Introduction to Flatland

Flatland

Flatland is an open-source toolkit for developing Multi-Agent Pathfinding algorithms in grid rail worlds. It provides a two-dimensional grid environment in which many agents can be placed, and each agent must solve one or more navigational tasks in the grid world. The library was developed by SBB, AIcrowd, and numerous contributors and AIcrowd research fellows from the AIcrowd community.

In this unit, you can implement any algorithms you have learnt for the Flatland challenge. Feel free to research and implement algorithms from the literature as well.

Part 1: Installation

Refer to Week 1 Installation Guide to Install Flatland on your computer.

You must update your flatland code and installation before starting the assignment:

- **Under flatland folder**
- *git stash save* *(In case you have unstaged changes made in week 1)*
- *git pull*
- *git stash pop*
- *python setup.py install*
- **Now, run command** *python -m pip list* **, you should see flatland-rl version is updated to 2.2.4**
- **(Hint, use** *python3* **instead of** *python* **if** *python3* **points to the right one on your machine)**

Part 2: Basic Rules with Flatland

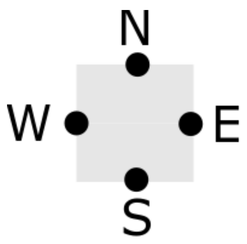
Characteristics:

- Flatland simulates a railway environment in a 2-dimensional grid space of any size.
- Flatland is a discrete-time simulation. A discrete-time simulation performs all actions within a constant time step.
- An agent can move in any direction on well-defined transitions from cells to cells.

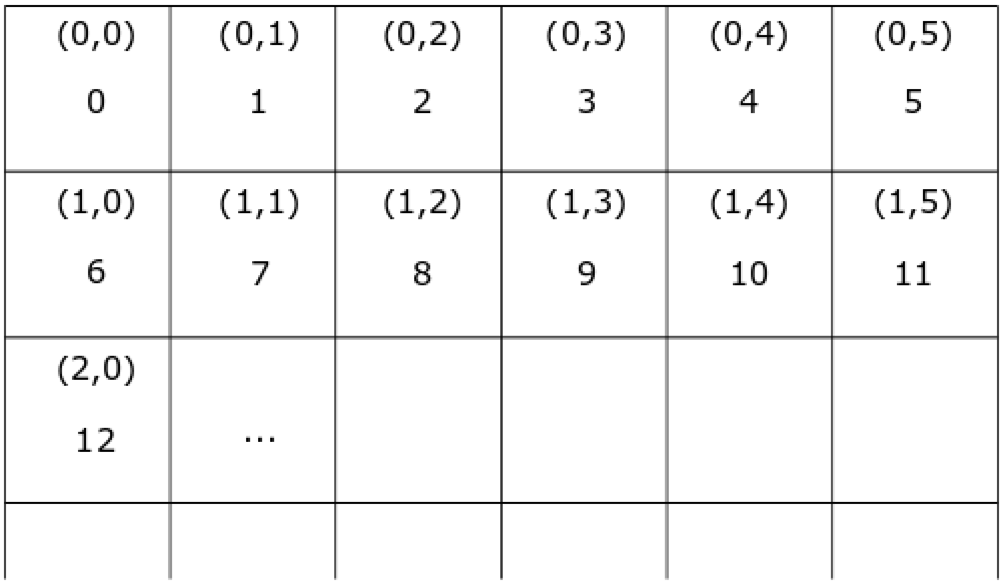
- The capacity of a cell is usually one. Thus usually only one agent can be located at a given cell.
- At each step, the agents can choose an action. For the chosen action the attached transition will be executed.
- While executing a transition, Flatland checks whether the requested transition is valid. If the transition is valid, the transition will update the agent's position.
- While executing a transition, a malfunction may happen which prevents agents from moving toward their intended target locations. Each malfunction lasts for a certain period of time and during the malfunction affected trains are unable to move. Malfunctions only occur in question 3.
- Also, each agent is assigned a start and goal location. Our target is to find paths for each agent while minimizing the sum of their individual path costs or total makespan.
- Each instance have a max time step limit, which is 1.5 * rail.height * rail.width. All agents need to arrive at their destination before this max time step limit.

