KDDCup Documentation

Release 0.1

1

Apr 02, 2021

CONTENTS:

CITY BRAIN CHALLENGE

In this challenge, we will provide you with a city-scale road network and its traffic demand derived from real traffic data. You will be in charge of coordinating the traffic signals to maximize number of vehicles served while maintaining an acceptable delay. We will increase the traffic demand and see whether your coordination model can still survive.

1.1 Problem

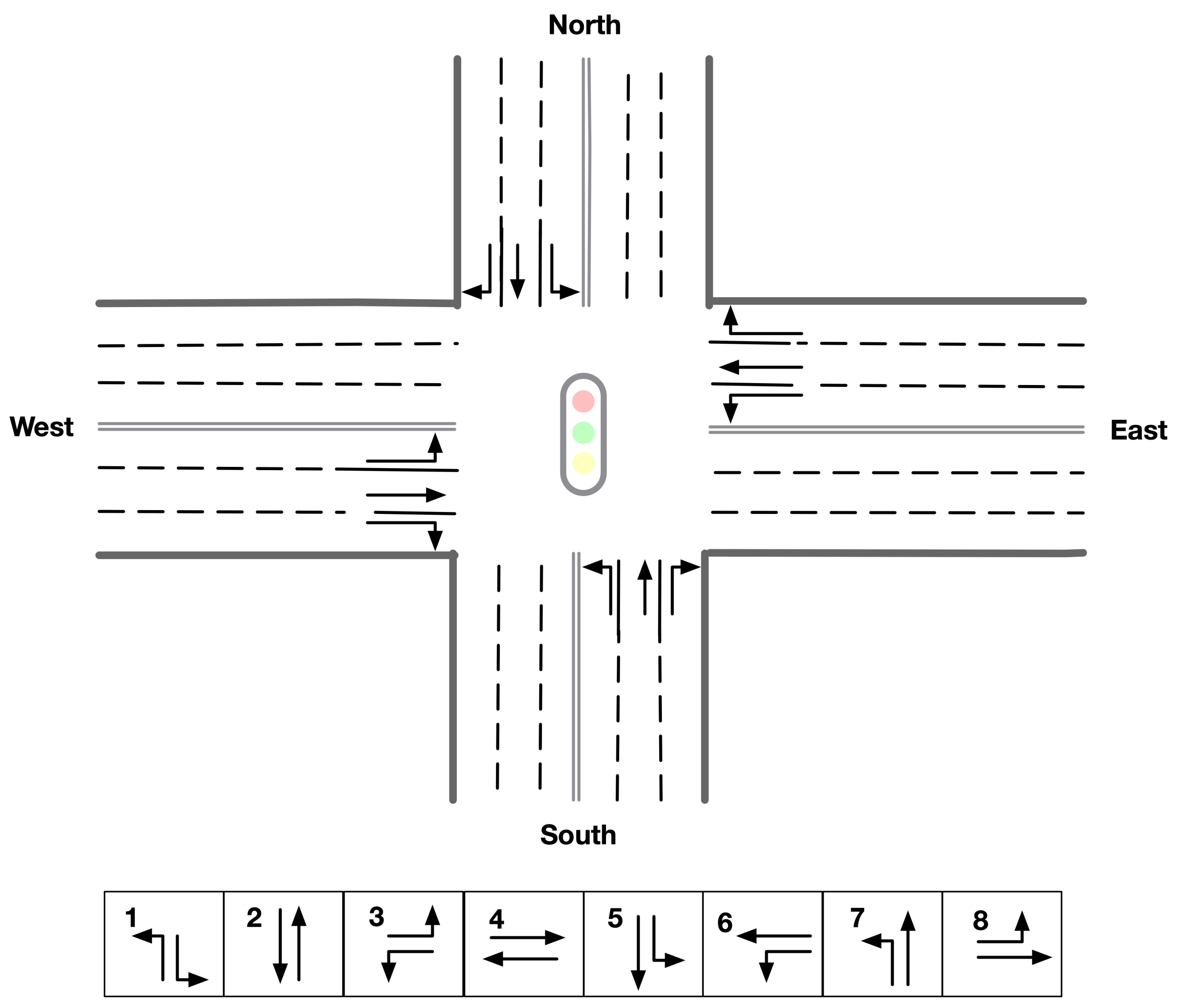
Traffic signals coordinate the traffic movements at the intersection and a smart traffic signal coordination algorithm is the key to transportation efficiency. For a four-leg intersection (see figure below), 1 of 8 types of signal phases can be selected each period of time step, serving a pair of non-conflict traffic movements (e.g., phase-1 gives right-of-way for eastbound and westbound through traffic). In this competition, participants need to develop a model to select traffic signal phases at intersections of a road network to improve road network traffic performance.

