ÉCOLE NATIONALE DES PONTS ET CHAUSSÉES

PROJET TDLOG 2021

OPTIMISATION DE TRAJET DANS UN SUPERMARCHE

Marwan Akrouch, Ilyes Belaouni, Abdennour El Harrak, Shuto Yotsumoto, Anatole Blanc

ABSTRACT :

Our software is designed to solve a problem of route optimization by passing through target points with the presence of obstacles, we took the case of a client who wants to get all the products on his list and leave in a optimal route.

The supermarket offers its client the possibility to choose the products he needs through a web interface and an optimal path will be calculated for him. This problem is a variant of the travelling salesman problem, we solved it in two different ways, we used operational research tools through Google OR libraries in python for the first solver, and Q-learning algorithm for the second. Since our algorithms are implemented in python we used django framework to build the web interface for the user.

KEYWORDS : Maze, travelling salesman, shortest path, python, django, javascript, Q-learning

Table des matières

1 Introduction et objectifs

C'en est fini des interminables errances dans les dédales des rayons des grandes surfaces à la recherche des produits désirés. Notre Logiciel permet aux consommateurs de préparer leurs courses à l'avance, il calcule un parcours optimisé selon leur liste de produits à acheter. Une fois en magasin, le client n'a plus qu'à suivre l'itinéraire qu'il aura imprimé préalablement. Dans la plupart des cas, le clients ne disposent que d'un temps limité pour faire leurs courses. Notre ambition est de réduire au maximum les courses "contraintes" pour laisser plus de place au shopping "plaisir".

Notre logiciel assez ambitieux est déstiné à résoudre un problème d'optimisation de trajet en passant par des points cibles avec présence d'obstacles. Ce type de problème, avec la modélisation qu'on considèrera, peut ce rencontrer dans plusieurs situations similaires dans laquelle trouver le plus court chemin peut faire économiser beaucoup d'argents comme par exemple un robot qui se déplace sur la planète mars. On a choisit le cas d'un client dans un supermarché car il nécessite également une interface pour l'utilisateur.

Notre but est d'avoir un algorithme qui résout ce type de problème non seulement pour un seul supermarché (un seul labyrinthe) mais pour n'importe lequel. Les objéctifs qu'on a tracé dés le début du travail :

— Un générateur de labyrinthe aléatoire.
— Un solveur qui prend en paramètre le labyrinthe.
— Une interface pour l'utilisateur.

Donc notre logiciel s'adapte au supermarché qui veut le lancer pour ses clients.

2 Modèle

on représente le supermarché comme une labyrinthe de longueur *width* et de hauteur *height* avec une entrée quelconque à droite et une sortie quelconque à gauche. La Labyrinthe est donc un ensemble de points $(n, m) \in \{0, \ldots, width - 1\}x\{0, \ldots, height - 1\}$ construite de telle sorte que :

— en absence de mur, l'arête entre deux points voisins est de poids 1.
— pour tout couple de points (n, m) et (n', m') il existe un chemin liant (n, m) et (n', m').
— chaque point contient un produit.

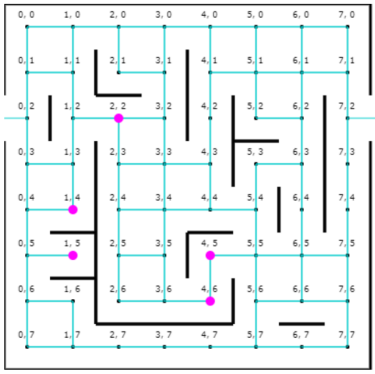


FIGURE 1 – Labyrinthe 8x8 : l'entrée est (0,2), la sortie est (2,2), les produits à récupérer en magenta.

Au départ, un client (ou un robot) est placé à la rentrée (le point (0,2) dans l'exemple précédent), et on souhaite qu'il prend un trajet optimal pour passer par tout les points où se trouvent les produits (points en magenta), sans prendre en considération l'ordre, puis arriver au point de sortie. La Figure 1 illustre un exemple de labyrinthe dans ce modèle.

3 Générer le labyrinthe et l'afficher

3.1 Générateur

(Marwan)

La première étape pour résoudre ce problème est de générer une labyrinthe selon le modèle décrit précédemment. Le'objectif est d'obtenir un graphe sous forme de dictionnaire qui à chaque point (n, m) associe une liste contenant les voisins auxquels il est lié directement par une arête, ce graphe sera un input primordiale au solveur.

Comme notre but est d'avoir un algorithme qui donne la solution pour tout labyrinthe vérifiant les critères du modèle on aura intérêt à avoir un générateur de labyrinthe aléatoire, i.e la présence des murs est aléatoire sous réserve de ne pas avoir des cellules enfermées. Pour cela on utilisera la bibliothèque *random*.

