnt experience from previous runs. They suggest normalizing, and / or, resetting pheromone values, however we assert that resetting would be similiar to re-computing the algorthm from scratch, since less information is obtained from previous runs. Angus and Hendtlass has made research on adapting Ant Colony Optimization to dynamic TSP problems using the normalization technique, which we will test and compare to our own.

Eyckelhof and Snoek suggests a shaking process for the Ant System, which is primarily used for changing edge weights when paths between cities becomes longer. [36]

In this experiment we will research the difference between dynamic and static max min ant system adapts to adding different checkpoints in random maps. We have conducted 30 different simulations, and in each simulation added upto 40 checkpoints for each checkpoint added we either rebuild the whole graph with the static version, or with the dynamic version run the ant system extra times with the same pheromone from the previous run. We use the Euclidean heuristic as the edge weights in the MMAS. Moreover, We use Random maps generations, but compare each specific run on the same map with the respective algorithm. In our dynamic ant system, we do not recalculate the τ_{min} and τ_{max} but reset the best tour. Pheromones for the new edges related to the new checkpoint are set to τ_{max} to encourage exploration of these. Since we reset the Best Tour, the new Global Best ant will update the trail limits, and the trail limits will already be set after the first iteration.

Firstly we investigated, how the Best Tour lengths were affected by this, and we see in the below diagram, they are both generally the same, and within the same confidence intervals, which makes sense, since we set the max iterations 1000 in the Max Min Ant System, and this seems to be enough to reach convergence across all checkpoint levels. This makes it more interesting for us to look further in how many iterations in each algorithm is required to reach the convergence, the approximated best tour length.

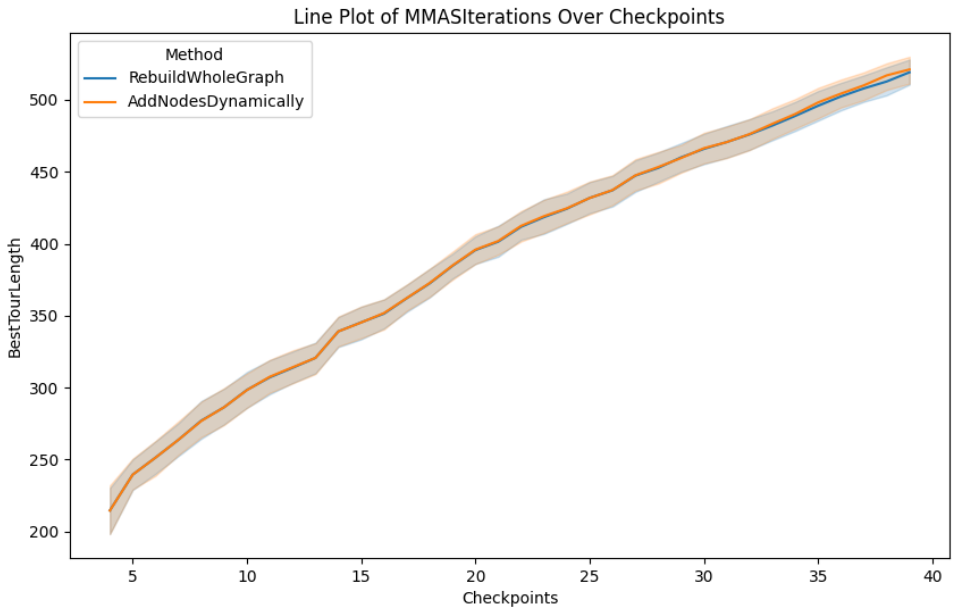


Figure 6.1: Line plot of Best Tour Length over checkpoints. The plot illustrates the length of the shortest path between all checkpoints. The shaded regions represent confidence intervals.

Below we have visualized in a line plot the number of the total iterations for each algorithm to reach the convergence of each checkpoint accumulated with the iterations from the previous checkpoints added. Here we see that the dynamic algorithm is definitely the best performing algorithm. We can also denote, that there is a clear linear correlation between number of checkpoints added and amount of iterations for the dynamic algorithm. On the other hand, the static algorithm has a one linear correlation till it hits around 25 checkpoints and then the slope increases.

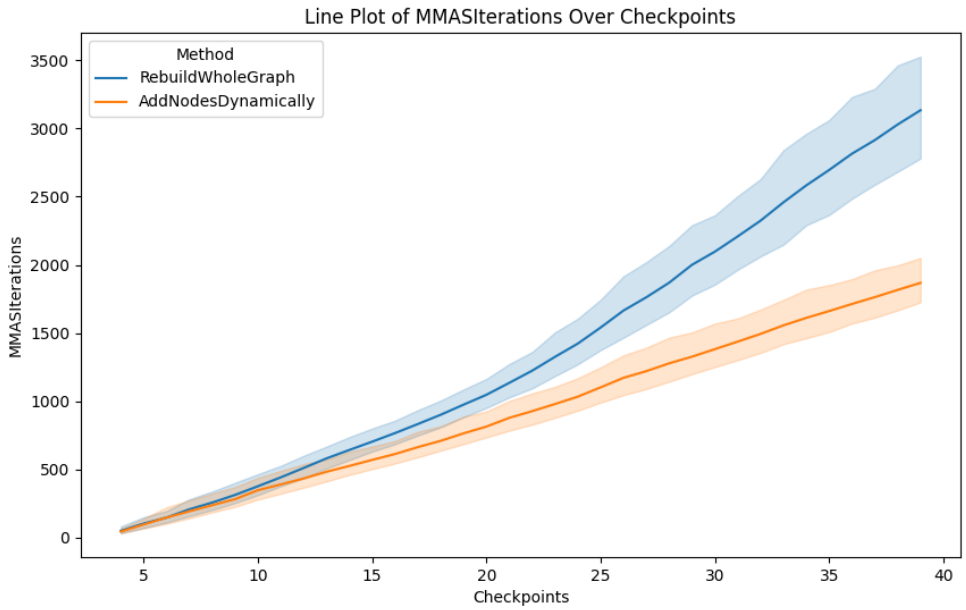


Figure 6.2: Line plot of MMASIterations over checkpoints. The plot illustrates the number of iterations required for MMAS to converge as the number of checkpoints increases. The shaded regions represent confidence intervals.

Since there is no previous heuristic information available in the static graph from the previous run, as the checkpoints increases, the influence of heuristics, will matter even more, since there are many more edges for the ants to choose from, it becomes even more difficult for the algorithm to convergence early, and thus we see a higher number of iterations.

The confidence intervals seems to increase on both methods, but more in the static method.

Below is a boxplot comparison for each checkpoint over the 30 simulation runs, here we can properly see the distribution of iterations. The dynamic method is more stable and reliable, while the static method starts to dramatically increase the distribution between iterations in the upper 50% of simulations, which indicates the method has more difficiult in reaching the convergence, we also see more outliers. Both algorithms have a relatively low median. The low median suggests that 50% of the time it convergence early given the parameters and heuristics.Given the Linear increase in iterations when added checkpoints indicates, we in general do not achieve stagnation, however, as the number of checkpoints increase so does the, bounds of the 3rd quantile, which suggests more stagnation happens, at larger amount of checkpoints.