1.2 Evaluation

Delay index will be calculated to measure road network traffic performance. The solutions with a lower delay index will be ranked higher. For a completed trip, the delay is computed as actual travel time divided by travel time at free-flow speed. For an uncompleted trip, the free-flow speed is used to estimate the travel time of rest of the trip. The delay index is computed as average trip delay over all vehicles served: $D = \frac{1}{N}\sum_{i=1}^{N} D_i$.

The trip delay D_i **of vehicle** i **is defined as** $D_i = \frac{TT_i + TT_i^r}{TT_i^f}$**, where,**

- TT_i: travel time of vehicle i;
- TT_i^r: rest of trip travel time, estimated with free-flow speed;
- TT_i^f: full trip travel time at free-flow speed



Traffic signal phase

1.3 Code

Participant will get a `starter-kit`. It contains:

```
# Participants must implement
agent/agent.py

# Participants could modify
agent/gym_cfg.py
cfg/simulator.cfg

# A simple demo of the environment
demo.py

# Scoring and simulation program
evaluate.py

# Other files and directory
data/flow_1x1_txt
data/flow_warm_up_1000.txt
data/roadnet_1x1.txt
data/roadnet_warm_up.txt
log/
out/
```

Participants should implement their algorithm in `agent.py`. And then execute `evaluate.py` to get scores. Participants could modify `simulator.cfg` and `gym_cfg.py`.

CHAPTER

TWO

TRY IT YOURSELF

Here, we provide guidelines for setting up the simulation environment and submitting results.

2.1 Installation guide

2.1.1 Installation via Docker

The simulator engine and the gym environment are incorporated into the docker image. You can pull it down to easily setup the environment. The latest image version is `0.1.0`, we will notify you if a new version is updated.

```
docker pull citybrainchallenge/cbengine:0.1.0
```

Then you can clone the code of the starter-kit.

```
git clone https://github.com/CityBrainChallenge/KDDCup2021-CityBrainChallenge-starter-
↪kit.git
```

After pulled down the docker image and cloned the starter-kit, you can run a docker container and run the code of the `starter-kit` repo.

```
docker run -it -v /path/to/your/starter-kit:/starter-kit citybrainchallenge/
↪cbengine:0.1.0 bash
cd starter-kit
python3 evaluate.py --input_dir agent --output_dir out --sim_cfg cfg/simulator.cfg
```

2.2 Run simulation

To check your simulation enviroment is ok, you can run `demo.py` in the starter-kit, where the `actions` are simply fixed. You need to overwrite the function of `act()` in `agent.py` to define the policy of signal phase selection (i.e., action).

```
import CBEngine
import gym
import agent.gym_cfg as gym_cfg

# load config
simulator_cfg_file = './cfg/simulator.cfg'
mx_step = 3600
gym_cfg_instance = gym_cfg.gym_cfg()
```

(continues on next page)

(continued from previous page)

```python
# gym
env = gym.make(
    'CBEngine-v0',
    simulator_cfg_file=simulator_cfg_file,
    thread_num=1,
    gym_dict=gym_cfg_instance.cfg
)

for i in range(mx_step):
    print("{}/{}".format(i,mx_step))

    # run one step simulation
    # you can use act() in agent.py to get the actions predicted by agent.
    actions = {0: 1}
    obs, rwd, dones, info = env.step(actions)

    # print observations and infos
    for k,v in obs.items():
        print("{}:{}".format(k,v))
    for k,v in info.items():
        print("{}:{}".format(k,v))
```

The meaning of `simulator_cfg_file`, `gym_cfg` is explained in APIs

Here is a simple example of a fixed time agent implemented at `agent.py` to coordinate the traffic signal. It use the *current_step* from observation to decide the phase.

```python
# how to import or load local files
import os
import sys
path = os.path.split(os.path.realpath(__file__))[0]
sys.path.append(path)
import gym_cfg
with open(path + "/gym_cfg.py", "r") as f:
    pass

class TestAgent():
    def __init__(self):
        self.now_phase = {}

        # pre-define time of green light
        self.green_sec = 40

        self.max_phase = 8
        self.last_change_step = {}
        self.agent_list = []
        self.phase_passablelane = {}

    ################################
    # load agent list
    # not suggest to modify this function.
    # agent_list is a list of agent_id (intersection id)
    def load_agent_list(self,agent_list):
        self.agent_list = agent_list
        self.now_phase = dict.fromkeys(self.agent_list,1)
        self.last_change_step = dict.fromkeys(self.agent_list,0)
```

(continues on next page)

