NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Final Project

Build Intelligent Agents for Tileworld Environment

School of Computer Science and Engineering
Programme: MSAI
Date: 21 April 2024
Team Members:

- Anushka Pandey - G2303190F
- Ntambara Etienne - G2304253K
- Pham Minh Khoi - G2303923E
- Su Su Yi - G2304221L
- Uwineza Joseph - G2303477F

I. Introduction

In this project, we develop agents that inhabit and operate within the given Tileworld environment using the MASON agent toolkit. The Tileworld is a chessboard-like grid on which there are agents, tiles, obstacles, and holes (2). An agent is defined as a unit square with the capability to perform a range of pre-set actions, typically consisting of moving one cell at a time in any of the four cardinal directions. A tile, also a unit square, can be transported by the agent. An obstacle is an assembly of grid cells that cannot be moved. A hole is a collection of grid cells, each capable of being filled by a tile when the tile is positioned over the hole cell, resulting in the disappearance of both the tile and hole cell, leaving an empty cell (2). When a hole is entirely filled, the agent is rewarded for its effort (2). The agent is aware in advance of the hole's worth; its primary objective is to maximize rewards by filling as many holes as possible (2). Furthermore, agents are assigned a fuel level, for which they are accountable for managing (1). To assist agents in sustaining their fuel levels, a fuel station is incorporated into the environment (1).

A Tileworld simulation unfolds dynamically. The process initiates in a state that is randomly produced by the simulator in accordance with a set of parameters and evolves incessantly over time (2). Objects (such as holes, tiles, and obstacles) materialize and vanish at rates dictated by the predetermined parameters, and the fuel station is randomly established at the outset and remains in a fixed position. The agent has the capacity to transport up to 3 tiles simultaneously and deplete one fuel at each progression. When the fuel quantity depletes to zero, the agent becomes immobile and remains stranded at the present location. The agent's visibility is limited and can only discern $\max(|x-X|, |y-Y|) \leq 3$ adjacent cells where (x, y) represents the agent's coordinates and (X, Y) represents the object's coordinates.

The main objective of this project is to get as many rewards as possible within a predefined number of steps of simulation. To achieve that, it is important to design effective strategies for agents to navigate the environment while achieving specific goals or objectives. This involves tasks such as finding optimal paths from starting locations to goal locations, avoiding collisions with other agents or obstacles, refueling in time, and potentially coordinating actions to optimize overall performance. We present further details of how each of our agent is designed in section III.

The Tileworld problem serves as a fundamental model in the field of multi-agent systems for evaluating agent architectures and behaviors within a simulated environment. Tileworld provides a structured framework to facilitate the study of planning, decision making, coordination, and pathfinding among multiple agents. It is used in various applications including robotics, autonomous systems, video games, and simulation environments. The following section describes the main algorithm we used for pathfinding, which is A* search.

II. A* Searching Algorithm

The A* (A star) search algorithm is a widely used pathfinding and graph traversal algorithm that efficiently finds the shortest path from a start node to a goal node. It combines elements of Dijkstra's algorithm, which focuses on finding the shortest path, with a heuristic approach that guides the search process more efficiently. In A*, each node is assigned a cost function $f(n)$, which is calculated as the sum of two values:

- $g(n)$: The actual cost from the start node to the current node n.
- $h(n)$: The estimated cost (heuristic) from the current node n to the goal node.

The formula for the cost function is:

$$f(n) = g(n) + h(n) \tag{1}$$

The algorithm uses a priority queue to explore nodes in order of their $f(n)$ value, prioritizing nodes with the lowest cost. This approach balances the exploration of known paths (using $g(n)$) and the estimated distance to the goal (using $h(n)$). The heuristic function $h(n)$ plays a critical role in the algorithm's efficiency by providing an estimate of the distance to the goal node, helping prioritize nodes closer to the goal, and minimize the exploration of less promising paths.