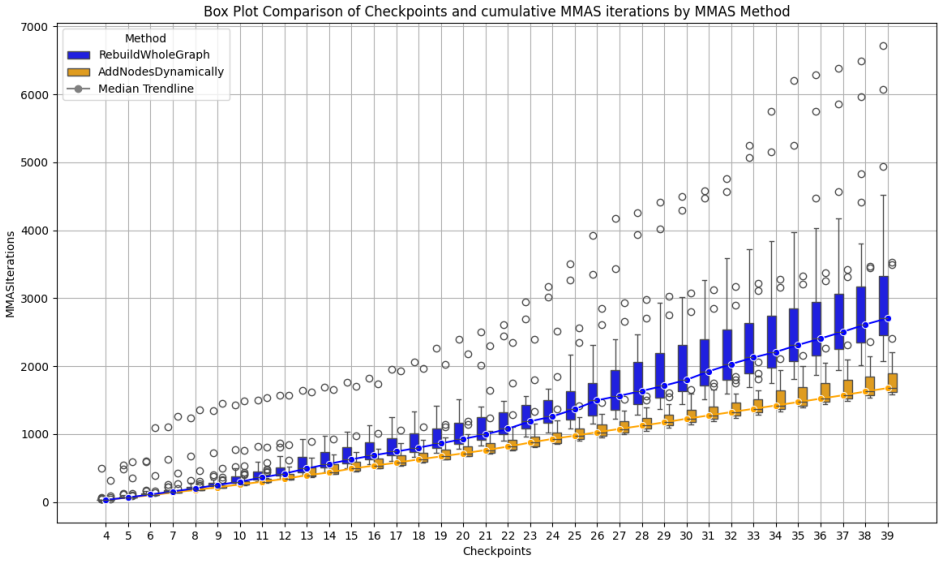
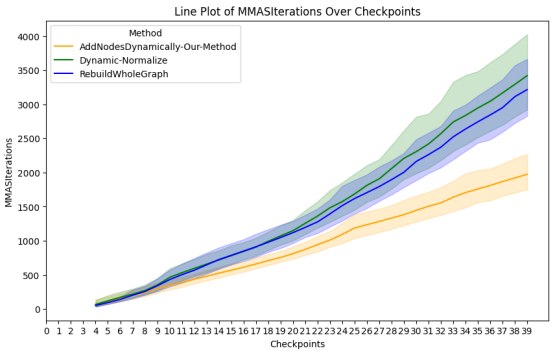


Figure 6.3: Box plot comparison of checkpoints and mmas iterations by method. The plot shows the distribution of cumulative mmas iterations for different numbers of checkpoints. The lines indicate the median trendline for each method.

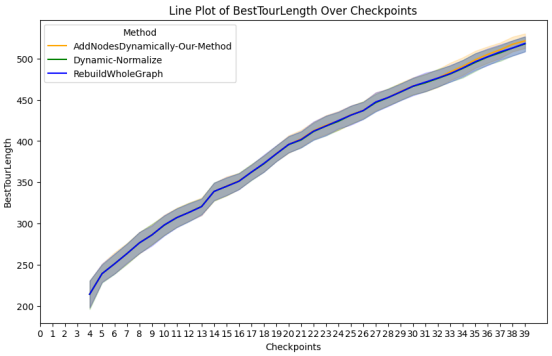
6.1.1 Comparison of normalization vs. our Method

In this section we try to investigate the normalization technique proposed by Angulus [6] in our 2D environments. When doing this comparision, it is important to remark, that Angulus implemented the Ant System, while we use the MMAS. We use the normalization by making a simple implementation, when we add a node to the MMAS graph, we will Normalize each nodes pheromones according to the maximum value of the neighbours. $\tau_{ij} \longleftarrow \tau_{ij}/\tau_{i,max}(t)$, and set the new edges in the graph to τmax . Moreover, in the paper, only results from Node removals was showcased.

On the below diagrams, we derive that our method performed significantly better, while the normalization technique, performs similarly to building the rebuilding the whole Graph, but actually worse on higher checkpoint levels. The paper didn't specify how pheromone should be initialized on the new edges added. We decide to keep our initial method in the simulation GUI.



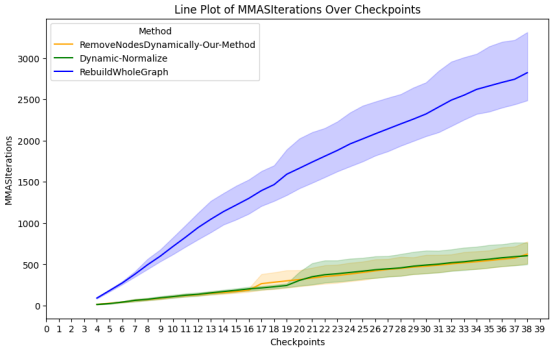
(a) Cumulative Iterations of checkpoint additions



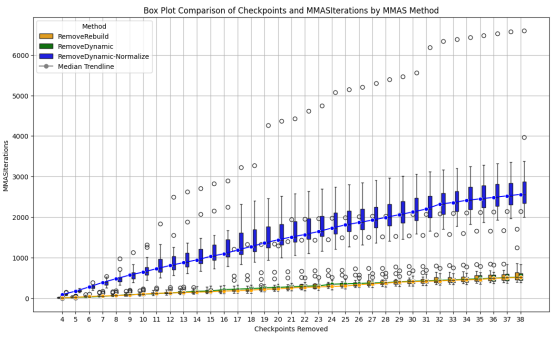
(b) Tour Length of checkpoints additions

6.1.2 Comparison techniques of Node Removals

In the previous sections, we only focused on the addition of checkpoints to the graph. Below, we see how the techniques, respond to removal of nodes, where we see the normalization technique and our method performs better than the static method. Both our and the normalization method perform quite similarly within the same confidence intervals, but it seems the normalization technique is slightly better in terms of the median in higher checkpoint levels. But in general, we cannot draw a conclusion of which of the two methods is best based on these results.



(a) Cumulative iterations of checkpoint removals



(b) Boxplots of checkpoint removals

Conclusion on dynamic vs static max min ant system

Both our dynamic and static methods achieved similar best tour lengths, indicating that with enough iterations, both methods can converge to optimal or near-optimal solutions. While our dynamic method required significantly fewer iterations to converge, particularly as the number of checkpoints increased and decreased. The static method showed increased iteration counts and wider distribution, which indicates less stability and predictability. In contrast the dynamic method proved to be more reliable and stable with narrower confidence intervals and fewer outlier compared to the static method. Moreover, the normalization technique, does not contribute to faster convergence in our problem space when adding checkpoints, but performs similarly as our method when removing checkpoints.

Overall, our implementation of a dynamic version of the Max Min Ant System shows a clear advantage in terms of iteration efficiency as the problem size increases with more checkpoints, which makes the dynamic MMAS a preferable choice for large-scale problems where checkpoints are added incrementally as in our 2D environment with agents.

