Master TRIED 2022

Research Project

Reinforcement Learning for Smart Mobility

ALTUNAY Daniel
SZEBERIN Richard

ARALDO Andrea

2022

Contents

1 Introduction

With populations on the rise, mobility is one of the main focus areas for all smart city projects. Sustainability objectives such as emissions reduction and accessibility improvement are often transformed into combinatorial optimization problems that are widely researched by transportation scientists (Biazzo et al., 2019) [1].

Both combustion engine and electric cars have environmental footprint: in addition to air pollution and manufacturing, traffic heavily impacts the life quality of citizens. Also, traditional transportation networks are highly centralized, which means outlying neighborhoods (mostly with a large proportion of lower income citizens) do not benefit from the advantages of public transportation, while paying a very high cost by having to live in a polluted environment. Thus, one of the most important urban development goals is to create transportation networks that favor an increased minimum accessibility throughout neighborhoods and lower emissions through reducing the total number of vehicles needed. Of course, we need to rely heavily on public transportation to achieve this goal, while augmenting traditional network components (metro, bus, tram, etc...) with innovative, low-emission alternatives to offer scheduled and on-demand services to communities with low accessibility (i.e. an on-demand electric van transport to a poorly served part of the Paris metropolitan area), as discussed in Basu et al. (2019).[2].

As such alternative solutions are very appealing to transportation authorities, our research attempts to extend a framework to deal with similar combinatorial optimization problems through graph embedding and reinforcement learning.

2 Problem definition

Table 2.1: Table of notations

NOTATION	DESCRIPTION
$G(V, E, w)$	Graph of nodes V, edges E, weight function w
n_{edges}	Number of edges to add to the graph
S	Partial solution (sequence of nodes)
$C_G(u)$	Closeness centrality of node u in graph G
$C_S(u)$	Closeness centrality of node u in graph G, with path S added to the edges
E_S	Set of edges with path S added
$m(G)$	Global graph accessibility metric
$\mathcal{N}(u)$	Neighbors of node u
Θ	Parameters of Q function and graph embedding
$h(S)$	adjacency matrix of the graph, given partial solution S
Reinforcement learning	
π	Policy (here, picking the node that will maximize $\hat{Q}$)
$r(S, v)$	Reward of taking action v, given a partial solution S
γ	Discount factor (high means more importance given to future rewards)
T	Number of steps in a RL episode
L	Number of episodes in the RL learning phase
S_t	Partial solution at step t
v_t	Action (node) chosen at step t
$\hat{Q}$	Q-learning function estimator (estimates future rewards following the action policy), which depends on Θ
R_{t_1,t_2}	Cumulative reward between steps t_1 and t_2
$\mathcal{M}$	Memory of the agent (partial solutions, actions, rewards)
Graph embedding	
$\mu_u^{(t)}$	Embedding of node u after t steps of embedding algorithm, which depends on Θ

2.1 Practical problem

Given a transport network (such as the bus network) in a given city (such as Paris), our goal is to add a new bus line, in order to make it easier for people in badly served bus stops to commute and join central points of the city.

2.2 Theoretical problem

Given a graph $G(V, E, w)$ with V the set of nodes, E the set of edges, and $w\colon E \to \mathbb{R}^+$ the weight function, and given a number of new edges n_{edges}, we want to add a path[1] of size n_{edges} to E so that we improve the overall accessibility of the graph. We will now discuss how we defined our accessibility metric.

2.2.1 Node accessibility metric : the Closeness Centrality

Given a connected[2] graph $G(V, E, w)$, we define the **closeness centrality** metric :

$$\forall u \in V,\ C_G(u) = \frac{1}{\sum_{v \in V} \mathrm{d}(u, v|E)} \tag{2.1}$$

where $\mathrm{d}(x, y|E)$ is the distance between u and v using the shortest path in E, using the edge weights.
This is the measure of the *centrality* of a node : the more central the node is in the graph, the *closer* it is to the other nodes.