In this project, the $g(n)$ function is the number of steps that the agent has taken to get to the current cell as all steps only cost one fuel unit. On the other hand, $h(n)$ is estimated using the square of Euclidean distance

$$h(n) = (x_{target} - x_{source})^2 + (y_{target} - y_{source})^2 \tag{2}$$

In multi-agent scenarios, A* can dynamically adjust paths based on the movements and actions of multiple agents. This allows each agent to find efficient paths while avoiding collisions and coordinating with other agents. A*'s flexibility and effectiveness make it a powerful tool for solving complex pathfinding problems in multi-agent environments.

III. Agents: Design, Results, Discussion

In this section, we will present the design and architecture of our agents and discuss the results achieved by the agents. As a group of five members, we each have developed our own agent. Through the subsections from A to E, each agent design is explained individually covering the architecture, results analysis, and discussion.

A. NtamAgent

Introduction The design and implementation of an agent-based simulation framework based on the Tile World simulation environment are examined in this research. In the simulation, agents function independently in a constantly changing setting, performing various activities including filling up tanks, gathering tiles, and placing tiles in holes. The main objective is to evaluate the agents' capacity for resource management and navigation in a limited area. The simulation provides a strong foundation for researching artificial intelligence and the behaviors of multi-agent systems since it employs Java to structure the environment, agents, and interactions.

Design Architecture The main Java classes that make up the simulation architecture each have specific functions inside the system. Grid size, object generation rates, and agent settings are among the setup options that control the overall dynamics of the simulation and are stored in the `Parameters` class. Serving as the focal point, `TWEnvironment` oversees all aspects of the simulation state, including the grid, agents, and object lifecycle. Subclasses of the `TWAgent` abstract class are used to instantiate agents, giving them the ability to customize behaviors while still having access to shared features like movement and sensing. A particular `TWAgent` implementation called `NtamAgent` has logic for making decisions depending on the agent's current state and external variables.The `TWAgentWorkingMemory` class manages memory by tracking viewed things and gradually deteriorating these memories to mimic forgetfulness. The modular design of the system allows for easy modifications and scalability due to the clear division of responsibilities.

Planning Algorithm Reactive decision-making is employed by the agents, taking into account their present perspective of the surroundings. `NtamAgent`'s decision points optimize actions like refueling, picking up, or placing tiles by taking into account elements like holes, the number of tiles, and proximity to fuel stations. To add to the simulation's unpredictable nature, the environment also creates and eliminates tiles, holes, and obstructions according to predetermined statistical distributions.

No	**Parameters**	**Values 1**	**Values 2**
1	Environment size	50x50	70x70
2	holeMean	0.2	1
3	tileMean	0.2	1
4	obstacleMean	0.2	1
5	tileDev	0.05f	0.5f
6	holeDev	0.05f	0.5f
7	obstacleDev	0.05f	0.5f
8	lifetime	100	50
9	Threshold	100	100
10	defaultFuelLevel	500	500
	Avg Score	**60**	**107**

Table 1: Results achieved under different environment settings in 180 seconds operating.

Results Analysis The agent's performance in two different environmental circumstances is displayed in the results shown in (Table 1). The agent achieved a score of 60 in 180 seconds in Values 1, despite the smaller grid and lower mean values for tile and hole production. In Values 2, the agent raised its score to 107 even though the environment was more difficult due to a wider grid and higher mean values. The agent's methods are resilient to the increasing frequency and unpredictability of barriers and objectives, as evidenced by the constant threshold and fuel level seen in both trials. Based on the statistics, it appears that the agent may be taking advantage of Values 2's denser surroundings in order to improve its score throughout the same time frame.

Drawback The current implementation's inefficient memory management, which results in possible scalability concerns as the environment size increases, is one of its main shortcomings. The grid is iterated to decay old memories. Furthermore, the simulation makes the assumption that the environment is static in between agent activities, which might not fully reflect the dynamics of the real world where changes happen more fluidly. The intricacy of actions that can be represented is limited because the agents' reactive nature prevents them from learning from prior experiences or long-term strategic planning.

Improvement Enhancements might involve the implementation of more complex memory decay algorithms that effectively manage big datasets, possibly by utilizing more complicated data structures. A more dynamic environment update system might make it possible for the simulation to replicate real-world circumstances more accurately. Agents could be able to adjust their methods over time and learn from past events by integrating machine learning techniques. Use other advanced algorithms that has the capabilities to run in different settings greater than 70x70.

