Hackathon Description

Welcome to Hacktrick!

In this hackathon, you will be required to implement agents that navigate through different layouts with lab components scattered around the layout.

Your agents should be able to build four different types of labs, with each lab having different requirements and specifications. We will be evaluating your agents based on the number of labs they build in the allotted time. More in-depth technical details are provided in the following sections.

There will be two different types of agents and gameplay:

1. Single Mode: Only one agent collecting the components and building the labs.
2. Collaborative Mode: Two agents working together in the same layout to build the required labs.

Finally, it is worth noting that there are no constraints on how you implement these agents. We will be providing you with tips on how to implement a reinforcement learning agent in this environment, but by no means do we require you to submit an RL-based solution. You are free to implement your solutions using any method you see fitting (Ex: rule-based agent).

1. Note: We would like to acknowledge that this problem is based on https://github.com/HumanCompatibleAI/overcooked_ai . We would also like to thank their team members for their helpful support.

Environment

States

This section contains details and snapshots of the different states in the game.

Layout:

- Empty space = ‘ ‘
- Counter = 'X'
- Projector Dispenser = 'O'
- Projector Dispenser = 'T'
- Projector Dispenser
- Construction Site = 'P'
- Container Dispenser = 'D'
- Delivery Location = 'S'

Lab types and components:

- Lab 1:
    - 1 Projector
    - 1 Solar Cell
- Lab 2:
    - 1 Projector
    - 2 Solar Cells
- Lab 3:
    - 2 Projectors
    - 2 Solar Cells
- Lab 4:
    - 1 Projector
    - 1 Solar Cell
    - 1 Laptop
- Lab 5:
    - 1 Projector
    - 2 Solar Cells
    - 2 Laptops

Building time and Points:

- Lab 1:
  - Build time = 10
  - Points = 20
- Lab 2:
  - Build time = 15
  - Points = 30
- Lab 3:
  - Build time = 20
  - Points = 40
- Lab 4:
  - Build time = 20
  - Points = 40
- Lab 5:
  - Build time = 30
  - Points = 70

Environment Action:

The environment has a list of 6 possible actions:

1- Up (0,-1)
2- Down (0,1)
3- Right (1,0)
4- Left (-1,0)
5- Stay (0,0)
6- "interact"

These actions can be accessed as a list in the ………. file by importing the Action class and accessing the list as follows: *Action.ALL_ACTIONS.*

Some useful methods in the Action class:

- `move_in_direction(point, direction): Takes a step in the given direction and returns the new point`
- `determine_action_for_change_in_pos(old_pos, new_pos): Determines an action that will enable intended transition`

Environment State

The environment returns a state object for each action taken. Its fields are divided as follows:

1. players**:** an array of player (either 1 or 2) containing the following information:
2. position: the player's position in the grid.
3. orientation: whether the player is facing up (0, -1), down (0, 1), right (-1, 0), left (1,0).
4. held_object: the type of the object that the player is holding.

2- objects: an array containing the objects currently on the counter. For example:

{'name': 'solar_cell', 'position': (1, 0)},

3- bonus_orders: an array containing the names of the bonus orders that can be performed in addition to the default ones

4- all_orders: and array containing a dictionary of the available orders that can be made. For example: [{'ingredients': ('projector', 'projector', 'projector')}]

5- Timestep: the current timestep of the environment since the current episode has started.

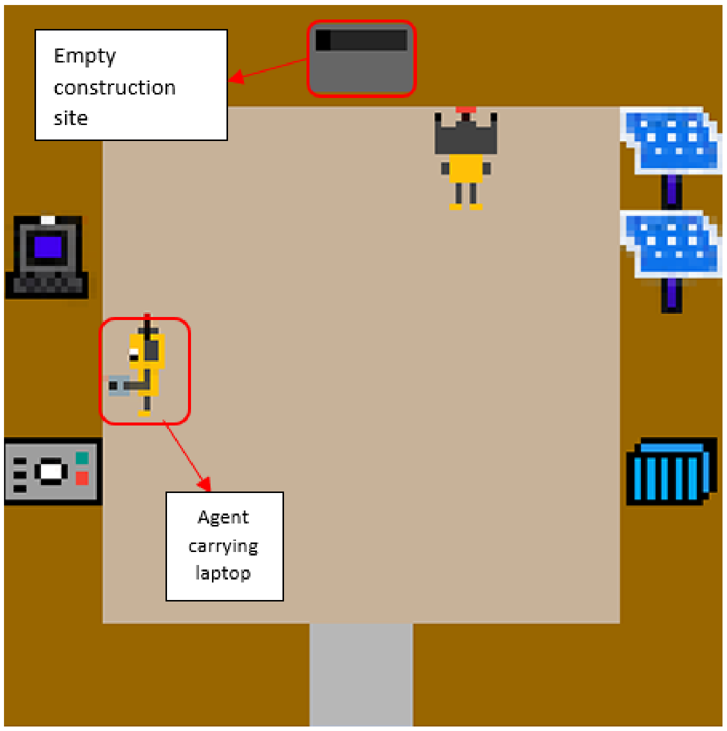
State example:

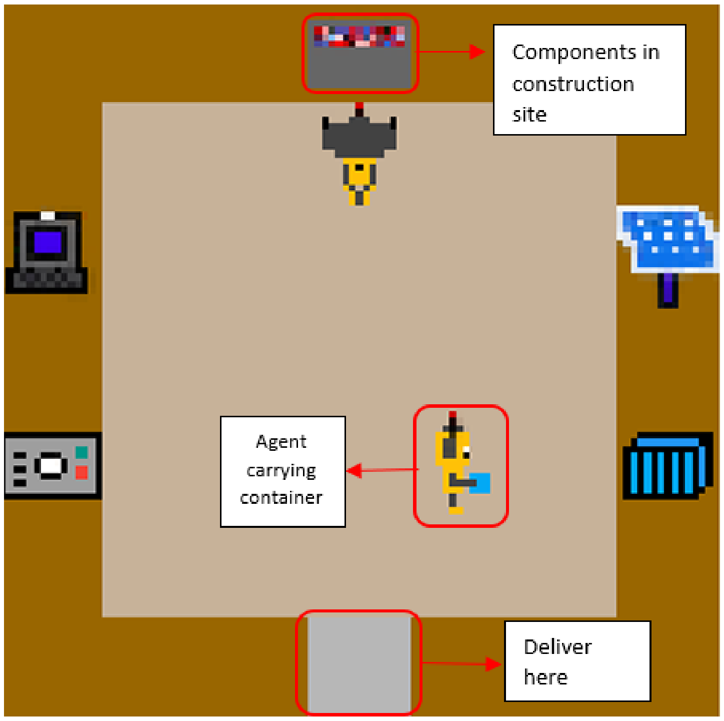
```
{'all_orders':
            [{'ingredients': ('projector', 'solar_cell')},
            {'ingredients': ('projector', 'solar_cell', 'solar_cell')},
             {'ingredients': ('projector', 'projector', 'solar_cell', 'solar_cell')},
            {'ingredients': ('laptop', 'projector', 'solar_cell')},
             {'ingredients': ('laptop', 'laptop', 'projector', 'solar_cell', 'solar_cell')}],
    'bonus_orders': [],
    'objects': [{'name': 'solar_cell', 'position': (1, 0)},
            {'name': 'solar_cell', 'position': (6, 3)},
             {'name': 'solar_cell', 'position': (6, 2)}],
    'players': [{'held_object': None,
             'orientation': (1, 0),
            'position': (5, 5)},
            {'held_object': None,
             'orientation': (-1, 0),
             'position': (5, 2)}],
    'timestep': 100
    }
```