Dans un fichier "Maze_generator.py" on introduit une classe **Cell** représentant une cellule d'une grille qui elle représentée par une classe **Grid**. Notre labyrinthe est représenté par une classe **Maze** qui hérite de **Grid**. Un objet Cell possède des attributs i et j qui sont les coordonnées du centre ainsi qu'une liste de booléen _walls de longeur 4 qui indique la présence ou non de mur dans chaque côté.

Le générateur se base sur l'agorithme de parcours en profondeur (Depth-First Search) en considérant une grille comme un graphe non orienté dont les noeuds sont les cellules (objet Cell), et chaque noeud est lié à ses 4 cellules voisines (moins que 4 aux bords). Dans une méthode *explorer* de **Maze**, on explore tout les cellules d'un objet *Grid* ayant des murs partout intialement, et à chaque fois qu'on est sur une cellule on supprime des murs aléatoirement. On veillera bien sûr que si un mur est supprimé celui à son dos l'est aussi. Pour que l'exploration soit optimale, on définit une matrice d'adjacence _visited_matrix un attribut de **Grid** qui pour chaque point indique s'il a été visité ou non, et on définit une file au sein de la méthode *explorer* à quelle on ajoute à chaque fois la dérnière position. L'algorithme s'écrit :

```
Cell = random cell;
remove some of Cell's walls randomly;
set Cell as visited;
stack = empty stack;
while not all cells visited do
    if not all Cell's neighbors visited then
        Cell = random not visited neighbor;
        remove some of Cell's walls randomly;
        set Cell as visited;
        add Cell to stack;
    else
        Cell = stack.pop() (i.e go back to previous cell);
    end
end
```

Algorithm 1: Random maze generator algorithm

Ensuite, on écrit du code qui assure que les bords du labyrinthe sont fermés à l'exception des positions de l'entrée et de la sortie qui doivent être ouvertes.

Pour obtenir le graphe représentant le labyrinthe dont aura besoin pour le solveur (graphe = dictionnaire en python), on utilise une méthode *generate_graph_cells* qui regarde simplement la présence de mur ou non en chaque point et lui associe une liste contenant les voisins accessibles.

Notons que, puisque c'est un supermarché, l'objet *Maze* prend en entrée une liste contenant les noms des produits, c'est une base de données dont on parlera ultérieurement dans la partie interface, et affecte à chaque cellule un produit de manière aléatoire.

3.2 Affichage

(Marwan)

on a voulu visualiser la labyrinthe bien avant que l'interface soit prête (surtout pour tester le solveur). Notre objectif initial était de faire une interface web, c'est pour cette raison qu'on a pensé a utiliser Javascript. Ce qui n'est pas évident puisque tout les informations qui définissent notre labyrinthe sont sur du code python. L'astuce qu'on a utilisé est la gestion des fichier en python, en créant un fichier .html surlequel on écrit du JS.

La méthode *save_to_html* de la classe **Maze** crée un fichier en mode écriture appelé "Maze.html" et écrit dessus des balise html standard, la plus important est celle su script. Le contenue de <script> dessin à l'aide de canvas < \ script> se construit à fur et à mesure qu'on parcourt les cellules de notre labyrinthe (en python), ici inteviennent les méthodes *draw* et *draw_graph* de Cell qui trace respectivement les murs en noir épais et les arêtes en cyan entre les points voisines accessibles entre eux.

Ensuite, on affiche les points où se trouve les produits cibles en magenta sachant que la méthode *save_to_html* prend en paramètre une liste contenant ces cellules cibles, elle prend aussi en paramètre une liste de coordonnées *path* qui sera la solution donnée par le solveur et ajoute le code JS qui permet de la tracer en rouge.

Enfin, on utilise *addEventListener* pour ajouter une fonctionnalité utile pour l'utilisateur : quand le curseur de la souris est sur une cellule, on affiche just en dessous du curseur un rectangle contenant le nom de produit qui s'y trouve.

Le fait d'écrire du JavaScript dans du python de cette manière est certainement désagréable et difficile à lire, c'était une approche qu'on a choisi pour les raisons suivante :

— L'objectif est une interface web, on utilisera JS en fin de compte.
— Faire tout en JS n'était pas une option puisqu'on aura besoin des bibliothèques python puissante pour le solveur.
— C'était une approche temporaire pour visualiser et tester.

Voir Figure 2 pour une illustration.

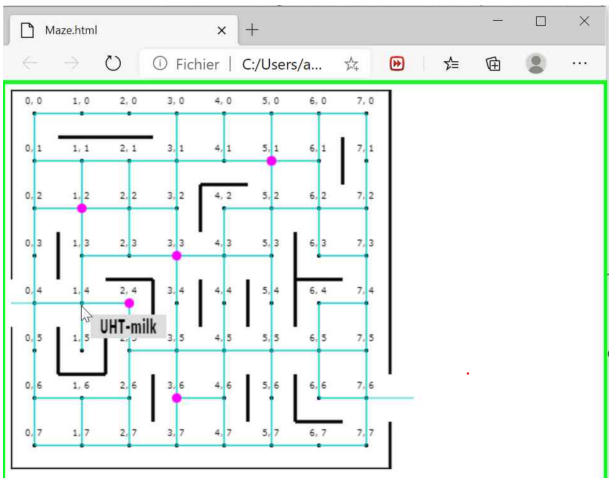


FIGURE 2 – Illustration du résultat d'affichage d'un plan du supermarché généré aléatoirement.

