

PotaG

Le social en circuit court

Projet Pluridisciplinaire d’Informatique Intégrative
Recherche de solutions innovantes dans les circuits courts et les jardins partagés

Travail réalisé par BEKHEDDA Maxence, GERMAIN Léo,
GIANELLI Thomas et WALTER Théo

Table des matières

1	Introduction	3
2	L'application	4
2.1	Base de données	4
2.1.1	Utilisateurs	5
2.1.2	Social	5
2.1.3	Ventes	6
2.2	Serveur web	7
2.2.1	app.py	8
2.2.2	potag_login.py	8
2.2.3	potag_social.py	10
2.2.4	potag_vente.py	11
2.2.5	potag_preferences.py	12
2.3	Algorithmes	14
2.3.1	Notion d'ordre	14
2.3.2	Tri Fusion	14
2.3.3	Backtracking	15
3	Tests et Performances	17
3.1	Tri Fusion	17
3.1.1	Tests	17
3.1.2	Performance	18
3.2	Backtracking	18
3.2.1	Tests	18
3.2.2	Performance	20
4	Gestion de projet	22
4.1	Cahier des charges fonctionnel	22
4.1.1	Auteurs de ce cahier des charges / groupe d'expression du besoin	22
4.1.2	Historique des modifications et révisions de ce document	23
4.1.3	Résultats et changements attendus	23
4.1.4	Parties prenantes	23
4.1.5	Contraintes de planning	23
4.2	To-Do List	24
4.2.1	To-Do List en vue de la première soutenance	24
4.2.2	To-Do List en vue des vacances de la Toussaint	25
4.2.3	Utilisation des To-Do List	26
4.3	Compte-rendus de réunion	26
4.3.1	Modèle type de compte-rendu	26
4.3.2	Utilisation des compte-rendus	27
4.4	Matrice RACI	28
4.5	Diagramme de Gantt	29
4.6	Post-mortem	29
5	Conclusion	31

1 Introduction

Notre objectif était de créer une application mettant en relation des particuliers produisant des fruits et légumes en nombre trop important pour leur consommation personnelle avec des acheteurs intéressés par la consommation de produits locaux.

Nous avons alors réalisé une application divisée en deux parties principales. La première prend la forme d'un réseau social où les utilisateurs peuvent partager des photos de fruits et légumes, de potagers ou bien de plats cuisinés à partir de certains produits. La seconde partie est quant à elle un espace de vente où seuls les utilisateurs inscrits avec un profil de producteur peuvent y publier des annonces de vente de fruits et légumes. Ces annonces seront alors visibles par les utilisateurs inscrits sous un profil d'acheteur qui pourront désormais acheter les produits qui les intéressent.

Notre système de vente utilise des points relais, appelés **lockers**, disposés à plusieurs endroits en ville. Lorsqu'un utilisateur décide d'acheter un produit à un producteur, un algorithme détermine alors un casier dans lequel venir déposer et récupérer le produit acheté.