6.2 The influence of heuristics in the Max Min Ant System

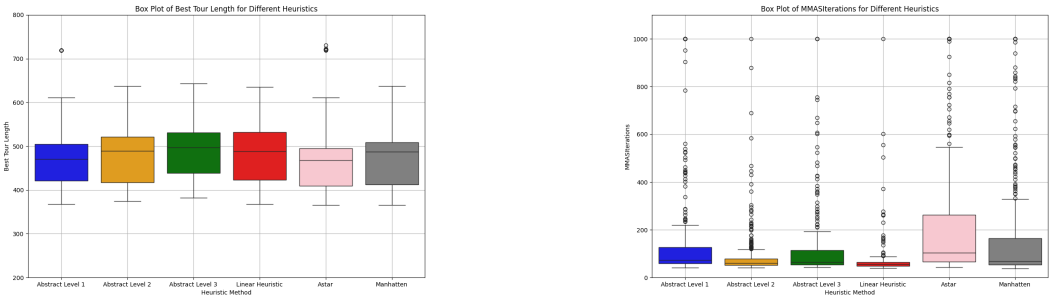
We test the heuristics under various experiments, where we change the amount of checkpoints, the map and the heuristic via different estimation of the length of the shortest path between checkpoints. Each experiment change is conducted separately.

The following shortest paths lengths were tested:

- A*
- HPA* abstraction level 1
- HPA* abstraction level 2
- HPA* abstraction level 3
- Euclidean distance
- Manhatten distance

To compare the results of each shortest tour between checkpoints, we calculated the A* shortest path length between each checkpoint along the shortest tour, which results in an fair best tour length measurement of the different heuristic experiments, since this will be the path the agent will actually walk. We conduct each experiment on a random map, with parameters: $density = 60\%$, $iterations = [17, 20]$, and different number of checkpoints. However, we cannot guarantee the amount of reachable checkpoints on each map, since checkpoints are placed randomly. Before constructing the MMAS graph, we check which points are reachable in the Map with A*. For each Map we run the MMAS algorithm 30 times to avoid drawing conclusions on random behavior. We use the same parameters for the MMAS with $alpha = 1.5$, and $beta = 4.5$ throughout all experiments.

Firstly, we conducted an experiment, where we try to compare the $BestTourLength$ of the different heuristics, where we saw this result:



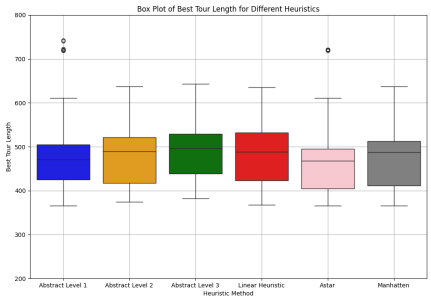
(a) Tour Lengths

(b) Iterarions

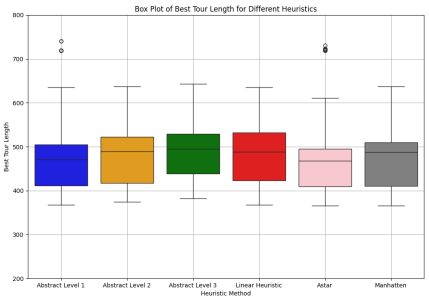
Figure 6.6: Comparison of Two Boxplots with Besttours and MMAS iterations in 30 maps with 30 iterations

In these diagrams we see that the A* yields the best tours, but it converges slower than the other heuristics. The second best is the Abstract level 1, and euclidean is the third best option. However, it is also important to notice the convergence where A* convergences much slower than the other algorithms. We will examine this further, but since we optimized the parameters of the MMAS on the $Berlin52$, we can also suspect that these parameters are better suited for a range of path lengths that are given by the different heuristics distances better, than the best for our problem instance. To prevent this, We

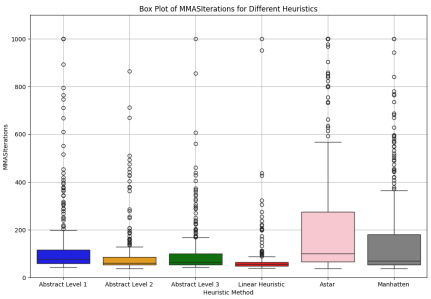
recall, that we operate in a different coordinate system than the $Berlin52$, since our is $100x100$. We therefore will there try to linearly transform the heuristic path lengths into the same interval, but with the same distributions to minimize the risk that the parameters are simply better suited for one heuristic than another. We should also note that scaling the values of the weights is similiar to changing the η in formula 2.4.2 however, we still find it interesting to determine how, the results and convergence time is determined by the heuristics.



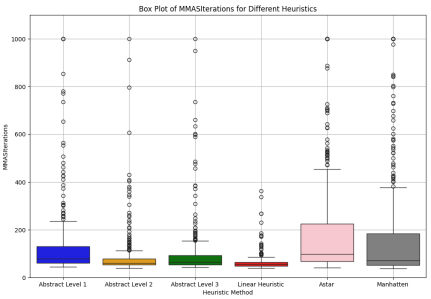
(a) Tour length when scaled 0 to 1 000



(b) Tour length when scaled 0 to 10 000



(c) Iterations when scaled 0 to 1 000

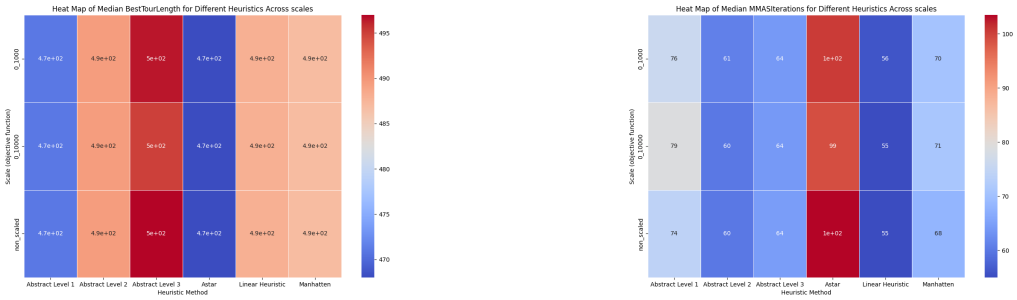


(d) Iterations when scaled 0 to 10 000

Figure 6.7: Comparison of Four Boxplots with Besttours and MMAS iterations in 30 maps with 30 iterations

We can see on the diagrams that the linear transformation had a no meaningful impact on the convergence time. Based on these 3 experiments, we can better determine which heuristic would be best for our 2D environment. Our initial thought were that A Star would be the best heuristic, since this is the actual cost of the agent walking the best tour, however, this experiment shows A* yields the BestTourLength but the convergence time is slower than Abstract Level 1, and on the below heat map it is clearly visible to see that, Abstract Level 1, and A* seems to generally produce the same BestTours. While Abstract level 1 converges significantly faster.

The is better seen in the generated median heat maps based on the best tour length and iterations:



(a) Heat Map of Median of Best tours length

(b) Heat Map Median of iterations stagnation

Figure 6.8: Comparison of Two Boxplots with Besttours and MMAS iterations in 30 maps with 30 iterations

We also note, that the euclidean heuristic performs similar Best Tours as the Abstract level 2, but the Abstract level 2 converges slower than the linear heuristic.