4 Solveur

4.1 Solveur par recherche opérationnelle

(Ilyes)

4.1.1 Première approche : programme linéaire en nombre entiers unique

Le problème consiste d'une part à trouver le plus court chemin entre un départ et une arrivée, et d'autre part à visiter les différents sommets du graphe dans lesquels les produits désirés se trouvent. Il s'agit ainsi d'un mélange entre deux problèmes de recherche opérationnelle bien connus :

— Problème de plus court chemin
— Problème du voyageur de commerce

La difficulté est qu'il ne s'agit ni d'un plus court chemin pur, ni d'un voyageur de commerce pur, mais bien d'un hybride des deux. D'un point de vue théorique, l'idée est de prendre en recherche opérationnelle la modélisation usuelle de chaque problème sous forme de programme linéaire en nombre entiers, et de fusionner les deux en réadaptant les contraintes.

On va se placer dans le cadre de la théorie des graphes. On modélise le supermarché sous forme de graphe $G = (V, E)$, avec ainsi les notations suivantes :

— G : graphe du supermarché.

— E : ensemble des arêtes de G.
— V : ensemble des sommets de V.
— U : ensemble des sommets contenant les produits à aller chercher.
— s : sommet rattaché au sommet de départ.
— t : sommet rattaché au sommet d'arrivée.

Notre variable de décision va être $x_{ij} \in \mathbb{N}$ le nombre de fois que l'on passe par l'arête ij, avec $(i,j) \in V^2$ deux sommets formant une arête de E.

La fusion des modèles implique le respect de trois contraintes principales :

(1) Une contrainte issue du modèle du plus court chemin : le flux de la trajectoire se conserve du début à la fin.

(2) Une contrainte issue du modèle de voyageur de commerce selon la formulation de Dantzig-Fulkerson-Johson : les boucles isolées indépendantes du trajet initial ne sont pas permises.

(3) Une autre contrainte issue du modèle de voyageur de commerce : on doit aller chercher les produits dans les sommets de U.

On peut ainsi construire un programme linéaire en nombre entiers pouvant potentiellement résoudre notre problème :

$$\min_{x \in \mathbb{Z}^E} \sum_{ij \in E} x_{ij}$$

$$s.c. \quad \begin{cases} \displaystyle\sum_{j \in V} x_{ij} - \sum_{j \in V} x_{ji} = \mathbb{1}_{\{i=s\}} - \mathbb{1}_{\{i=t\}} & \forall i \in V & (1) \\ \displaystyle\sum_{i \in Q} \left(\sum_{j \in Q \setminus \{i\}} x_{ij} \right) \leq |Q| - 1 & \forall Q \subsetneq V, |Q| \geq 2 & (2) \\ \displaystyle\sum_{j \in V, ij \in E} x_{ij} \geq 2 & \forall i \in U & (3) \\ x_{ij} \geq 0 & \forall ij \in E & (4) \end{cases}$$

La contrainte (2) rend le problème NP-difficile comme c'est le cas pour le problème du voyageur de commerce. Le nombre de Q possibles vaut :

$$|\mathcal{P}(V)| - \binom{|V|}{2} - \binom{|V|}{1} - 2 = 2^{|V|} - \frac{|V|!}{2(|V|-2)!)} - \frac{|V|!}{(|V|-1)!)} - 2 = 2^{|V|} - \frac{|V|}{2}(|V|-3) - 2$$

On voit bien que le nombre de Q possibles est exponentiel par rapport à $|V|$. Or c'est ce problème là qui a amené à l'échec de cette première approche : dès que $width$ et $height$ dépassent 5, une erreur Python indique que la mémoiré a été saturée. On identifie facilement que le stockage de tous les Q possibles dans une liste de listes ne permettra pas de résoudre notre problème avec une taille un minimum intéressante.