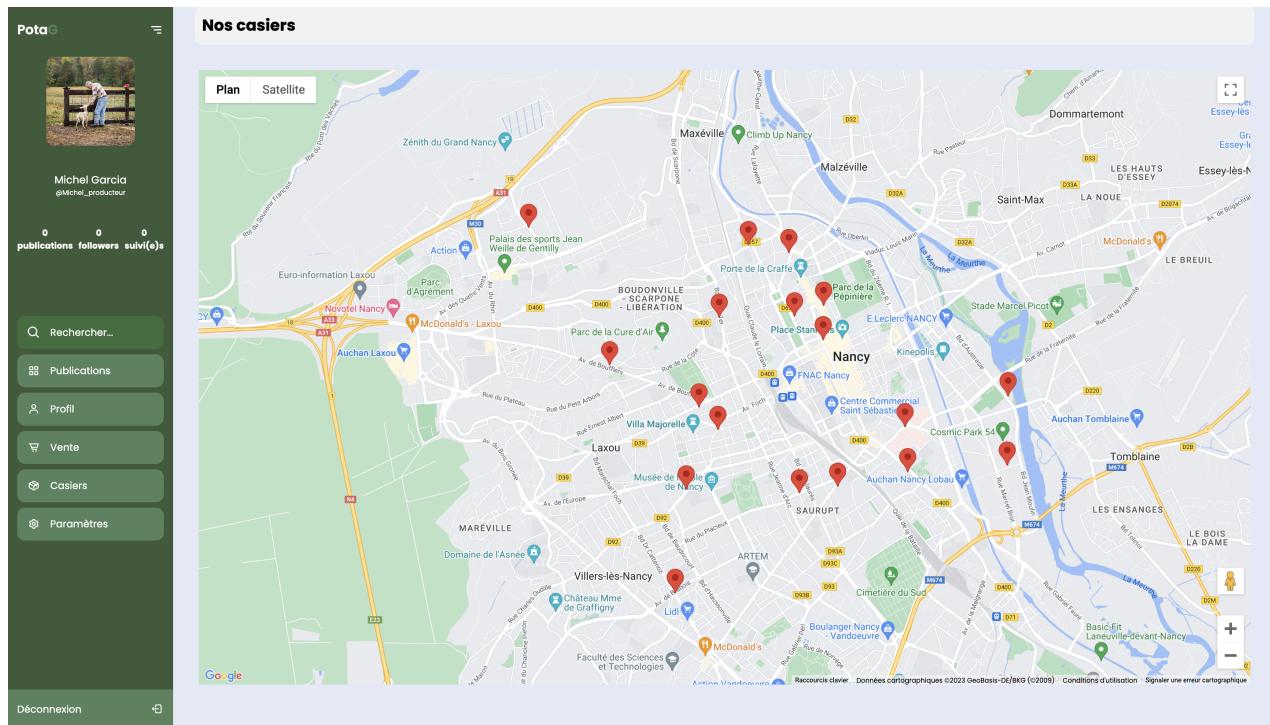


FIGURE 1 – Aperçu de la page **lockers** de notre application

2 L'application

2.1 Base de données

L'utilisation de notre base de données est au coeur de notre application. Pour cela nous avons dû dans un premier temps réfléchir à une structure permettant de réaliser les fonctionnalités que nous voulions implémenter. Nous avons alors établi le schéma relationnel¹ suivant :

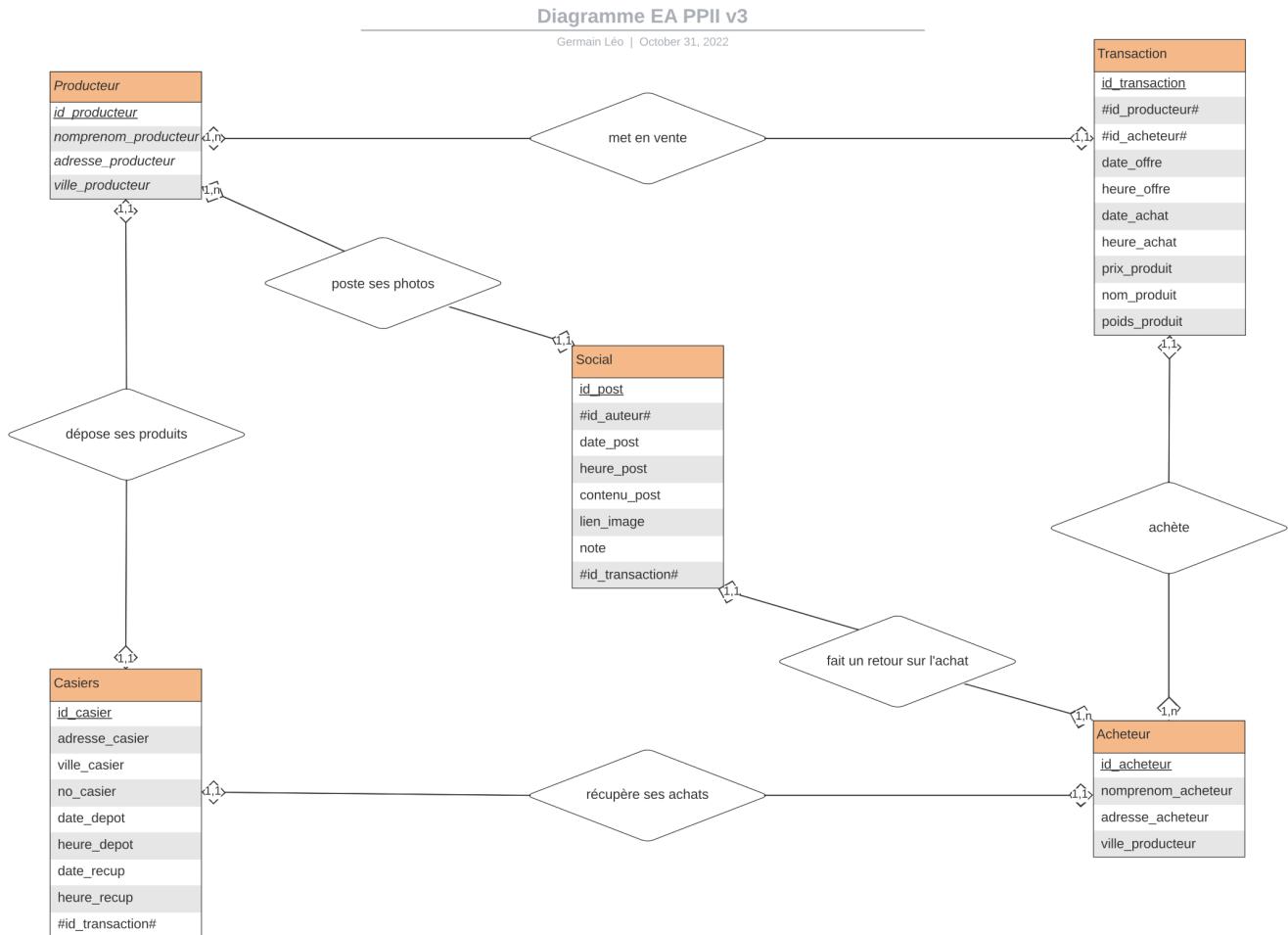


FIGURE 2 – Schéma relationnel de la première version de notre base de données

Néanmoins, au fur et à mesure que nous implémentions des fonctionnalités dans notre application, nous avons réalisé que notre base de données ne permettait pas d'obtenir les résultats désirés.

Nous avons fait évoluer la base de données afin qu'elle réponde au mieux à nos besoins. La base de donnée obtenue est détaillée ci-dessous.

