16-891 Multi-Robot Planning and Coordination

Instructor : Jiaoyang Li   TAs: Philip Huang and Jingtian Yan

Spring 2026 HW2: Robust MAPF, Target Assignment, Execution Policies, and Realistic Simulations

Assignment Deadline: 11:59 PM, March 16th 2026

This is a newish assignment. Please report errors and bugs to the course staff via email or Piazza. Thank you!

You have just been hired as CTO in a new warehouse automation startup. Congratulations! During your interviews with the CEO, you have demonstrated your newly acquired expertise in fundamental Multi-Agent Path Finding algorithms (e.g., CBS, PBS, Prioritized Planning, etc.) and convinced her that MAPF algorithms are the best choice for planning efficient paths for ground robots in her new company. She promptly fired her current CTO Ben Bitdiddle and put you in his place.

You begin your first week at the company by copy-pasting your CBS code from HW1 and... by the end of the week you have a few damaged robots and an unhappy CEO. The following issues emerged when running vanilla CBS for the warehouse floor:

1. Even though your planner produced collision-free paths, those are only valid when robots move at constant speeds and can break or accelerate instantaneously. This is an unrealistic assumption, of course. You noticed that even your most reliable robots could not perfectly execute their planned paths. The minor deviations between planned and executed paths caused your robots to collide with each other despite following their "safe" paths.

2. The company's first product is a package sortation system. The input for a path finding call is not an explicit goal configuration for each robot, but instead a set of goals (e.g., package pickup locations) that *some* robot needs to reach. The CEO asks you to include an optimized assignment between available robots and each one of those locations.

3. Some of the robots at the company were faulty. They slowed down or even stopped randomly. Since this option was not captured in your planning algorithm... you guessed it – collisions.

4. The planner produces collision-free paths for all robots, but they do not consider robot dynamics. You need a more realistic simulation to better understand your algorithm's performance with complicated real robot dynamics.

In this assignment, we will improve on the MAPF code you developed for HW1 and handle the scenarios you have identified during your first week in industry. In particular, we will:

1. Implement k-Robust CBS [1] so that robots do not move too close behind each other. This would make it less likely for small delays to affect execution.
2. Implement a combined multi-agent target assignment and path finding algorithm [4].
3. Implement an execution policy for handling arbitrary delays in real-time [2].
4. Test your MAPF planner in a more realistic simulation testbed based on [5].

Honor Code

As a student, you may discuss with your peers, but at no point should you seek assistance on your code from them, from any generative AI tools, or from the internet. Your code submission should be yours and yours only. This assignment has a lot of learning and intuition waiting to be acquired if only you followed this Honor code and remained loyal to your learning!

Grading Rubrics

This assignment is worth 20% of your final grade and has 20 points in total: 12 for the implementation problems and 8 for the written problems. The point distribution is included in the respective task sections and summarized below:

Algorithm	Implementation Points	Written Points
k-Robust CBS	1.5	1
Target Assignment	5	2.5
Execution Policy	3.5	1.5
Realistic Simulation	2	3

To get all points for each implementation problem, your code must pass all corresponding test cases on Gradescope. For k-robust CBS and TA-CBS, your solvers are expected to find optimal paths with the minimum sum of costs. The Execution policy problem is open-ended, and your score will depend on the quality (makespan) of your solutions and the time it takes to obtain them per timestep. Section 3 elaborates on this. Test cases (`instances/test_*.txt`) are released with the code template for validating your implementation before submission.

Submission Details

Code submission: Compress the following files into one .zip file and upload to Gradescope

- `single_agent_planner.py`
- `kr_cbs.py`
- `ta_random.py`
- `ta_distance.py`
- `hungarian.py`
- `ta_cbs.py`
- `execution_manager.py`

- `any other Python files used for Task 4`

Write-up submission: Fill in `writeup-template.tex` and upload the PDF version to Gradescope

0 Task 0: Preparing for the Project

This section will look awfully similar to the one in the first assignment. Feel free to skip it.