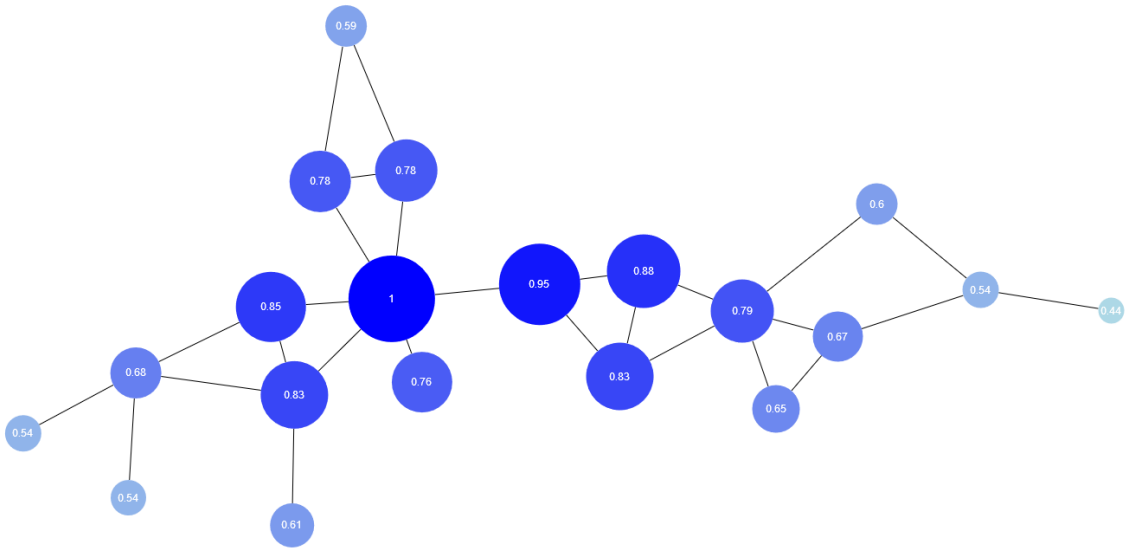


Figure 2.1: Random graph with closeness centrality values

[1] In graph theory, a path in a graph is a finite or infinite sequence of edges which joins a sequence of vertices which, by most definitions, are all distinct (and since the vertices are distinct, so are the edges). *Wikipedia*

[2] A graph is said to be connected if every pair of vertices in the graph is connected. This means that there is a path between every pair of vertices. *Wikipedia*

In Paris, *Châtelet Les-Halles* has a very high CC value, since it is well connected to almost everywhere else in Paris. It is served by many lines :

- RER : A, B, D
- Subway : 1, 4, 7, 11, 14
- Bus : 21, 38, 47, 58, 67, 69, 70, 72, 74, 75, 76, 85, 96

2.2.2 Graph accessibility metric

We have defined the *closeness centrality* metric for a node in a graph. Now, we want to define a metric which will apply to a **whole graph**, so that we can evaluate its global efficiency in terms of accessibility.

Different ideas came to us. The first one was to sum all of the CC (closeness centrality) measures of all the nodes in the graph : $\sum_{u \in V} C_G(u)$.
The problem is, the same importance (or weight) is given to all of the CCs, while we prefer to avoid the smaller ones. Our fear was that it would favor making a very central node, while ignoring the outer ones.

Our second idea was to add an exponent to the CC value, so that in order to have a high overall value, we need to avoid small singular values : $\sum_{u \in V} C_G(u)^\gamma$, with $\gamma < 1$. We would have kept $\sum_{u \in V} \sqrt{C_G(u)}$, for example, as an arbitrary choice.

After discussing with our mentor, we finally decided to put the importance on the least accessible node, so that we avoid having one lone node in the graph.

Thus, here is the final global graph accessibility we decided to keep :

$$m(G) = \min_{u \in V} C_G(u) \tag{2.2}$$

This way, the algorithm will not even bother looking at already highly connected nodes, since there will be no point (in terms of global accessibility maximizing) in improving their closeness centrality.