(continued from previous page)

```
    ################################


    def act(self, obs):
        """ !!! MUST BE OVERRIDED !!!
        """
        # here obs contains all of the observations and infos
        observations = obs['observations']
        info = obs['info']
        actions = {}


        # preprocess observations
        # get a dict observations_for_agent that contains the features of all agents.
        observations_for_agent = {}
        for key,val in observations.items():
            observations_agent_id = int(key.split('_')[0])
            observations_feature = key[key.find('_')+1:]
            if(observations_agent_id not in observations_for_agent.keys()):
                observations_for_agent[observations_agent_id] = {}
            observations_for_agent[observations_agent_id][observations_feature] = val

        for agent in self.agent_list:
            # select the now_step
            # change phase for a certain period of time
            for k,v in observations_for_agent[agent].items():
                now_step = v[0]
                break
            step_diff = now_step - self.last_change_step[agent]
            if(step_diff >= self.green_sec):
                self.now_phase[agent] = self.now_phase[agent] % self.max_phase + 1
                self.last_change_step[agent] = now_step

            # construct actions
            actions[agent] = self.now_phase[agent]
        return actions
```

2.3 Evaluation

`evaluate.py` is a scoring program that output the scores of your agent. It is the same as the evaluate program on the server. So you'd like to check your agent's behaviour by execute

```
python evaluate.py --input_dir agent --output_dir out --sim_cfg cfg/simulator.cfg
```

Then result will be output at the `starter-kit/out/scores.json`

2.4 Results

Results will be saved as `starter-kit/out/scores.json`, the data format of results is exemplified as follows.

```
{
  "success": true,
  "error_msg": "", // if "success" is false, "error_msg" stores the exception
  "data": {
    "total_served_vehicles": 1047, // if "success" is false, here it returns -1
    "delay_index": 2.3582080966292374 // if "success" is false, here it returns -1
  }
}
```

2.5 Visualization

The CBEngine can log replay file. You can follow the following steps to visualize the intermediate results of your algorithm. Here *mapbox token* and *yarn* are required.

1. Put the `lightinfo.json`, `roadinfo.json`, `time*.json` from */log* to */ui/src/log*
2. modify */ui/src/index.js*

```
mapboxgl.accessToken = Your_Token;
this.maxTime = max_value_of_*_of_time*.json
```

3. cd to */ui*

```
yarn
yarn start
```

4. open *localhost:3000* with your browser

Here are some Tips:

- *Sky blue* indicates left-turning cars, *dark blue* indicates straight ahead cars, and *dark green* indicates right-turning cars.
- The color of signal is meaningless.
- Lines indicate roads. The color of the line represents the average speed of the road.

2.6 Make a submission

1. To submit the models for evaluation, participants need to modify the starter-kit and place all the model-related files (including but not limited to `agent.py` and deep learning model files) into the `agent` folder. Compress the agent folder and name it as `agent.zip` to make the submission. Note that you need to directly compress the `agent` folder, rather than a group of files.
2. Participants need to train their models offline and submit the trained models along with `agent.py`, which will load them.
3. All submissions should follow the format of our sample code in starter-kit . Hence, please do not modify any file outside the `agent` folder, except the `.cfg` file (The `.cfg` file can be revised to incorporate different training traffic).

4. If your model need to import or load some files, please put them to the `agent` folder and make sure to use the absolute path. Examples are shown at the beginning of fixed time `agent.py`.
5. Please also make sure to only use the packages in the given docker file, so that your code can be executed at the evaluation platform.
6. Participants can report the python package required to build the model if these packages are not included in the current docker environment. The support team will evaluate the request and determine whether to add the package to the provided docker environment.
7. Participants are responsible for ensuring that all the submissions can be successfully tested under the given evaluation framework.

CHAPTER

THREE

ENVIRONMENT - CBENGINE

CBEngine is a microscopic traffic simulation engine that can support city-scale road network traffic simulation. CBEngine can support fast simulation of road network traffic with thousands of intersections and hundreds of thousands of vehicles. CBEngine is developed by the team from Yunqi Academy of Engineering. This team will provide timely support for this competition. The safety distance car following and lane-changing models used in CBEngine are similar to SUMO (Simulation of Urban Mobility). The road network and traffic flow input data for CBEngine are compatible with the commonly used traffic simulators such as SUMO and VISSIM. The following sections describe the format of the road network and traffic flow input data. This description will help you to setup the engine with input data.

3.1 Data format

3.1.1 Roadnet File Format

Road network data

The road network file contains the following three datasets.

- **Intersection dataset** Intersection data consists of identification, location and traffic signal installation information about each intersection. A snippet of intersection dataset is shown below.

```
92344 // total number of intersections
30.2795476000 120.1653304000 25926073 1
30.2801771000 120.1664368000 25926074 0
...
```

  The attributes of intersection dataset are described in details as below.