0.1 Installing Python 3

This project requires a Python 3 installation with the `numpy` and `matplotlib` packages. On Ubuntu Linux, download python by using:

```
sudo apt install python3 python3-numpy python3-matplotlib
```

On Mac OS X, download Anaconda 2019.03 with Python 3 from `https://www.anaconda.com/distribution/#download-section` and follow the installer. You can verify your installation by using:

```
python3 --version
```

On Windows, download Anaconda 2019.03 with Python 3 from `https://www.anaconda.com/distribution/#download-section`.

On Ubuntu Linux and Mac OS X, use `python3` to run python. On Windows, use `python` instead.

You can use a plain text editor for the project. If you would like to use an IDE, we recommend that you download PyCharm from `https://www.jetbrains.com/pycharm/`. The free community edition suffices fully, but you can get the professional edition for free as well, see `https://www.jetbrains.com/student/` for details.

0.2 The Assignment Code

Clone the GitHub repository with the followin gcommand

```
  git clone --recursive https://github.com/philip-huang/16891_HW2_Sp26.git
```

You should have all the necessary files to start the assignment. The main driver code sits in `run_experiments.py`. Some helper functions are provided along the way, so it is recommended to use them and not change anything unless the section has a `TODO`. This code builds on that for HW1.

1 k-Robust Conflict Based Search

k-Robust CBS (proposed by [1]) is a variant of Conflict-Based Search (CBS) that enforces a margin of k time-steps between robots. In other words, robots can be *delayed* for up to k time-steps and still be able to carry out their plans without colliding with each other. In this first part of the homework, you will modify your CBS implementation from Assignment 1 to be k-robust.

Before you begin, take a look at `kr_cbs.py`. This file is almost identical to `cbs.py` from the first assignment. There are minor changes to the signature of the constructor of the planning algorithm to allow for passing in the k parameter.

```
class KRCBSSolver(object):
  ...
  def __init__(self, my_map, starts, goals, k=0):
```

You may begin by copying over your code from the previous assignment for CBS and the low-level A* planner. Once you do that (without modifying anything), you should be able to run the command:

```
python run_experiments.py --instance instances/exp3_2.txt --solver KRCBS --k 2
```

Note the last argument specifying the k parameter. You should be able to see a visualization where each agent has a tail of length k steps (denoted by $2k$ circles). Your solution will likely be invalid in the k-robust sense and the script will report `(agent-tail)` collisions.

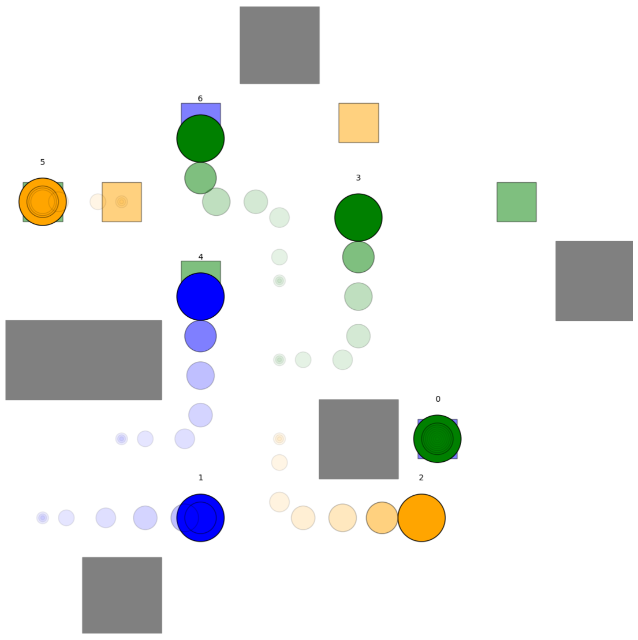

Figure 1: kR-CBS visualization. The "tail" following each agent denotes reserved space that other agents are not allowed to collide with. This is a safety margin of k time-steps. ($k = 4$ in this example.)

1.1 Implementing k-Robust CBS

To implement k-Robust CBS, otherwise known as kR-CBS, you will modify the way your code identified conflicts and the way it resolves them with constraints.