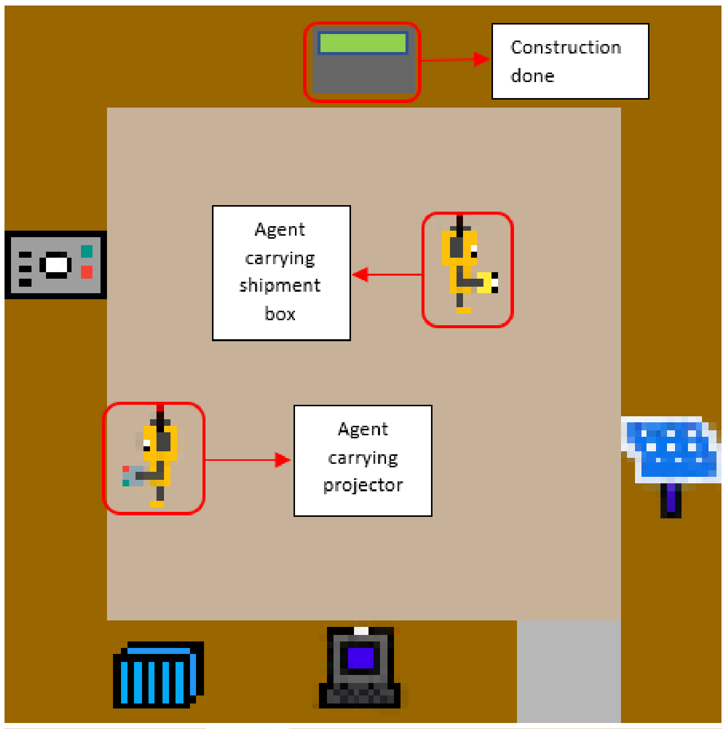
The environment has several helper functions that can be used to analyze the current state and they can be found in the …/mdp/…._mdp in the OvercookedGridWorld class. Examples of such functions:

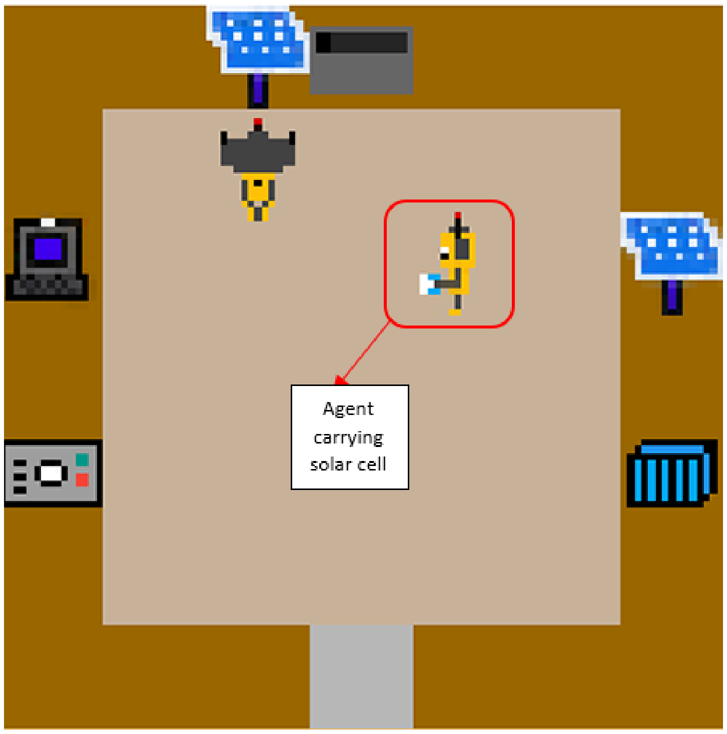
1- *get_container_dispenser_locations*()
2- *get_laptop_dispenser_locations*()
3- *get_construction_site_states*()
4- *get_empty_counter_locations*()
5- *get_partially_full_construction_sites*()