Attribute Name	Example	Description
latitude	30.279547600	local latitude
longitude	120.1653304000	local longitude
inter_id	25926073	intersection ID
signalized	1	1 if traffic signal is installed, 0 otherwise

- **Road dataset** Road dataset consists information about road segments in the network. In general, there are two directions on each road segment (i.e., dir1 and dir2). A snippet of road dataset is shown as follows.

```
2105 // total number of road segments
28571560 4353988632 93.2000000000 20 3 3 1 2
1 0 0 0 1 0 0 1 1 // dir1_mov: permissible movements of direction 1
```

(continues on next page)

(continued from previous page)

```
1 0 0 0 1 0 0 1 1 // dir2_mov: permissible movements of direction 2
28571565 4886970741 170.2000000000 20 3 3 3 4
1 0 0 0 1 0 0 1 1
1 0 0 0 1 0 0 1 1
```

The attributes of road dataset are described in details as below

Attribute Name	Example	Description
from_inter_id	28571560	upstream intersection ID w.r.t. dir1
to_inter_id	4353988632	downstream intersection ID w.r.t. dir2
length (m)	93.2000000000	length of road segment
speed_limit (m/s)	20	speed limit of road segment
dir1_num_lane	3	number of lanes of direction 1
dir2_num_lane	3	number of lanes of direction 2
dir1_id	1	road segment (edge) ID of direction 1
dir2_id	2	road segment (edge) ID of direction 2
dir1_mov	1 0 0 0 1 0 0 1 1	Every three digits form a group to indicate whether left-turn, through and right-turn movements are permissible for a lane of direction 1. For example, '0 1 1' indicate a shared through and right-turn lane.
dir2_mov	1 0 0 0 1 0 0 1 1	Every three digits form a group to indicate whether left-turn, through and right-turn movements are permissible for a lane of direction 2. |

- **Traffic signal dataset** Note that, we assume that each intersection has no more than four exiting approaches. The exiting approaches 1 to 4 starting from the northern one and rotating in clockwise direction. Here, -1 indicates that the corresponding exiting approach is missing, which generally indicates a three-leg intersection.

```
107 // total number of traffic signals
1317137908 724 700 611 609 // inter_id, approach_id
672874599 311 2260 3830 -1
672879594 341 -1 2012 339
```

The attributes of road dataset is described in details as below

Attribute Name	Example	Description
inter_id	1317137908	intersection ID
approach0_id	724	road segment (edge) ID of northern approach
approach1_id	700	road segment (edge) ID of eastern approach
approach2_id	611	road segment (edge) ID of southern approach
approach3_id	609	road segment (edge) ID of southern approach

Example

Here is an example 1x1 roadnet `roadnet.txt` .

```
5
30 120 0 1
31 120 1 0
30 121 2 0
29 120 3 0
30 119 4 0
4
0 1 30 20 3 3 1 2
1 0 0 0 1 0 0 0 1
1 0 0 0 1 0 0 0 1
0 2 30 20 3 3 3 4
1 0 0 0 1 0 0 0 1
1 0 0 0 1 0 0 0 1
0 3 30 20 3 3 5 6
1 0 0 0 1 0 0 0 1
1 0 0 0 1 0 0 0 1
0 4 30 20 3 3 7 8
1 0 0 0 1 0 0 0 1
1 0 0 0 1 0 0 0 1
1
0 1 3 5 7
```

Here provides an Illustration of example above.

3.1.2 Flow File Format

Flow file is composed by flows. Each flow is represented as a tuple (*start_time*, *end_time*, *vehicle_interval*, *route*), which means from *start_time* to *end_time*, there will be a vehicle with *route* every *vehicle_interval* seconds. The format of flows contains serval parts:

- The first line of flow file is *n*, which means the number of flow.
- The following *3n* lines indicating configuration of each flow. Each flow have 3 configuration lines.
  - The first line of flow configuration indicating *start_time*, *end_time*, *vehicle_interval*.
  - The second line of flow configuration indicating the length of route of this flow : *k*.
  - The third line of flow configuration indicating the *route* of this flow. Here flow's route is defined by *roads* not *intersections*.

```
n
flow_1_start_time   flow_1_end_time flow_1_interval
k_1
flow_1_route_0      flow_1_route_1  ...     flow_1_route_k1

flow_2_start_time   flow_2_end_time flow_2_interval
k_2
flow_2_route_0      flow_2_route_1  ...     flow_2_route_k2

...

flow_n_start_time   flow_n_end_time flow_n_interval
```

(continues on next page)