Notre nouvelle base de données se compose de 8 tables :

- Producteur
- Acheteur
- Social
- Casier

1. Les cardinalités des associations ne sont pas correctes sur ce graphique

- Follow
- Like
- File_attente
- Transaction

Toutes ces tables ont été créées grâce à un programme Python et au module `sqlite3`. Voici un extrait de code nous ayant permis de générer la base de données :

```

1 con = sqlite3.connect("../potag_main.db")
2 cur = con.cursor()
3
4 cur.execute("CREATE TABLE producteur ( \
5     id_producteur VARCHAR(100) PRIMARY KEY, \
6     nom VARCHAR(40), \
7     prenom VARCHAR(40), \
8     mail VARCHAR(100), \
9     password VARCHAR(100), \
10    profile VARCHAR(100), \
11    adresse_producteur VARCHAR(40), \
12    ville_producteur VARCHAR(40), \
13    casier INT );")

```

Listing 1 – Code Python permettant de créer la table `producteur`

2.1.1 Utilisateurs

Les tables `producteur` et `acheteur` sont construites de manière identique. Selon le profil que l'utilisateur renseigne lors de la création de son compte, il sera enregistré dans l'une des deux. La clé primaire de ces deux tables est l'`id_acheteur` (ou `id_producteur`). Il correspond au nom d'utilisateur qui a été choisi, sachant que pour celui-ci soit unique, un test est réalisé à l'inscription. Le `nom` et le `prénom` de l'utilisateur sont également renseignés à l'inscription sans contraintes particulières. L'adresse mail de l'utilisateur est elle aussi ajoutée à la colonne `mail` lors de l'inscription, à condition qu'elle respecte le format suivant `exemple@nom.domaine` (exemple, nom et domaine doivent contenir au minimum un caractère). Ensuite, le mot de passe est enregistré dans la colonne `password`. Pour cela, il doit respecter 4 conditions : être de plus de 8 caractères, contenir au moins une majuscule, une minuscule et un chiffre. La colonne `profile` contiendra quant à elle le chemin vers la photo de profil de l'utilisateur. Ce champ peut être rempli ou non par l'utilisateur, tout comme son adresse et sa ville. Enfin, la colonne `casier` est remplie dans le cas d'une transaction. Elle contient pour le producteur ou pour l'acheteur l'identifiant du casier dans lequel il doit aller déposer ou récupérer ces produits.

2.1.2 Social

Ensuite vient la table `social`, qui permet d'accéder à toutes les publications faites par les utilisateurs.

	<code>id_post</code>	<code>id_auteur</code>	<code>date_post</code>	<code>heure_post</code>	<code>contenu_post</code>	<code>lien_image</code>
1	1	Maxence	2023-01-01	17:59	Trop chouette	static/images/publications/Maxence1.png
2	2	Thomas	2023-01-01	18:00	Super genial	static/images/publications/Thomas1.png
3	3	Léo	2023-01-01	18:00	trop bien j'étais la bas aussi	static/images/publications/Léo1.png
4	4	Théo	2023-01-01	18:01	beau temps?	static/images/publications/Théo1.png

FIGURE 3 – Exemple de contenu dans la table `social`

La première colonne est la clé primaire, c'est l'identifiant de la publication. Il est ajouté de manière automatique à chaque fois qu'une image est publiée grâce à la fonction `AUTOINCREMENT`. L'`id_auteur` cor-

respond à l'`id_acheteur` de l'utilisateur ayant fait la publication. La date et l'heure sont elles aussi ajoutées automatiquement et correspondent au moment auquel l'utilisateur publie son post. Le `contenu_post` contient une description en moins de 300 caractères que l'utilisateur a choisi d'associer à sa photo. Enfin le `lien_image` correspond au chemin d'accès à la photo publiée.

La table `follow` est extrêmement simple mais elle nous est très pratique. À chaque fois qu'un utilisateur (le `follower`) décide de suivre le profil d'un autre utilisateur (le `followed`), une ligne est ajoutée dans la table. Cela nous permet alors d'accéder facilement à notre nombre de followers, ou bien à la liste des profils que l'on suit.

```
1 COUNT * FROM follow WHERE followed = Thomas; --compte le nombre de followers de Thomas
2 SELECT followed FROM follow WHERE follower = Maxence; --renvoie les profils suivie par
   Maxence
```

Listing 2 – Exemple de requête SQL sur la table `follow`

La table `like` est elle aussi très simple. Lorsque un utilisateur "like" un post, une ligne est ajoutée avec l'`id_post` (l'identifiant unique de la publication) et l'`id_liker` (le nom d'utilisateur ayant aimé la publication). Grâce à cette table, on peut alors savoir facilement combien de fois une publication a été likée ou combien de publications ont été likées par un utilisateur.

	<code>id_post</code>	<code>id_liker</code>
1	4	Théo
2	1	Théo
3	3	Théo
4	1	Léo
5	3	Léo

FIGURE 4 – Exemple de contenu dans la table `like`

Dans cet exemple, la publication numéro 3 a été likée deux fois, et l'utilisateur Théo a aimé trois publications.