6.2.1 Conclusion on the influence of different heuristics in MMAS

Considering the complexity of implementing HPA* as a heuristic versus using the euclidean heuristic, this heuristic is very attractive to use due to its simplicity. However, in conclusion the Abstract level 1, is the best heuristic to use with these parameters and number of checkpoints. It is also important to denote, that as the number of checkpoints increase, likelihood of them being next to each other increases, and thus the positive results of using the euclidean heuristic should increase, as it will better approximate the real path lengths, than with lower checkpoints.

6.3 HPA* efficiency and accuracy of approximations of shortest paths

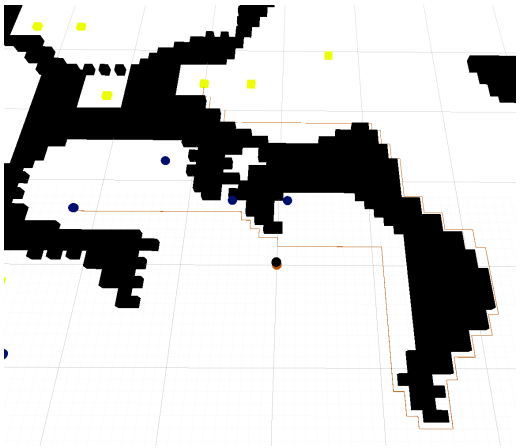
6.3.1 Experiment setups for HPA*

Before proceeding with the experiments, we first need to define a good experiment setup, so we can test the algorithm effienctly in various scenarios. Therefore, we choose to have a map size of $n * n$ where $n = 100$, unless other specified. Firstly, we did some manual tests to find good parameters to the CellularAutomata algorithm with various complexities, and obstacles.

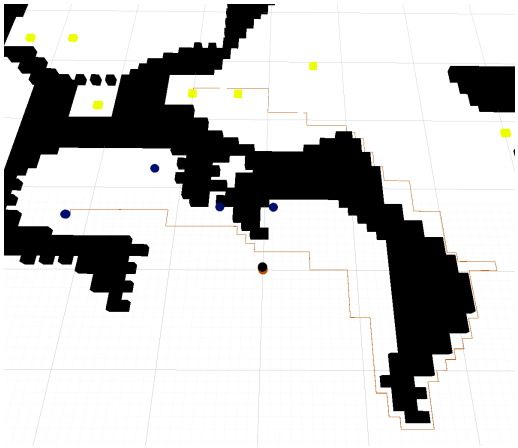
We did also investigate how the two points (start and goal), should be placed in the map, to achieve good experiments. Based on these observations we see that the Success rate is highest with 53% when $Start$ is fixed and $Goal$ is randomly placed, this indicates that these placements in the map contains a set of easier scenarios, since we also noted that the number of Explored nodes is less when placing both checkpoints randomly in the map: $Random(start, goal)$, but the paths lengths are almost the same, this indicates that the $Random(start, goal)$ yields more difficult experiments than $Fixedstart, RandomGoal$.

Having both checkpoint and goal completely fixed resulted in the lowest success rate, but also in longer paths and more explored notes, however, walls are often placed in the corners, thus this yields to lowest success rate. The$Random(start, goal)$ yield more difficult paths, than $Start$ is fixed and $Goal$ is randomly placed. Therefore we choose to proceed with complete Random placement of the start and goal in the maps throughout our experiments.

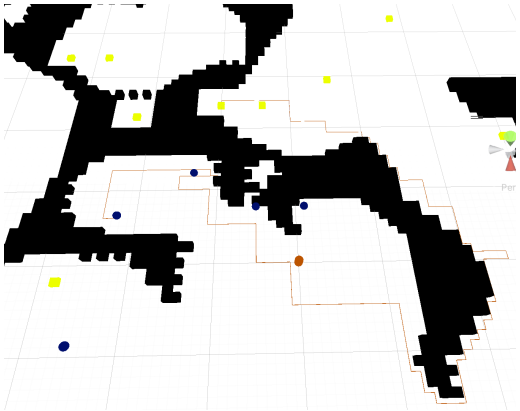
First we will make an qualitative assessment with a visualisation of path finding in the environment on AStar lowest level, abstract level 1, abstract level 2 and abstract level 3.



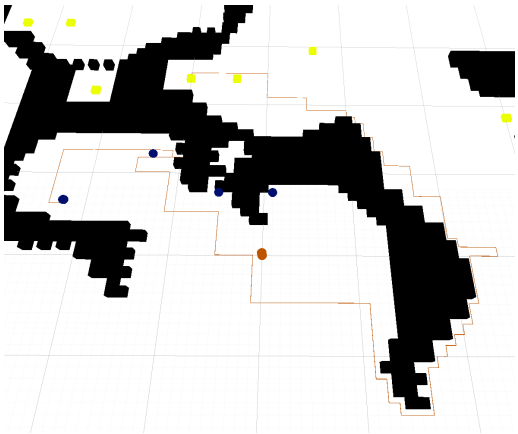
(a) A*



(b) Abstraction level 1



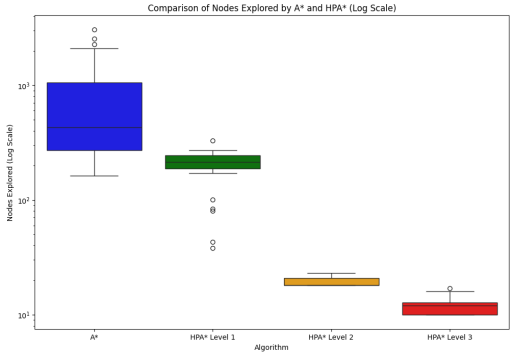
(c) Abstraction level 2



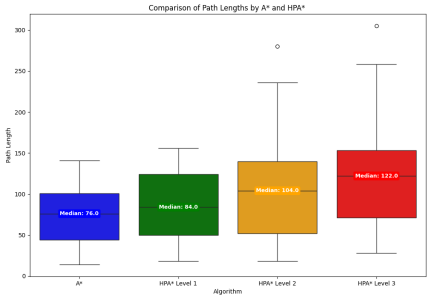
(d) Abstraction Level 3

Figure 6.9: Comparison of four shortest paths on 4 abstraction levels

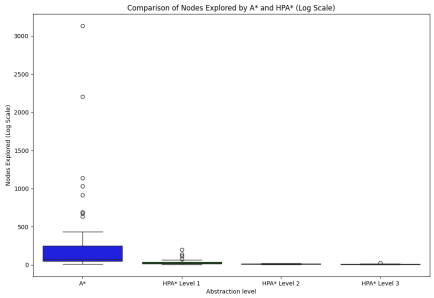
We see that in this case, A* clearly finds the shortest path, while abstraction level 1 seems better than the two other abstraction levels, since it doesn't take any large detours for cluster entrances. The Abstraction level 2 and 3, produce the same path, and generally take more detour as abstraction level 1, this is likely because the start and goal, is within a bound of cluster at level 3. In this next section, we will explore how the amount of nodes explored is correlated with the abstract and refined paths.