Grid:

A rectangular grid of integer shape (dim_x, dim_y) defines the spatial dimensions of the environment. We use North, East, West, South as orientation indicators where North is Up, South is Down, West is left and East is Right.



Each cell in the grid environment can be located by (row, column) coordinate. Here is an example:



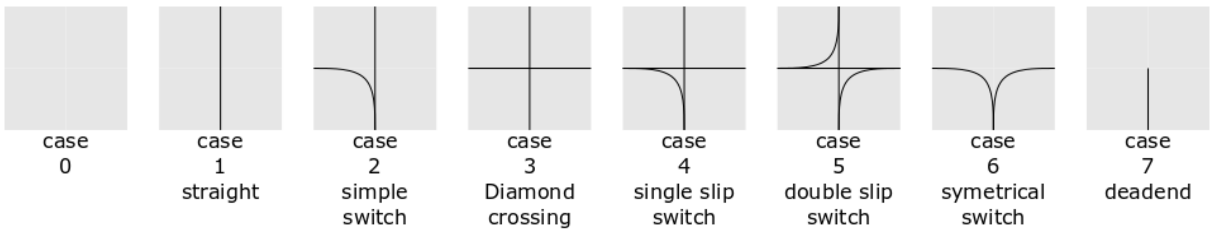
(0,0) 0	(0,1) 1	(0,2) 2	(0,3) 3	(0,4) 4	(0,5) 5
(1,0) 6	(1,1) 7	(1,2) 8	(1,3) 9	(1,4) 10	(1,5) 11
(2,0) 12	...				

A pair of row and column coordinates can also be converted to a single integer by row × map_width + column.

Tile Types:

Each Cell within the simulation grid consists of a distinct tile type which in turn limits the movement possibilities of the agent through the cell. For Flatland problems, 8 basic tile types are defined to describe a rail network. An invariant across each tile in the railway network is that there are at most two branches available at any junction point with any entry direction.

The following image gives an overview of the eight basic tile types. These can be rotated in steps of 45° and mirrored along the North-South of the East-West axis.



- Case 0 represents a wall, thus no agent can occupy the tile at any time.
- Case 1 represents a passage through the tile. While on the tile the agent can make no navigation decision. The agent can only decide to either continue, i.e. passing on to the next connected tile, or wait at the current tile.
- Case 2 represents a simple switch, thus when coming to the top position (south in the example) a navigation choice (West or North) must be taken. Generally, the straight transition (S->N in the example) is less costly than the bent transition. Therefore in Case 2 the two choices may be rewarded differently. Case 6 is identical to case 2 from a topological point of view, however, there is no preferred choice when coming from the South.
- Case 3 can be seen as a superposition of Case 1. As with any other tile at maximum one agent can occupy the cell at a given time.
- Case 4 represents a single-slit switch. In the example, a navigation choice is possible when coming from the West or South.
- In Case 5 coming from all directions, a navigation choice must be taken.
- Case 7 represents a dead-end, thus only stop or backward motion is possible when an agent occupies this cell.

Execution and Termination

- For each question, the simulator will execute the computed plan for each agent.
- Execution proceeds by applying the next planned action for each agent in some specified ordering determined by the simulator.
- Execution stops when any agent is unable to execute their next planned action. At this point the simulator calls a replan function and all agents have an opportunity to recompute their paths.
- Evaluation of current instance terminates, when one of the following conditions is met:
    - All agents reach their destination collision-free.
    - All of the planned actions of all agents are executed or,
    - The maximum number of timestep is reached

Part 3: How to interact with the flatland environment.

Information related to question 1 and question 2

Flatland environment

All the information about the map is stored in an object of **GridTransitionMap.** In the example of the next chapter, the variable **rail** is the object of **GridTransitionMap.**

Width and height:

- Get map width by **rail.width**
- Get map height by **rail.height**

Get available transitions:

cell_transitions = rail.get_transitions(x, y, direction) and in return you get an array with possible transitions in each of the four cardinal directions **[North, East, South, West]**. The **x** and **y** is the location coordinate of the current cell where **x** in [0, height] and **y** in [0, width].