Snapshots




Components in construction site
Agent carrying container
Deliver here


Construction done
Agent carrying shipment box
Agent carrying projector

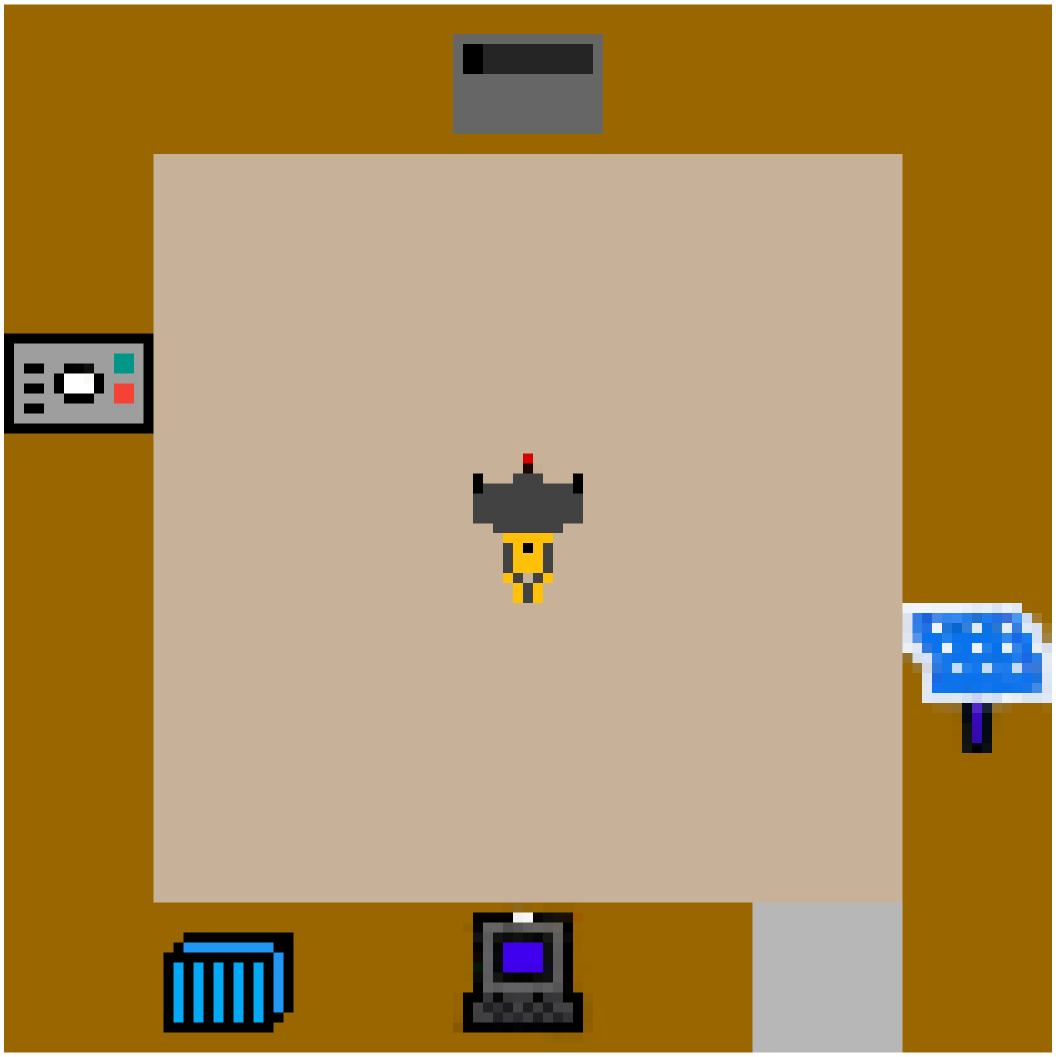

Agent carrying solar cell

Hackathon Phases

This section contains images and details of the different levels throughout the hackathon.

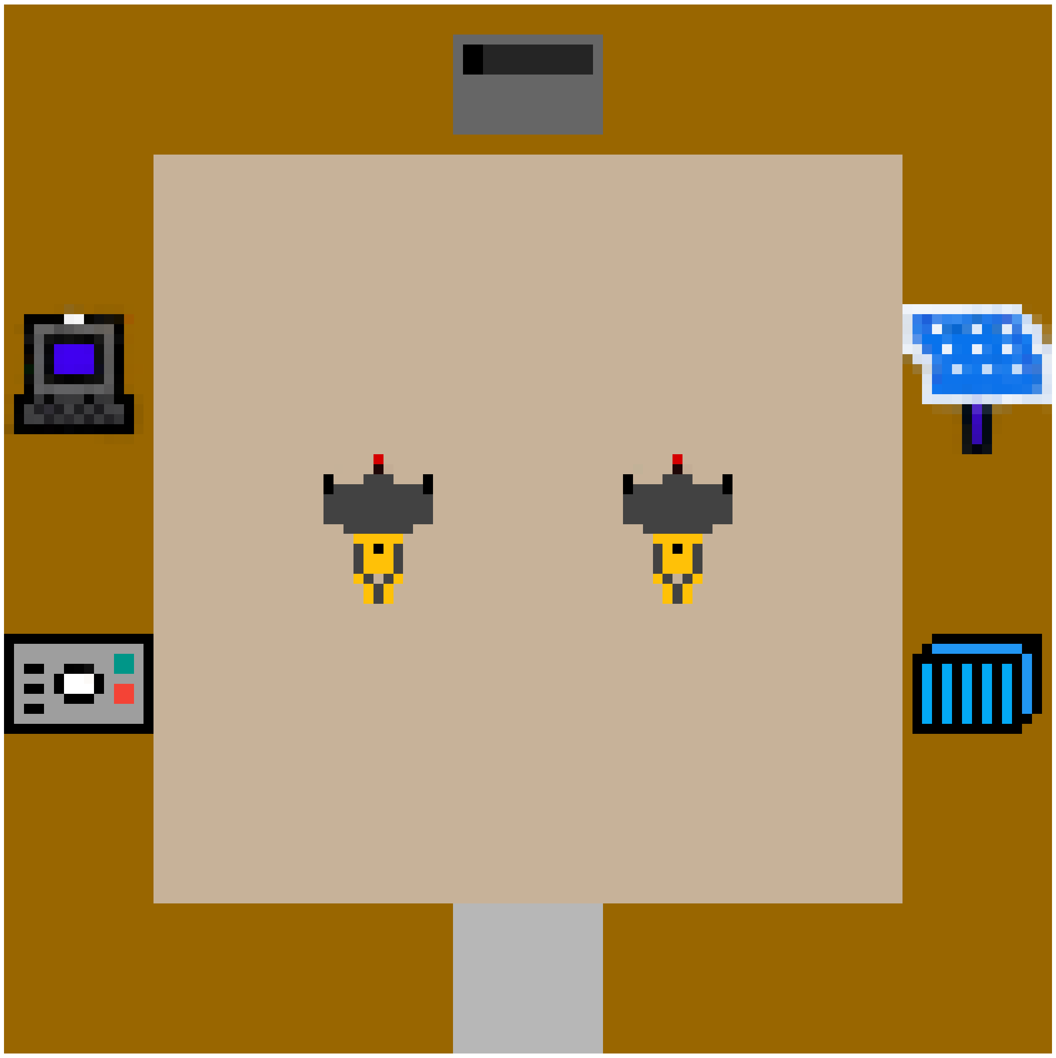
Leaderboard Single Player:

- Empty layout with no obstacles.
- Contains projectors, solar cells, laptops, 1 construction site, and 1 delivery location scattered around the layout.