Conflicts are now defined as follows. Let's say we have a CT node N in which two agents i and j have associated paths $\boldsymbol{\pi}^i, \boldsymbol{\pi}^j$. Let $\boldsymbol{\pi}^i_t$ denote the location of agent i at time-step $t \in \mathbb{Z}^{\geq 0}$. The agents i and j are conflicting within N if there exists some $\Delta \in \{0, 1, \ldots k\}$ such that $\boldsymbol{\pi}^i_t = \boldsymbol{\pi}^j_{t+\Delta}$. This is also called a k-delay conflict.

Constraints are imposed on individual agents' low-level planners to resolve k-delay conflicts. Let's say that two agents i, j conflict at vertex v, i.e., $\boldsymbol{\pi}^i_t = \boldsymbol{\pi}^j_{t+\Delta} = v$. k-Robust CBS imposes two vertex constraints, one on each agent, $\langle i, v, t\rangle$ and $\langle j, v, t+\Delta\rangle$. In class we have also talked about multiple different ways to apply constraints in kR-CBS. For this assignment, you are free to implement any type of constraint you would like. The original constraints proposed in the paper may not be sufficient for solving all test cases. Please consult [1] and the course notes for more details.

Task 1 (1.5 points) Implementation
Please **implement** kR-CBS in `kr_cbs.py` and submit the associated code files.

Task 2 (1 points) Written
Answer the following question in your writeup.
For any $k \geq 1$, is there a need to find edge conflicts and impose edge constraints in kR-CBS?
Is it problematic to run kR-CBS with $k = 0$? Explain your answer.

How Will Your Code Be Evaluated?

The autograder will run your code on various maps and expect your resulting paths to be k-robust and have optimal cost. You can check the autograder cases and expected cost in this csv.

2 Target Assignment and Path Finding

Excellent! Your CEO is *slightly* happier! With your robots colliding less now that they have a safety margin added to their plans, she convinced customers that uninterrupted operations would begin within a month. No pressure, but I think you should get to work.

The second key component of the system is a task assignment module. Remember, the input to every planning call is not a mapping between agent and goal position, but instead a set $\mathcal{T} = \{q_1, q_2, \ldots, q_n\}$ of goals (also called targets) that need to be reached. Each goal $q_g \in \mathcal{T}$ is a 2D coordinate (i.e., $q_g \in \mathbb{Z}^2 \quad \forall g \in \{1, \ldots, n\}$) vertex on our graph, and needs to be reached by *some* agent. This is known as the *anonymous MAPF* problem. We'll use the words "task" and "target" interchangeably in this homework.

2.1 Naive Target Assignment Strategies I

One simple (but uninformed) way to assign tasks to robots is by randomly assigning goals to agents. Starter code is provided in `ta_random.py` and can be run with

```
python run_experiments.py --instance instances/test_1.txt --solver TA-RANDOM
↪  --k 2
```

Here, any assignment computed will be passed to kR-CBS for path finding.

Task 3 (1 points) Implementation
Please implement the random assignment strategy in `ta_random.py`. You'll notice that this is a derived class of `KRCBSSolver` so there is no need to repeat much of the functionality there. The only required modification is setting `self.goals` in a random order in the constructor.

Task 4 (0.5 points) Written, Empirical
For the instance `test_1.txt` and $k = 2$, please run a few experiments (at least 10), record the sums of costs across experiments and report their minimum, maximum, mean, and variance.

2.2 Naive Target Assignment Strategies II

Another way to perform task assignments is by distance. Here, we say that the *cost* to assign an agent i to a goal $q_g \in \mathcal{T}$ is $d_g^i = \text{dist}(q_{\text{start}}^i, q_g)$. With $\text{dist}(q_{\text{start}}^i, q_g)$ being the distance of the shortest path between the start position of agent i and some goal position q_g.

In this method, we want to assign all agents to goals such that the overall assignment cost is minimized. Formally, the minimum cost assignment is the ordered set of goal positions T, which is a permutation of $\mathcal{T}$ (a permutation within the symmetric group $S_{\mathcal{T}}$). We use T^i to index into the set T at index i.

$$\arg\min_{T \in S_{\mathcal{T}}} \sum_{i=1}^{n} \text{dist}(q_{\text{start}}^i, T^i)$$

This assignment T will again be passed to kR-CBS like before.