B. CustomAgent

Introduction The Custom Agent class provides the design implementation of an agent customized from the standard TWAgent class in the TileWorld simulation package which helps to provide a dynamic environment, where different agents interact with each other and the tileworld infrastructure to compete to collect rewards (earned by collecting a tile and filling a hole with it).

Design Architecture The architecture of the `CustomAgent` class consists of four main components listed below:

Fuel Level Check: At each time step, the agent assesses its current fuel level. If the fuel level falls below a predefined threshold, indicating a potential risk of running out of fuel, the agent initiates a plan to navigate towards the fuel station for refueling.

Path Planning to Fuel Station: When the fuel level is deemed insufficient, the agent employs a pathfinding algorithm, likely A* search, to compute an optimal path to the nearest fuel station. This ensures that the agent can refuel in a timely manner to avoid running out of energy during its tasks.

Tile Capacity Check: If the fuel level is above the threshold, indicating that the agent has sufficient energy reserves, it proceeds to evaluate its current tile capacity.

Tile Management: The tile management module includes different reasoning procedure in order to decide the suitable action for the tiles

1. Tiles Carry Capacity: If the agent already possesses three tiles, which is its maximum carrying capacity, it prioritizes finding a suitable location to drop one of the tiles. This prevents inefficiencies caused by carrying excess tiles beyond the capacity limit.

2. Tile Pickup: If the agent has not reached the three-tile limit and is currently positioned on a cell containing a tile, it picks up the tile to add to its inventory. This action ensures that the agent maximizes its tile collection capacity as it navigates the environment.

3. Tile Drop-off: Conversely, if the agent is situated on a cell with a hole and it possesses at least one tile in its inventory, it strategically drops one of the tiles into the hole. This action effectively manages the agent's tile inventory and contributes to achieving its objectives within the environment.

Discussion Motivated Exploration: The architecture incentivizes agents to explore the environment actively, driven by the goal of maximizing rewards. This motivation encourages comprehensive exploration and facilitates the discovery of valuable resources and strategic locations.

A* Pathfinding: The utilization of the A* search algorithm for pathfinding ensures that agents can efficiently navigate the complex and dynamic terrain of the Tile World environment. A* search is renowned for its effectiveness in finding optimal paths, enabling agents to reach their destinations promptly while conserving energy.

Adaptability to Dynamic Environment: The design of the agent architecture allows for seamless operation in dynamic environments where conditions and objectives may change over time. This adaptability ensures that agents can respond effectively to new challenges and opportunities as they arise.

Improvement Communication Module Implementation: Introducing a communication module enables enhanced collaboration and coordination among agents operating within the Tile World environment. By facilitating the exchange of information and strategies, agents can optimize their collective efforts and achieve superior performance in achieving shared goals.

Fuel Management Optimization: Monitoring the distance from the fueling station at each time step and proactively planning movements to ensure that the agent's current fuel level is adequate for returning to refuel if necessary enhances fuel efficiency and prevents energy depletion incidents.

Exploration Strategy Refinement: Implementing mechanisms to penalize agents for excessive random exploration and initiating re-strategization procedures in such instances promotes more strategic exploration behaviors. By discouraging aimless wandering and encouraging targeted exploration, agents can utilize their resources more effectively and achieve objectives more efficiently.

Results The experiments result in Table 3 illustrate the operating performance in two different configurations given in the requirements

No	**Parameters**	**Values 1**	**Values 2**
1	tileMean	0.2	2
2	holeMean	0.2	2
3	grid_size	50x50	80x80
4	obstacleMean	0.2	2
5	tileDev	0.05f	0.5f
6	holeDev	0.05f	0.5f
7	obstacleDev	0.05f	0.5f
8	lifetime	100	30
	Avg Score	**69**	**109**

Table 2: Setting of the environment under two fixed configurations.

C. AgentCC

Introduction AgentCC class extends the TWAgent class and is designed to navigate and operate efficiently within the Tileworld environment. With a focus on fuel management, decision-making, and pathfinding, AgentCC aims to accomplish tasks such as picking up and putting down tiles, refueling at designated stations, and optimizing its movements to maximize rewards.