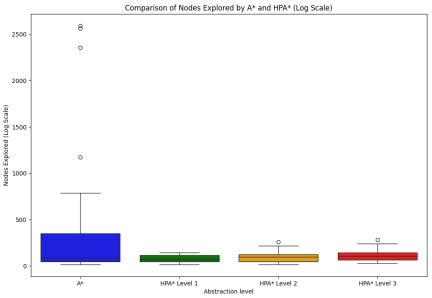
(a) Nodes explored on different abstraction levels (log scale)



(b) Path lengths on different abstraction levels



(c) Nodes explored on abstract paths across abstraction levels



(d) Nodes explored on refined paths across abstraction levels

Figure 6.10: Overview of nodes explored and path lengths across different abstraction levels

Algorithm	AvgNodesExplored	RelativeDif	AvgPathLength	PathsFound (%)
$AStar$	312.659091	0.0	69.863636	57.0
$Level1$	34.136364	-89.081922	16.022727	49.0
$Level2$	10.045455	-96.787090	8.886364	49.0
$Level3$	7.886364	-97.477648	5.795455	44.0

Table 6.1: Metrics for **abstract path** on average Nodes Explored, Relative Difference in nodes explored from A*, Average Path Length and Paths Found Percent

In the table above we see that the amount of nodes explored on both abstraction levels is significantly less than A*, and thus the computational complexity of calculating the shortest path is significantly reduced. We see that on abstract level 1, it is reduced with 89%, and on level 2 it is reduced by 96.8%, and on level 3 by 97.5 %. However, on the contrary the success of the algorithms decreases by 8 percentage points on level 1 and 2, while on level 3 by 13 percentage point. This is most likely caused by the grouping and merging of clusters while converting to intra to inter edges information is lost in this process.

Algorithm	AvgNodesExplored	RelativeDif	AvgPathLength	PathsFound (%)
$AStar$	349.255814	0.0	72.139535	57.0
$Level1$	75.744186	-78.312691	83.279070	50.0
$NLevel2$	95.627907	-72.619523	103.674419	50.0
$Level3$	110.0	-68.504461	119.511628	43.0

Table 6.2: Metrics for **refined path** on average Nodes Explored, Relative Difference in nodes explored from A*, Average Path Length and Paths Found Percent

An interesting finding when computing the refined paths without using cached paths, is that the Nodes explored increases on the abstraction levels, while the paths lengths also increases, and the success is 7 percentage point lower on level 1 and 2, but on level 3 it is 7 percentage points lower than 1 and 2. One could instead cache the paths, but the the finding shows that building larger clusters at higher abstraction levels is more computationally expensive than small clusters at lower abstraction level when producing refined paths.

6.3.2 Comparison with ”the Near optimal path finding (HPA*)”

The paper by Botea, Müller and Schaeffer [2] claims to be able to guarantee 1% accuracy based on their experiments. We have a significant less accuracy, however, they also underlines that the amount of entrances, and cluster size should be optimized to the problem. Our conversion of lower levels to higher levels seems to be the main difference between our implementation and the pseudo code outlined in the paper, however, this also saves us significant computation when producing higher abstraction levels, but at the cost of accuracy. They did not showcase their experiments on random map generation but mention that they obtained similar accuracy on 1%. It is also worth noticing, that we do not compute the path Smoothing, however, this would also produce shorter paths in specially in open areas, where the shortest path currently take a detour to an entrance between clusters as seen in the images on 6.9, and this would lower the difference in path lengths between the HPA* and A*. Another optimization would be to introduce more caching and better handling of entrances, such as grouping entrances on higher abstraction levels, but still allowing the refined path to enter a cluster in other open areas. As an example, this could create a new concept of both refined and abstract entrances. We did conduct some tests of clusters size 5, and 10, but did see that on these map types, there was slightly fewer nodes explored on abstraction level 2, when the intitial cluser size was 5, however, in future work, it could be interesting to determine the optimal cluster sizes and entrances based on different parameters such as map size, and complexity.

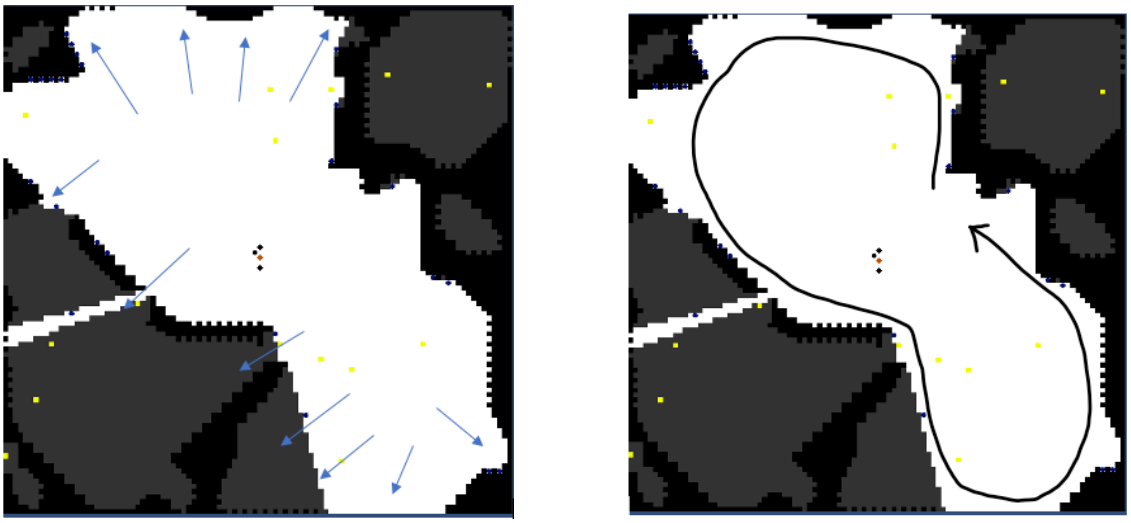
6.3.3 Conclusion on HPA* efficiency and accuracy compared to A*

The study on the HPA* and comparison with A*, shows that the amount of explored nodes are approximately 10-fold reduced which gives significant computational efficiency when using HPA* has an abstraction of the real path. When refining the path to get a real walkable path for an agent, the nodes explored are reduced by almost a 5 fold for abstraction level 1, while the paths are generally 14% longer. The higher abstraction levels for refined paths still explores fewer nodes than A*, but not fewer than level 1, while the paths are also longer.

6.4 Frontier Based Exploration

6.4.1 Evaluation and Results

Two different exploration strategies was tested and compared. The first being recomputing centroids each scan and the other one being only recomputing centroids when there is no more reachable centroids left on the map. The first strategy is the most efficient solution in terms of minimizing the total amount of steps taken by the agents. The reason being that each agent explores the map in depth first search manner in there respective directions. The other strategy explored the map in a circular manner like a breath first search which was inefficient. Since they ended up moving back and forth between the different frontier regions instead of staying at there respective positions. Furthermore, The chance of agents grouping together is increased, where the most desirable scenario in general is when agents are spread out. This can been seen in the following figures.



(a) Depth first search manner

(b) Breath first search manner

The blue arrows on the figure (a) represents the directions the agents will continue to explore in. Where the figure (b) shows that agents will walk around in a circular manner along the frontier border. It is obvious to see that in figure b the distance or amount of steps is greatly increased since it has to revisit frontier regions. A test was conducted to see the difference in amount of steps between the two strategies. Where a step is defined as one iteration of the agents, the experiment is done when they have fully discovered the map. We will define strategy A as the depth first search manner and B as the breath first search manner. The results can be seen below.