2.3 Real-life applications

Once we have built an algorithm that can find the optimal solution for a given graph and a given number of new edges to add, we can apply it to a city. Since this is hardly possible to link two unlinked subway or RER stations, the main focus will be on bus lines.

We would have to represent the city bus infrastructure with a graph (`Python` library `OSMnx` lets us do this). Then, we would also need to find a € cost estimation of adding a bus line, depending on the number of served bus stops (knowing that we do not need to create any additional stops). With this cost, we can then choose a value of n_{edges}, and run our algorithm on the graph. In the end, we could have a sequence of bus stops that would represent the best bus line to add in order to improve the least accessible areas, where there is already a bus stop. This could be proposed to RATP (*Régie Autonome des Transports Parisiens*) as a possible concrete project to work on, or any big city's public transport authority.

3 First approaches with simple heuristics

Using Python library `networkx`, we generated a random graph G with its corresponding set of n nodes V and set of edges E, using a customized graph based on the Barabási–Albert model. Then, we chose a number n_{edges} of edges to add to the graph, using several strategies : **randomness**, **naivety**, and **greediness**.

For these algorithms, we used several parameters to characterize our problem. First, we assumed the nodes (bus stops) cannot be added or removed. Also, we added constraints to our model in order to model the characteristics of real world problems:

- `n_nodes_total`: number of nodes in the graph (bus stops in the city)
- `n_edges`: number of edges in the graph (number of connections between the stops)
- `x_size`: x dimension of the map on which the nodes are located
- `y_size`: y dimension of the map on which the nodes are located
- `n_stops_line`: the maximum number of nodes (bus stops on the line)
- `km_budget`: the maximum distance covered by the added path (a kilometer budget for the new line)
- `n_repeat`: the maximum number of times a specific node can be added to the path
- `n_parallel_edges`: the maximum number of edges to connect the same two nodes

While we considered several different parameter settings during the development of our model, we will use a specific combination of parameters to compare the results of the algorithms:

Using some of the above parameters, we generated our initial graph shown in Figure 3. In this case, we observe a minimum closeness centrality value of 0.21 (node 7).

n_stops_total	10	n_stops_line	3
n_edges	9	km_budget	1.5
x_size	1	n_repeat	1
y_size	1	n_parallel_edges	1

Table 3.1: Parameters

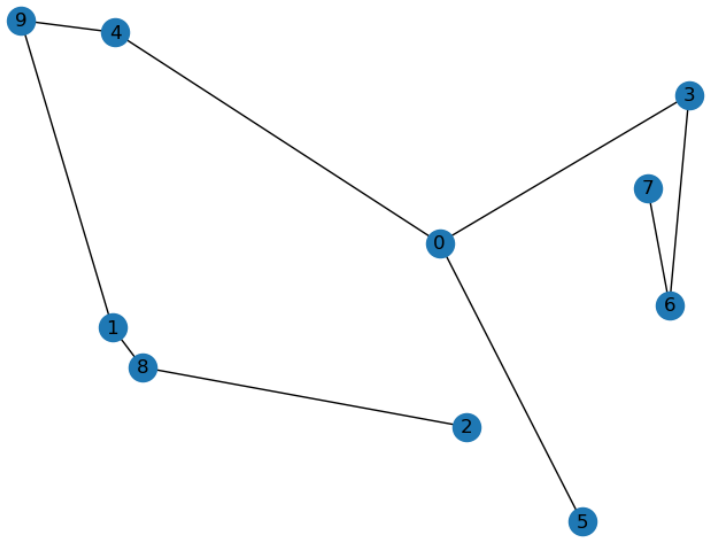


Figure 3.1: Initial graph

3.1 Random algorithm

First, we implemented an algorithm that chooses randomly from all available nodes. We describe a node as available if it has at least one non-neighboring node and it has not been visited by the algorithm before.