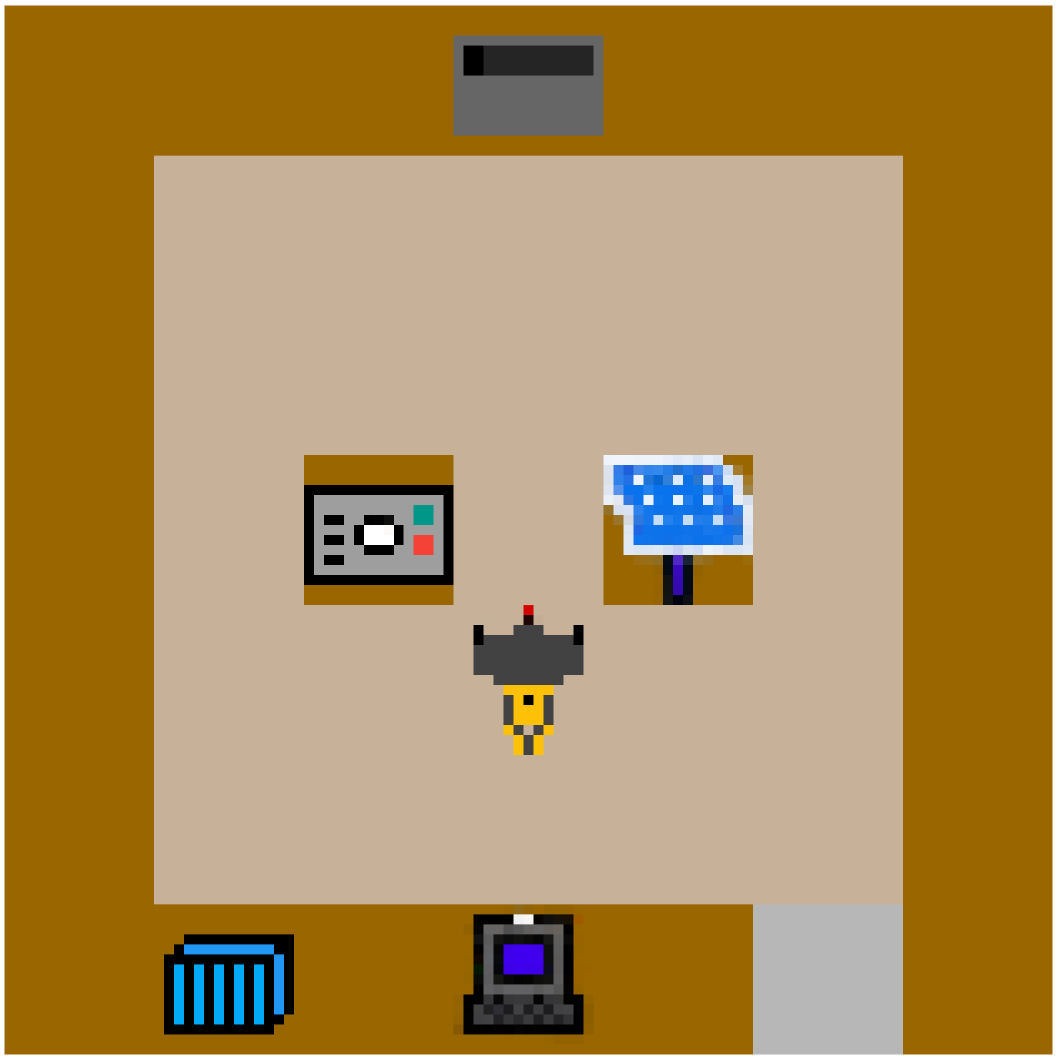
Leaderboard Collaborative Mode:

- Empty layout with no obstacles.
- Contains projectors, solar cells, laptops, 1 construction site, and 1 delivery location.



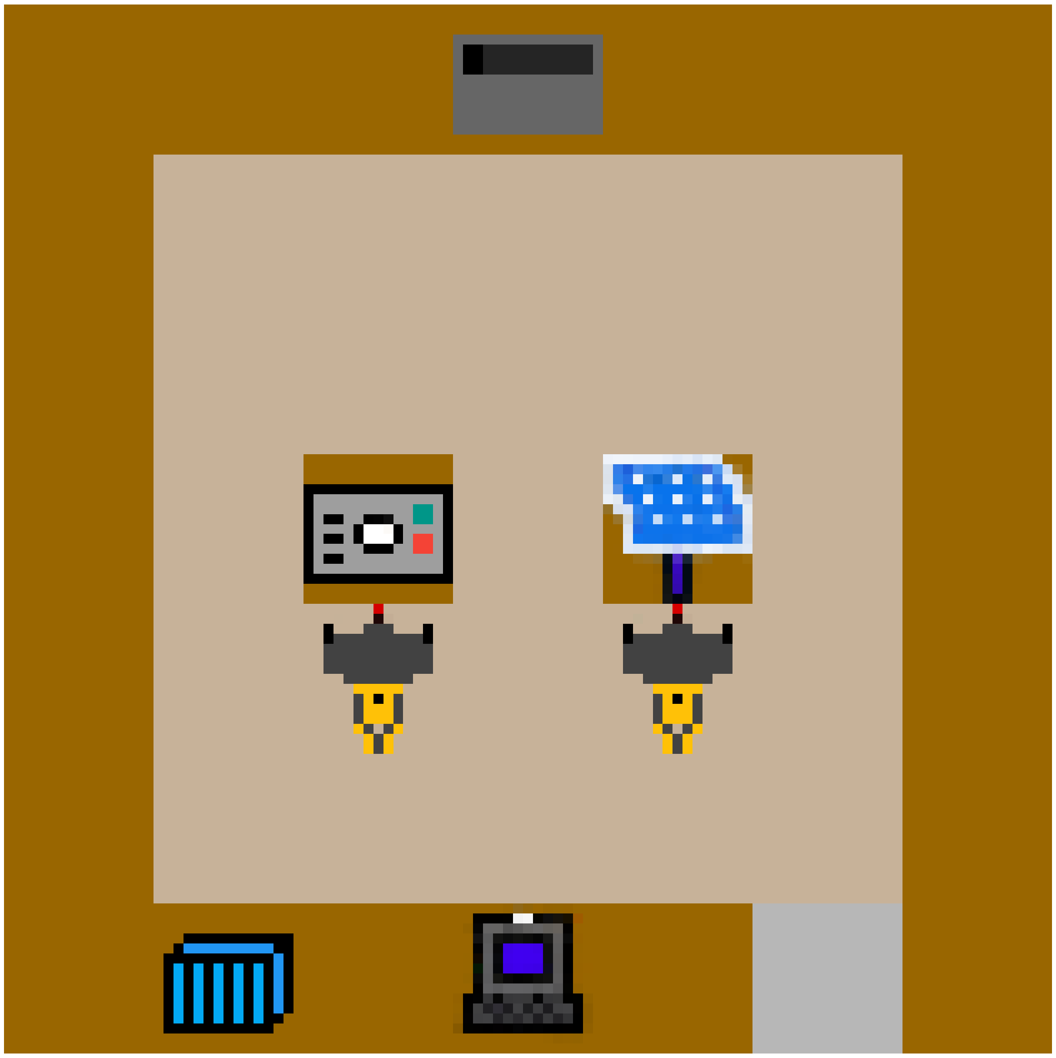
Round of 16 Single Player:

- Two obstacles in the middle with components on top of them.
- Contains projectors, solar cells, laptops, 1 construction site, and 1 delivery location.



