CityFlow Documentation

Release 0.1

Huichu Zhang

Dec 04, 2020

Contents:

CHAPTER 1

Introduction

CityFlow is a multi-agent reinforcement learning environment for large scale city traffic scenario.

Checkout these features!

- a microscopic traffic simulator which simulates the behavior of each vehicle, providing highest level detail of traffic evolution.
- support flexible definitions for road network and traffic flow
- provides friendly python interface for reinforcement learning
- **Fast!** Elaborately designed data structure and simulation algorithm with multithreading. Capable of simulating city-wide traffic. See the performance comparison with SUMO[2].

See *Quick Start* to get started.

[2] SUMO home page

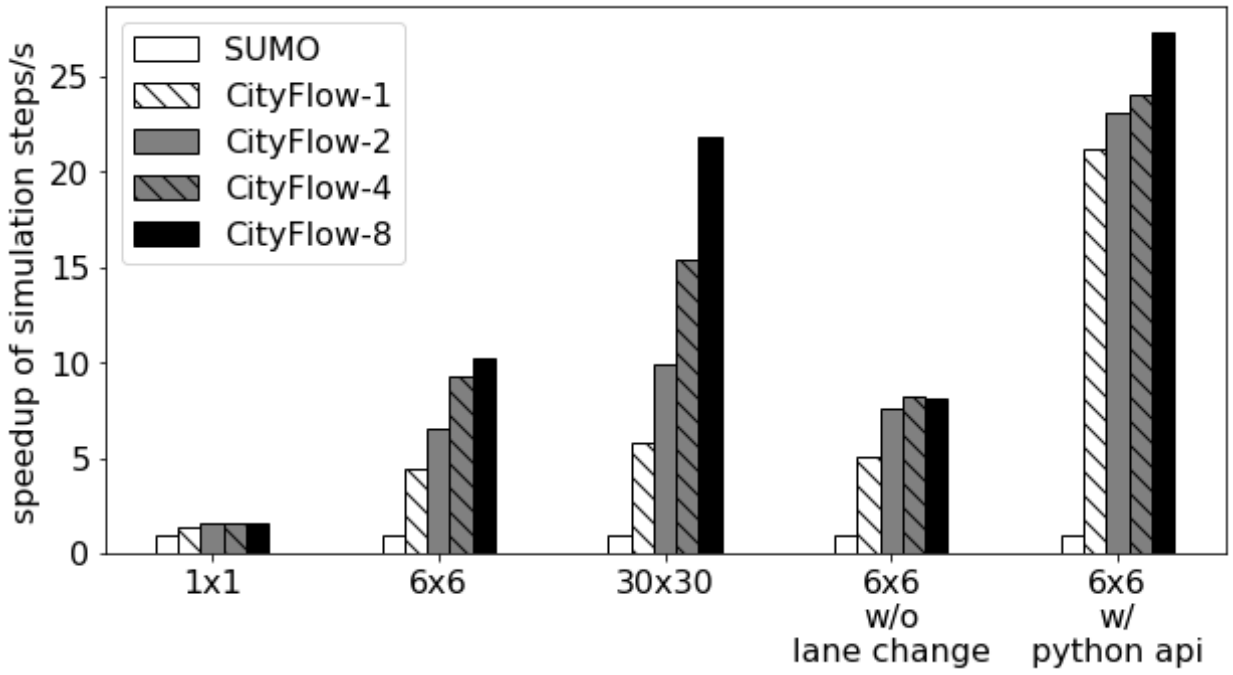

Fig. 1: Performance comparison between CityFlow with different number of threads (1, 2, 4, 8) and SUMO. From small 1x1 grid roadnet to city-level 30x30 roadnet. Even faster when you need to interact with the simulator through python API.

CHAPTER 2

Installation Guide

2.1 Docker

The easiest way to use CityFlow is via docker.

```
docker pull cityflowproject/cityflow:latest
```

This will create docker image `cityflow:latest`.

```
docker run -it cityflowproject/cityflow:latest
```

Create and start a container, CityFlow is out-of-the-box along with miniconda with python3.6.

```
import cityflow
eng = cityflow.Engine
```

2.2 Build From Source

If you want to get nightly version of CityFlow or running on native system, you can build CityFlow from source. Currently, we only support building on Unix systems. This guide is based on Ubuntu 16.04.