3.2 Naive algorithm (exhaustive search)

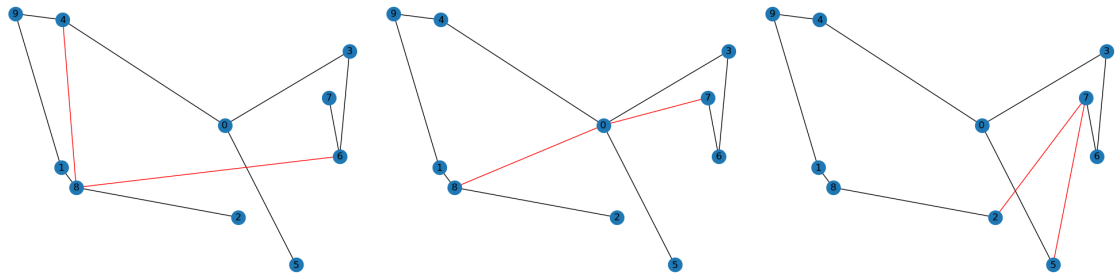
In order to find the best result by iterating through all possible bus lines, we performed an exhaustive search by calculating scores for all paths available, based on the same availability criteria. The naive method finds the optimal solution with our parameter setup.

3.3 Greedy algorithm

In the third alternative, we use a 0-step greedy approach. Thus, the algorithm always looks for the node with the lowest closeness centrality value when adding a new edge. While greedy methods are not really efficient in finding the optimal solution, they provide us with a computationally inexpensive alternative.

3.4 Results comparison

The three algorithms found different solutions to the problem, as visually represented in Figure 3.4. Furthermore, we observed that while our greedy and random algorithm runs with a similar time complexity, finding the optimal solution through exhaustive search approach takes much longer.



(a) Random **(b)** Naive **(c)** Greedy

Figure 3.2: Solutions found by the algorithms

Min CC	Random	Naive	Greedy
stop 1	0.21	-	0.21
stop 2	0.32	-	0.29
stop 3	0.36	0.41	0.34
execution time	0.74s	63.98s	0.58s

Table 3.2: Results and execution times

As we observed that some of the methods are fast but inaccurate, and exhaustive search is very expensive, our objective is to develop an algorithm which combines randomness to reduce complexity while finding a nearly optimal solution. In the next chapters, we will explore the application of reinforcement learning to similar problems.

4 Reinforcement learning on transportation graphs

As our baseline for our research project, we used this paper[3] from Hanjun Dai et al. (2018). It managed to implement a Python solution to solve three well known problems, using a graph embedding method (setting optimal parameters using a deep learning module called `structure2vec` from Torch) and Reinforcement Learning.

Since the mentioned problems are similar to ours, we decided to use their code and adapt it. First, let us explain what they learned to solve in the paper.

4.1 Paper problems : MVC, MAXCUT, TSP

The three problems the authors of the paper worked on are the Minimum Vertex Cover (MVC), the Max-Cut (MAXCUT), and the Traveling Salesman Problem (TSP). They all consist in finding an optimal sequence of nodes in a given graph, in order to maximize a certain metric (an edge cover ratio, a sum of distance, etc.)

These are well known among the graph optimization problems.

4.1.1 Minimum Vertex Cover

Given a graph G, find a subset of nodes $S \subseteq V$ such th at every edge is covered, i.e. $(u, v) \in E \Leftrightarrow u \in S$ or $v \in S$, and $|S|$ is minimized.

4.1.2 Max-Cut

Given a graph G, find a subset of nodes $S \in V$ such that the weight of the cut-set $\sum_{(u,v)\in C} w(u, v)$ is maximized, where cut-set $C \subseteq E$ is the set of edges with one end in S and the other end in $V \setminus S$.

4.1.3 Traveling Salesman Problem

Given a set of points in 2-dimensional space, find a tour of minimum total weight, where the corresponding graph G has the points as nodes and is fully connected with edge weights corresponding

to distances between points; a tour is a cycle that visits each node of the graph exactly once.

4.2 Paper problem resolution method

4.2.1 Graph embedding : structure2vec

The authors decided to use a graph embedding deep learning structure. Its purpose is to *featurize* the nodes in the graph, with its properties in regards to its neighborhood. This help the algorithm understand the usefulness of each node. To understand better how it is built, see 4.1 definition of the node embedding.