Design and Architecture The architecture of an agent outlines the arrangement and interactions of its internal components, facilitating its efficient operation within the environment. These components typically comprise mechanisms for perception, reasoning, learning, and action. Reactive agent architecture is used for designing AgentCC where the agent reacts directly to the environment without needing a complex internal model and makes decisions based on the current state of the surroundings.

Fuel Management: AgentCC utilizes a threshold-based approach to manage its fuel constraint. When its fuel level reaches to or drops below the pre-defined threshold, it plans a route to the fuel station for refueling. This helps prevent the agent from running out of fuel during its tasks and keeps it running smoothly without getting stuck. Additionally, when the agent is already at the fueling station, it checks if its fuel level is lower than the default level. If it is, the agent refuels to make sure it has enough fuel for its next adventures. This approach ensures the agent always has the energy it needs to keep going without any interruptions.

Planning: During the agent's think process, it first checks its memory to identify nearby holes and tiles. If the agent is already carrying tiles and finds itself at the same location as a hole, it will proceed to put down the tile in that hole. However, if there is a hole closer to the agent than the nearest tile, it will adjust its path to move towards the hole and then put down the tile because it is designed to prioritize filling the hole to get reward. On the other hand, if the agent encounters a tile at its current location, it will pick it up. If there are tiles nearby but none at the agent's location, it will move towards the nearest tile. In cases where there are neither tiles nor holes nearby, the agent will opt to move in a random direction, allowing it to explore and potentially discover new tiles or holes in its environment.

After completing its thought process, the agent proceeds to act based on its decisions. It employs the A* algorithm to find optimal paths in its environment, ensuring efficient navigation. When calculating distances to tiles and holes, the agent uses the Manhattan distance metric, providing a straightforward measure of proximity. Additionally, during random exploration, the agent actively avoids moving towards the edges of its environment, minimizing the risk of getting stuck or encountering obstacles. This cautious approach enhances the agent's ability to explore its surroundings effectively while mitigating potential navigation challenges.

Result Analysis The average rewards achieved by AgentCC in two different parameter settings is shown in Table 3. The simulation is run for 5000 time steps and the initial fuel level is 500. The fuel threshold is set to 75 for both configurations. We do 10 experiments for each configuration and take the average. Under values 1 setting, the agent scores the average of 61 and under values 2, the agent obtains the average score of 159.

No	**Parameters**	**Values 1**	**Values 2**
1	tileMean	0.2	2
2	holeMean	0.2	2
3	grid_size	50x50	80x80
4	obstacleMean	0.2	2
5	tileDev	0.05f	0.5f
6	holeDev	0.05f	0.5f
7	obstacleDev	0.05f	0.5f
8	lifetime	100	30
	Avg Score	**61**	**159**

Table 3: Results obtained by AgentCC under two sets of configuration in 5000 time steps with 500 fuel level at the start.

Discussion and Improvement Advantages: The approach incorporates a proactive fuel management system, which ensures that the agent maintains sufficient fuel reserves to sustain its operations. By defining a threshold for fuel level and planning a path to refuel when the fuel level drops below this threshold, the agent minimizes the risk of running out of fuel during critical tasks, thereby enhancing its reliability and operational efficiency. Additionally, the agent's decision-making process is guided by a set of clear and concise rules, facilitating efficient navigation and task execution. By considering the proximity of holes and tiles stored in memory, the agent can prioritize actions such as picking up or putting down tiles based on their spatial relationships. Furthermore, the utilization of the A* algorithm for pathfinding, coupled with the Manhattan distance metric for distance calculation, enables the agent to navigate its environment optimally, reducing computational overhead and improving overall performance.

Drawback: However, despite its strengths, the approach has some limitations. One notable drawback is its reliance on predefined thresholds and rules, which may not always account for dynamic changes or unforeseen circumstances in the environment. For instance, the fixed fuel threshold and deterministic decision-making process may lead to suboptimal outcomes in complex or unpredictable scenarios, potentially limiting the agent's adaptability and robustness. Furthermore, the use of a simplistic random movement strategy when no tiles or holes are nearby may result in inefficient exploration and resource gathering, particularly in large or sparsely populated environments. Additionally, while the A* algorithm is effective for pathfinding, its computational complexity may become prohibitive in resource-constrained or real-time applications, leading to delays in decision-making and action execution.