CityFlow has little dependencies, so building from source is not scary.

1. Check that you have python 3 installed. Other version of python might work, however, we only tested on python with version >= 3.5.
2. Install cpp dependencies

```
sudo apt update && sudo apt install -y build-essential cmake
```

3. Clone CityFlow project from github.

```
git clone https://github.com/cityflow-project/CityFlow.git
```

4. Go to CityFlow project's root directory and run

```
pip install .
```

5. Wait for installation to complete and CityFlow should be successfully installed.

```
import cityflow
eng = cityflow.Engine
```

2.3 For Windows Users

For Windows users, it is recommended to run CityFlow under Windows Subsystem for Linux (WSL) or use docker.

CHAPTER 3

Quick Start

3.1 Installation

If you have not installed CityFlow yet, *Installation Guide* is a simple guide for installation.

3.2 Create Engine

```
import cityflow
eng = cityflow.Engine(config_path, thread_num=1)
```

- `config_path`: path for config file.
- `thread_num`: number of threads.

3.2.1 Arguments In Config File

- `interval`: time of each simulation step (in seconds). An `interval` of 0.5 means for each simulation step, the system will move 0.5 seconds forward. For example, if a car is at location 0 with speed 10m/s, after one simulation step, it will move to location 10*0.5=5.
- `seed`: random seed.
- `dir`: root directory, all file path will be relative to this directory.
- `roadnetFile`: path for roadnet file.
- `flowFile`: path for flow file.
- `rlTrafficLight`: whether to enable traffic light control through python API. If set to `false`, default traffic light plan defined in `roadnetFile` will be used.
- `saveReplay`: whether to save simulation for replay. If set to `true`, `roadnetLogFile` and `replayLogFile` are required.

- `roadnetLogFile`: path for roadnet replay file. This is a special roadnet file for replay, not the same as `roadnetFile`.
- `replayLogFile`: path for replay. This file contains vehicle positions and traffic light situation of each simulation step.
- `laneChange`: whether to enable lane changing. The default value is 'false'.

For format of `roadnetFile` and `flowFile`, please see *Roadnet File Format*, *Flow File Format*

Note: Runnable sample roadnet and flow files can be found in `examples` folder.

You can generate grid roadnet and flow files using *tools/generate_grid_scenario.py*

For example, you can generate a 2x3 roadnet with predefined traffic light plan and a corresponding flow file with

```
python generate_grid_scenario.py 2 3 --roadnetFile roadnet.json --flowFile flow.json --dir . --tlPlan
```

3.2.2 Sample Config File

Note: Runnable sample config files can be found in `examples` folder.

```
{
    "interval": 1.0,
    "seed": 0,
    "dir": "data/",
    "roadnetFile": "roadnet/testcase_roadnet_3x3.json",
    "flowFile": "flow/testcase_flow_3x3.json",
    "rlTrafficLight": false,
    "saveReplay": true,
    "roadnetLogFile": "frontend/web/testcase_roadnet_3x3.json",
    "replayLogFile": "frontend/web/testcase_replay_3x3.txt"
}
```

3.3 Simulation

To simulate one step, simply call `eng.next_step()`

```
eng.next_step()
```

3.4 Data Access API

`get_vehicle_count()`:

- Get number of total running vehicles.
- Return an `int`

`get_vehicles(include_waiting=False)`:

- Get all vehicle ids
- Include vehicles in lane's waiting buffer if `include_waiting=True`
- Return an `list` of vehicle ids

`get_lane_vehicle_count()`:

- Get number of running vehicles on each lane.
- Return a `dict` with lane id as key and corresponding number as value.

`get_lane_waiting_vehicle_count()`:

- Get number of waiting vehicles on each lane. Currently, vehicles with speed less than 0.1m/s is considered as waiting.
- Return a `dict` with lane id as key and corresponding number as value.

`get_lane_vehicles()`:

- Get vehicle ids on each lane.
- Return a `dict` with lane id as key and list of vehicle id as value.