2.1.3 Ventes

La table `transactions` présente de nombreuses similitudes avec la table `social`, elle est en quelque sorte son équivalent pour les producteurs. En effet, les producteurs ont la possibilité de faire des publications, mais celles-ci apparaissent dans un onglet séparé et sont accompagnées d'un prix. C'est pour cette raison qu'en plus de reprendre les colonnes de la table `social`, une colonne `prix_produit` est ajoutée, contenant le prix en euros de l'article mis en vente. De plus cette table comprend également une colonne `vendu` qui indique si un produit a été vendu ou pas encore.

```
1 cur.execute("CREATE TABLE transactions (
2     id_transaction INTEGER PRIMARY KEY AUTOINCREMENT,
3     id_producteur VARCHAR(20) REFERENCES Producteur(id_producteur),
4     lien_image VARCHAR(100),
5     date_offre DATE,
6     prix_produit FLOAT(7),
7     description VARCHAR(200),
8     vendu VARCHAR(100));")
```

Listing 3 – Code Python permettant de créer la table `transaction`

La table `casier` est au coeur de notre système de vente. Sa taille est fixe, elle contient 396 lignes. Tous nos points de vente se trouvent à Nancy, et ils sont au nombre de 18. Dans chacun de ces points de vente,

nous avons considéré qu'il y a 22 casiers ; d'où les 396 lignes : une par casier. À chaque casier est affecté un identifiant unique, une adresse, la ville dans laquelle il se trouve (dans notre cas c'est toujours Nancy), et un nombre de 0 à 21 unique à chaque point de vente. Enfin, une dernière colonne indique grâce à un booléen si le casier est actuellement utilisé, ou s'il est libre.

20	19	15 Avenue Anatole France Nancy	Nancy	19	0
21	20	15 Avenue Anatole France Nancy	Nancy	20	1
22	21	15 Avenue Anatole France Nancy	Nancy	21	1
23	100	118 Avenue de Boufflers Nancy	Nancy	0	0
24	101	118 Avenue de Boufflers Nancy	Nancy	1	0

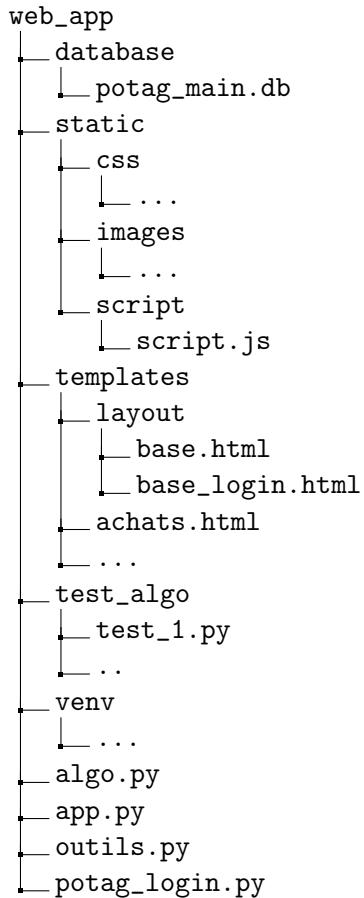
FIGURE 5 – Extrait de la table `casiers`

On peut voir un casier vide et deux casiers pleins au point de vente 0 du 15 Avenue Anatole France ainsi que deux casiers vides au point de vente 1 situé au 118 Avenue de Boufflers.

La table `file_attente` permet de savoir quels produits ont été achetés. Lorsqu'un utilisateur achète un produit, on insère dans la colonne `acheteur` le nom de l'utilisateur et dans la colonne `vendeur`, le nom du producteur ayant mis en vente le produit. On considère donc qu'un acheteur ne peut acheter qu'un produit à la fois, et que de même, un producteur ne peut vendre qu'un produit à la fois.

2.2 Serveur web

Notre serveur web fonctionne avec le framework Flask en python. Avec Flask, on peut accéder à des pages html dynamiques. Pour cela on utilise des fonctions python qui renvoient les fichiers html avec des variables. L'arborescence du projet est la suivante :



```

potag_preferences.py
requirements.txt
potag_social.py
potag_vente.py

```

Le serveur web fonctionne avec un fichier flask_app : `app.py`. Pour ne pas avoir toutes les fonctions dans le fichier `app.py`, les fonctions sont séparées dans 6 fichiers python en fonction de leurs utilisations :

```

— algo.py
— outils.py
— potag_login.py
— potag_social.py
— potag_vente.py
— potag_preferences.py

```

On importe ces 6 modules dans `app.py`.

2.2.1 app.py

L'`app.py` est celle lancée pour faire fonctionner le serveur web. Depuis le module flask, on importe Flask qui permet de configurer le serveur.

```

1 app = Flask(__name__)
2 app.config['SECRET_KEY'] = 'UneCléSecrète',
3 app.config["SESSION_PERMANENT"] = False
4 app.config["SESSION_TYPE"] = "filesystem"
5 app.config['UPLOAD_FOLDER'] = 'static/images/'

```

Listing 4 – Code Python permettant de configurer le serveur

Il faut ensuite déclarer les routes à flask. Elles sont rangées en 4 catégories : login, social, ventes et préférences.

```

1 # Login
2 import potag_login
3 app.add_url_rule('/login', view_func=potag_login.login, methods=["POST", "GET"])
4 ...
5
6 # Social
7 import potag_social
8 app.add_url_rule('/publications', view_func=potag_social.publications, methods=["POST", "GET"])
9 ...