Improvement: To address the limitations of the current approach and further enhance the agent's capabilities, several key improvements could be implemented. Firstly, enhancing memory management mechanisms would enable the agent to store and retrieve information more efficiently, improving its ability to make informed decisions based on past experiences. Implementing techniques such as prioritized memory storage or dynamic memory allocation could optimize memory usage and facilitate faster access to relevant information. Additionally, fostering communication and collaboration with other agents in the environment could enable the exchange of valuable information and resources, enhancing collective decision-making and task execution. Implementing communication protocols and mechanisms for sharing knowledge and coordinating actions could promote synergy among agents and improve overall system performance. Furthermore, optimizing the planning process for paths to the fuel station could minimize fuel consumption and enhance the agent's energy efficiency. Incorporating factors such as terrain topology, fuel station proximity, and expected fuel consumption rates into path planning algorithms could enable the agent to identify more fuel-efficient routes and reduce unnecessary energy expenditure. Moreover, developing an adaptive refueling strategy that takes into account the current state of the environment would enhance the agent's ability to manage fuel resources effectively. By considering factors such as distance to the fuel station, number of tiles carried, and proximity to the nearest hole, the agent could dynamically adjust its refueling decisions to optimize energy usage and ensure continuous operation without unnecessary interruptions. Overall, by implementing these improvements, the agent could achieve greater autonomy, efficiency, and adaptability, enabling it to perform more effectively in diverse and dynamic environments.

D. MyTWAgent

Introduction The `MyTWAgent` class is an extension of the `TWAgent` in the TileWorld simulation, which is designed to provide an interactive, dynamic environment where agents can perform various tasks such as collecting tiles, filling holes, and navigating around obstacles. The class specifically encapsulates the behaviors and strategies necessary for an agent to function autonomously within this environment, addressing fundamental challenges like fuel management, object interaction, and movement strategies.

Design `MyTWAgent` inherits from `TWAgent`, thereby utilizing the framework established for basic agent functionalities, such as movement, sensing, and memory. Key attributes added in `MyTWAgent` include:

1. Fuel Management: The agent keeps track of its fuel levels and the location of a fuel station `(fuelStationX, fuelStationY)`. A critical attribute `FUEL_THRESHOLD` dictates when the agent should prioritize refueling.

2. Autonomous Decision Making: Implemented through the `think()` method, decisions are made based on the agent's current state, including its fuel level, possession of tiles, and proximity to relevant objects (tiles and holes).

The agent architecture contains some necessary methods that allow the agent to perform planning, take action, and object navigation. Critical methods and functionalities such as `think()` is central to the agent's autonomous behavior, this method evaluates the agent's current needs (e.g., refueling, picking up or dropping tiles) and environmental conditions to decide the next action. `act(TWThought thought)` function allows the agent to execute the action decided in the `think()` method, handles movement and interactions with tiles and holes, and manages exceptions like blocked paths. Navigation helpers methods such as `getDirectionToNearestFuelStation()` and `getRandomDirection()` aid in navigating the grid environment efficiently. Utility methods include functions for handling blocked paths `(handleBlockedCell())` and consuming fuel `consumeFuel()`, ensuring the agent operates within its operational constraints.

Results The simulation results from deploying `MyTWAgent` can be evaluated based on several metrics such as efficiency in task completion, fuel usage, and adaptability to environmental changes. Key observations include:

1. Fuel Efficiency: Concerns on how effectively MyTWAgent manages its fuel, and how often it needs to refuel, The agent's ability to minimize unnecessary movements and effectively plan its path to the fuel station is crucial.

2. Task Completion: Efficiency in collecting and depositing tiles, and how the agent prioritizes these tasks relative to its fuel state.

3. Adaptability: The agent's ability to handle dynamic changes in the environment, such as appearing or disappearing tiles and holes, and how it navigates around obstacles.