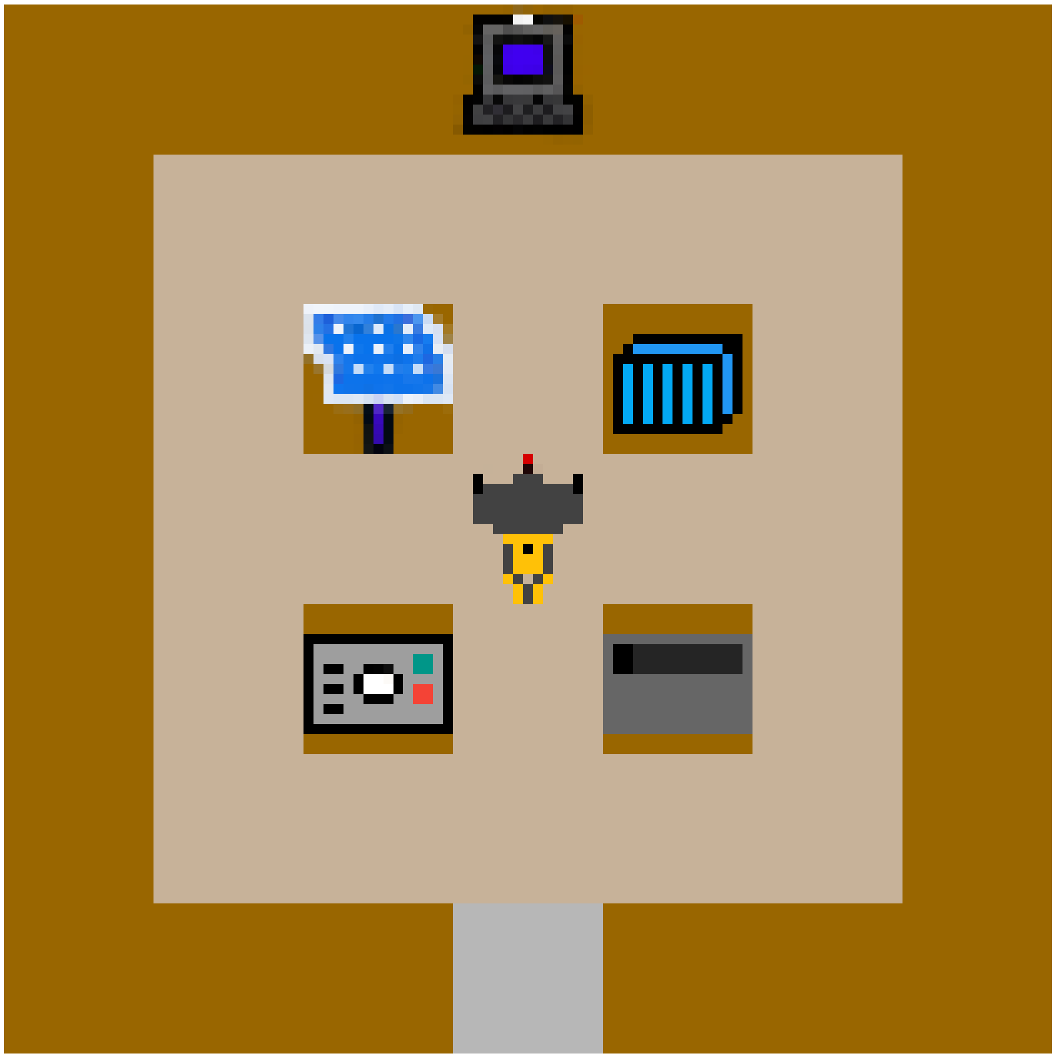
Round of 16 Collaborative:

- Two obstacles in the middle with components on top of them.
- Contains projectors, solar cells, laptops, 1 construction site, and 1 delivery location.



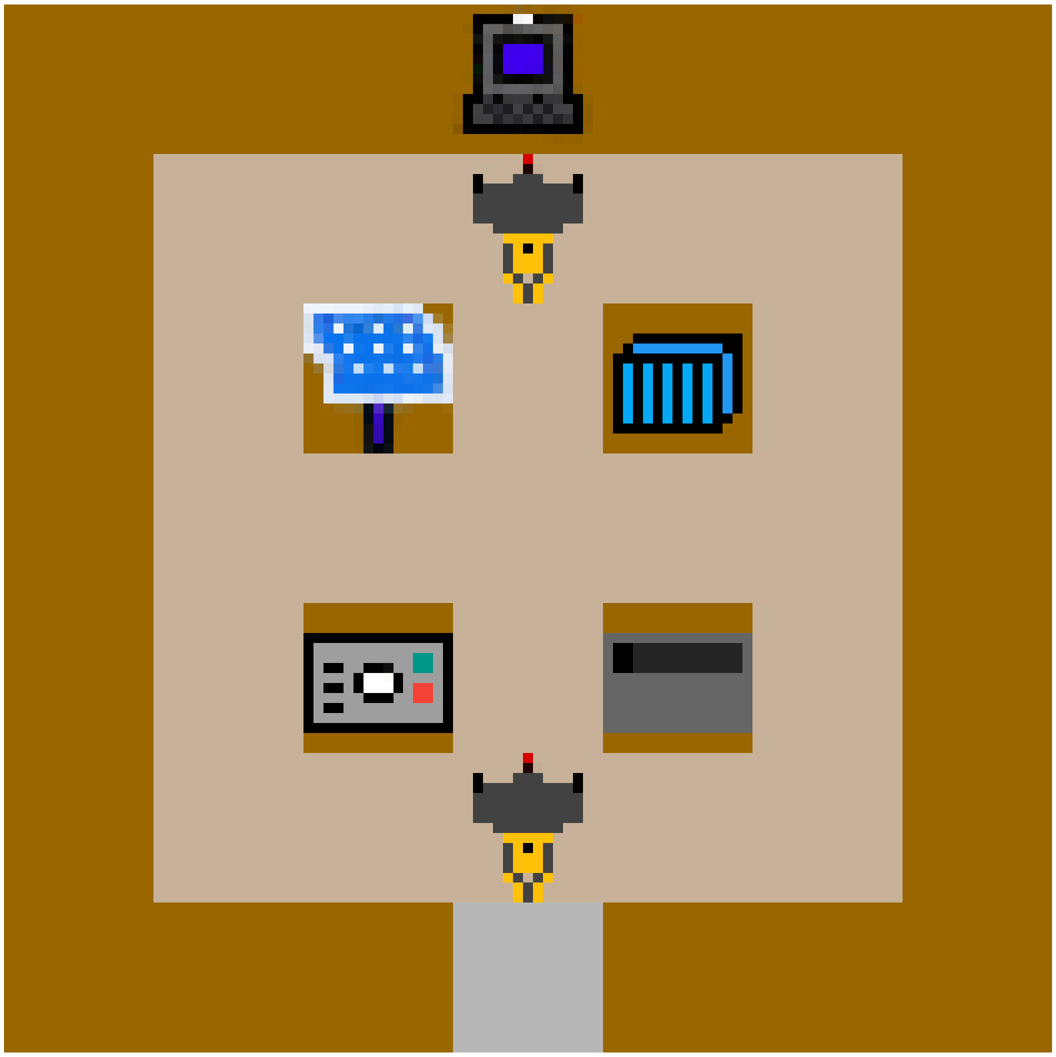
Quarter Final Single Player:

- Four obstacles in the middle with components on top of them.
- Contains projectors, solar cells, laptops, 1 construction site, and 1 delivery location.