Agents	A	B
1	979	1332
4	361	504
8	217	384

Table 6.3: Strategies compared in amount of steps

They are tested on the same map with a density on 20 with 1, 4 and 8 agents. We can see in the table above that strategy A is performing significantly better than B in all scenarios. Another interesting experiment to conduct is how many scans are made by the agents in the two strategies. Especially in multi agent environments, because the total amount of scans are somehow related to the positioning of the agents. If agents are clustering

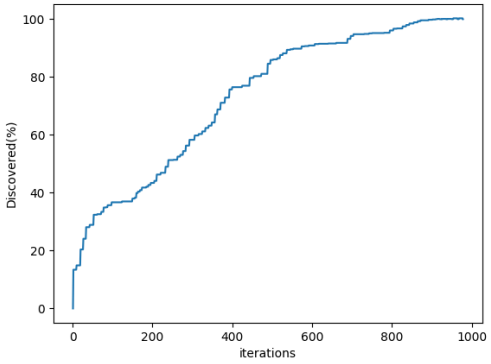
together, then a higher amount of scans occurs, since areas on the map are more likely to be scanned multiple times. The following table shows the total amount of scans from the same map used earlier.

Agents	A	B
1	89	100
4	103	106
8	106	140

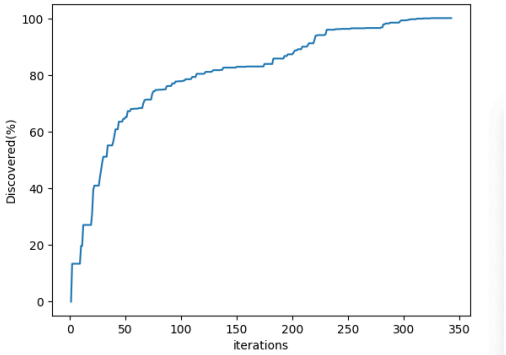
Table 6.4: Strategies compared in amount of scans

We can see that strategy A performs slightly better with 1 and 4 agents and significantly better with 8 agents. This is also expected since strategy B will group them more together than strategy A.

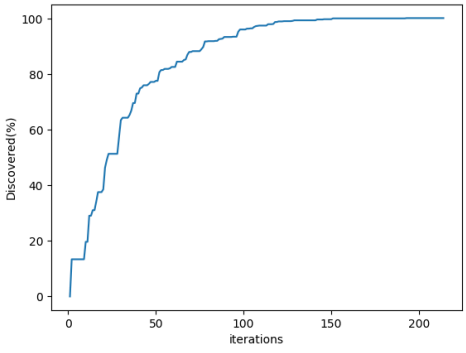
In this section a convergence analysis is used to check the rate of convergence in single and multi agent environments for the frontier based algorithm implemented. Furthermore, the algorithm is only using strategy A in these experiments. This analysis will give insight into how quickly the algorithm approaches its goal with different amount of agents. The experiments are done on maps with a density around 20 and a map size on 100. The plots below shows the amount of discovered area on the y-axis and the amount of iterations on the x-axis. figure a, b and c shows the convergence rate for 1, 4 and 8 agents respectively.



(a) Convergence rate 1 agent



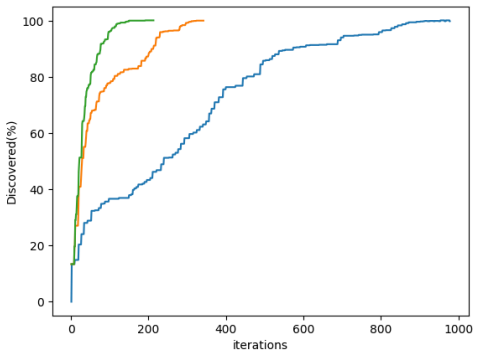
(b) Convergence rate 4 agents



(c) Convergence rate 8 agents

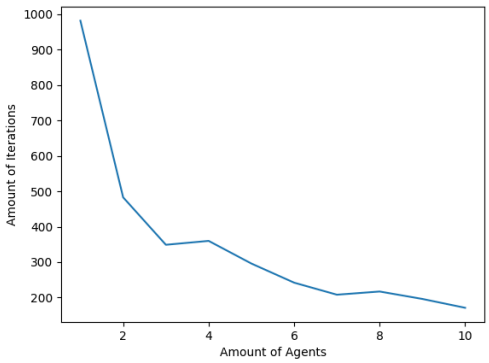
There is significant increase in convergence rate when using multiple agents. When increasing the amount of agents from 1 to 4 the amount of steps is nearly three times less. When going from 1 to 8 agents then its close to four times less. This means that multiple

agents greatly increases the convergence rate. The shape of the graphs clearly shows that the convergence rate when using 4 and 8 agents are significantly better. Since the shape of graph for one agent follows a more linear tendency. The plot below shows the three graphs combined.

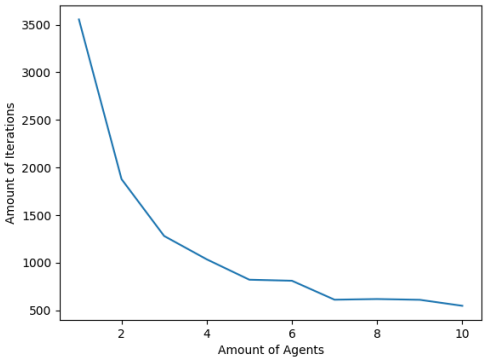


(a) Combined convergence rate

This asks the next question, is there an optimal amount of agents? To answer this question a test was made for each amount of possible agents going from 1 to 10. The map used had the size 50 and 100 with moderate density. The plot from the two experiments can be seen below.

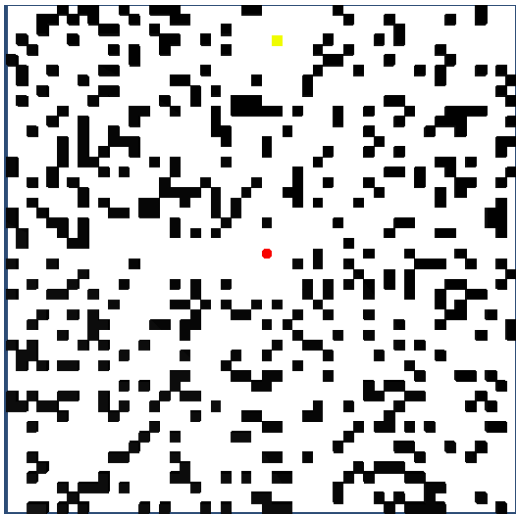


(a) Amount of iterations with map size 50



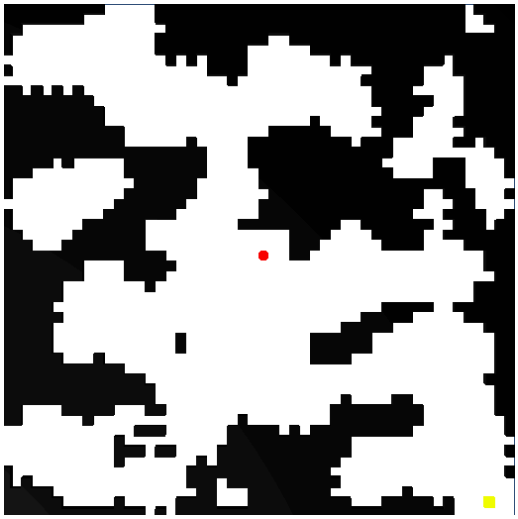
(b) Amount of iterations with map size 100

It can be seen on the two plots that the graph begins to level out around 4 agents. This means that the optimal amount of agents is around 4 since the iterations doesn't change that much compared to the interval from 1 to 4. This makes sense since the agents doesn't communicate their respective positions. Which creates the possibility of agents ending up in the same space. This has biggest negative impact on the total amount of iterations. In the table below the frontier based algorithm using strategy A is tested on low, medium high density maps. Low density maps are open with few walls scattered randomly throughout the map. Medium sized maps looks like open maze structures. High density maps looks like cave structures. The different type of maps can be seen below.