```

Listing 5 – Code Python permettant de déclarer la route `login` et `publications`

Dans l'exemple précédent, la première route concernée est `/login` et la fonction correspondante est `potag_login.login()`. L'appel à une route sur le navigateur lance la fonction python associé. On remarque que les fonctions sont importées depuis les modules cités précédemment. Les informations de connexion sont enregistrées grâce au module `flask.session`.

```

1 session["name"]="Didier"
2 session["compte"]="producteur"

```

Listing 6 – Code Python permettant d'ajouter un élément dans session

2.2.2 potag_login.py

C'est dans ce module que sont regroupées toutes les fonctions liées à la connexion :

- `signup()` pour s'inscrire

- `passwordchoice()` pour choisir un mot de passe
- `login()` pour se connecter
- `password()` pour indiquer le mot de passe de connexion
- `logout()` pour se déconnecter
- `forgot()` en cas d'oubli de mot de passe

Elles permettent de créer un compte et de se connecter. Prenons comme exemple la fonction `login()`.

```

1 def login():
2     session.clear()
3     if request.method == "POST":
4         session["name_temp"] = corrige_espace(request.form.get("name"))
5         db=sqlite3.connect(DATABASE)
6         cursor=db.cursor()
7         cursor.execute("SELECT id_acheteur FROM acheteur WHERE id_acheteur= ?", (session
8 ["name_temp"], ))
9         login=cursor.fetchall()
10        if login==[]:
11            cursor.execute("SELECT id_producteur FROM producteur WHERE id_producteur= ?"
12 , (session["name_temp"], ))
13            session["compte"]="producteur"
14        else:
15            session["compte"]="acheteur"
16        db.close()
17        return redirect("/password")
18    return render_template("login.html")

```

Listing 7 – Fonction Python `login`

Pour pouvoir afficher la page html il nous faut le module `flask.render_template`. Notre fonction retourne donc `render_template("login.html")` ce qui permet de renvoyer le fichier html. En cas de requête POST, c'est-à-dire quand on clique sur "valider", on ajoute la donnée de "nom d'utilisateur" dans session. On détermine si c'est un acheteur ou un producteur puis on ajoute l'information dans session. Enfin, on redirige vers /password. On interroge la base de donnée grâce au module `sqlite3`. Les pages html, qui sont dans templates/, viennent s'ajouter à `base_login.html` qui se trouve dans templates/layout/. Elles utilisent `inscription.css` dans static/css/.

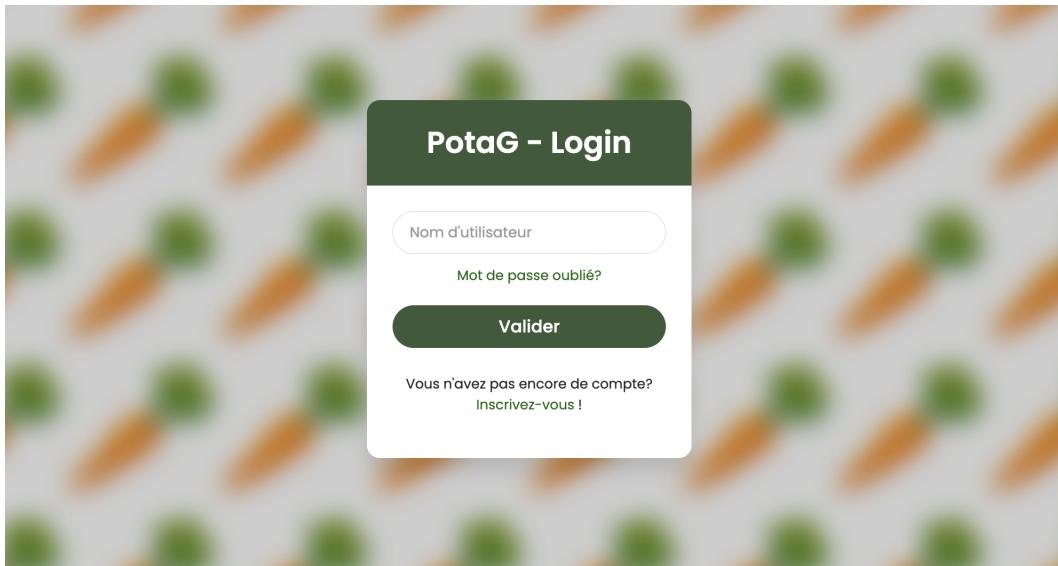


FIGURE 6 – Formulaire de la page `login.html`

2.2.3 potag_social.py

C'est dans ce module que sont regroupées toutes les fonctions liées à la partie réseau social de notre application :

- publications() pour voir les publications d'autres utilisateurs
- publicationliked() pour voir les publications likées
- addpublication() pour publier une image
- profil() pour voir son profil
- deletepost() pour supprimer une publication
- visitprofile(username) pour voir le profil d'autres utilisateurs
- profilsfollowed() pour voir les profils suivis
- followers() pour voir les profils qui nous suivent
- recherche() pour effectuer une recherche