Depending on the tile type, at most two of the returned directions will be valid transitions. **Be careful with the heading direction!** A different heading **direction** for the same cell will return different available transitions. Thus you need to correctly specify the current position and direction. These important pieces of information are both parts of your agent's state! Also, be careful about the heading **direction**'s influence on heuristics!

Being unable to find valid paths:

In question 2, it is possible that a very small number of agents' paths cannot be found by your algorithm as moving obstacles may traverse through the start location of your agent. You can simply return an empty list in this case, and the agent with an empty list will not show on the map.

Note, this only happens to not more than 5 or 6 instances, with only 1~2 agents having no paths.

Information related to question 3

Get agent information:

Agents information is stored in **agents** which is a list of **EnvAgent objects**. An EnvAgent object contains the following attribute:

- **initial_position**
  A tuple of (x,y). The initial location of the agent.
- **initial_direction**
  An integer in [0,1,2,3]. The initial heading direction of the agent.
- **target**
  A tuple of (x,y). The goal location of the agent.

- **position**
  A tuple of (x,y). The current location of the agent, this is used when a plan is in execution, which means .
- **direction**
  An integer in [0,1,2,3]. The current heading direction of the agent.
- **status**
  An integer in [0,1,2]. The current status of the agent.
- **deadline**
  An integer indicates the expected latest arrival timestep of the agent.
- **malfunction_data**
  A dictionary stores data related to malfunctions.

You will use the **initial_position, initial_direction,** and **target** of each agent to plan paths for them. The **position** and **direction** will be used on the execution stage. Agents arriving at goal locations after the **deadline** timestep will receive a penalty.

The heading direction of the train is represented by an integer:

- 0: North.
- 1: East.
- 2: South.
- 3: West.

The status is an integer:

- 0: The agent is not active (not on the map).
- 1: The agent is active (on the map).
- 2: The agent reaches the goal location.
- 3: The agent reaches the goal location and is removed from the map.

Executing your coordinated plan

By default, when plan execution starts, the **status** of the agent is 0 and the **position** is None. By giving a move forward command, the status turns to 1, **position=initial_position** and you can start executing your plan. When the agent reaches its goal location the status of the agent turns to 3.

During execution, it is possible your planned actions may fail due to malfunctions (i.e., the train experiences a mechanical problem) . The **malfunction_data** object is a dictionary that contains the following information:

- "malfunction" : int
  A number that indicates how many timesteps **n** the malfunction lasts for. When ***n > 0*** the agent will not be able to execute any command for the next **n** timesteps. In other words, the agent must wait in place for the malfunction to pass (i.e., until the train is repaired!).

Replan

When a new malfunction occurs or if two agents plan to execute a pair of incompatible actions (i.e., leading to a collision), the system will call a **replan** function to provide a new set of feasible paths.

The new paths should consider that agents can only wait at the current location during the malfunction timesteps. Some extra information that is available to the **replan** function:

- **current_timestep**
  At which timestep during the execution, the replan function is called.
- **existing_paths**
  The paths from the previous get_path/replan function call.
- **new_malfunction_agents**
  A list of ids of agents that have a new malfunction happens at the current time step (Does not include agents already have malfunctioned in past timesteps).
- **failed_agents**
  A list of ids of agents that failed to reach their intended location at the current timestep. It usually includes new malfunction agents, other agents influenced by the malfunction agents (i.e., knock-on effects), and agents with failure execution due to collisions in their current plans.

Should find paths for all agents:

In question 3, you have full control on all agents. Thus, we expect you to find paths for all agents.

Part 4: Code for Assignment 1.

Get Piglet assignment 1 branch

We release the assignment 1 code base as a separate branch in piglet.

Follow the following steps to switch between branches in piglet and get assignment 1 branch.