To find the assignment T, we will be using the Hungarian algorithm [3] in this homework. You will find started code in `hungarian.py`. Run experiments with:

```
python run_experiments.py --instance instances/test_1.txt --solver TA-DISTANCE
↪  --k 2
```

Task 5 (1 points) Implementation
Implement distance-based assignment in `ta_distance.py`. To do so, you will need to also "implement" the Hungarian Algorithm in `hungarian.py`. You are allowed to (and encouraged to) use `linear_sum_assignment` from `scipy.optimize`. This function is essentially all you need for this part – it includes an implementation of the Hungarian algorithm.

Task 6 (0.5 points) Written
Report the sum of costs obtained using this target assignment strategy in `test_1.txt` with $k = 2$. How does it compare to the values obtained via random assignment?

Task 7 (0.5 points) Written
Give an example of a scenario where this approach would yield a **sub-optimal** assignment in terms of sum of costs.

Task 8 (0.5 points) Written
Give an example of a scenario where this approach would yield an invalid assignment. I.e., a set of goals that cannot be reached by their assigned agents.

Note I: Both of the naive methods we have explored so far are hierarchical and do not have feedback. That is, an assignment is computed first, kept fixed, and then path finding is carried out. Normally such methods are not optimal and could also be incomplete.

2.3 Target Assignment Conflict-Based Search

Great. You have experimented with two naive hierarchical ways to assign targets to agents. You and your CEO think this is a good start, and you also noticed some issues with them (hopefully, you identified those too in the written questions above). So you decide to try your hand at computing an *optimal* assignment by jointly considering the assignments of agents to targets and the associated path costs computed under the assignments. After looking for a method that will produce valid globally optimal assignments in terms of the sum of costs, you converge on a variant of Iterative-Target-Assignment CBS (ITA-CBS) [4] that you call TA-CBS.[1]

The first and most major difference between CBS and TA-CBS is the structure of CT nodes.

- In CBS, CT nodes need to keep track of *at least* the current sum of costs, the current paths per agent, and the constraint set per agent.

[1]**Note 0:** The algorithm TA-CBS was never published under this name. It is a variant of ITA-CBS. **Note 1:** The only difference between TA-CBS and ITA-CBS is that ITA-CBS uses the Dynamic Hungarian algorithm while TA-CBS uses the good-old Hungarian algorithm. Both algorithms solve for an optimal target assignment, with the only difference being that Dynamic Hungarian leverages previous assignments to inform new queries and the regular Hungarian algorithm solves for the optimal assignment from scratch. You can still follow the pseudocode in [4] as is. **Note 2:** Technically you'll be implementing k-Robust TA-CBS in this assignment.

- TA-CBS CT nodes keep track of the same data as CBS with an additional matrix M_c that holds the current shortest path cost for each agent from its start to each goal (while respecting the constraints imposed on the agent).

The way TA-CBS operates is fairly simple. Following a similar structure to CBS, on each CT node split, TA-CBS creates two child CT nodes with a new constraint imposed on one agent in each of the two new nodes. In CBS, one path finding call would be done per agent. In TA-CBS, $|\mathcal{T}|$ calls are done – it computes the shortest path for an agent between its start position to *all goals*. This information is stored in M_c, and a currently-optimal target assignment is computed based on M_c. The pseudocode for ITA-CBS is included here (Algorithm 1) for reference. See the paper for symbol definitions.