Prenons comme exemple la fonction publications. C'est une fonction assez lourde car elle permet l'affichage des publications de chaque utilisateur accompagnées de leurs likes.

```
1 def publications():
2     active=recup_infos()
3     if not active:
4         return redirect('/')
5     info=session["infos_perso"]
6     if request.method=="POST":
7         idpost=request.form.get("idpost")
8         db = sqlite3.connect(DATABASE)
9         cursor= db.cursor()
10        cursor.execute("SELECT id_post FROM like WHERE (id_post, id_liker) = (?, ?)", (
11            idpost, session["name"]))
12        res=cursor.fetchall()
13        if len(res)>0:
14            cursor.execute("DELETE FROM like WHERE (id_post, id_liker) = (?, ?)", (
15            idpost, session["name"]))
16        else:
17            cursor.execute("INSERT INTO like (id_post, id_liker) VALUES (?, ?)", (idpost,
18                session["name"]))
19            db.commit()
20            db.close()
21        db = sqlite3.connect(DATABASE)
22        cursor= db.cursor()
23        cursor.execute("SELECT id_post, id_auteur, lien_image, contenu_post, date_post,
24        heure_post FROM social ORDER BY date_post")
25        posts_temp=cursor.fetchall()
26        posts=[]
27        for post in posts_temp:
28            post = list(post)
29            cursor.execute("SELECT count(id_liker) from like where id_post=?;",[post[0]])
30            likes = cursor.fetchall()[0][0]
31            post.append(likes)
32            cursor.execute("SELECT id_post FROM like WHERE (id_post, id_liker) = (?, ?)", (
33            post[0], session["name"]))
34            res=cursor.fetchall()
35            if len(res)>0:
36                post.append("true")
37            else:
38                post.append("false")
39            posts.append(post)
40        db.close()
41        posts.reverse()
```

```
37     return render_template("publications.html", compte=info, posts=posts)
```

Listing 8 – Fonction Python publications

On remarque tout d'abord l'utilisation de outils.recup_info qui permet, en plus d'ajouter les informations de l'utilisateur dans session, de vérifier que l'utilisateur est connecté. En effet si session["name"] n'existe pas alors recuper_info renvoie False et on est renvoyé vers /login. Ce test est réalisé dans chaque fonction car il évite toutes les erreurs liées à une personne non connectée. Si l'utilisateur était bien connecté, on récupère des infos sur son profil dans la variable info.

Si une requête de type POST est effectuée sur la page, c'est que le bouton de "like" a été cliqué par l'utilisateur. Il existe alors deux cas de figure :

- La publication était déjà likée, l'utilisatuer veut donc retirer son like,
- La publication n'était pas likée, l'utilisateur vient donc de la liker.

Pour savoir dans quel cas nous sommes, il nous faut vérifier si la base de données contient le like entre l'utilisateur et la publication en question. Puis selon la situation, il nous faut alors ajouter ou supprimer cette ligne dans la table like.

Dans tous les cas, la page doit afficher les publications. On effectue donc une requête dans la base de données pour récupérer toutes les informations de toutes les publications, que l'on trie par ordre chronologique. Pour chacune de ces publications, on va ensuite calculer combien de fois elles ont été likées et si elles sont likées par l'utilisateur connecté. Une fois toutes ces informations relevées, on retourne la page publications.html en affichant les publications par ordre antichronologique pour voir les publications les plus récentes en premier.

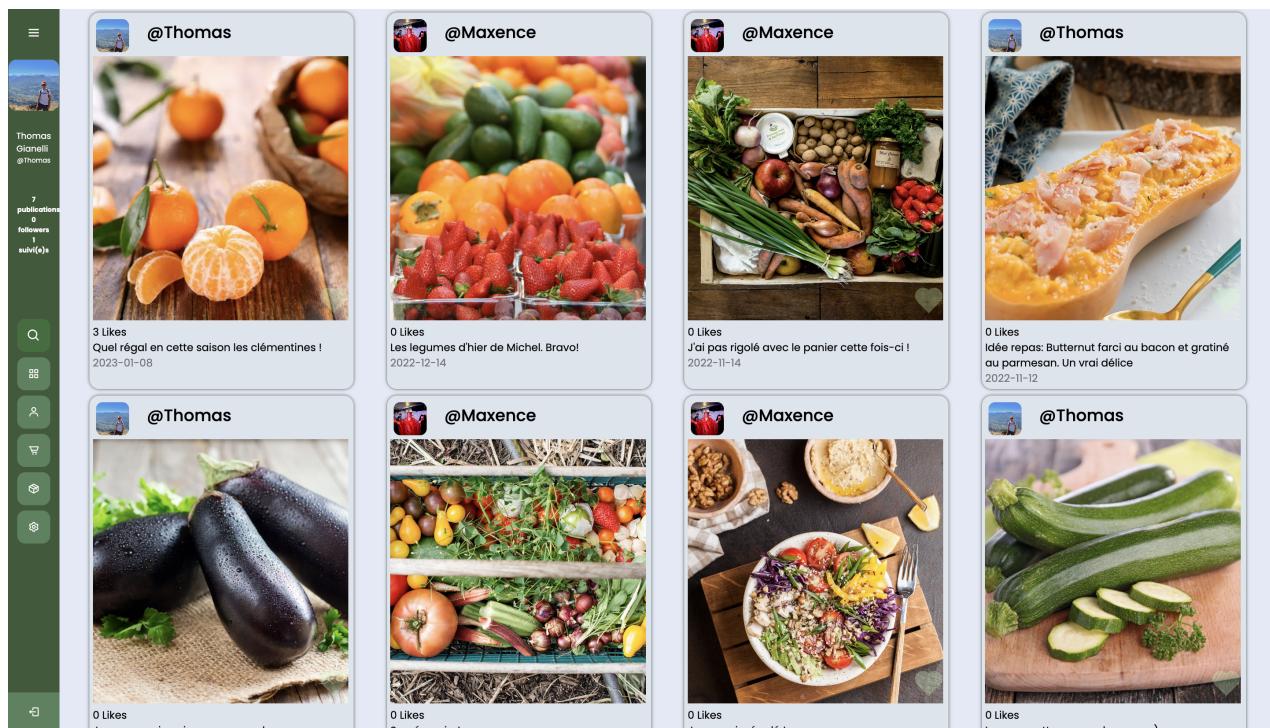


FIGURE 7 – Rendu de la page publications.html

Voici donc un exemple de rendu de la page publications.html.