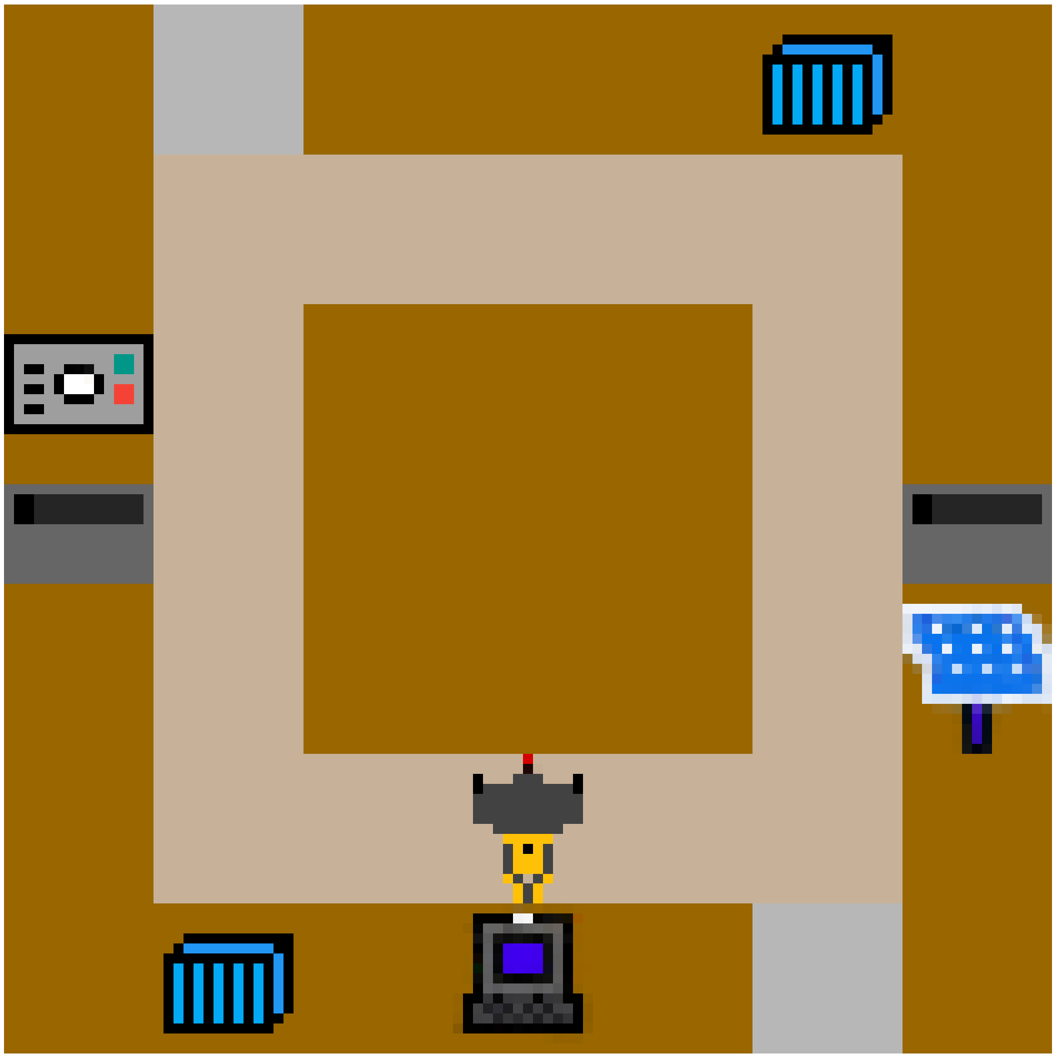
Quarter Final Collaborative:

- Four obstacles in the middle with components on top of them.
- Contains projectors, solar cells, laptops, 1 construction site, and 1 delivery location.



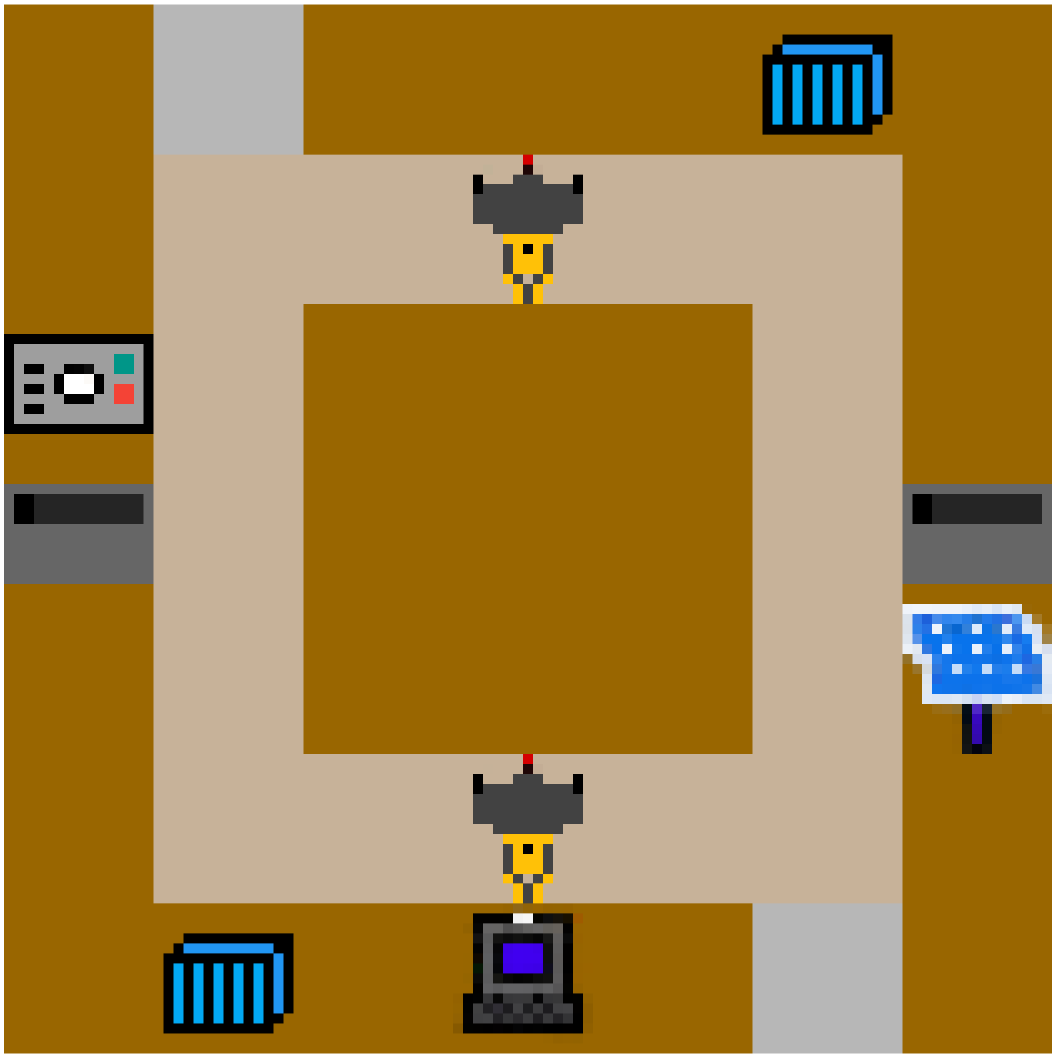
Semi Final Single Player:

- The agent can only move in a ring around the center of the layout.
- Contains projectors, solar cells, laptops, 2 construction sites, and 2 delivery locations.