1. Assuming you are currently working in the “single-agent-prac” branch, and want to checkout to the “assignment1_2025” branch. (To know what branch you are currently working on, use ‘git branch’ command under piglet-public)
2. Discover new branches in the remote repository:
   `git fetch` this command should show what it discovered from the remote repo.
3. If you made any changes in the “single-agent-prac” branch, you should **stage** and **commit** your changes to the piglet (which saves all your changes and allows you to checkout to another branch):
   `cd` to the piglet-public folder.
   `git add *` this command will stage all your changes in piglet
   (or use git add path/to/changed/file to stage a single file)
   `git commit -m "Describe what you did"` this command will commit (save) all your changes.
4. Checkout to the target assignment branch:
   `git checkout assignment1_2025`

You could discover more geek style usage of git from the git tutorials we recommended on week 1, which could speed up your version management and development efficiency.

Question scripts

You will find question1.py, question2.py, and question3.py in the root folder of the piglet.

- question1.py requires you to implement a single-agent pathfinding algorithm for flatland.
- question2.py requires you to avoid conflicts with existing paths.
- question3.py requires you to implement a multi-agent pathfinding algorithm, and a replan function to handle malfunctions.

Each question python script includes a get_path() function with different parameters. You need to implement your algorithm inside the get_path() function and return the required data.

On default get_path() functions include some dummy implementation. They only can generate some dummy paths. Replace them with your implementation. Also, you can read these dummy implementations to learn how we use parameters and what kind of data we return.

Test your implementation

To test your implementation, run command (Make sure your conda virtual environment is activated):

```
python question1.py
```

You can replace question1.py with question2.py or question3.py for other questions.

The script will output statistical data at the same time.

Debug and visualizer

You can find two variables "debug" and "visualizer" in each question script.

If "debug" is set to True, the script will:

- output detailed execution information.
- pause program if conflict happened or meet other errors.
- output conflict details.

If "visualizer" is set to True, the script will:

- show your plan with a nice visualizer.
- show agent id if "debug" also True.

If "test_single_instance" is set to True, the script will:

- evaluate only instances specified by "level" and "test".

Submission of Assignment

Follow instructions in the assignment description to submit the zip file to moodle.

Follow the Contest Server submission instructions to submit to the contest server.

Part 5: Practice - Navigate a train with your keyboard (Optional).

This part helps you learn more about Flatland. Follow these instructions to create a keyboard navigated train driving program with Flatland.

1. Create an empty python file
2. Insert following codes to import necessary modules that will be used in this practice:

```
#import necessary modules that this python scripts need.
from flatland.envs.rail_generators import complex_rail_generator
from flatland.envs.schedule_generators import complex_schedule_generator
from flatland.envs.rail_env import RailEnv
from flatland.utils.rendertools import RenderTool, AgentRenderVariant
import time
```

3. Insert following codes to initialize Flatland Railway environment:

```
#Initialize Flatland Railway environment
railWaySettings = complex_rail_generator(
                   nr_start_goal=10, # Number of start location and train station pairs
                   nr_extra=10, # Extra rails between each pair of start and goal.
                   min_dist=10, # Minimum distance between start and goal locations
                   max_dist=99999,
                   seed=999)

env = RailEnv(  width=15, #Width/columns of grids
               height=15, #Height/rows of grids
               rail_generator= railWaySettings,
               number_of_agents=1) # How many trains are enabled.
env.reset() # initialize railway env

# Initiate the graphic module
render = RenderTool(env, gl="PILSVG",
                         show_debug=False,
                         screen_height=500,  # Window resolution height
                         screen_width=500)  # Window resolution width
render.render_env(show=True, frames=False, show_observations=False)
window=render.gl.window  #We will use this window object to capture user input
```

4. Insert following codes to capture keyboard press event and store the key to a variable:

```
#Capture key press event from window and set pressed key to pressedKey variable
pressedKey = None #store pressed key here
keyMap={"Left":1,#Turn left
       "Up":2,#Go forward
       "Right":3,#Turn right
       "Down":4#Stop
       }
def _keypress(event):
   global pressedKey
   if event.keysym in keyMap.keys():
      pressedKey = keyMap[event.keysym] #retrieve corresponding actions from keyMap
window.bind("<KeyPress>", _keypress) # bind keypress event to our function
```