4.1.2 Deuxième approche : diviser pour régner

Une autre approche aurait été de chercher un moyen de gérer le stockage des différents Q possibles sous une forme bien optimisée, en faisant par exemple appel à des fichiers CSV extérieurs. Après mûre réflexion, nous avons préféré profiter de toute la puissance de la librairie Google-OR pour laquelle un grand travail d'optimisation de qualité a été réalisé par Google pour résoudre le problème de voyage de commerce

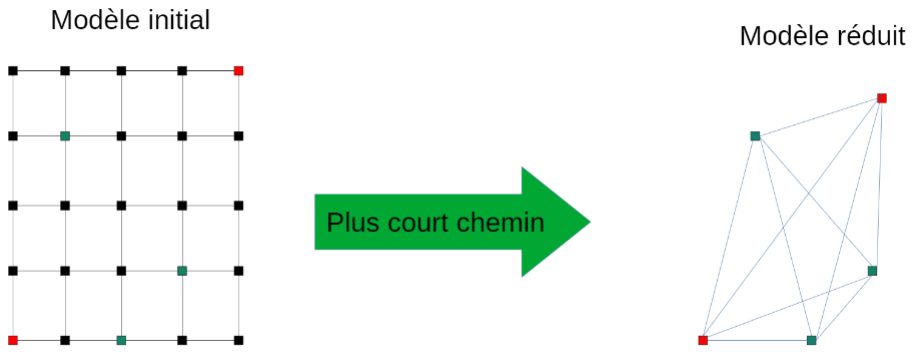
notamment. Comme notre modèle ne rentre pas exactement dans le formalisme nécessaire à la librairie, l'idée n'est plus de fusionner les deux grands modèles sous-jacents, mais plutôt de les séparer.

On peut ainsi diviser le problème général en deux catégories de sous-problèmes :

— Problèmes de plus court chemin : quel est le plus court chemin entre deux sommets ?

— Problème du voyageur de commerce : comment atteindre tous les sommets choisis au mieux ?

Le problème du plus court chemin nous permet de passer d'un modèle initial à un modèle réduit qui ignore tous les chemins inutiles et ne prend en compte que les plus courts chemins entre deux sommets qui nous intéressent (parmi le départ, l'arrivée et les sommets de U).



Le modèle réduit est ainsi parfaitement adapté au problème classique de voyageur de commerce, et peut être résolu avec le formalisme de la librairie Google-OR. Une fois la solution trouvée, il suffit de reconstruire la solution dans le formalisme de notre modèle initial.

4.1.3 Implémentation du modèle

On modélise G sous forme de dictionnaire, avec comme clés les différents sommets du graphe (c'est-à-dire appartenant à V), et comme valeurs les différents sommets voisins de la clé correspondante. À partir d'un G, d'un départ, d'une arrivée et d'un U, on peut initier une classe "Model" qui nous servira tout le long de la résolution.

Le sous-problème du plus court chemin peut être résolu par un algorithme efficace en $O(n^2)$ comme Dijkstra. Néanmoins, cela nécessite d'adopter une méthodologie différente de celle adoptée jusqu'à présent avec une minimisation de fonction sous différentes contraintes. Ce sous-problème étant de loin le plus facile à résoudre, son optimisation excessive n'est que peu utile. Le solver SCIP est utilisé pour résoudre les programmes linéaires en nombre entiers.

Ainsi, la résolution des problèmes de plus court chemin se fait avec le modèle explicité précédemment en enlevant la contrainte (2) qui nous avait posé problème. De plus, $x_{ij} \in \{0, 1\}$ pour tout $ij \in E$, car repasser deux fois par la même arête n'est plus nécessaire dans ce cas, et on revient bien à un problème de s-t plus court chemin classique en reliant dans le graphe s au point de départ et t au point d'arrivée. Pour chaque plus court-chemin à calculer, on garde le même *Model* et on change simplement le départ et l'arrivée auxquels viennent se greffer les sommets s et t.

Les résultats de cette première partie nous permettent de construire la matrice de distance spécifique au problème de voyageur du commerce. Plus précisément, selon la terminologie utilisée par Google-OR, il

s'agit d'un *Vehicle Routing Problem* que l'on a adapté à notre cas. Le travail d'optimisation de Google nous permet ainsi de résoudre le deuxième sous-problème sans trop de difficultés.

Enfin, à partir de la solution trouvée, on reconstruit la solution générale du modèle sous un format qui va pouvoir se lier aux autres parties de ce projet.

4.2 Solveur par apprentissage

(Marwan)

4.2.1 Algorithme de Q-learning

Le Q-learning fait référence à un algorithme de formation d'agents afin qu'ils convergent vers la sélection d'actions optimales. Cette technique ne nécessite aucun modèle initial de l'environnement.

En général, l'idée de Q-learning est d'apprendre une fonction valeur-action Q (s, a) qui, pour un état s donné et une action a donnée, renvoie la récompense cumulative attendue que l'agent obtiendra jusqu'à la fin de l'épisode (dans notre cas, un épisode est une instance du plus court chemin ou TSP). Une fois que nous avons une version bien comportée d'une telle fonction, et si nous pouvons nous permettre d'énumérer toutes les actions possibles, nous pouvons simplement laisser l'agent choisir les actions a qui maximisent le retour attendu estimé Q (s, a) de n'importe quel état s.