(a) Low density map

(b) Medium density map



(c) High density map

The total amount of iterations can be seen in the two tables below. The first table represents map sizes on 50 and the other one on 100.

agents/density	low	medium	high
1	954	608	235
4	365	249	140
8	224	232	98

Table 6.5: Amount of iterations on map size 50

agents/density	low	medium	high
1	3583	2621	624
4	1168	876	281
8	618	676	282

Table 6.6: Amount of iterations on map size 100

6.4.2 Conclusion/Discussion

We can conclude that a exploring strategy using a depth first search manner is significantly more effective than a breath first search both in single and multi agent environments. Furthermore, the convergence rate is increased in multi agent environments, meaning that the frontier based algorithm implemented is more suited for using multiple agents. The total amount of scans between single and multi agent environment is the same. The optimal amount of agents when using a depth first search strategy is 4.

7 Conclusion

We have implemented and evaluated how MMAS, A*, HPA*, shadow casting and frontier based exploration can be used by the agents. We also implemented simple random map generator to test and verify our algorithms in similar environments as computer games. Our findings from the evaluation of our implementation of the MMAS shows that the MMAS can be well adapted for the dynamic TSP. The dynamic adaption was further evaluated, and showed that it was significantly more efficient than a static adaption, when removing and adding new checkpoints. A simple approach of setting pheromone to τ_{max} for new edges and resetting the best tour performed best. For checkpoint removal normalizing or simply removing edges perform similarly but both better than the static method. In addition, we also evaluated different heuristics, and determined that the optimal heuristic is using the HPA* abstraction level 1 to determine path lengths between checkpoints, which balances convergence time, and best tour length. The implementation complexity of HPA* was extensive, and using the linear heuristic performed worse but is recommended for simple implementations. The nodes explored with HPA* is dramatically reduced than using A* directly which decrease the computational overhead significantly. However, this comes with the cost of path precision. We also found, that when refining paths, the nodes explored is lower on all abstraction levels than directly using A*. However, abstract level 1 explored fewer nodes than abstraction level 2 and 3, and resulted in shortest paths. Abstraction level 1 is most attractive for refining paths in our implementation. Symmetric shadow casting algorithm is an effective choice for large open maps because of the computational efficiency. It can scan large areas of the map in real time both in single and multi agent environments. When interpreting the environment using diamond shaped walls (instead of square shape) the field of view is greatly increased with no drawbacks. This decreases the overall amount of scans made by each agent. With the right implementation it is fast, consistent and symmetric. There is a possibility that tiles are detected through walls, which needs to be taken into consideration when using frontier based strategies. We can conclude that a exploring strategy using a depth first search manner is significantly more effective than a breath first search both in single and multi agent environments. Furthermore, the convergence rate is increased in multi agent environments, meaning that the frontier based algorithm implemented is more suited for using multiple agents. There is no difference between the total amount of scans in single and multi agent environments. The optimal amount of agents when using a depth first search strategy is 4. Finally, we have visualized all algorithms in Unity to create a better understanding of how the algorithms work, and by that, providing a framework for game developers, to use our algorithms as part of their game.

Bibliography

[1] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. doi: 10.1109/TSSC.1968.300136.

[2] Adi Botea, Martin Müller, and Jonathan Schaeffer. “Near optimal hierarchical path-finding (HPA*)”. In: *Journal of Game Development* 1 (Jan. 2004).

[3] Thomas Stützle and Holger Hoos. “MAX-MIN ant system”. In: 16 (Nov. 1999).

[4] Michalis Mavrovouniotis et al. “Ant Colony Optimization Algorithms for Dynamic Optimization: A Case Study of the Dynamic Travelling Salesperson Problem [Research Frontier]”. In: *IEEE Computational Intelligence Magazine* 15.1 (2020), pp. 52–63. doi: 10.1109/MCI.2019.2954644.

[5] Lawrence Johnson, Georgios Yannakakis, and Julian Togelius. “Cellular automata for real-time generation of”. In: (Sept. 2010). doi: 10.1145/1814256.1814266.

[6] Daniel Angus and Tim Hendtlass. “Dynamic Ant Colony Optimisation”. In: *Applied Intelligence* 23.1 (2005), pp. 33–38.

[7] 15-112: Fundamentals of Programming and Computer Science. *Fundamentals of Maze Generation*. https://www.cs.cmu.edu/~15112-f22/notes/student-tp-guides/Mazes.pdf. 2022.

[8] Stefan Gustavson. “Simplex noise demystified”. In: (Jan. 2005).

[9] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010.

[10] Edsger W Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271. url: https://doi.org/10.1007/BF01386390.

[11] Ira Pohl. “Bi-directional search”. In: *Machine Intelligence* 6.124-140 (1969), p. 2. url: https://www.semanticscholar.org/paper/Bi-directional-and-heuristic-search-in-path-Pohl/e609ab97186834bd97d3fbc0c97a19744d752c23.

[12] Kenny Daniel et al. “Theta*: Any-Angle Path Planning on Grids”. In: *CoRR* abs/1401.3843 (2014). arXiv: 1401.3843. url: http://arxiv.org/abs/1401.3843.

[13] Anthony Stentz. “The Focussed D* Algorithm for Real-Time Replanning”. In: *Robotics Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213 U. S. A.* 1995. url: https://robotics.caltech.edu/~jwb/courses/ME132/handouts/Dstar_ijcai95.pdf.

[14] Daniel Harabor and Alban Grastien. “Online Graph Pruning for Pathfinding On Grid Maps.” In: vol. 2. Jan. 2011. url: https://www.researchgate.net/publication/221603063_Online_Graph_Pruning_for_Pathfinding_On_Grid_Maps.

[15] Xiao Cui and Hao Shi. “A*-based Pathfinding in Modern Computer Games”. In: 11 (Nov. 2010).

[16] Rina Dechter and Judea Pearl. “Generalized best-first search strategies and the optimality of A*”. In: *J. ACM* 32.3 (July 1985), pp. 505–536. issn: 0004-5411. doi: 10.1145/3828.3830. url: https://doi.org/10.1145/3828.3830.

[17] Robert Geisberger et al. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. In: May 2008, pp. 319–333. isbn: 978-3-540-68548-7. doi: 10.1007/978-3-540-68552-4_24.

[18] Paul Tozour and I. Austin. “Building a near-optimal navigation mesh”. In: (Jan. 2002).