The embedding depend on parameters Θ, which are learned by *Stochastic Gradient Descent*, trying to minimize the loss (see (4.2)).

4.2.2 Reinforcement learning : Q-learning

The idea behind using **reinforcement learning** is to train our algorithm on several graphs, so that it is able to find the optimal solution for a new unseen graph. Thus, we first generate a graph distribution $\mathbb{D}$, $\mathscr{D} = \{G_i\}_{1 \geq i \geq m}$, with potential different sizes. This will make the model able to find a solution even in a city with a different size.

The authors defined an exploration probability ε, which determines whether the agent chooses a random new node or the one it thinks fits the best. While we let it in the following pseudo code, we made $\varepsilon = 1$ in our case.

Algorithm 1 Q-learning pseudo code

Initialize experience replay memory $\mathscr{M}$ to capacity N
for episode $e = 1$ **to** L **do**
    Draw graph G from distribution $\mathbb{D}$
    Initialize the state to empty $S_1 = ()$
    for step $t = 1$ **to** T **do**
        $v_t = \begin{cases} \text{random node } v \in \overline{S_t}, & \text{with probability} \varepsilon \in]0,1[\\ \arg\max_{v \in \overline{S_t}} \hat{Q}(h(S_t), v|\Theta), & \text{otherwise (following policy } \pi) \end{cases}$
        Add v_t to partial solution: $S_{t+1} := S_t \cup \{v_t\}$
        if $t \geq n$ **then**
            Add tuple $(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t)$ to $\mathscr{M}$
            Sample random batch from $B \overset{iid.}{\sim} M$
            Update Θ by *SGD* over (4.2) for B
return Θ

This is a combination of *fitted Q-iteration* and *n-step Q-learning*, which have both been explained in more details in [3].

An episode corresponds to a complete sequence of node additions, starting from an empty solution $()$, until the termination criterion is fulfilled. Each step in an episode corresponds to adding a new node v_t to S_t.

4.3 Our problem : *MMCC* (Max Min Closeness Centrality)

While our problem is similar to the *TSP*, we need to change several parameters and metrics in order to adapt the code. We decided to name our problem *Max Min Closeness Centrality* (MMCC), since our goal is to maximize the minimum of the closeness centrality of the nodes in the graph.

4.3.1 Basic parameters

For all of these, we will have the same definition :

- $G = (V, E, w)$: given graph
- V : set of nodes in G
- E : set of edges in G
- w: $E \to \mathbb{R}^+$ the weight function (always 1 in our case, since we consider the ride duration between two bus stops is generally equal in Paris)
- $n = |V|$: number of nodes in graph G

We saw that all three problems require a solution being a sequence of nodes. Thus, we need to define and work with a partial solution being a sequence of nodes and not a set of edges.
The bus line is eventually represented by $S = (v_1, v_2, \ldots, v_{|S|}), v_i \in V$.

4.3.2 Metric, reward function

Here is the exact definition of the **closeness centrality** metric, given the partial solution S :

$$\forall x \in V,\ C_S(x) = \frac{1}{\sum\limits_{y \in V} \mathrm{d}(x, y|E_S)}$$

where $\mathrm{d}(x, y|E_S)$ is the distance between x and y via the shortest path using edges in E_S, with E_S the set of new edges, given the partial solution S, given by :

$$E_S = E \cup \left(\bigcup_{i=1}^{|S|-1} \{(v_i, v_{i+1})\} \right)$$

Since our global metric is different than theirs (see (2.2)), we define our reward r when taking an action v, i.e., adding node v to the current partial solution S, thus adding edge $(v_{|S|}, v)$ to the graph. The reward is defined as follows:

$$\begin{aligned} r\colon V^{|S|} \times V &\to \mathbb{R}^+ \\ (S, v) &\longmapsto \min_{u \in V} C_{S \cup \{v\}}(u) - \min_{u \in V} C_S(u) \end{aligned}$$

Intuitively, this gives us the *gained accessibility* by taking the given action (picking v), while being in the given state (represented by S).