`get_vehicle_info(vehicle_id)`:

- Return a `dict` which contains information of the given vehicle.
- The items include:
  - `running`: whether the vehicle is running.
  - If the vehicle is running:
    - `speed`: The speed of the vehicle.
    - `distance`: The distance the vehicle has travelled on the current lane or lanelink.
    - `drivable`: The id of the current drivable(lane or lanelink)
    - `road`: The id of the current road if the vehicle is running on a lane.
    - `intersection`: The next intersection if the vehicle is running on a lane.
    - `route`: A string contains ids of following roads in the vehicle's route which are separated by `' '`.
- Note that all items are stored as `str`.

`get_vehicle_speed()`:

- Get speed of each vehicle
- Return a `dict` with vehicle id as key and corresponding speed as value.

`get_vehicle_distance()`:

- Get distance travelled on current lane of each vehicle.
- Return a `dict` with vehicle id as key and corresponding distance as value.

`get_leader(vehicle_id)`

- Return the id of the vehicle in front of `vehicle_id`.
- Return an empty string `""` when `vehicle_id` does not have a leader

`get_current_time()`:

- Get simulation time (in seconds)

- Return a `double`

`get_average_travel_time()`:

- Get average travel time (in seconds)
- Return a `double`

3.5 Control API

`set_tl_phase(intersection_id, phase_id)`:

- Set the phase of traffic light of `intersection_id` to `phase_id`. Only works when `rlTrafficLight` is set to `true`.
- The `intersection_id` should be defined in `roadnetFile`
- `phase_id` is the index of phase in array `"lightphases"`, defined in `roadnetFile`.

`set_vehicle_speed(vehicle_id, speed)`:

- Set the speed of `vehicle_id` to `speed`.
- The vehicles have to obey fundamental rules to avoid collisions so the real speed might be different from `speed`.

`reset(seed=False)`:

- Reset the simulation (clear all vehicles and set simulation time back to zero)
- Reset random seed if `seed` is set to `True`
- This does not clear old replays, instead, it appends new replays to `replayLogFile`.

`snapshot()`:

- Take a snapshot of current simulation state
- This will generate an `Archive` object which can be loaded later
- You can save an `Archive` object to a file using its `dump` method.

`load(archive)`:

- Load an `Archive` object and restore simulation state

`load_from_file(path)`

- Load a snapshot file created by `dump` method and restore simulation state.
- The whole process of saving and loading file is like:

```
archive = eng.snapshot() # create an archive object
archive.dump("save.json") # if you want to save the snapshot to a file

# do something

eng.load(archive)
# load 'archive' and the simulation will start from the status when 'archive'is␣
↪created

# or if you want to load from 'save.json'
eng.load_from_file("save.json")
```

`set_random_seed(seed)`:

- Set seed of random generator to `seed`

`set_vehicle_route(vehicle_id, route)`:

- To change the route of a vehicle during its travelling.
- *route* is a list of road ids (doesn't include the current road)
- Return true if the route is available and can be connected.

3.6 Other API

`set_replay_file(replay_file)`:

- `replay_file` should be a path related to `dir` in config file
- Set `replayLogFile` to `replay_file`, newly generated replays will be output into `replay_file`
- This is useful when you want to look at a specific episode for debugging purposes
- This API works only when `saveReplay` is `true` in config json

`set_save_replay(open)`:

- Open or close replay saving
- Set `open` to False to stop replay saving
- Set `open` to True to start replay saving
- This API works only when `saveReplay` is `true` in config json

CHAPTER 4

Roadnet File Format

Roadnet file defines the roadnet structure. CityFlow's roadnet mainly consists of intersections and roads (see them as nodes and edges of a graph).

- *Road* represents a directional road from one *intersection* to another *intersection* with road-specific properties. A *road* may contain multiple *lanes*.
- *Intersection* is where roads intersects. An *intersection* contains several *roadlinks*. Each *roadlink* connects two roads of the intersection and can be controlled by traffic signals.
- A *roadlink* may contain several *lanelinks*. Each lanelink represents a specific path from one lane of incoming road to one lane of outgoing road.

