This supplementary material provides additional detailed descriptions of the experiment including the dataset, parameters, experiment setup and design, and code implementation. The implemented codes can be found in `https://github.com/waldyjoe/MADPRP`.

A Data

Multi-Agent Dynamic Police Patrol Dispatching and Rescheduling Problem (MADPRP) is essentially a multi-agent version of the problem described in [1]. We refer our readers to [1] for detailed problem and model descriptions. While we try to mimic the real-world problem setting as close as possible, we evaluate our proposed approach with synthetically-generated data due to classified nature of the data. Nevertheless, data pertaining to police sector boundaries is obtained from publicly-available data provided by the local government authority (see `https://data.gov.sg/dataset/singapore-police-force-npc-boundary`).

Prior to running the experiments, there are several data preparation steps needed to be done. Firstly, we need to create the multi-sector police patrol environment that includes and not limited to police patrol sector boundaries, patrol areas, patrol teams, point-to-point travel time matrix, neighbourhood relationship amongst the agents and patrol time requirement per patrol area. Secondly, we need to create a simulator to simulate the realizations of dynamic events in a given planning horizon.

In this section, we make references to our code directory (in this format ***/folder/subfolder/***) to point the readers to the location where those data and the relevant codes are stored for ease of reading and reproducibility. Note that due to file size limit only a subset of the dataset is provided in the github repository.

A.1 Environment Setup

Hexagonal grids of diameter 2.22 km each are drawn over the local police sectors. Each grid represents a patrol area. We assume that patrol teams have the flexibility to plan their own routes within a patrol area. We use QGIS, an open source GIS software (`https://www.qgis.org/en/site/`) to draw the hexagonal grids and the resulting mapfile can be found in ***/data/RawData/grid0.02_npc.geojson***. Figure 1 shows a sample grid map of 3 police sectors.

To determine the minimum patrol time requirement per patrol area, we use the number of road nodes in a patrol area as a proxy. The road nodes data are represented as coordinates and we use QGIS to label each node with the corresponding patrol area grid. The data pertaining to the number of road nodes for each patrol area can be found in ***/data/RawData/grids_node_count.pkl*** and the data source can be found in:

`https://data.gov.sg/dataset/master-plan-2019-road-name-layer`.

To generate the point-to-point travel time matrix from each patrol area, we first compute the medoid of each patrol area. We use medoid instead of centroid because of the imbalance distribution of

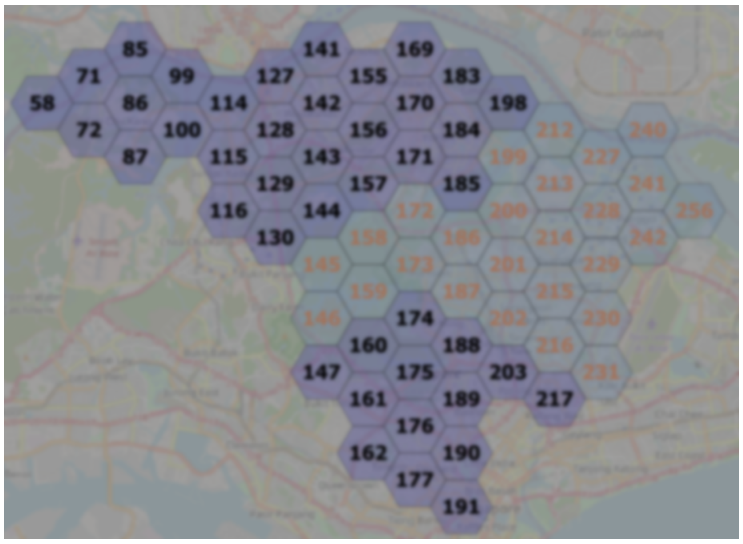

Figure 1: Hexagonal grids drawn over 3 police sectors. Image is intentionally blurred for anonymity purposes.

road (patrol) nodes in a given patrol area. Thus, medoid represents a good proxy of where patrols are more likely to be taking place. To calculate the point-to-point travel time, we use the routing API provided by Open Source Routing Machine (OSRM) (`http://project-osrm.org/`). The data pertaining to the coordinates of the medoids and the travel time matrix can be found in ***/data/RawData/grids_medoid.pkl*** and ***/data/RawData/travel_time_matrix.pkl*** respectively.