No	**Parameters**	**Values 1**	**Values 2**
1	tileMean	0.2	1
2	holeMean	0.2	1
3	obstacleMean	0.2	1
5	tileDev	0.05f	0.5f
6	holeDev	0.05f	0.5f
7	obstacleDev	0.05f	0.5f
8	Enviroment size	50×50	80×80
	Scores under 60 seconds	**12**	**23**

Table 4: Shows the setting of the environment under two different values at each parameter.

As shown from the table 4, the agent in the TileWorld simulation achieves a higher score with more densely populated settings (Values 2), likely due to increased interactions with objects like tiles and holes. The variability in object generation may offer more scoring opportunities, suggesting that the agent's strategies are effective in busy environments. Frequent object interactions enhance performance even if they introduce navigational challenges. These findings highlight the importance of environmental parameters in influencing agent efficiency and the need for adaptable agent strategies to optimize performance under varying conditions.

Discussion

`MyTWAgent` demonstrates basic decision-making capabilities that align closely with survival instincts (like refueling) and task-oriented objectives (managing tiles and holes). However, the intelligence of the agent can be further enhanced by integrating more complex algorithms for pathfinding and perhaps machine learning techniques for dynamic strategy optimization based on past outcomes. The current implementation provides a solid foundation, but scalability can be an issue as the complexity of the environment or the number of agents increases. Performance optimizations and more sophisticated memory management techniques could be necessary to handle larger or more complex simulations. While `MyTWAgent` operates effectively in isolation, real-world scenarios often require coordinate effort among multiple agents. Extending the `communicate()` method to support inter-agent collaboration could lead to more sophisticated interaction patterns and collective problem-solving strategies. The `MyTWAgent` class in the TileWorld simulation framework exemplifies a robust model for autonomous agents in a simulated environment, with fundamental capabilities in navigation, task management, and self-preservation.

Improvements Future work should focus on enhancing the agent's decision-making framework, improving performance, and enabling multi-agent collaboration to better simulate complex real-world scenarios. This agent serves not only as a functional component within the TileWorld simulation but also as a valuable model for studying autonomous behavior in controlled environments.

E. Nearest-first Search and Random Maximum Explore Agent (KhoiAgent)

Following the previous architecture, we will introduce an additional memory module which assigns scores to the grid cells, helping to encourage the agent to explore the unknown area of the environment.

Memory The default memory implemented in the package consists of a simulated map of the environment which stores the positions of the entities such as tiles, holes, obstacles, and the fuel station. Each cell in this 2-dimensional array stores the entity and the time step, at which the cell is observed by the agent. This memory is updated after every time step by setting the value of those outdated cells to null by comparing the current time step with the stored time step of the cells. There are some other functions that help to update the object in the cell or remove the object from the cell `removeAgentPercept()`, retrieve the nearest tile `getNearestTile()` or hole `getNearestHole()`. Except from the default design of the `TWAgentWorkingMemory` implemented in the MASON package. We have added a copy of the environment which stores the scores for each grid cell which indicates the potential values that the agent would get if it is heading towards that area. This module is basically a 2-dimensional array with the same size as the map, and it is initialized with the values of 1 for every cell. During the operation process, all the values of the cells located in the agent's vision range will be reset to 0, and others will be incremented by 1 after each step of the environment. Figure 1 shows the transition

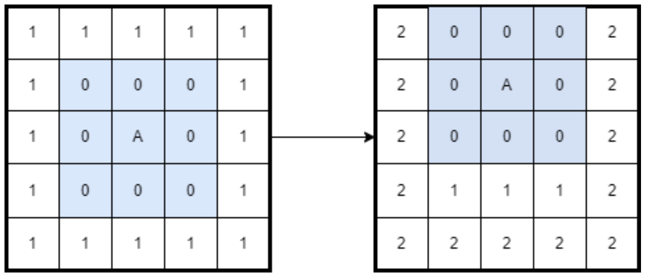

Figure 1: The transition from time step $t - 1$ (left) to time step t (right) and the update of the score map.

between time step $t - 1$ to time step t. The score map will be updated to ensure the agent always explores the unknown area.