Algorithm 1 ITA-CBS algorithm (directly from [4])

```
Input: Graph $G$, start locations $\{s_i\}$, target locations $\{g_i\}$, target matrix $A$
Output: Optimal TAPF solution
 1: OPEN = PriorityQueue()
 2: $\Omega^0 = \emptyset$
 3: for each $(i, j) \in \{1, \cdots, N\} \times \{1, \cdots, M\}$ do
 4:     if $A[i][j] = 1$ then
 5:         $M_c^0[i][j]$ = shortestPathSearch($G$, $s_i$, $g_j$, $\Omega_0$)
 6:     else
 7:         $M_c^0[i][j] = \infty$
 8:     end if
 9: end for
10: $\pi_{ta}^0$ = optimalTargetAssignment($M_c^0$)
11: $c^0, \pi^0$ = getPlan($\pi_{ta}^0$, $M_c^0$)
12: $H_0 = \{c^0, \Omega^0, \pi^0, \pi_{ta}^0, M_c^0\}$
13: Insert $H_0$ to OPEN
14: while OPEN not empty do
15:     $H_{cur}$ = OPEN front node; OPEN.pop()
16:     Validate $H_{cur}.\pi$ until a conflict occurs
17:     if $H_{cur}.\pi$ has no conflict then
18:         return $H_{cur}.\pi$
19:     end if
20:     $(i, j, t)$ = getFirstConflict($H_{cur}.\pi$)
21:     for each agent $k$ in $(i, j)$ do
22:         $Q = H_{cur}$
23:         if $(i, j, t)$ is vertex conflict then
24:             $Q.\Omega = Q.\Omega \cup (k, v_t^k, t)$
25:         else
26:             $Q.\Omega = Q.\Omega \cup (k, v_{t-1}^k, v_t^k, t)$
27:         end if
28:         for each $x$ with $A[k][x] = 1$ do
29:             $Q.M_c[k][x]$ = shortestPathSearch($G$, $s_k$, $g_x$, $Q.\Omega$)
30:         end for
31:         $Q.\pi_{ta}$ = optimalTargetAssignment($Q.M_c$)
32:         $Q.c, Q.\pi$ = getPlan($Q.\pi_{ta}$, $Q.M_c$)
33:         Insert $Q$ to OPEN
34:     end for
35: end while
36: return No valid solution
```

Task 9 (3.0 points) Implementation
Please implement TA-CBS in `ta_cbs.py`.

Task 10 (0.5 points) Written
Compare the performance of the naive target assignment methods to that of TA-CBS. Do so for `test_1.txt` with $k = 2$ and for two other maps of your choice. Report planning times and sums of costs. Which planners are more efficient, are the differences in cost significant?

How Will Your Code Be Evaluated?

The autograder will run your code on various maps and expect your resulting sum-of-costs to be optimal. Your paths still need to be k-robust as before.

Note II: In real-world warehouses, the flow of goals (e.g., package pickup and delivery locations) is normally given by a *stream*. That is, a continuous sequence of tasks that robots need to execute. This problem, otherwise known as *Lifelong MAPF* or *Lifelong Multi-Agent Pickup and Delivery (MAPD)* is not addressed by our formulation here. We solved the integrated task assignment and path finding problem in a one-shot manner.

3 Execution Policy

Excellent! Your CEO is becoming confident in your ability to set up a robust multi-robot planning stack for the company. Your system can now assign goals to robots and plan paths with safety margins. Good work. Being confident in your algorithms, which seemed to be working well in all the simulated scenarios you tried, you scheduled a set of experiments with the real robots to run overnight. On the following morning, you were greeted with *"Good morning, Destroyer"* by the mechanics at your company. More robots have collided overnight and were damaged. This led you to realize that the current system has a major flaw – it does not handle situations where robots are delayed by an arbitrarily long amount of time. In the current system, if a robot is delayed for more than k time-steps it is in danger of colliding with other robots.

To model this situation you have created a more realistic simulator. In the new simulator, you (we) now give single-step commands to agents and allow (some of) them to fail in the execution of commands with probability p. The simulator then gets feedback with execution success information. This is closer to how real robots would execute MAPF paths. You'll find an abstract `ExecutionManager` class in `execution_manager.py` that handles this behavior when instantiated in `run_execution_experiments.py`. We also include an example of a naive execution manager that simply returns the next step in the planned path regardless of other agents in `execution_manager.py`: `TACBSExecutionManager`. Try it out with this command:

```
python3 run_execution_experiments.py --instance instances/exp0.txt --k 1
--fail_prob 0.5
```

Your task in this section is to devise an execution policy that would adaptively send actions to robots based on the progress of the entire multi-robot system. This is an open-ended problem that you can solve however you'd like. The Lecture 9 slides could be very helpful here. Before you begin, consider (and please respond to) this question:

Task 11 (0.5 points) Written
Consider a greedy policy: a policy that commands agents to step forward along their paths as long as their next cell is unoccupied. What could go wrong when using this policy in general MAPF problems? (By "general MAPF problems" we mean MAPF problems on the regular 4-connected grid graphs. Those are the regular problems we have been looking at throughout the class so far.)