Pour construire la fonction Q, l'idée est de lancer de nombreuses épisodes avec notre agent et d'améliorer itérativement la fonction. Au cours de chaque épisode, dans l'état s, l'agent peut utiliser une politique avide - où il prend l'action un Q (s, a) de maximisation - et il peut également prendre des actions complètement aléatoires de temps en temps (afin de continuer à explorer). Au cours de nombreux épisodes, nous pouvons stocker les récompenses cumulatives qui ont finalement été obtenues en tant que «conséquence» de certaines paires état / actions (s, a).

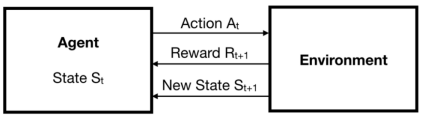


FIGURE 3 – Q-learning : interaction entre l'agent et l'environnement

Avant que l'apprentissage ne débute, la fonction Q est initialisée arbitrairement (le plus souvent nulle). Ensuite, à chaque choix d'action, l'agent observe la récompense et le nouvel état (qui dépend de l'état précédent et de l'action actuelle). Le cœur de l'algorithme est une mise à jour de la fonction de valeur. La définition de la fonction de valeur est mise à jour à chaque étape de la façon suivante :

$$Q[s,a] = (1-\alpha)Q[s,a] + \alpha(r[s,a] + \gamma \underset{a'}{max} Q[s',a']) \tag{5}$$

où s' est le nouvel état, s est l'état précédent, a est l'action choisie, r est la récompense reçue par l'agent, α est un nombre entre 0 et 1, appelé facteur d'apprentissage, et γ est le facteur d'actualisation. Un épisode de l'algorithme finit lorsque s_{t+1} est une état final.

Le facteur d'apprentissage α détermine à quel point la nouvelle information calculée surpassera l'ancienne. Si $\alpha = 0$, l'agent n'apprend rien. A contrario, si $\alpha = 1$, l'agent ignore toujours tout ce qu'il

a appris et ne considèrera que la dernière information. Le facteur d'actualisation γ détermine l'importance des récompenses futures. Un facteur 0 rendrait l'agent myope en ne considérant seulement les récompenses courantes, alors qu'un facteur proche de 1 ferait aussi intervenir les récompenses plus lointaines. Si le facteur d'actualisation est proche ou égal à 1, la valeur de Q peut diverger, ce qui n'est pas apprécié.

4.2.2 Implémentation

On aura besoin uniquement de la bibliothèque *numpy* pour implémenter cet algorithme. Notons que numpy contient *random*.

On considère qu'un état pour notre agent est de se trouver à (i, j) une cellule du labyrinthe, une action consiste en le passage à une cellule voisine $\in \{(i-1, j), (i+1, j), (i, j-1), (i, j+1)\}$, et les récompenses sont représentées par une matrice R carrée d'ordre $width \times height$ telle que
$R(i + width \times j, i' + width \times j')$ est la récompense du passage de l'état (i, j) à l'état (i', j'), et la matrice Q est elle même carrée d'ordre $width \times height$, elle est nulle au départ et actualisée à chaque itération par la formule (5).

Dans un premier temps, on définie la matrice de récompense R de la manière ci-dessous, on rappelle que le générateur nous donne un dictionnaire qu'on appellera *graph* qui à chaque cellule associe les cellules directement accessibles :

$$\begin{cases} R[(i,j),(i',j')] = -\infty & \text{si} (i', j') \notin graph[(i, j)] \\ R[(i,j),(i',j')] = 500 & \text{si} (i', j') \in graph[(i, j)] \text{ et } (i', j') \in target\_products \cup \{sortie\} \\ R[(i,j),(i',j')] = -1 & \text{si} (i', j') \in graph[(i, j)] \text{ et } (i', j') \notin target\_products \end{cases} \tag{6}$$

On rappelle que $(i, j) \iff i + width * j$.
En définissant la matrice R de cette manière, on teste l'agorithme pour différentes valeurs de α et γ. On remarque que notre agent n'atteint pas la sortie, mais il choisit plutôt de rester en un point où se trouve un produit à récupérer ! on a même essayer d'affecter à la sortie une récompense plus grande que les produits à récupérer : si elle est très grande il sort sans récupérer tout les produits, si elle n'est pas suffisamment grande il se comporte comme dans le cas qui précède.

La solution est de diviser le problème en deux sous-problèmes comme on a fait pour le solveur par recherche opérationnelle : un plus court chemin et un voyageur de commerce.