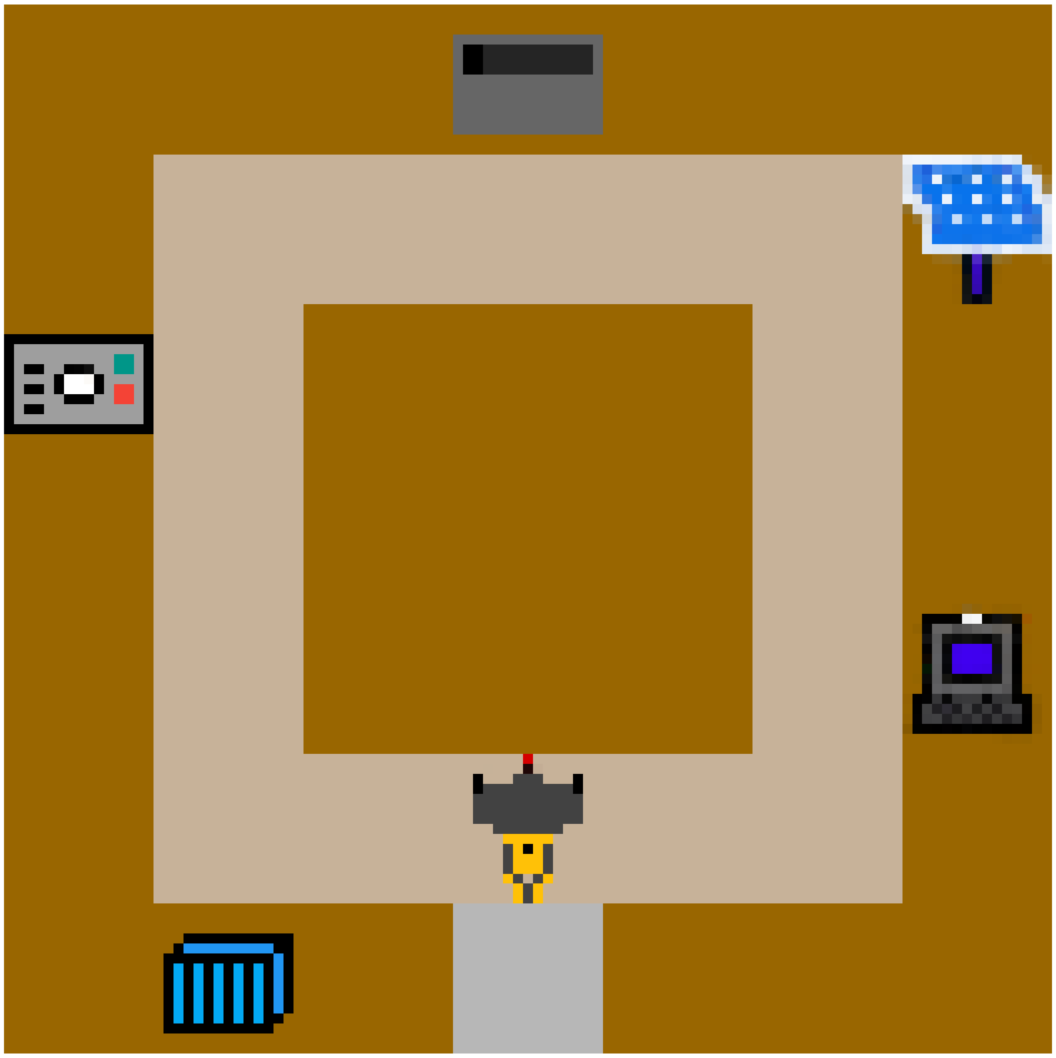
Semi Final Collaborative:

- The agents can only move in a ring around the center of the layout.
- Contains projectors, solar cells, laptops, 2 construction sites, and 2 delivery locations.



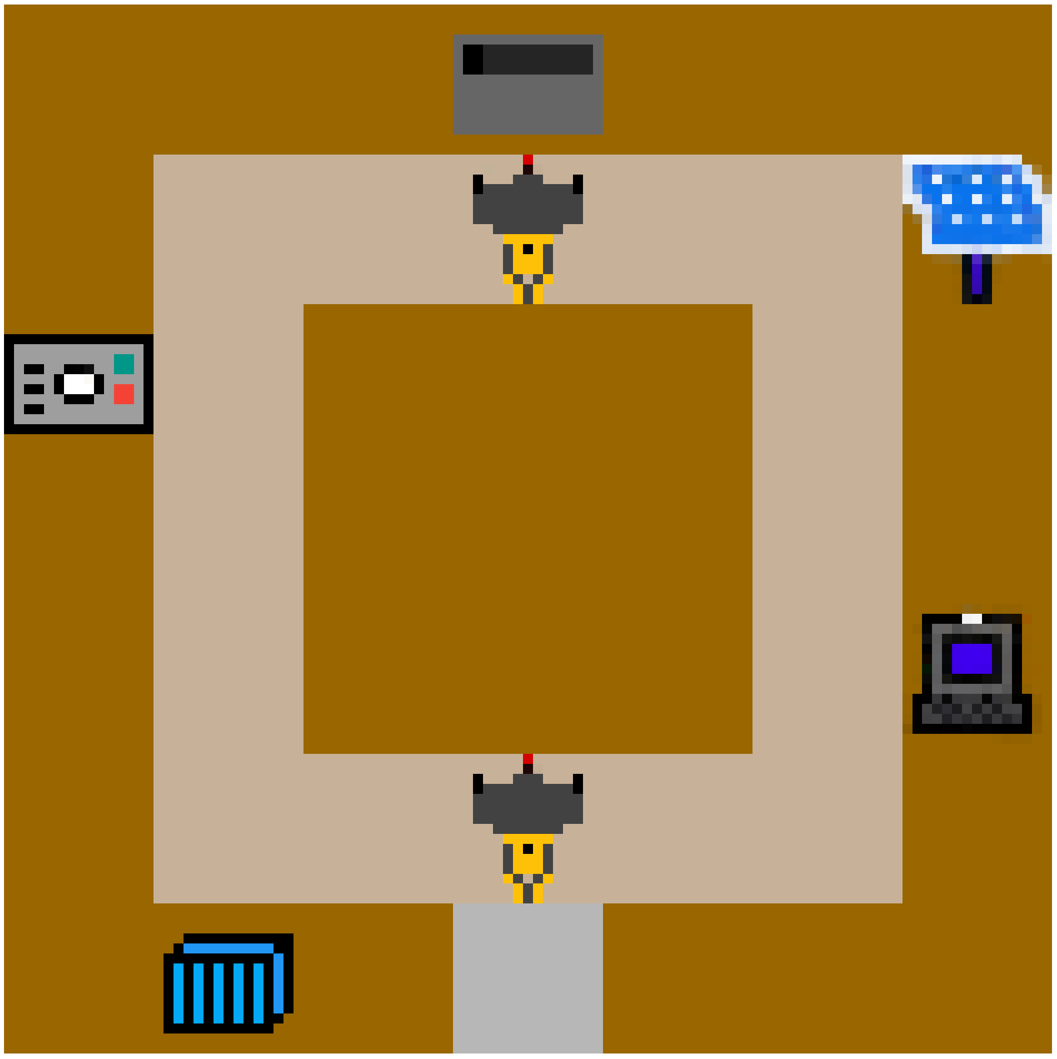
Final Single Player:

- The agent can only move in a ring around the center of the layout.
- Contains projectors, solar cells, laptops, 1 construction site, and 1 delivery locations.