One promising framework to explore for your execution policy is a Temporal Plan Graph (TPG). A TPG is good for decentralized execution under arbitrary delays and is fairly simple to implement. If you choose to take this route, please consult the Lecture 9 slides for details. Please note that there are some limitations to MAPF solutions that can be executed under the guidance of a TPG. For example, a MAPF solution that includes multiple agents following each other directly in a loop (like in Fig. 2) would result in none of them moving under the TPG. One "hack" for ensuring that no such situations exist in a MAPF plan is by having it be 1-robust.

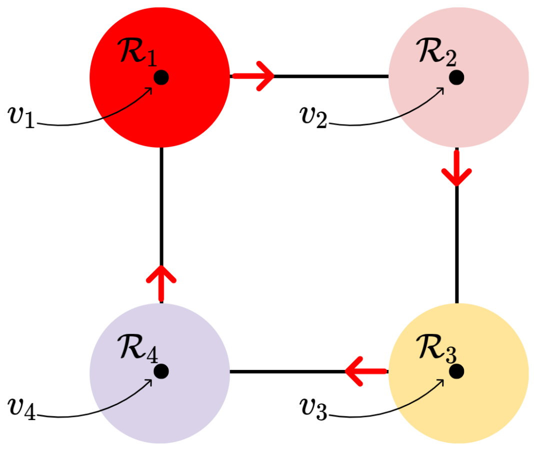

Figure 2: A very small graph that only consists of the vertices v_1, v_2, v_3, v_4 and single step agent plans moving in a loop (e.g., the plan for agent $\mathcal{R}_1$ is $\{v_1, v_2\}$).

Task 12 (1 points) Written
Please draw the TPG the corresponds to the MAPF solution depicted in Fig. 2 and explain why, when robots request a *next step* positions from the TPG, they will all end up waiting. This written problem aims to help you build intuition for how TPGs work and their limitations.

Task 13 (3.5 points) Implementation
Implement your execution policy in the class `WorksReallyWellExecutionManager` within the `execution_manager.py`.
We have provided some starter code there if you want to implement a TPG, but feel free to change your implementation as long as `get_next_location_for_all_agents` and `feedback_successful_agent_ids` functions can interact successfully with the `run_execution_experiments.py` and your agents never collide.
Pass the argument `-execution_manager WorksReallyWell` to evaluate your own policy. Tests `test_20`, `test_21`, `test_22`, `exp0`, and `exp2_2` are nice for evaluating the correctness of your execution monitor.

How Will Your Code Be Evaluated?

The autograder will run your code on various maps and check for validity in the motions of the agents. Shorter sums of costs will be given higher scores. As long as your agents never collide, you will be getting the vast majority of the points for this implementation task.

4 SMART Simulator Integration

It has been a month since you deployed the ITA-CBS algorithm with execution policy in your warehouse. The robots are moving, and there are no more overnight collisions! Customers are happy with the operations, and your CEO is pleased. In fact, business is so good that orders are pouring in quickly and the wearhouse is not keeping up with the increasing demand. Your CEO urges you to keep improving the performance and throughput of the automated warehouse even more!

You went back to your desk and started exploring ideas in the literature. Your CEO has just recently forwarded you a paper titled SMART (A Scalable Multi-Agent Realistic Testbed) [5]. The paper claims that SAMRT has a more realistic simulation, a built-in execution monitoring framework, and can scale to thousands of the robot. Since the code is publicly available, you decided to give it a try to better evaluate its claim and determine whether it can be used to improve your MAPF algorithms.