Plus court chemin :

on applique l'algorithme de Q-learning pour trouver le plus court chemin et la plus courte distance entre chaque deux points de l'ensemble { entrée, sortie } ∪ target _ products, ce qui rend ce solveur beaucoup moins rapide que le précédent . Pour le faire pour deux points s_1 et s_2, notre agent se trouve initialement en s_1 avec une matrice Q nulle qu'il actualise à chaque itération avec $\alpha = 0.2$ et $\gamma = 0.9$, et on définit la matrice des récompenses par :

$$\begin{cases} R[(i,j),(i',j')] = -\infty & \text{si} (i', j') \notin graph[(i, j)] \\ R[(i,j),(i',j')] = 5000 & \text{si} (i', j') \in graph[(i, j)] \text{ et } (i', j') = s_2 \\ R[(i,j),(i',j')] = -1 & \text{si} (i', j') \in graph[(i, j)] \text{ et } (i', j') \neq s_2 \end{cases} \tag{7}$$

Notons que le graphe est non orienté donc on n'a pas besoin de recalculer le plus court chemin de s_2 à s_1, il suffit d'inverse celui de s_1 à s_2.

Après l'étape d'entraînement, on pose l'agent en s_1 et on le fait choisir un chemin vers s_2 qui maximise Q à chaque pas.

Voyageur de commerce :

Ici, les états sont les points de l'ensemble { entrée, sortie } ∪ target _ products et une action pour l'agent, lorsqu'il est en un état, est le passage en un état différent .

- Q est nulle au départ.
- $\alpha = 0.4$
- $\gamma = 0.7$
- La matrice de récompense est telle que : $R[s_1, s_2] = 1/d(s_1, s_2)$ $\forall s_1, s_2$ des états tq $s_1 \neq s_2$, avec $d(s_1, s_2)$ la plus courte distance entre s_1 et s_2 calculée à l'étape précédente, si $s_1 = s_2$ la récompense est nulle.

Après entraînement du model, on pose notre agent à l'entrée et on le fait parcourir un chemin maximisant Q, sauf qu'il ne laissera pas nécessairement la sortie jusqu'à la fin ! puisque pour lui c'est un état comme les autres. L'astuce est de modifier la matrice R en considérant que la sortie est beaucoup plus éloignée des autres états, $i.e$ pour tout état s_1 différent de la sortie on rajoute à $d(s_1, sortie)$ la quantité $width \times height$.

Résultats :

Le solveur n'est pas parfait et moins meilleur que le premier en terme de précision et de temps de calcul, il y a beaucoup d'améliorations à faire !

Quand le nombre de produits choisis est (moins de 5) le solveur est relativement rapide et donne souvent de bons résultats, plus ce nombre est grand et plus la taille de la labyrinthe est grande moins les résultats sont bons et plus le calcul est lent. N'oublions pas que le code est écrit entièrement en python et n'utilise que numpy, il serait meilleur d'écrire le code en un language plus rapide, C++ par exemple. Dans ce cas, il faut faire une liaison avec le reste du logiciel fait en python et django.

4.3 Liaison générateur solveur

(Marwan)

Le générateur et les deux solveurs sont chacun sur un fichier .py, "maze_generator" pour le premier, "graph_solver" pour le solveur par recherche opérationnelle et "Qlearning_solver" pour celui par apprentissage. Afin de pouvoir tester notre logiciel on a créé dans un premier temps un fichier "main.py" qui importe les trois fichier, et dans lequel on suit les étapes suivante :

- Générer le labyrinthe.
- Adapter le graphe obtenue pour le donner au solver. (le 1er solver prend un graphe dont les points sont des str et non des tuples)
- Trouver la solution à l'aide des méthodes du solver.
- Afficher la solution à l'aide de la méthode dédiée dans le générateur.

Le "main.py" jouera le rôle de "views.py" de django qu'on aborde dans la séction suivante.

5 Architecture Django

(Anatole) Afin de générer un labyrinthe de manière responsive avec l'utilisateur nous avons fait le choix de développer une interface web en utilisant le framework Django qui s'appuye sur Python.

5.1 Django, le framework Python pour le web development

Django a été créé entre 2003 et 2005 par une équipe de développeur indépendante qui créait des sites web pour des magazines et journaux. Il a depuis connu plusieurs améliorations et en est désormais à sa version 3.1. Selon les sondages menés par stackoverflow, Django est d'ailleurs devenu le huitième framework le plus utilisé au monde en 2020.

Django dispose de nombreux avantages lorsqu'il s'agit de développer une application web. Le premier est évidemment que celui-ci repose sur le langage Python, langage de haut-niveau facilement adaptable pour tous les objectifs ; qui est aussi le langage utilisé pour générer le labyrinthe et le solveur. Ainsi, Django dispose d'une communauté active ainsi que d'une documentation fournie et très qualitative ce qui le rend facilement utilisable et agréable pour résoudre les différents écueils auxquelles nous avons pu nous heurter. Django est totalement gratuit et open-source, comme le veut la philosophie du langage Python.

De plus, Django a été pensé par ses créateurs comme une solution *all-inclusive*. Ainsi toutes les fonctionnalités d'un site web ont été pensées comme des librairies supplémentaires que l'on peut aisément adapter à notre projet. Django est ainsi un framework qui s'adapte à tous les types de projet web, de notre petit site de supermarché à des applications énormes telles que Spotify ou Instagram.