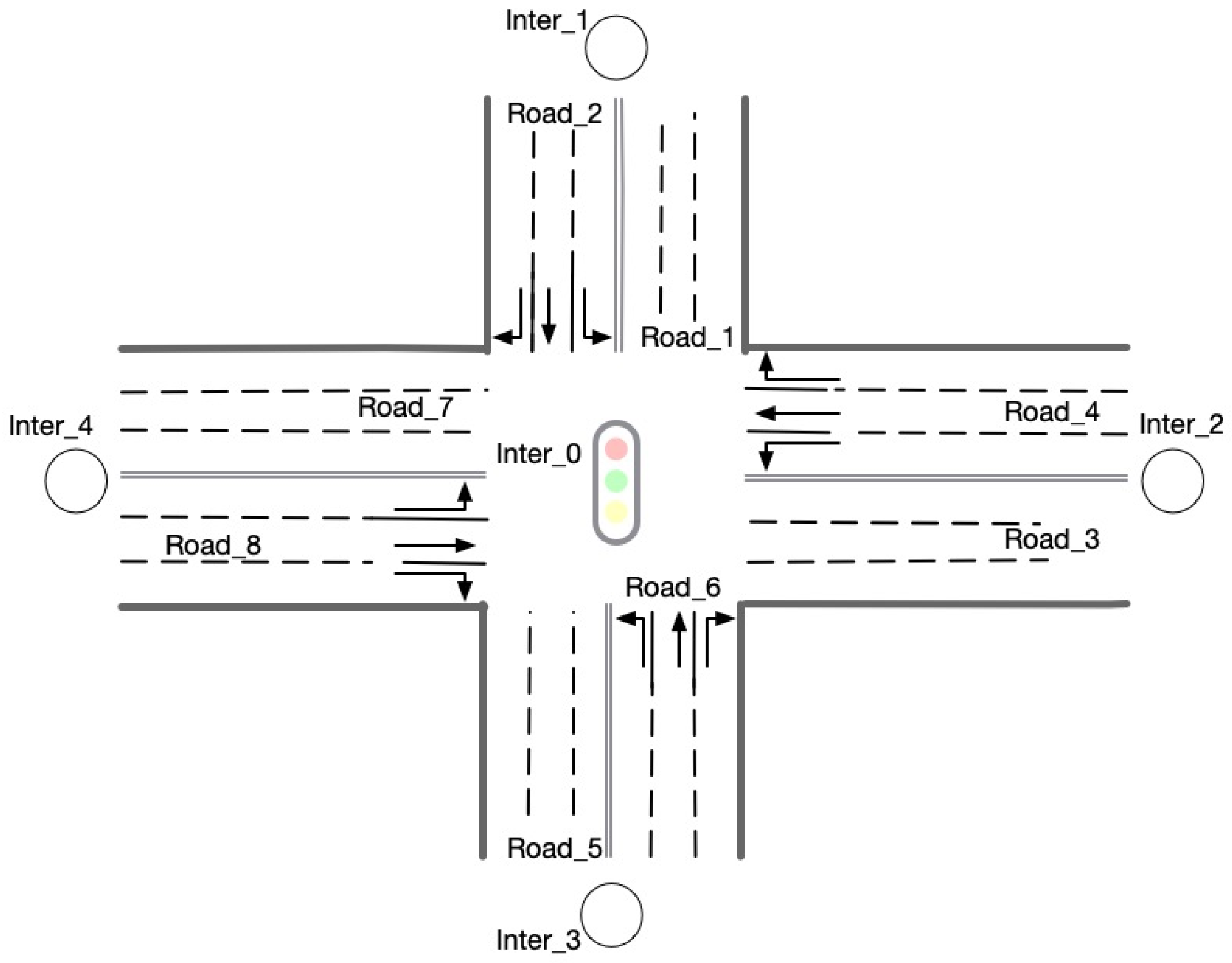


Fig. 1: Illustration of a 1x1 roadnet

(continued from previous page)

```
k_n
flow_n_route_0       flow_n_route_1  ...     flow_n_route_k
```

Here is an example flow file

```
12
0 100 5
2
2 3
0 100 5
2
2 5
0 100 5
2
2 7
0 100 5
2
4 5
0 100 5
2
4 7
0 100 5
2
4 1
0 100 5
2
6 7
0 100 5
2
6 1
0 100 5
2
6 3
0 100 5
2
8 1
0 100 5
2
8 3
0 100 5
2
8 5
```

3.2 Observations

Participants will be able to get a full observation of the traffic on the road network at every step, including vehicle-level information (e.g., position, speed) and lane-level information (e.g., average speed of each lane, number of vehicles on each lane). These observations will be helpful for decision-making on the traffic signal phase selection. Detailed description the features of *observation* can be found in `agent/gym_cfg.py`.

The format of observations could be found at annotation in code blocks in observation format.

3.3 Actions

For a traffic signal, there are at most 8 phases (1 - 8). Each phase allows a pair of non-conflict traffic movement to pass this intersection. Here are illustrations of the traffic movements and signal phase.

