Task Planning for Unmanned Vehicle Delivery

Team 24
ZHU HAOYU, WANG GUANDING, ZENG CONG

Our project focuses on the path planning of unmanned vehicles (UVs) for parcels delivery across different communities within an urban environment. We constructed an environment that includes both a map and an unmanned vehicle, and used a random policy along with 3D-rendered animations to visualize the interactive process of the entire system.

1. Code explanation

Our environment code consists of two Python files, **gym_env.py** and **viewer_part.py**, which are used for creating the RL environment and performing 3D visualization, respectively. In **gym_env.py**, a main function is implemented that interacts with the environment using a random policy. It also calls the visualization module in **viewer_part.py** to display the movement of the UV between depots and communities within the city.

- gym_env:
  The gym_env module contains two classes, World and PointsEnv. The PointsEnv class defines the dynamic RL environment for interaction, while the World class is responsible only for storing and computing the map and distance information generated by PointsEnv.
  When running gym_env.py, the main function first calls PointsEnv to generate a map containing four depots and one hundred task points. Each task point is then assigned a random parcel quantity drawn from a uniform distribution between 5 and 20. After that, one of the depots is randomly selected as the starting point of the UV. At each step, the environment calculates the set of reachable target points, and the agent selects the next destination among the feasible ones. The environment then updates the state and reward accordingly. Once the delivery round is completed, when the UV returns to the depot, the main function calls viewer_part to visualize the entire delivery process.
- viewer_part:
  The viewer_part module builds a three-dimensional scene of urban parcel delivery by unmanned vehicles based on the Ursina engine. Upon initialization, the program creates a window and lighting setup, places a map plane on the scaled ground, and positions the camera at an oblique overhead angle above the scene.
  The scene consists of a robot and three types of nodes: depots, task points, and visited task points. The visualization of the vehicle’s path is implemented using two methods: current_path_line and path_lines, which represent the current movement trajectory and the historical trajectory, respectively.
  When the unmanned vehicle arrives at a task point, a delivery animation is triggered on the right side of the vehicle. After the animation finishes, the corresponding node is marked as visited.

2. Existing Libraries

- Ursina

  Ursina is a high-level Python framework for 3D and 2D development built on top of Panda3D. It encapsulates rendering, window management, input handling, user interface, camera control, and entity management into a unified Entity abstraction. This framework enables us to efficiently and intuitively create 3D visualizations of the urban unmanned vehicle delivery scenario.

3. Reflections/Lessons Learned

- Map Generation

  We initially experimented with three methods for map generation, manually designing depots and task points, fully random generation, and random generation of task points followed by clustering to generate depot. Then, we employed an optimization solver (OR-Tools) to solve then.

  Our findings showed that manually designed grid-based maps were too simple to solve, which means RL method is meaningless. Both the fully random and random-plus-clustering approaches often produced depots located very close to each other, which is unrealistic.Therefore, we designed a set of customized rules for map generation. Specifically, we constrained the minimum distance between depots as well as their distance from the map boundaries. Depots were then randomly generated in the four quadrants until all constraints were satisfied. Finally, 25 task points were generated around each depot following a Gaussian distribution.
- Masking or Negative Reward

  To prevent UV to potentially selecting invalid points, we initially attempted to assign a large negative reward to such illegal actions. However, experiments showed that this approach led to unstable training. The learning agent often became stuck receiving large negative rewards and learned nothing. Therefore, we adopted a masking strategy instead. In this approach, invalid actions are marked in gym_env and their corresponding output values are later set to negative infinity, ensuring that the softmax operation assigns zero probability to these invalid actions.
- Identifying Invalid Points

  In our initial implementation, the legality of a target point was determined only by whether it was within the remaining travel distance of the UV. However, during testing, we observed that after the UV moved to a reachable point, its remaining distance further decreased, sometimes making it impossible to return to a depot. To address this issue, we extended the legality check by also considering the shortest distance from each point to the nearest depot. A point is regarded as valid only if:

  remain_distance > distance(to next point) + distance(next point to nearest depot).

  This is also the reason why we implemented a separate World class, dedicated to calculating and storing distance information within the environment.