Final Collaborative:

- The agents can only move in a ring around the center of the layout.
- Contains projectors, solar cells, laptops, 1 construction site, and 1 delivery locations.



Model training

- The given code contains a reinforcement learning template using “ray” library. The policy network can be modified from “human_aware_rl/ppo/ppo_rllib.py” file by adding a suitable architecture.
- The given trainer uses a PPO trainer from ray for which all the parameters are specified in the “human_aware_rl/ppo/ppo_rllib_client.py”. The PPO training algorithm in ray uses several losses including an entropy loss for which the library could use a scheduler which parameters can be finetuned based on the layout. These parameters are:
    1. *entropy_coeff_horizon*: the number of steps which the scheduler perform till it reaches the end value *entropy_coeff_end*
    2. *entropy_coeff_end*: the final value that will be used by default after the scheduler finishes.
- The policy is afterwards used in “human_aware_rl/rllib/rllib.py” in the class “*RlLibAgent*”
- The step function of the original environment returns four variables:
    1. *next_state*: the new state of the environment as explained in the state section
    2. *sparse_reward*: the score achieved in this timestep (not cumulative). For example, if during this timestep an order is completed, the environment returns its reward as the score.
    3. done: a flag signaling whether the episode is done or not
    4. info: a dictionary containing information such as two types of extra reward values as follows:
        - '*sparse_r_by_agent*': a list of two values representing the timestep score achieved for each player.
        - '*shaped_r_by_agent*': a list of two values representing the shaped reward score achieved for each player (refer to the next point for more details)
        - *'phi_s*': an internal reward calculated for the current timestep based on how optimal the current state is.
        - '*phi_s_prime*': an internal reward calculated for the new timestep based on how optimal the new state is after the taken action.

  Note: the functions that calculate the *phi_s* value is not optimized for the given problem. Feel free to implement your own algorithm or modify it.

- In the “*OvercookedMultiAgent*” class, the step function is defined to use the default reward function of the environment using either a calculated reward inside the environment based on how good each action (the delta between *phi_s_prime* and *phi_s*) or a shaped reward is based on a dictionary defined in the “ppo_rllib_client.py” for example:

```
rew_shaping_params = {
                "PLACEMENT_IN_POT_REW": 3,
                "DISH_PICKUP_REWARD": 3,
                "SOUP_PICKUP_REWARD": 5,
                "DISH_DISP_DISTANCE_REW": 0,
                "POT_DISTANCE_REW": 0,
                "SOUP_DISTANCE_REW": 0,
            }
```

Each of these functions are obtained from the returned dictionary “info” from the step function and an example are present in the environment class.

A simulated annealing algorithm produces a factor (r*eward_shaping_factor*) which is used on the choosing extra reward to guarantee that the model after a number of steps starts depending more on the game score itself instead of the calculated reward. The simulated annealing has three parameters that could be finetuned if needed based on the layout and policy network:

1. *reward_shaping_factor*: the default value that gets used in the simulated annealing and gets used after the algorithm reaches its end.
2. *reward_shaping_horizon*: the number of steps of the simulated annealing after which the produced factor is defaulted into the *reward_shaping_factor*.

The step function converts the state object to a set of boolean map channels with 1 in case the channel corresponding to the object or play is in this location using the *lossless_state_encoding()* function. in case the participant wanted to implement their own RL agents, it is advised to use this representation of the state.

Running & Submitting

Installation

- cd hacktrick_ai/
- `pip install -e .`

Code Structure Overview

hacktrick_ai_py contains:

`mdp/`:

- hacktrick_mdp.py: main Hacktrick game logic.
- hacktrick_env.py: environment classes built on top of the Hacktrick mdp.
- layout_generator.py: functions to generate random layouts programmatically.

`agents/`:

- agent.py: location of agent classes.
- benchmarking.py: sample trajectories of agents (both trained and planners) and load various models.

`planning/`:

- This directory contains some logic that might help you in implementing a rule-based agent.
- You are free to disregard this directory and implement your own functions.
- If you find any functions that make your implementation easier, or even as a guide/starter, feel free to use them.

Agents

In `hacktrick_agent.py` you will find two base classes `MainAgent()` and `OptionalAgent()`. Implement according to the following cases.

- In single mode, implement only one of them but make sure you are testing and submitting the correct agent.
- In collaborative mode, implement both classes if you want to implement different agent logic.
- In collaborative mode, implement one class only if you want to apply the same logic on both agents.

Run and test your agents locally

Follow the steps in this notebook \hacktrick_client\hackathon_tutorial.ipynb

Submit your solution

- In hacktrick_agent.py you will find two base classes MainAgent() and OptionalAgent(). Implement your logic in these classes.
- Run this command python3 client.py --team_name=TEAM_NAME --password=PASSWORD --mode=MODE --layout=LAYOUT_NAME.

APIs Documentation

Participants in the hackathon shall submit their work using the submission script using the following command

```
python3 client.py --team_name= team --password=PASSWORD --
layout=leaderboard_single
```

The script receives the following parameters

`team_name`	*your team name*
`password`	*your password*
`layout`	*One of the 6 layouts handed to you.*

The script automatically establishes connection with our servers after it authenticates your credentials.

The server and the client communicate through an open socket connection. Once the game is created using the `create` event, the main logic of the game is generated by an ongoing transmission of states and actions. The submission script receives a state from the `state_pong` event and in turn sends an action using the `action` event to the server. This flow goes on until the end of the game which is signaled by the server to the client through the `end_game` event.

`Event`	Type	Description
`connect`	Emitted	This event connects the client to the server.
`create`	Emitted	This event is emitted inside the main function of the submission script and it creates an instance of the game.
`start_game`	Received	
`state_pong`	Received	This event returns the current state of the game.
`action`	Emitted	This event is used to send a single action to the server in the single mode.
`action_collaborative`	Emitted	This event is used to send an array of actions (2 actions) to the server in the collaborative mode.
`end_game`	Received	This event is sent by the server and it signals the end of the game. The game score is sent to the client only in the leaderboard phase.
`creation_failed`	Received	This event is sent by the server if any of the game parameters were incorrect.
`disconnect`	Received	
`authentication_error`	Received	This event is sent by the server if the team name and password do not match.

Pre-requisite packages for the submission script

The following packages should be installed on your machine to run the submission script. ***Please Note*** that you **must** ensure that the exact versions are installed on your side for compatibility.

```
python-socketio[asyncio_client]==4.6.0
python-engineio==3.13.0
```