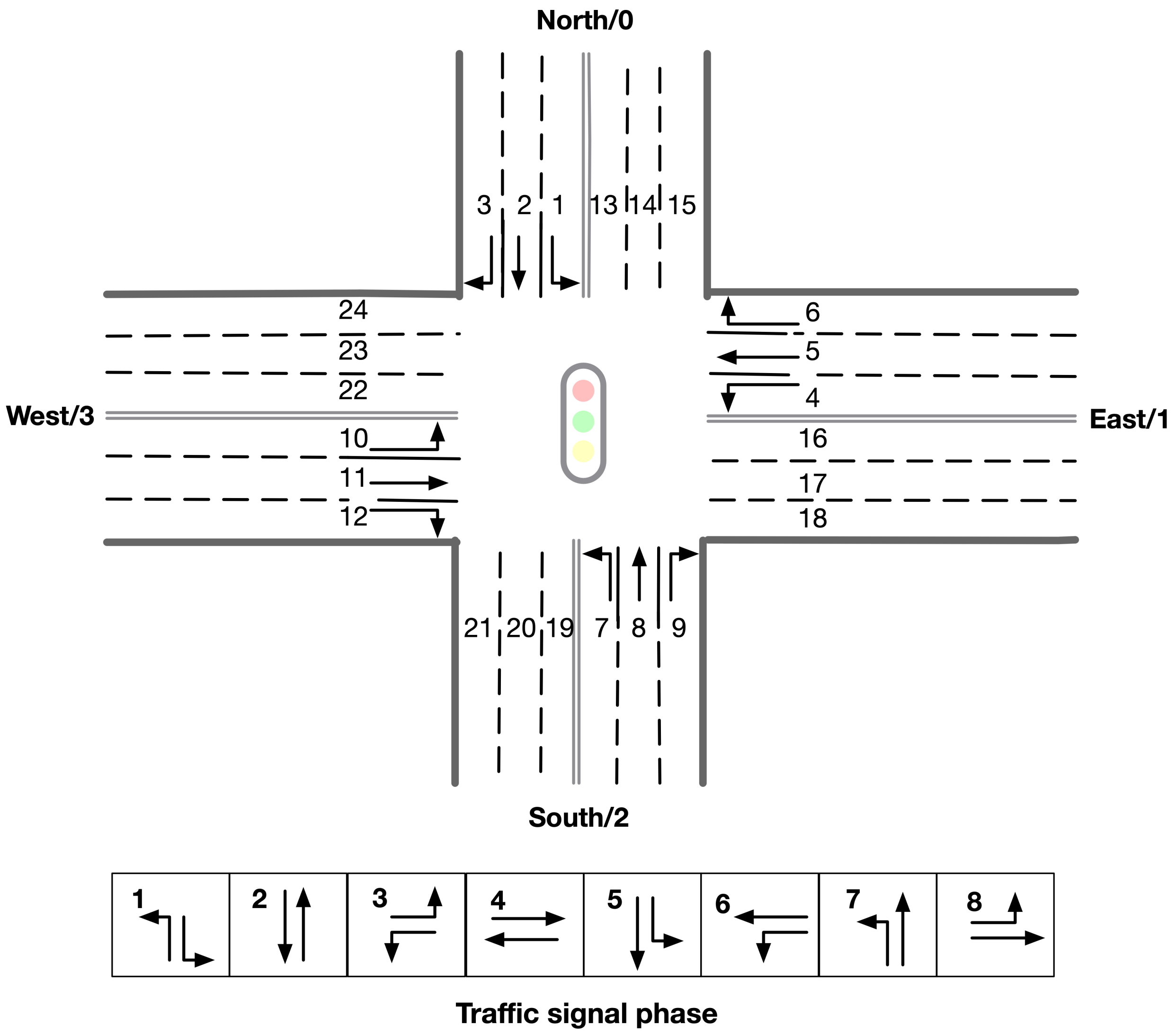


Fig. 2: Phase and lane ordering

For example, if an agent is at phase 1, *lane_1* and *lane_7* along with all right turning lanes are passable. The index of the lanes in *observation* and *reward* could be found in observation format.

There are a total of 8 different types of phases for a standard four-way intersection. To simplify, only the first 4 signal phases (1, 2, 3, 4) are open to participants at this stage. You can also learn how to set the traffic signals with the information given on the APIs page.

The action is defined as the traffic signal phase for each intersection to be selected at next step. If an agent is switched to a different phase, there will be a 5 step period of 'all red' at this agent, which means all vehicles could not pass this intersection. If continuously switched to different phase, agent would be always 'all red'. So we don't recommend you to switch agent to a different phase frequently.

CHAPTER

FOUR

API FUNCTIONS

Based on the CBEngine, we provide APIs that share the similar parameters as the OpenAI Gym environment.

4.1 Simulation Initialization

```
env = gym.make(
        'CBEngine-v0',
        simulator_cfg_file=simulator_cfg_file,
        thread_num=1,
        gym_dict=gym_cfg_instance.cfg
    )
```

***simulator_cfg_file*:**

- the path of simulator.cfg
- be used for initialize engine

Example

```
#configuration for simulator

# Time Parameters
start_time_epoch = 0
max_time_epoch = 3600

# Roadnet file and flow file used to simulate
road_file_addr : ./data/roadnet_warm_up.txt
vehicle_file_addr : ./data/flow_warm_up_1000.txt


# Log configuration
# Don't change the value of report_log_mode
report_log_mode : normal
# Log path
report_log_addr : ./log/
# Log interval
report_log_rate = 10
# Log configuration to track the vehicle. Don't change the value
warning_stop_time_log = 100
```

***thread_num*:**

- the thread number used for engine

gym_dict:

- the configuration used for initialize gym
- a dict
- The meaning of it is clarified at next section.
- stored in /agent/gym_cfg.py, as a member variable of class `gym_cfg`.