To use SMART, you have two options - local installation, or a public web demo. **We recommend using the web version**. If you decide to install locally (feel free to skip this part and use the web demo), we have already included the code as a git submodule under the `smart` directory. Navigate to that directory and follow the installation instruction in the README. Specifically, you need an Ubuntu 22.04 machine (or using the Windows on Linux Subsystem) to install the ARGoS simluation. Download the ARGoS simulator from here, and run `sudo apt install ./argos3_simulator-3.0.0-x86_64-beta59.deb` to install. Then, under the `smart` folder, run

```
git submodule init && git submodule update
mkdir build && cd build
cmake .. && make -j
```

to install. You should be able to run the simulation locally with

```
python3 run_sim.py --map_name=random-32-32-20.map \
--scen_name=random-32-32-20-random-1.scen \
--num_agents=50 --path_filename=example_paths_xy.txt --flip_coord=0
```

If you have problem with display, you could also run the simulation in headless mode with `-headless=True`

The smart simulator parses the map and the agents' paths differently than our code base. Thus, we have provided a way to save your planned path in a SMART-compatible format. To do so, you can run the `run_experiment.py` with the `-smart` option to generate the agent paths. For simplicity, we will not do target assignment and only run a naive CBS planner. SMART natively uses an execution manager similar to the TPG to control the robots based on the plan, so we can also plan path with or without the k-robust setting.

```
python3 run_experiments.py --solver KRCBS --k 0 --batch \
--instance instances/test_0.txt --smart
```

This generates three files in the `smart/generated` folder - the map file `test_0-krcbs-k0.map`, the scenarios file including start and goal locations `test_0-krcbs-k0.scen`, and the robot paths `test_0-krcbs-k0-paths_xy.txt`.

You can run and then upload the corresponding map file, scenario, and solution of the `Input` tab in the web demo. Once you have successfully uploaded the files, click the Simulate button. You should then see the robots move in simulation. The original sum of cost of the MAPF plan, as well as the new makespan and sum of cost under realistic robot dynamics, should be reported in the Statistics tab. If you would like to update your map or trajectory, you may need to refresh the webpage, re-upload the files, and re-run the simulation.

Your task in this section is to try running existing MAPF algorithm in the SMART simulator and assess its performance. Then, you will be tasked with an open-ended question to implement a different MAPF solver to improve your plan in the SMART simulator.

Task 14 (1 points) Written, Empirical
Run the existing MAPF plan on the SMART simulator on the tests `test_0`, `test_1`. Report the actual simulated average cost from SMART for `k=0` and `k=2`. What did you see?

Task 15 (1 points) Written
Explain the reason why there is a difference between the original sum of cost from your discrete path and the robot execution time simulated under SMART. What are the differences in terms of robot behavior in these simulations? Why would changing the `k` affect the performance?

Now that you have more experience with the SMART simulator, your next task is to find a new way to improve your MAPF algorithm to work better on a more realistic simulator. For example, consider your answer to the previous question and think about how some of the existing assumptions in the original planner affect real-world performance. You may want to modify the robot action model and the cost of certain actions. You are allowed to modify the code however you like and use any external library you want.

Task 16 (2 points) Implementation
Implement a new strategy to improve the average costs of your MAPF plan on the SMART simulator. Test it on the `test_0`, `test_1`, `test_2` and `test_15`. In your submission, you should upload the code used for your implementation. Report the simulated makespan and average costs in SMART before and after your changes.

Task 17 (1 points) Written
Discuss your strategy to improve the performance (e.g. sum of costs) in the SMART simulator. Explain your implementation and walk us through your code.

How Will Your Code Be Evaluated?

There will be no autograder in this section. You are required to submit your code and explain key decisions you made in your write-up. A lower sum of cost will be given higher scores. The question is open-ended, so the vast majority of the points will be awarded based on the merit of your idea and your implementation effort.

References

[1] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. Robust multi-agent path finding. In *Proceedings of the International Symposium on Combinatorial Search*, volume 9, pages 2–9, 2018.

[2] Wolfgang Hönig, TK Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Multi-agent path finding with kinematic constraints. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 26, pages 477–485, 2016.

[3] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[4] Yimin Tang, Zhongqiang Ren, Jiaoyang Li, and Katia Sycara. Solving multi-agent target assignment and path finding with a single constraint tree. In *2023 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, pages 8–14. IEEE, 2023.

[5] Jingtian Yan, Zhifei Li, William Kang, Kevin Zheng, Yulun Zhang, Zhe Chen, Yue Zhang, Daniel Harabor, Stephen F. Smith, and Jiaoyang Li. Advancing mapf towards the real world: A scalable multi-agent realistic testbed (smart), 2025.