4.3.3 Graph embedding

For the node embedding, we will keep the same structure : $\mu_u^{(T)}$ is embedding of the node u at the T^{th} graph embedding (`structure2vec`) iteration (paramaterized by Θ), given by :

$$\begin{aligned}\mu_u^{(0)} &= 0 \text{ (initialization)}\\ \mu_u^{(t+1)} &\leftarrow F\left(x_v, \{\mu_u^{(t)}\}_{u\in\mathcal{N}(v)}, \{w(u,v)\}_{u\in\mathcal{N}(v)}|\Theta\right)\\ &= \text{relu}\left(\theta_1 x_v + \theta_2 \sum_{u\in\mathcal{N}(v)} \mu_u^{(t)} + \theta_3 \sum_{u\in\mathcal{N}(v)} \text{relu}(\theta_4 w(u,v))\right)\end{aligned} \tag{4.1}$$

where $\mathcal{N}(v)$ is the set of neighbors of v in graph G, and F is a generic nonlinear mapping such as a neural network or kernel function.

The parameters Θ are updated at each step of an episode t by performing a gradient step, to minimize the squared loss :

$$(y - \hat{Q}(h(S_t), v_t|\Theta))^2 \tag{4.2}$$

where $y = \gamma \max_{v'} \hat{Q}(h(S_{t+1}), v'|\Theta) + r(S_t, v_t)$ for a non-terminal state S_t.

4.3.4 Q-learning

In Q-learning, the Q function determines the potential gain (long term reward), given a current state (here S), an action (here v), and a policy (that we could write π).

$$Q(S,v) = \mathbb{E}[r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots] = \mathbb{E}[r_0] + \gamma Q(S \cup \{v\}) \tag{4.3}$$

where γ is a number between 0 and 1, r_0 the reward of taking action v, and r_i the reward for taking i^{th} best action after v, following π policy.

In Q-learning, we try to find an estimator $\hat{Q}$ of the Q function, such as $\hat{Q}(S,v) \approx Q(S,v)$.

We have $\hat{Q}(S,v) \approx r_0 + \gamma \hat{Q}(S \cup \{v\}, v')$ (where v' is the next best action/node to choose, based on π). This explains (4.2) : our goal is to find the best parameters Θ so that we minimize $\hat{Q}(S,v) - \left(r_0 + \gamma \hat{Q}(S \cup \{v\}, v')\right)$, and thus the squared loss in the previous equation. It is important to note that we have a determinist transition : once a node v is chosen, $S \leftarrow S \cup v$.

For the Q-learning, we will keep the same Q function estimator :

$$\begin{aligned}\hat{Q} &: \mathcal{M}_n(\{0,1\}) \times \overline{S} \to \mathbb{R}^+\\ &(h(S), v) \longmapsto \hat{Q}(h(S), v|\Theta) = \theta_5^T \text{relu}([\theta_6 \sum_{u\in V} \mu_u^{(T)}, \theta_7 \mu_v^{(T)}])\end{aligned} \tag{4.4}$$

where $\{\theta_i\}_{1\le i\le 7} = \Theta$ are model parameters learned using `structure2vec` (the deep learning architecture from `Torch`), and $[\cdot,\cdot]$ is the concatenator operator.

The policy is defined as follows : $\pi(v|S) := \arg\max_{v'\in\bar{S}} \hat{Q}(h(S), v')$

As for the adjacency matrix $h(S)$, we define it this way :

$$h(S) = (a_{i,j})_{(i,j)\in[\![1,n]\!]^2} \Big/ a_{i,j} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E \\ 0, & \text{otherwise} \end{cases}$$

If $i = j$, then v_i and v_j are not connected together (a node cannot be connected to itself).

5 Working on the algorithm and the code

We found two implementations of the paper algorithm on Github. The first one[1] was written in `C++`, and was hard to make work : it required `Docker`, `cuda`, a Linux environment (that we managed to use via `WSL 2`). Though, we realized it was hard to modify in order to adapt it to our problem, since we have a poor knowledge of `C++`.