Random Maximum Explore The maximum exploration random walk is shown in Figure 2. The algorithm will iterate through each possible moving action and compute the cumulative exploration scores, which are coloured in orange, purple, yellow, and green in the figure. The algorithm will only take into consideration the directions that are not blocked by the obstacles. Therefore, sometimes the directions with the highest cumulative score are not chosen to be the possible actions and result in several equal scores actions. In this case, all the actions will be chosen randomly but it still not is a blind random walk.

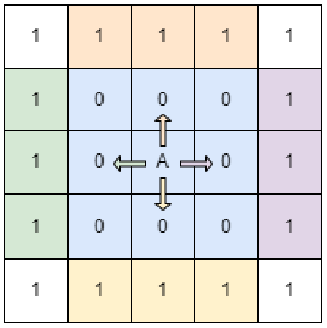

Figure 2: Maximum searching algorithm.

Planning The planning process is not different much from the previous architecture except for switching the order of the fuel checking by placing it after checking the default actions such as `PICKUP`, `PUTDOWN`, and `REFUEL`. The reason for this placement is that they do not cost any fuel to perform those actions and they provide instance rewards like `PICKUP` will save fuel for the agent instead of finding a tile later, `PUTDOWN` will give agent rewards immediately and also it is the main goal for the agent to operate. The `REFUEL` action will also check if the agent's fuel level is lower than 90% of the max fuel level in order to avoid the infinite loop of this action.

The fuel checking process implemented in `checkFuel()` function will only activate if the agent has the location of the fuel station stored in memory. This is to avoid random searches when the fuel level is low. The agent will aim to maximize rewards before it stops working. Therefore, the algorithm will have the following flow:

All the find path functions mentioned in the Algorithm1 `findPathToStation()`, `findPathToTile()`, and `findPathToHole()` have the same format by replacing the entity that we are searching for. The pseudo codes of them are shown in the Algorithm 2. The path-finding algorithm is A^* algorithm and it will only return `null` if the algorithm reaches the maximum searching step but still has not found the entity or the path is blocked by obstacles.

Experiment results The experiments of the agent are shown in Table 5. It is clear that under the same setting given in the requirement of this project, the exploring procedure has leveraged the chance for the agents to figure out the location of the fuel station, which has led to an unlimited amount of fuel for the agent to operate. With the smaller environment size, the agent will have a higher probability of figuring out the location of the fuel station. Therefore, it is reasonable that the result in the smaller environment, at 371.3, is much higher than in the large environment, at 204.4. Hyperparameters, including the fuel threshold and the maximum memory limit for tiles, holes, and obstacles, are adjusted in accordance with the environment to enhance their performance. The fuel threshold is determined by the formula $(\texttt{xDimension} + \texttt{yDimension})/2$, while the memory limit for both tiles and holes is set to $\texttt{0.3} \times$

Algorithm 1: Planning module pseudo code for Max-Explore Agent

```
urgent ← false
if canPickupTile then
    return PICKUP
else if canPutdownTile then
    return PICKUP
else if fuel level $<$ 0.9 max fuel then
    return REFUEL
end ifurgent ← checkFuel()
if urgent = true then
    return findPathToStation()
else
    if checkCarriedTile() = 3 then
        return findPathToHole()
    else if checkCarriedTile() = 0 then
        return findPathToTile()
    else
        if distance to tile $<$ distance to hole then
            return findPathToTile()
        else
            return findPathToHole()
        end if
    end if
end if
```

Algorithm 2: Finding path module pseudo code

```
TWEntity e
AstarPathGenerator astar
path ← astar.findPath(e)
if path = null then
    return maxExploreSearch()
else
    return path
end if
```

Parameters.lifetime. These scaling adjustments enable the agent to respond more effectively to environmental changes, thereby improving its overall performance.

Architecture analysis and future improvements The agent is built on a hybrid framework, which not only holds a symbolic depiction of the environment in its memory, but also has the ability to react to changes, thus operating effectively in a variable environment. The max explore approach encourages the agent to explore uncharted areas, enhancing the chances of locating the fuel station. The adjusted parameters allow the agent to adjust to the environment without the necessity for manual parameter modifications.