2.2.4 potag_vente.py

Nous avons regroupé dans ce module les fonctions liées à la vente :

- ventes() pour voir les légumes vendus par les producteurs

- addventes() pour ajouter des légumes
- lockers() pour voir l'emplacement des lockers
- achats() pour voir ses achats

Les fonctions ventes et addventes fonctionnent sur le même principe que publications et addpublications, à la différence que les likes deviennent des achats. La fonction achats fonctionne sur le même principe que followers.

Lors d'un achat, l'objet n'est plus visible pour les autres, l'acheteur et le vendeur sont ajoutés en file d'attente et le nombre d'achats augmente.

Un des avantages de flask est l'utilisation de boucle dans l'html afin d'afficher un nombre de publications ou d'achats qui dépend de la base de donnée.

```

1  {%for achat in achats%}
2      <div class="ligne">
3          <div class="photo">
4              
7              <a href="/visitprofile{{achat[1]}}">{{achat[1]}}</a>
8          </div>
9          <div class="nb_publications">
10             description: {{achat[2]}}
11         </div>
12         <div>
13             <form action="/achats" method="POST">
14                 <div class="annulation">
15                     <input type="hidden" name="idpost" value={{achat[0]}} >
16                     <button type='submit'>Annuler</button>
17                 </div>
18             </form>
19         </div>
20     </div>
21 {%endfor%}
```

Listing 9 – Boucle dans le code html de achats.html

Ici le bloc est répété pour chacun des éléments de la liste achats. On peut aussi appeler des variables avec {{variable}}.



FIGURE 8 – Exemple de rendu de la page achats.html

2.2.5 potag_preferences.py

Enfin, dans ce module nous avons regroupé toutes les fonctions relatives aux paramètres de l'utilisateurs :

- parametres() pour voir les différents paramètres
- profile_picture() pour changer sa photo de profil
- changeaddress() pour changer son adresse
- changepassword() pour changer son mot de passe
- deleteaccount() pour supprimer son compte

Nous allons cette fois-ci un petit peu détailler la fonction `profile_picture()` car nous n'avons pas encore présenté la façon de charger une image sur notre application.

```

1 def profile_picture():
2     active=recup_infos()
3     if not active:
4         return redirect('/')
5     if request.method == "POST":
6         file = request.files['file']
7         filename=file.filename # On note le nom du fichier envoyé par l'utilisateur
8         if filename=='': # Si le nom est vide c'est qu'aucun fichier n'a été envoyé
9             error="Veuillez sélectionner une image"
10            return render_template("profilepicture.html", error=error)
11        file.save(UPLOAD_FOLDER+ 'profils/' + 'pp' + session["name"] + '.png') # On
12        enregistre le fichier en le renommant
13        db = sqlite3.connect(DATABASE)
14        cursor= db.cursor()
15        chemin= UPLOAD_FOLDER + 'profils/' + 'pp' + session["name"]+'.png'
16        if session["compte"]=="acheteur":
17            cursor.execute("UPDATE acheteur SET profile = ? WHERE id_acheteur = ?", (
18            chemin, session["name"]))
19        else:
20            cursor.execute("UPDATE producteur SET profile = ? WHERE id_producteur = ?", (
21            chemin, session["name"]))
22        db.commit() # On insère dans la base de données le chemin du fichier
23        db.close()
24        return redirect("/profil")
25    info=session["infos_perso"]
26    return render_template("profilepicture.html", compte=info)

```

Listing 10 – Fonction Python `profile_picture.html`

Comme presque toutes les fonctions de notre application, on commence par vérifier si un utilisateur est connecté. Dans ce cas, on récupère des infos relatives à son profil.

Si une requête de type `POST` est effectuée sur la page, on récupère le fichier envoyé par l'utilisateur et le nom du fichier. Si ce nom est la chaîne vide, c'est qu'aucun fichier n'a en réalité été envoyé par l'utilisateur. Dans ce cas on l'invite à envoyer de nouveau un fichier. Dans le cas contraire, c'est que l'utilisateur a envoyé un fichier au format `.jpeg` ou `.png` car seules ces formats sont acceptés par notre code `html`. On enregistre alors le fichier dans un dossier prédéfini en prenant soin de le renommer selon un schéma bien précis. Une fois le fichier enregistré, on insère dans la base de données le chemin menant à l'image, afin de pouvoir la récupérer facilement.