The neighbourhood relationship matrix of the sectors can be found in ***/data/RawData/adjacency_matrix.csv***. This is used to build communication network for the baseline model, MAVFA-C-H.

/data/processed_data.pkl is the main input file that integrates all the above data and represents a multi-sector police patrol environment that is used subsequently by the simulator to run training and testing experiments. The following files contain the codes to set up the modelling environment that have been described in this section: ***PreProcessInputData.py, /preprocess/PreProcessGeoFile.py*** and ***/preprocess/ProcessInputData.py***.

A.2 Simulator

We defined a planning horizon of length T as a 12-hour shift discretized to 10-minute time periods. The simulator is developed to simulate multiple scenarios of any given shift. Each scenario consists of initial patrol schedules of all agents and the occurrences of dynamic event throughout a given shift.

A.2.1 Initial Schedule

We derive the initial schedule of an agent by formulating and solving a mathematical model based on the static version of the police patrol scheduling problem. The code implementation of this mathematical model can be found in ***/model/SetCover.py***. Initial schedules sample generator code can be found in ***InitialScheduleSampleGenerator.py***. The sample initial schedules that have been generated can be found in ***/data/SampleInitialSchedule/2/Sector_X/***.

A.2.2 Incident Generation

Each scenario also consists of a list dynamic events in a given shift. Each incident consists of its arrival time, sector where the incident takes place, location (represented by patrol area) and how long the incident takes to be resolved (resolution time). We model the inter-arrival time of the dynamic incident using a Poisson process with λ as the rate of occurrences of dynamic incidents per hour. The probability distribution of an incident happening in a given patrol area in a given sector is generated based on its minimum patrol time requirement (we assume that areas that require longer patrol time has a higher likelihood for an incident to occur) and a random factor. This probability distributions can be found in ***/data/location_pdf_X.pkl***. We model the resolution time as gamma distribution with the following key parameters a, $scale$ and loc (see `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.gamma.html` for detailed descriptions of the parameters). We choose gamma distribution because of the assumptions that short, less serious incidents occur more often than long, serious incidents. The incident and scenario generation codes can be found in ***/data/ScenarioGenerator.py***.

A.2.3 Training and Testing Data

We synthetically generate 100 samples of initial patrol schedules for each sector. 70% of the sample is used for training while 30% is used for testing. The sample initial schedule for training can be found ***/data/Training/2/Sector_X/*** while those used for testing can be found in ***/data/Testing/2/Sector_X/***.

For reproducibility, we created training instances for 10,000 training episodes in which each training episode contains the initial schedule and the corresponding set of scenario of occurrences of dynamic events. These instances can be found in ***/data/Training/instances/2/***. Similarly for testing, we created the exact test cases that we run for the experiments and they can be found in ***/experiment/test_cases/***. Each test case file consists of an initial patrol schedule and multiple numbers of realizations of dynamic incidents to simulate different scenarios that may take place in a given shift.

A.3 Model Parameters (MAVFA)

This section covers the parameters used solely to implement our proposed MAVFA where we represent the different components of the joint value function as neural networks. Here are the

Hyperparameter	**Value**
γ	0.99
buffer size	1e5
batch size	64
τ (for soft update)	1e3
learning rate	5e-4
learn frequency	every 10 steps
update frequency	every 20 steps
ε-decay rate	25000
ε-decay start	0.9
ε-decay end	0.05

Table 1: List of hyperparameters used by MAVFA.

details of the implemented networks:

- **Encoder.** We represent the encoder network as a fully-connected neural network with 2 hidden layers with 128 and 64 nodes respectively with encoding dimension of 5.
- **Local Value Network.** We represent the encoder network as a fully-connected neural network with 2 hidden layers with 64 and 32 nodes respectively.
- **Mixing Network.** We represent the encoder network as a fully-connected neural network with 2 hidden layers with 64 and 32 nodes respectively.

We use ReLU as activation function and Adam optimizer for all networks. Table 1 summarizes the hyperparameters used in MAVFA. These parameters are either passed as arguments prior to running the model or hard-coded in the parameter file (see ***/constant/Settings.py***). The implementation codes for MAVFA can be found in ***/mavfa/***). The trained parameters for our proposed approach can be found in ***/mavfa/NoComms/BR/parameter/***).