Example of `gym_dict`

```
gym_dict = {
    'observation_features':['lane_speed','lane_vehicle_num']
}
```

4.2 Environment Configuration: gym_cfg.py

`gym_cfg.py` in `agent` folder defines the configuration of gym environment. Currently it contains *observation features*. There are two options in *observation features*, namely *lane_speed* , *lane_vehicle_num*, which determines the content of observations you get from the `env.step()` api. You must write at least one of the two features.

```
class gym_cfg():
    def __init__(self):
        self.cfg = {
            'observation_features':['lane_speed','lane_vehicle_num']
        }
```

self.cfg:

- store the configuration of gym
- 'observation_features' indicates the return observation feature of the gym instance. Currently *lane_speed*, *lane_vehicle_num* are available

4.3 Simulation Step

`step(actions)`:

- Simulate one step
- The format of action is specified below.
- return observation, reward, info, dones
- The format of observations, rewards, infos and dones is specified below.

```
observation, reward, dones, info = env.step(action)
```

actions:

- Required to be a dict:

```
``{agent_id_1: phase_1, ... , agent_id_n: phase_n}``
```

- Set *agent_id* to some *phase* (The figure below demonstrates the allowed traffic movements in each phase)
- The phase is required to be an integer in the range [1, 8] (note there is no 0)

- The initial phases of all agents are set to 1
- The phase of an agent will remain the same as the last phase if not specified in the dict *actions*
- *Attention*: If an agent is switched to a different phase, there will be a 5 step period of 'all red' at this agent, which means all vehicles could not pass this intersection. If continuously switched to different phase, agent would be always 'all red'.

observations:

- a dict
- format: `{key_1: observations_values_1, key_2: observations_values_2}`
- The key is "{}_{}".format(agentid,feature) where feature is given by `gym_cfg.py`
- Format of observations_values:

```
# observation values:

# lane_speed sample: [13, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2,␣
↪-2, -2, -2, -2, -2, -2, -2, -2, -2, -2]
# The first value is current step
# There are 24 lanes left. The order of their roads is defined in 'signal' part␣
↪of roadnet file
# the order is :inroad0lane0, inroad0lane1, inroad0lane2, inroad1lane0 ...␣
↪inroad3lane2, outroad0lane0, outroad0lane1 ...
# Note that, [lane0, lane1, lane2] indicates the [left_turn lane, approach lane,␣
↪right_turn lane] repespectively of the corresponding road.
# The order of roads are determined clockwise.
# If there is a -1 in the signal part of roadnet file (which indicates this road␣
↪doesn't exist), then the returned observation of the corresponding lanes on␣
↪this road are also 3 -1s.
# -2 indicating there's no vehicle on this lane

# lane_vehcile_num sample [13, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0,]
# The first value is current step
# There are 24 lanes left. The order of their roads is defined in 'signal' part␣
↪of roadnet file
# the order is :inroad0lane0, inroad0lane1, inroad0lane2, inroad1lane0 ...␣
↪inroad3lane2, outroad0lane0, outroad0lane1 ...
# If there is -1 in signal part of roadnet file, then the lane of this road is␣
↪filled with three -1.

# Sample Output
{
"12530758427_lane_speed": [13, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2,
↪ -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2],
"12530758427_lane_vehicle_num" : [13, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
↪ 0, 0, 0, 0, 0, 0, 0, 0, 0,],
}
```