We found another implementation[2] in `Python`, which was easier to understand and to run. We exclusively worked with it for the rest of our project.

5.1 Structure

Here is the structure of the code :

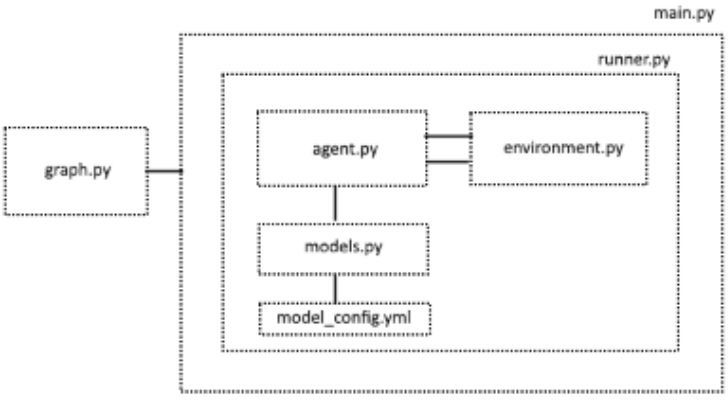


Figure 5.1: Code structure of the Python implementation

Basically, we run a **main.py** file, which creates many random graphs (following a certain distribution) using **graph.py**. Then, it runs the **runner.py** file, which instantiates :

[1] Official implementation (C++) : `https://github.com/Hanjun-Dai/graph_comb_opt`
[2] Unofficial implementation (Python) : `https://github.com/DavidLee-4032/Graph_Opt`

- an environment for the reinforcement learning, depending on the problem to solve (MVC, MAXCUT), using **environment.py**. We added the new environment *MMCC* ourselves.
- an agent, which will operate inside the environment and modify it, and define required objects and methods for the Q-learning, using **agent.py**
- a deep learning model via `Torch`'s `structure2vec` which will update parameters Θ (4.1 and 4.4) for the most efficient graph embedding, using **models.py** and **model_config.yml**

The Q-learning function is defined in **model.py**.

A more in-depth description can be found on David Lee's Github repository (Python implementation).

5.2 Adapting the code for *MMCC*

5.2.1 Arguments

We needed to add a nw parameter representing the constraint of the maximum budget we would have. Ineed, we do not want out algorithm to create edges between all of the nodes, even though it would make the best graph accessibility-wise : creating each edge has a real-life cost.

Because we have a new parameter n_{edges}, we had to add a new argument to parse in the **main.py** file. As a reminder, this parameter defines the exact number of edges to add to our graph to create the path that will maximize the minimum closeness centrality of the graph nodes.

Also, we added a new modality for the *environment_name* argument, which is *MMCC*. The only two options were *MVC* and *MAXCUT*.

5.2.2 Environment

Since we are adding a new problem (*MMCC*), we needed to update the environment. The candidate nodes for an action at a given step are defined in **environment.py**. Inside, we called the memory of the agent to access the partial solution (or bus line) S, and access the last node added. This was a hard task, since the agent memory is not easily accessible from the environment.

Indeed, the environment belongs to the agent, but no agent belongs to the environment, in terms of class hierarchy. The reason is that for *MVC* and *MAXCUT*, the graph is not modified during the algorithm, and the order of the partial solution is not important (they used sets of nodes and not sequences). We had to work it out with all of the different classes, creating new attributes.

Once we have the potential candidate nodes, we added the reward calculation, which is different than for *MVC* and *MAXCUT* (see 4.3.2 for the reward). Only nodes which were not already in S were considered as potential candidates.