The drawbacks of the architecture are that it is lacking of a communication method and the searching procedure does not have scalability. Firstly, the agents operate individually without communication which has caused them to run out of fuel and not effectively distribute the resources. An improvement to this could be constructing a communication protocol for all agents which allows them to share

No.	**Parameters**	**xZZZZValues 1**	**Values 2**
1	Enviroment size	50×50	80×80
2	tileMean	0.2	2
3	holeMean	0.2	2
4	obstacleMean	0.2	2
5	tileDev	0.05	0.5
6	holeDev	0.05	0.5
7	obstacleDev	0.05	0.5
8	lifeTime	100	30
	Avg. Score	**371.3**	**204.4**

Table 5: Experiments results of Max-explore Agent under 2 sets of parameters in 5000 steps.

the location of the fuel station, tiles, and holes. For instance, when one agent has carried the maximum number of tiles while the other agents are struggling to find a tile then the agents that are in need of tiles can send a message that contains the ”sender” identity, the ”receiver” identity and the content of messages such as "Agent1 Agent2 REQUEST-TILE" and the receiver can reply by adding the location of the tile to the message "Agent2 Agent1 REPLY-TILE-X-Y". This will avoid sharing the memory, which is crucial in data security matters, yet still allows the agent to communicate for their need. Secondly, all the agents are searching the whole map which will cause redundant movements, and consume more memory to design each agent whilst not able to utilize the existing lifetime of the tiles and the holes. Therefore, a possible solution to this is that we can use a divide-and-conquer procedure by dividing the map into columns or rows, then, assigning each area with an agent. This will limit the working area of each agent and allow them to return back to the previous area faster in order to avoid the tile from disappearing. Moreover, each agent can have a broader picture of their territory and can decide whether to request other agents in helping to collect tiles or fill the holes when they do not have the capability to do so. Last but not least, every agent will only have to store the map of their territory instead of the whole environment, which is extremely effective in large-scale environments.

IV. Conclusion

In conclusion, the MASON TileWorld environment offers a rich and versatile platform for exploring and implementing various agent architecture designs. Through this project, we have the first understanding of intelligence agent design. It is crucial to give the agent the goals to achieve and also understand the trade-off between reactive and proactive. We also learn different ways to design an agent, from deliberative architectures, reactive architectures, and hybrid architectures which take into account the best of both worlds. The choice of architecture design should be informed by the specific goals and requirements of the task at hand. For instance, in tasks demanding speed and adaptability, a reactive or hybrid approach may be preferable. For more complex, goal-oriented tasks, a deliberative or hybrid model might be more effective. Despite we were not able to build the best agent architecture that is suitable for the environment, it still is valuable foundation knowledge for us to design autonomous agents and combine it with more advanced techniques such as deep learning or reinforcement learning in the near future.

References

[1] M. Lees. *A history of the Tileworld agent testbed*. Sch. Comput. Sci. Inf. Technol. Univ. Nottingham, Nottingham, 2002.

[2] M. E. Pollack and M. Ringuette. *Introducing the Tileworld: Experimentally evaluating agent architectures*. In AAAI, volume 90, pp. 183–189, 1990.

[3] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992.

[4] B. Li, S. Mabu, and K. Hirasawa. *Tile-world—a case study of genetic network programming with automatic program generation*. In 2010 IEEE International Conference on Systems Man and Cybernetics (SMC), pp. 2708–2715, 2010.

[5] "Human-competitive results produced by genetic programming," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 251–284, Sep. 2010. [Online]. Available: http://link.springer.com/article/10.1007/s10710-010-9112-3

[6] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. Oxford, England: U Michigan Press, 1975, vol. viii.

[7] H. Iba. "Emergent cooperation for multiple agents using genetic programming," in *Parallel Problem Solving from Nature — PPSN IV*, ser. Lecture Notes in Computer Science, H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, Eds. Springer Berlin Heidelberg, Jan. 1996, no. 1141, pp. 32–41. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-61723-X_967

[8] C. Ryan, J. J. Collins, and M. O. Neill. "Grammatical evolution: Evolving programs for an arbitrary language," in *Genetic Programming*. Springer, 1998, p. 83–96. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0055930