A.4 Model Parameters (Others)

Table 2 summarizes the rest of the parameter values used in our experiment and the corresponding description. These are parameters used in building the environment, simulator and in running the heuristic. These parameters are either passed as arguments prior to running the model or hard-coded in the parameter file (see ***/constant/Settings.py***).

Parameter	**Value**	**Description**
λ	2	Rate of occurrences of dynamic incidents per hour
a	3	Parameter for incident resolution time distribution
loc	10	Parameter for incident resolution time distribution
τ_{target}	20 minutes	Response time target
τ_{max}	3 time periods	Any incident must be assigned within this time
$\|T\|$	720 minutes	Duration of a shift
β_p	2.5	Rate of decay used in U_p
β_r	25	Rate of decay used in $f_r(x_k)$
P_{max}	0.4	Hamming distance in terms of %
ε_h	0.3	Probability of exploring random chain in rescheduling heuristic based on ejection chains
K	100	Max iteration for iterative best response algorithm
ε_b	0.7	Probability of improving the current best joint plan in iterative best response algorithm

Table 2: List of model parameters use and their corresponding descriptions.

A.5 Experiment Results

All the output data pertaining to the experimental results (Phase 1) can be found in ***/mavfa/NoComms/BR/output/*** and ***/mavfa/NoComms/BR/parameters/*** where the former contains the training statistics while the latter contains the trained parameters. Note that ***/mavfa/NoComms/BR/*** refers to our proposed approach MAVFA-BR-H. Separate subfolders are created for different variations of the model namely MAVFA-H and MAVFA-C-H and all the models and the corresponding training output files can be found in ***/mavfa/***. Meanwhile, all the output data pertaining to the experimental results (Phase 2) can be found in ***/experiment/2/Sector_EFL/***. There are two output files per model where one contains the output pertaining to solution quality and another pertaining the computational time. The codes to preprocess and analyze these outputs can be found in ***/analysis/***)

B Code

The codes are structured as follows:

- In the main directory, there are 3 python scripts for training, testing and generating sample initial schedules. The names of the scripts are self-explanatory. There is a ***READ_ME*** file to provide step-by-step guide on how to use run the various python scripts.
- ***analysis***. Contains the python scripts to process the output training and testing files.
- ***constants***. Contains the settings file where all the hardcoded parameters are stored and read from.
- ***data***. Contains the raw and processed input files.
- ***entity***. Contains the base classes in building the model and the environment.
- ***experiment***. Contains the output files of the experiment results (Phase 2).
- ***mavfa***. Contains the multi-agent RL model and the corresponding training output files.
- ***model***. Contains all non-RL models such as the rescheduling heuristic, simulator and mathematical model to generate initial schedules.
- ***output***. Output files generated after running the experiments will be stored here.
- ***preprocess***. Contains scripts to preprocess the raw input files.
- ***util***. Contains common utility functions.
- ***vfa***. Contains the single-agent RL model and the corresponding training output files.

The codes are written and run in Python 3.7 with the following key libraries (sorted in alphabetical order):

- beautifulsoup4 (to parse the raw input data)
- haversine (to compute travel distance matrix)
- geojson (to process geospatial data)
- pandas
- ray (to generate training experiences in parallel)
- scipy
- torch

To generate the initial schedules (see Section A.2.1), Python API of any CPLEX that is compatible to the corresponding python version is required. The experiments are run on a server with the following configurations: Rocky Linux 8.6, 64-Core processor and 384GB RAM.

B.1 Implementation Consideration

The main challenge of learning policy to make complex decision directly is that it is very computationally expensive to run numerous training episodes. To illustrate, assuming each decision takes a realistically reasonable time of 20s and there are 30 dynamic incidents per day, one simulation episode takes 10 mins. To train the model in magnitude of a thousand will take at least 10,000 mins or at least a week if training episodes are done in sequence.

To address this scalability issue, we parallelize the generation of experiences. As each training episode is independent of each other, multiple episodes can be run in parallel resulting in faster learning process as more experiences are being generated at any one time. This distributed RL mechanism relies one a single learner but multiple workers to generate experiences.

References

[1] Waldy Joe, Hoong Chuin Lau, and Jonathan Pan. “Reinforcement Learning Approach to Solve Dynamic Bi-objective Police Patrol Dispatching and Rescheduling Problem”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 32. 2022, pp. 453–461.