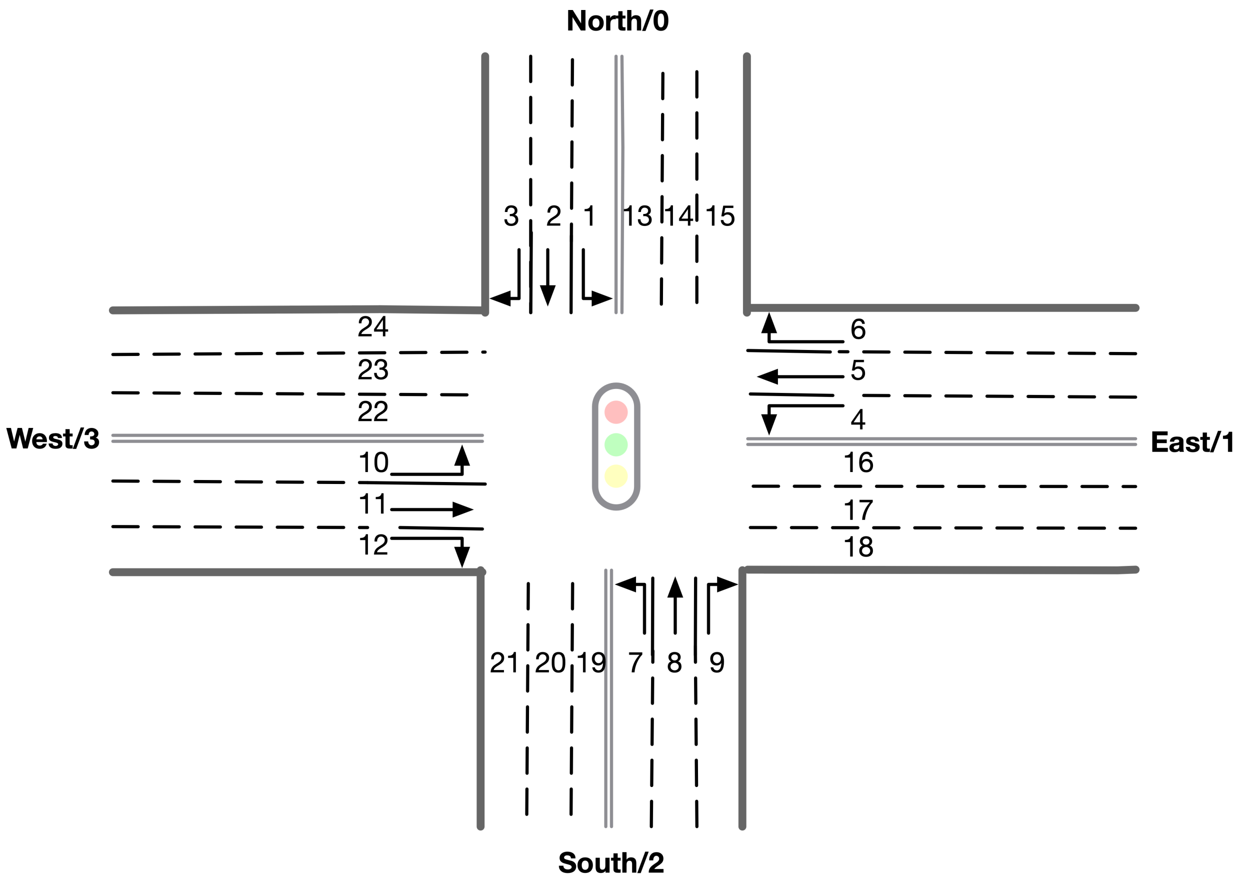
rewards:

- a dict
- {*agent_id_1*: *reward_values_1*, . . . , *agent_id_n*: *reward_values_n*}
- Format of reward_values:

```
# reward value:
# a list of tuples indicating (in_number of this step, out_number of this step)
# The length of the value list is 24. The order of their roads is defined in
↪'signal' part of roadnet file and the same as in observation.
# The order is :inroad0lane0, inroad0lane1, inroad0lane2, inroad1lane0 ...␣
↪inroad3lane2, outroad0lane0, outroad0lane1 ...
# If there is a -1 in the signal part of roadnet file (which indicates this road␣
↪doesn't exist), then the returned observation of the corresponding lanes on␣
↪this road are also 3 -1s.

# Sample Output
{
0: [(0,0),(0,1),(1,0),(0,0),(0,0),(0,1),(1,0),(0,0),(0,0),(0,1),(1,0),(0,0), (0,
↪0),(0,1),(1,0),(0,0),(0,0),(0,1),(1,0),(0,0),(0,0),(0,1),(1,0),(0,0)]
}
```

Here is an illustration of the lane index in *observation* and *reward*.



info:

- a dict
- {*vehicle_id_1*: *vehicle_info_1*, . . . , *vehicle_id_m*: *vehicle_info_m*}

```
"vehicle_info": {
0: {
    "distance": [259.0], # The distance from this vehicle to the start point of␣
↪current road.
    "drivable": [29301.0], # Current lane of this vehicle.
    "road": [293.0], # Current road of this vehicle.
    "route": [293.0, 195.0, 207.0, 5.0, 67.0, 70.0, 88.0, 92.0, 76.0, 18.0], #␣
↪Route of this vehicle (starting from current road).
    "speed": [0.0], # Current speed of this vehicle.
    "start_time": [73.0], # Time step of creation of this vehicle.
    "t_ff": [112.716] # Travel time of this vehicle assuming no traffic signal␣
↪and other vehicle exists.
    },
...
}
```

***dones*:**

- a dict
- {*agent_id_1*: *bool_value_1*, . . . , *agent_id_n*: *bool_value_n*}
- Indicating whether the simulation of an agent is ended.

4.4 Simulation Reset

`reset:`

- Reset the simulation
- return a tuple: (observation, info)
- reset the engine

```
observation , info = env.reset()
```

4.5 Other interface

We offer two extra interfaces:

`set_warning(flag):`

- set flag as False to turn off the warning of invalid phases. The warning will be issued if a green phase to an inexistent lane.

`set_log(flag):`

- set flag as False to turn off logs for a faster speed when training. Note that the score function will not work if the logging is turned off.