[19] Scott Kirkpatrick, C. Gelatt, and M. Vecchi. “Optimization by Simulated Annealing”. In: *Science (New York, N.Y.)* 220 (June 1983), pp. 671–80. doi: 10.1126/science.220.4598.671.

[20] Haris Sriwindono Alexander. “The Comparison of Genetic Algorithm and Ant Colony Optimization in Completing Travelling Salesman Problem”. In: *EAI Endorsed Transactions on Energy Web* 7.28 (2019), e4. doi: 10.4108/eai.20-9-2019.2292121. url: https://eudl.eu/pdf/10.4108/eai.20-9-2019.2292121.
[21] Marco Dorigo. “Optimization, Learning and Natural Algorithms”. PhD thesis. Italy: Politecnico di Milano, 1992.
[22] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. MIT Press, 2004.
[23] Renu Jangra and Ramesh Kait. “Analysis and comparison among Ant System; Ant Colony System and Max-Min Ant System with different parameters setting”. In: *2017 3rd International Conference on Computational Intelligence Communication Technology (CICT)*. 2017, pp. 1–4. doi: 10.1109/CIACT.2017.7977376.
[24] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. “Distributed Optimization by Ant Colonies”. In: Jan. 1991.
[25] M. Dorigo, V. Maniezzo, and A. Colorni. “Ant system: optimization by a colony of cooperating agents”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.1 (1996), pp. 29–41. doi: 10.1109/3477.484436.
[26] Emil Lundt Larsen. “Visualizing and Evaluating the Working Principles of Nature-Inspired Optimization Metaheuristics”. Supervisor: Carsten Witt. Bachelor’s thesis. June 11, 2018: DTU Compute, B.Sc. Software Technology, 2018.
[27] A. Ford. “Symmetric Shadowcasting”. In: (2020).
[28] B. Yamauchi. “A frontier-based approach for autonomous exploration”. In: *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA’97. ’Towards New Computational Principles for Robotics and Automation’*. 1997, pp. 146–151. doi: 10.1109/CIRA.1997.613851.
[29] Anirudh Topiwala, Pranav Inani, and Abhishek Kathpal. *Frontier Based Exploration for Autonomous Robot*. 2018. arXiv: 1806.03581 [cs.RO].
[30] Robert C. Martin. *Clean Architecture*. Prentice Hall, 2017.
[31] justcoding121. *Advanced Algorithms Repository*. https://github.com/justcoding121/advanced-algorithms/tree/a7bfe1555f1a623525415ab91725a3ff26b7cf69. Accessed: 2024-04-30. 2023.
[32] A. Milazzo. “Roguelike Vision Algorithms”. In: (2014).
[33] Gerhard Reinelt. *TSPLIB - A Library of Sample Instances for the TSP (and related problems) from Various Sources and of Various Types*. Accessed: 2024-04-30. 1995. url: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.
[34] Nikolaus Hansen. *The CMA Evolution Strategy: A Tutorial*. 2023. arXiv: 1604.00772 [cs.LG].
[35] Guillermo Leguizamón and Enrique Alba. “Ant colony based algorithms for dynamic optimization problems”. eng. In: *Studies in Computational Intelligence* 433 (2013), pp. 189–210. issn: 18609503, 1860949x. doi: 10.1007/978-3-642-30665-5_9.
[36] Casper Eyckelhof and Marko Snoek. “Ant systems for a dynamic TSP”. In: vol. 2463. Sept. 2002, pp. 88–99. isbn: 978-3-540-44146-5. doi: 10.1007/3-540-45724-0_8.

8 Appendix

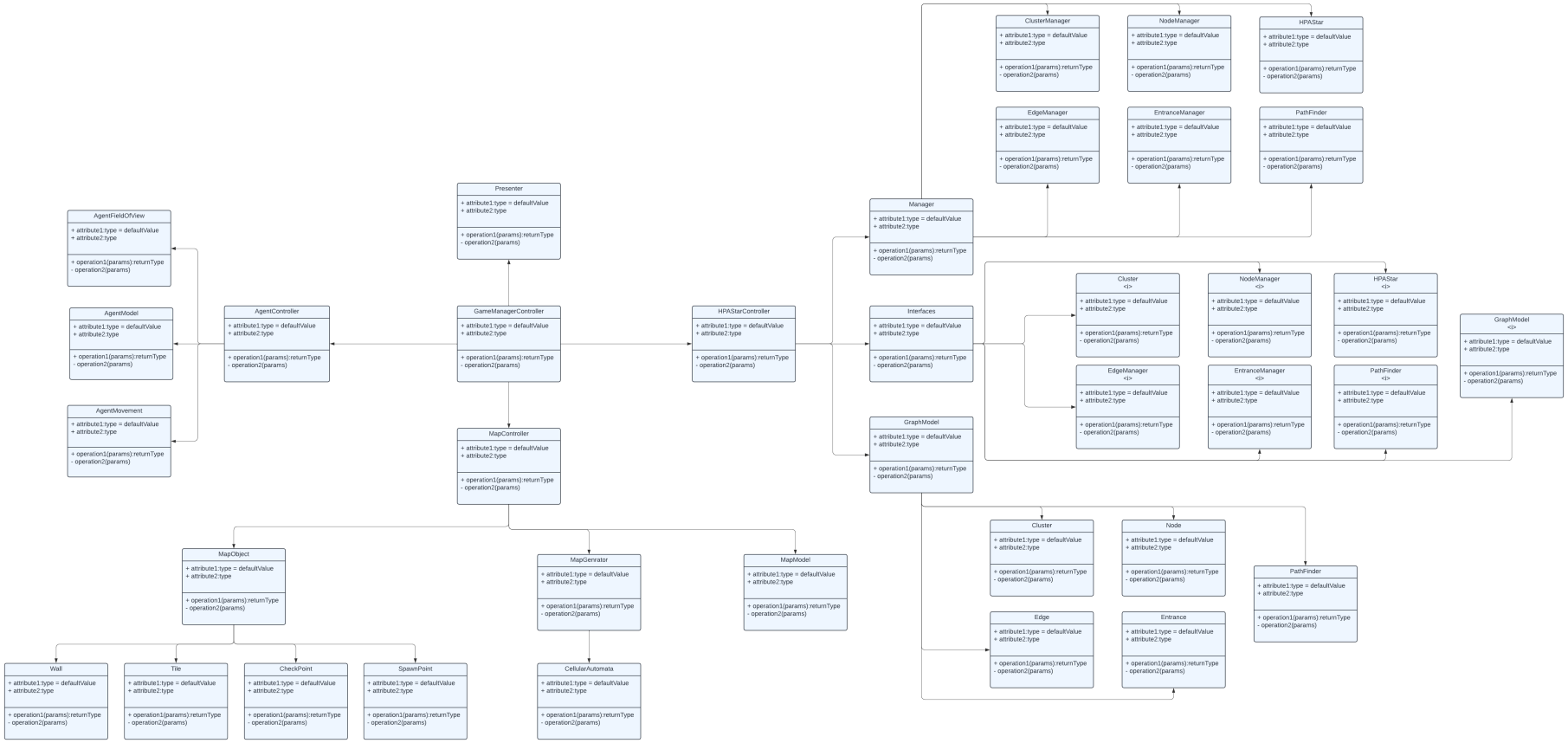


Figure 8.1: UML Class Diagram