Now let's see a sample roadnet file and we'll explain the meaning of each components. Relax, the definition of field is quite straight forward, if you are familiar with modern road networks. For the following json file, `[]` means this field is an array, but we will only show one object for demonstration.

Note: Runnable sample roadnet files can be found in `examples` folder.

Sample `roadnet.json` with explanation.

```
{
  "intersections": [
    {
      // id of the intersection
      "id": "intersection_1_0",
      // coordinate of center of intersection
      "point": {
        "x": 0,
        "y": 0
      },
      // width of the intersection
      "width": 10,
      // roads connected to the intersection
      "roads": [
```

(continues on next page)

(continued from previous page)

```
        "road_1",
        "road_2"
      ],
      // roadLinks of the intersection
      "roadLinks": [
        {
          // 'turn_left', 'turn_right', 'go_straight'
          "type": "go_straight",
          // id of starting road
          "startRoad": "road_1",
          // id of ending road
          "endRoad": "road_2",
          // lanelinks of roadlink
          "laneLinks": [
            {
              // from startRoad's startLaneIndex lane to endRoad's endLaneIndex lane
              "startLaneIndex": 0,
              "endLaneIndex": 1,
              // points along the laneLink which describe the shape of laneLink
              "points": [
                {
                  "x": -10,
                  "y": 2
                },
                {
                  "x": 10,
                  "y": -2
                }
              ]
            }
          ]
        }
      ],
      // traffic light plan of the intersection
      "trafficLight": {
        "lightphases": [
          {
            // default duration of the phase
            "time": 30,
            // available roadLinks of current phase, index is the no. of roadlinks␣
↪defined above.
            "availableRoadLinks": [
              0,
              2
            ]
          }
        ]
      },
      // true if it's a peripheral intersection (if it only connects to one road)
      "virtual": false
    }
  ],
  "roads": [
    {
      // id of road
      "id": "road_1",
      // id of start intersection
```

(continues on next page)

(continued from previous page)

```
      "startIntersection": "intersection_1",
      // id of end intersection
      "endIntersection": "intersection_2",
      // points along the road which describe the shape of the road
      "points": [
        {
          "x": -200,
          "y": 0
        },
        {
          "x": 0,
          "y": 0
        }
      ],
      // property of each lane
      "lanes": [
        {
          "width": 4,
          "maxSpeed": 16.67
        }
      ]
    }
  ]
}
```