5. Insert following codes to define controller function, which is the most important part you should pay attention:

```
#Define Controller
def my_controller(number_agents,timestep=False):
   _action = {}
   global pressedKey #declare pressedKey is a global variable

   if timestep:
       #In the first frame of flatland, agents aren disabled,
       #  use any action to enable agents.
       for i in range(0,number_agents):
           _action[i] = 2
       return _action

   #Retrieve pressed action and put it into an action dictionary.
   #We only have 1 agent in this practice,
   #  thus we assign the pressed action only to agent 0
   if pressedKey is not None:
       _action[0] = pressedKey
   else:
       _action[0] = 0
   pressedKey = None
   return _action
```

6. Insert following codes to create the main loop of this practice:

```
#Main loop
cost_dict={} #Record cost for each agent
timestep = -1 #all agents are disabled in the first makespan, so we start from -1..
```

```python
all_done = False
while not all_done:
   #Call controller function to get the action dictionary for all agents
   _action = my_controller(len(env.agents), timestep == -1)
   #Pass action dictionary to railway env and execute all actions.
   obs, all_rewards, done, info = env.step(_action)
   all_done = done["__all__"]

   #count cost for each agent
   for key,value in done.items():
       if value == False and key != "__all__":
           if key not in cost_dict.keys():
               cost_dict[key] = 1
           else:
               cost_dict[key] += 1
   timestep+=1
   render.render_env(show=True, frames=False, show_observations=False)
   time.sleep(0.25)

print("Sum of individual cost: ", sum(cost_dict.values()))
print("Total makespan: ", makespan)
```

7. Run the python program under the virtual environment you created. If you use a Python IDE, you should try to find a virtual environment setting and select the correct one. Making sure the railway window is in the front when you press array keys.

Hint:

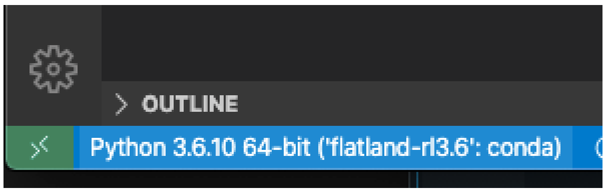
Flatland Documentation:

The Flatland simulator has a rich API with many useful functions and descriptions of how they work. You may find it helpful to explore the official documentation when developing your planner.

Selecting a virtual environment:

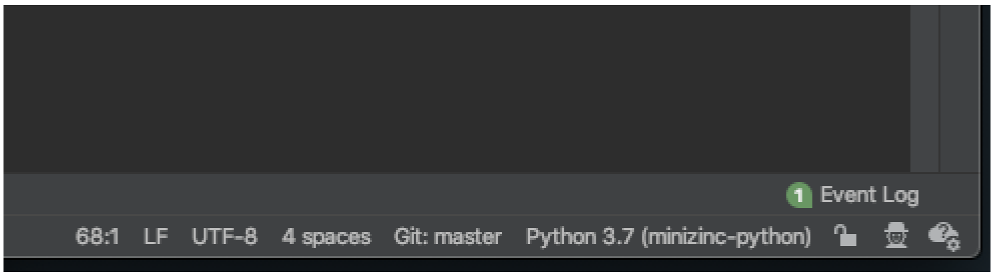
When editing with VS code or Pycharm, the default environment is likely base python. This environment does not have the necessary libraries for the Flatland environment. For this reason you will likely need to specify the correct virtual environment you have already set up for the Flatland challenge.

VS Code:
1. Install python extension for your VS Code.
2. Select a virtual environment by clicking the “Python” text on the bottom-left corner of the window:



Pycharm:

1. Click the “Python” text on the bottom-right corner to select virtual environment:


1 Event Log
68:1 LF UTF-8 4 spaces Git: master Python 3.7 (minizinc-python)