We created a new graph corresponding to the graph with the edge between the last and potential node added, computed the minimum closeness centrality on it, and calculated the delta value (see (2.2)) with the actual graph (before the agent action).

```
elif self.name=="MMCC" :
    done=False
    last_node = self.agent.memory[-1][1] #last bust stop (node number, integer)
    graph_temp = self.graph_init.g.copy()
    graph_temp.add_edge(last_node, node)
    cc = nx.closeness_centrality(graph_temp)
    reward = cc[min(cc, key=cc.get)]
    change_reward = reward-self.last_reward
    if change_reward<=0:
        done=True
    self.last_reward = reward
    return (change_reward,done)
```

Figure 5.2: Reward calculation (code written by us)

5.2.3 Agent

The *memory* is stored in an agent attribute `agent.memory`. It consists in a tuple of size 4 :

1. `permu` : used by `structure2vec` for the graph embedding, not to be modified
2. `action` : the chosen node v_t to add to S_t
3. `reward` : the reward corresponding to the action
4. `aux` : a metric calculated on the graph, only used for *MVC* or *MAXCUT* (we discarded it)

```
def remember(self, permu, action, reward, aux):
    self.memory.append((permu, action, reward, aux))
```

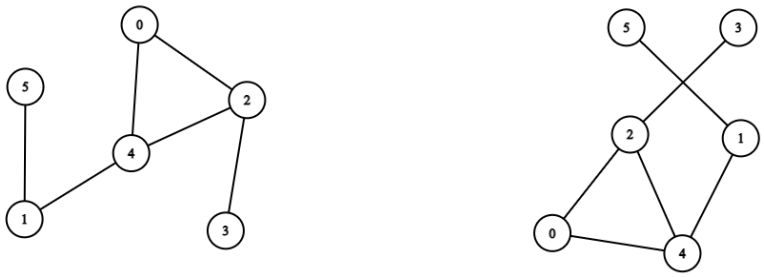
Figure 5.3: *remember()* method from class Agent

Each time an action is taken (with the method `Agent.act()` inside the environment, the `remember` method is called, and the agent `memory` is updated.

5.2.4 Graphs

When working with the library `networkx` to understand how graphs work in `Python`, we realized that nodes did not have a specific location.

For example, both these graphs are equal, in regards to `networkx` :



(a) Graph shape 1 **(b)** Graph shape 2

Figure 5.4: Two visual shapes for the same actual graph

However, we wanted to lock the coordinates of the nodes in the graph, so that it was easier to visualize and understand changes. Also, it made no sense to change the shape of what is supposed to represent a city's transport infrastructure, which has obviously a permanent shape.

We used a `networkx` function for this. However, we believe it might be a lot harder with the graph corresponding to a whole city : we would need file with all of the coordinates of all of the bus stops. Since we are only working with small graphs for now, we do not address this problem.

5.2.5 Termination criterion

We have our own termination criterion, that we define using our new parameter. A given game in an environment stops when the following condition is verified :

```
len(S)-1 == n_edges
```

with S the partial solution (sequence of nodes), and n_{edges} the number of edges to add (given as an input to the algorithm).

In `Runner.loop()`, this criterion appears :

```
if len(self.agent.memory) - 1 >= self.agent.env.n_edges_max :
    done=True
```

Figure 5.5: Termination criterion (code written by us)

A `break` statement is then used to get out of the current episode loop.

Bibliography

[1] Indaco Biazzo, Bernardo Monechi, and Vittorio Loreto. General scores for accessibility and inequality measures in urban areas. *Royal Society Open Science*, 2019. URL: https://www.researchgate.net/publication/335454832_General_scores_for_accessibility_and_inequality_measures_in_urban_areas.

[2] Rounaq Basu, Andrea Araldo, Arun Akkinepally, Bat-hen Nahmias Biran, Kakali Basak, Ravi Seshadri, Neeraj Deshmukh, Nishant Kumar, Carlos Lima Azevedo, and Moshe Ben-Akiva. Automated mobility-on-demand vs. mass transit: A multi-modal activity-driven agent-based simulation approach. *Transportation Research Record Journal of the Transportation Research Board*, 2018. URL: https://www.researchgate.net/publication/335454832_General_scores_for_accessibility_and_inequality_measures_in_urban_areas.

[3] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *College of Computing, Georgia Institute of Technology*, 2018. URL: https://arxiv.org/pdf/1704.01665.pdf.