5.2 L'architecture MVT

Django a été conçu en suivant l'architecture MVT, c'est-à-dire que l'organisation d'un site web sous ce framework s'articule autour de trois pôles :

— Modèle
— Vue
— Template

5.2.1 Modèle

Les modèles sous Django permettent d'interagir avec les bases de donnée auxquelles est relié le projet. Ainsi l'une des grandes forces de Django est de pouvoir interagir de manière rapide et facile avec des bases de données SQL afin d'aller chercher des données et les restituer à l'utilisateur. Par défaut, Django utilise des bases de donnée SQlite 3. Ces bases de données SQL permettent de stocker et de retrouver rapidement les informations. Pour encore plus de scalabilité, Django peut aussi prendre en compte des bases de donnée PostgreSQL ou MySQL. Les modèles Django s'appuient sur un ORM (*Object Relational Mapping*) qui traduit les résultats d'une requête SQL en objets Python. Ainsi lorsque l'on lance une requête Python, on peut récupérer les résultats dans des listes ou dictionnaires Python qui sont facilement utilisables pour la génération de l'interface.

5.2.2 Template

Un template est un fichier HTML destiné à recevoir des objets Python. Ainsi Django comprend des lignes de code Python au coeur d'un fichier HTML qui permettent d'utiliser des objets Python sur l'interface de l'utilisateur.

5.2.3 Vue

L'objectif de la vue est de recevoir une requête HTTP et de retourner en fonction de cette requête une réponse HTTP. La vue permet ainsi d'effectuer différentes en action en fonction de ce qui est nécessaire :

— Si une interaction avec une base de données est requise, la vue appelle un modèle afin de récupérer des informations puis gère ensuite ces informations pour les envoyer au gabarit dans la forme adéquate.
— Si un template est nécessaire, la vue l'appelle.

Finalement, chaque vue est associée à un URL (ou une forme d'URL) stocké dans le fichier urls.py

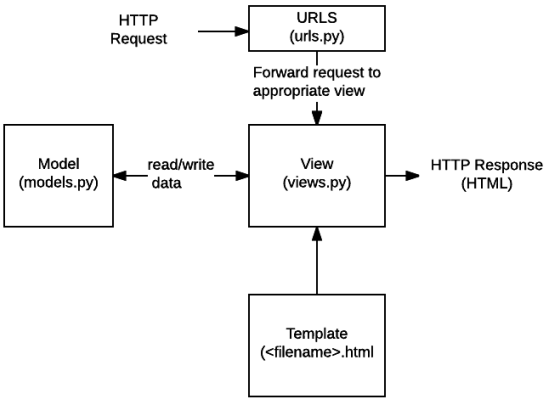


FIGURE 4 – Fonctionnement de Django

6 L'architecture Django pour le projet supermarché

(Abdennour)

6.1 Accès à l'interface Django

Pour travailler sur l'architecture de notre interface Django, on a opté pour la *Virtualenv* qui permet d'accueillir nos scripts python dans un environnement isolé. Ce choix a pour but d'assurer un traitement clair et propre de notre interface.

L'accès au serveur recueillant notre interface se fait via *Windows Powershell* et en tapant certaines commandes. Et ça aprés avoir installer la *Virtualenv* et la version Django==3.0.2 . Les commandes se font comme suit :

— Démarrer *Windows Powershell* en tant qu'un administrateur.
— *cd* vers le '*path*' du projet sur votre ordinateur.
— *pipenv shell* pour démarrer l'environnement virtuel.
— *activate* pour l'activer. A cette étape normalement, le *powershell* doit afficher une syntaxe pareille à celle-ci (*(project folder's name) PS path*>).
— *cd* vers *src* du projet.
— *cd* vers *supermarket*. C'est l'*App* principale de l'interface.
— *python manage.py createsuperuser*. Cette commande a pour but de vous donner le droit d'accèder à la structure et la base de données de l'interface comme étant un adminstrateur.A cette étape-là, vous devez taper votre nom d'utilisateur et votre mot de passe.

— *python manage.py makemigrations*. Cette commande génére la nouvelle base de données sous format *SQLITE* ;
— *python manage.py migrate*. Cette commande transmet la nouvelle base de données vers la base de données par défaut *db.sqlite3*.
— *python manage.py runserver*. Le *powershell* va vous générer le lien HTTP vers le serveur de l'interface.

6.2 Architecture de l'interface

6.2.1 Coeur de l'interface Django

Dans notre cas, l'App *Supermarket* fait référence au coeur de notre interface Django. Elle contient le fichier *settings.py*, c'est-à-dire la configuration structurelle de l'interface, à travers laquelle on gére les Apps installées (les composantes de l'architecture), la base de données, les chemins vers le dossier source de l'interface et le dossier source des *templates*. Ainsi que, les différents aspects de la sécurité et de la protection de données des utilisateurs.