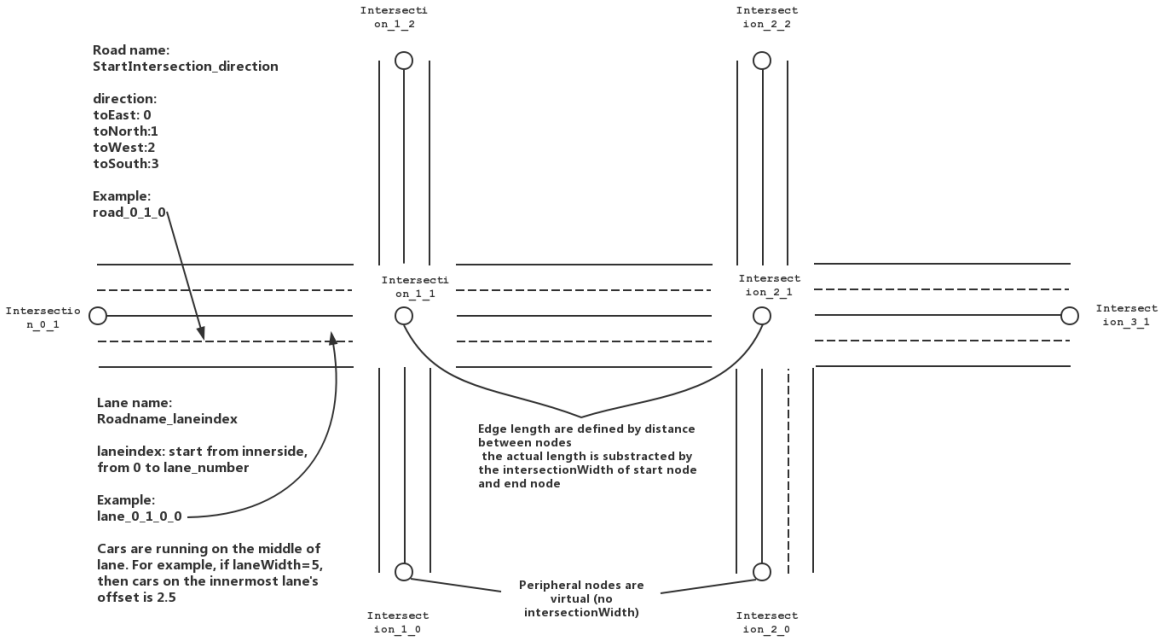

Fig. 1: Illustration of a 1x2 grid roadnet.

You can convert SUMO roadnet files into CityFlow format using tools/Converter/converter.py

For example, the following code converts a sumo roadnet file, atlanta.net.xml, to CityFlow format.

```
python converter.py --sumonet atlanta_sumo.net.xml --cityflownet atlanta_cityflow.json
```

CHAPTER 5

Flow File Format

Flow file defines the traffic flow. Each flow contains following field:

- **`vehicle`: defines the parameter of vehicle.**
  - length: length of the vehicle
  - width: width of the vehicle
  - maxPosAcc: maximum acceleration (in m/s)
  - maxNegAcc: maximum deceleration (in m/s)
  - usualPosAcc: usual acceleration (in m/s)
  - usualNegAcc: usual deceleration (in m/s)
  - minGap: minimum acceptable gap with leading vehicle (in meter)
  - maxSpeed: maximum cruising speed (in m/s)
  - headwayTime: desired headway time (in seconds) with leading vehicle, keep *current speed * headwayTime* gap.
- `route`: defines the route, all vehicles of this flow will follow the route. Specify the source and the destination, optionally some anchor points and the router will connect them with shortest paths automatically.
- `interval`: defines the interval of consecutive vehicles (in seconds). If the interval is too small, vehicles may not be able to enter the road due to blockage, it will be held and let go once there are enough space.
- `startTime`, `endTime`: Flow will generate vehicles between time [startTime, endTime] (in seconds), including `startTime` and `endTime`.

Note: Runnable sample flow files can be found in `examples` folder.

CHAPTER 6

Replay

6.1 Start

1. enter the `frontend` folder and open `index.html` in your browser.
2. choose the roadnet log file (as defined by `roadnetLogFile` field in the config file, **not 'roadnetFile'**) and wait for it to be loaded. When it has finished loading, there will be a message shown in the info box.
3. choose the replay file (as defined by `replayLogFile` field in the config file) .
4. choose the chart data file (optional, see section *Chart* below).
5. press `Start` button to start the replay.

6.2 Control

- Use the mouse to navigate. Dragging and mouse wheel zooming are supported.
- Move the slider in Control Box to adjust the replay speed. You can also press `1` on keyboard to slow down or `2` to speed up.
- Press `Pause` button in Control Box to pause/resume. You can also double-click on the map to pause and resume.
- Press `[` or `]` on keyboard to take a step backward or forward.
- To restart the replay, just press `Start` button again.
- The `debug` option enables displaying the ID of vehicles, roads and intersections during a mouse hover. **This will cause a slower replaying**, so we suggest using it only for debugging purposes.

6.3 Chart

The player supports showing the change of different metrics in a chart simultaneously with the replay process.

To provide required data, a log file in a format as shown below is needed:

The first line is the title of the chart.

Each row stands for a time step and each column stands for a specific metric. For example, to track vehicle numbers of three crossroads respectively, we need three columns and each column stands for the vehicle number of a certain crossroads.

In one row, numbers are separated by one or more spaces or tabs.

The numbers in one column will be shown as points connected by one line in the chart.

Note: Make sure that each row is corresponding with the right time step.

6.4 Notes

- To get the example replay files, run `download_replay.py` under `frontend` folder.
- If you create a new Engine object with same `replayLogFile`, it will clear the old replay file first
- Using `eng.reset()` won't clear old replays, it will append newly generated replay to the end of `replayLogFile`
- You can change `replayLogFile` during runtime using `set_replay_file`, see set-replay-file