L'App *Supermarket* contient également le fichier *urls.py*. Celui-ci gére les urls disponibles via l'interface. Il fait appel aux fonctions définies dans *views.py*. En plus, il dispose des fichiers *asgi.py* et *wsgi.py* qui communique des objets exécutables *application* au code python et permet de les générer à l'aide des commandes prédéfinies conformément aux standards ASGI et WSGI.

6.2.2 Structure de L'App principale *products_ database*

6.2.2.1 *models.py*

Le fichier *models.py* gére les modéles mis en oeuvre par l'interface. Ils sont définis sous forme de classes Python qui hérite sa structure de django.db.models.Model. Chaque attribut du modéle représente un champ de base de données. En gros, les modéles sont ceux qui définissent la structure des objets qui seront stockés dans la base de données. Dans notre fichier *models.py*, on a crée deux modéles :

— *class Product (models.Model)* : chaque objet de cette class est défini par 4 attributs : (*order*, *identity*, *date*, *name*). La représentation *str* de l'objet de Product() est *name*.
— *class CheckedProduct(models.Model)* : chaque objet de cette class est défini par un seul attribut : (*productsList*). La représentation *str* d'un objet de CheckedProduct() est *productsList*.

Le modéle *Product* gére les produits disponibles dans le supermarché. Pourtant que le modéle *CheckedProduct* gére les produits choisis par l'utilisateur en générant une liste contenant tous ses choix.

6.2.2.2 *admin.py*

Le stockage des objets de ces deux modéles dans la base de donnéées *db.sqlite3* doit faire objet d'une permission spéciale qui se fait via le fichier *admin.py*. En gros, il permet à l'utilisateur admin d'enregister la nouvelle base de données transmise.

6.2.2.3 *forms.py*

Dans ce fichier, on peut créer des formulaires à travers différentes méthodes. Chaque formulaire est une *class* python qui hérite soit de *forms.ModelForm* ou *forms.Form*. Chaque attribut de l'objet du *form* est défini par une certaine structure propre à Django. Dans notre cas, on a crée une formulaire qui prend

les porduits stockés et les affiche à l'utilisateur. Puis, il les sauvegarde en tant que des objets de la class *CheckedProduct*.

6.2.2.4 *views.py*

Ce fichier est celui qui gére l'affichage de l'interface . C'est le coeur de l'affichage pour Django. Chaque fonction définie dans *views.py* prend comme paramétre la *request* de l'utlisateur et retourne la *template* et le *context*. La *template* est un fichier HTML stocké dans un dossier nommé *templates*. Il prend comme instance l'objet modéle mis en *context*. Dans notre cas, on a crée six fonctions :

- *database_upload* : elle permet à l'utilisateur de déposer n'importe quelle fichier *.csv* contenant la base de données de produits disponibles dans le supermarché. Après le dépôt, les produits seront des objets de la class *Product*.
- *show_database* : elle permet d'afficher la base de données déposée par ordre sous format *(order, identity, date, name)*.
- *user_selection_save* : elle permet à un utilisateur donné de choisir les produits qu'il veut acheter via le formulaire crée dans *forms.py*. Et puis, elle sauvegarde les produits choisis comme des objets de la class *CheckedProduct*. Après, l'utilisateur sera dirigé vers l'*url* correspondant au choix du solveur du traitement *choose_solver*.
- *show_checked_products* : elle permet d'afficher l'historique des choix des utilisateurs. En effet, cette fonction génére une base de données tellement cruciale qui pourra être exploiter pour optimiser le rendement du supermarché.
- *choose_solver* : elle permet à l'utilisateur de choisir un solveur parmi le Qlearning_solver et le RO_solver. Le choix de l'utilisateur *user_choice_solver* est ainsi modifié.
- *show_path* : la fonction récupére la liste des produits disponibles dans le supermarché afin de générer la labyrinthe de taille ($width \times height$). Elle génére également les cellules du graphe. Puis, elle prend la liste des produits choisis par l'utilisateur et attribue à chaque produit des coordonnées de position. Le modéle du graphe est prêt. Le solveur choisi par l'utilisateur tourne et la solution est générée ainsi que le chemin optimal. La fonction retourne le fichier *Maze.html* qui sera stockée dans *templates*.

Références :

[1] Depth-first search, https ://en.wikipedia.org/wiki/Depth-first_search .

[2] Travelling salesman problem, https ://en.wikipedia.org/wiki/Travelling_salesman_problem.

[3] How to apply reinforcement learning to order-pick routing in warehouses, https ://machine-learning-company.nl/en/technical/how-to-apply-reinforcement-learning-to-order-pick-routing-in-warehouses-including-python-code/

[4] Introduction to Q-learning, https ://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56

[5] Try-Django, https ://github.com/codingforentrepreneurs/Try-Django

[6] Django documentation, https ://docs.djangoproject.com/

[7] HTML documentation, https ://developer.mozilla.org/