FIGURE 9 – Page de changement de la photo de profil

2.3 Algorithmes

Dans notre projet PotaG, lors de l'achat du produit d'un producteur par un acheteur, un casier doit leur être attribué afin de procéder à l'échange. Il ne faut pas que le casier attribué soit trop loin pour les deux personnes concernées. Aussi, il faudrait essayer d'harmoniser les distances entre chaque partie et le casier, et cela pour plusieurs transactions à la fois. C'est pourquoi nous avons mis en place une liste d'attente qui est traitée par l'algorithme une fois remplie. L'algorithme en question doit donner la meilleure suite de casiers pour les couples de la liste d'attente.

2.3.1 Notion d'ordre

Un couple est constitué de l'id d'un acheteur et d'un vendeur : (Thomas, Michel). Nous allons d'abord récupérer la distance pour l'acheteur et pour le vendeur à chaque casiers. Nous aurons donc une liste avec des tuples contenant : la distance acheteur, la distance vendeur, le numero de casier :

$$[(1, 3, 0), (2, 3, 1), \dots, (a_1, a_2, a_3)]$$

on répète cela pour chacun des couples de la liste d'attente. Il nous faut ensuite trier ces listes. Mais comment juger de l'ordre pour des tuples (en sachant que le numéro de casier n'entre pas en compte). L'ordre lexicographique (ordre du premier élément puis du deuxième) n'est pas judicieux car il pénalise le vendeur. Après quelques tests, un ordre semble intéressant : $\sqrt{a_1^2 + a_2^2}$.

En posant

- $a = (a_1, a_2)$
- $b = (b_1, b_2)$
- $\sqrt{a_1^2 + a_2^2} < \sqrt{b_1^2 + b_2^2} \implies a < b$

On obtient un ordre qui favorise les petites distances mais aussi les petits écarts entre a_1 et a_2 .

2.3.2 Tri Fusion

Nous avons besoin de trier les listes car nous voulons savoir quel casier est le meilleur pour un couple acheteur-vendeur donné. Si le premier casier n'est pas disponible, on choisira le suivant et ainsi de suite. Parmi les tris que l'on connaît, le tri fusion se place comme un bon candidat : il est simple à mettre en œuvre, présente une complexité optimale en $n \times \log(n)$ pour une liste de taille n et fonctionne parfaitement avec des listes. Il est basé sur le principe algorithmique diviser pour régner. Le tri fusion fonctionne de manière récursive : pour trier une liste A, on trie la moitié droite et la moitié gauche puis on les fusionne. Un tableau à un seul élément est déjà trié.

```
1 def tri_fusion(tableau):  
2     if len(tableau) <= 1:  
3         return tableau  
4     pivot = len(tableau)//2  
5     gauche = tri_fusion(tableau[:pivot])  
6     droite = tri_fusion(tableau[pivot:])  
7     return fusion(gauche,droite)
```

Listing 11 – Appel recursif

Pour fusionner deux tableaux triés, on parcourt les deux tableaux parallèlement en ajoutant dans le tableau final le plus petit des deux éléments. C'est donc ici l'on implémente l'ordre choisi.

```
1 def fusion(tableau1,tableau2):  
2     indice_tableau1 = 0  
3     indice_tableau2 = 0  
4     taille_tableau1 = len(tableau1)  
5     taille_tableau2 = len(tableau2)  
6     tableau_fusionne = []
```

```

7     while indice_tableau1 < taille_tableau1 and indice_tableau2 < taille_tableau2:
8         val1 = sqrt(tableau1[indice_tableau1][0]**2 + tableau1[indice_tableau1
9             ][1]**2)
10        val2 = sqrt(tableau2[indice_tableau2][0]**2 + tableau2[indice_tableau2
11             ][1]**2)
12        if val1 < val2:
13            tableau_fusionne.append(tableau1[indice_tableau1])
14            indice_tableau1 += 1
15        else:
16            tableau_fusionne.append(tableau2[indice_tableau2])
17            indice_tableau2 += 1
18        while indice_tableau1 < taille_tableau1:
19            tableau_fusionne.append(tableau1[indice_tableau1])
20            indice_tableau1+=1
21        while indice_tableau2 < taille_tableau2:
22            tableau_fusionne.append(tableau2[indice_tableau2])
23            indice_tableau2+=1
24    return tableau_fusionne

```

Listing 12 – Fusion de deux tableaux triés selon l'ordre choisi précédemment

Dans notre projet l'appel est fait sur des listes de taille n = "le nombre de casier". Dans notre exemple sur Nancy il y a 18 casiers donc $n = 18$.

2.3.3 Backtracking

Il nous faut maintenant trouver la meilleure suite de casiers afin que chacun des couples se voit attribué le casier avec le plus faible score ($\sqrt{distance_{acheteur}^2 + distance_{vendeur}^2}$) possible. L'algorithme est le suivant pour 3 couples, deux casier, un d'une place et l'autre de deux places :

- On suppose que l'on a pour chaque couple la liste des distances aux casiers triée. Le troisième élément étant le numéro du casier (0 ou 1 dans notre exemple).

	Pierre-Paul	Théo-Antoine	Thomas-Michel
Choix 0	(1,4,0)	(2,4,1)	(1,3,0)
Choix 1	(2,8,1)	(3,5,0)	(2,4,1)

- On choisit d'abord le premier choix pour chacun.

	Pierre-Paul	Théo-Antoine	Thomas-Michel
	(1,4,0)	(2,4,1)	(1,3,0)
	(2,8,1)	(3,5,0)	(2,4,1)

- On parcourt ensuite les couples. On vérifie qu'il y a de la disponibilité dans le casier voulu. Si c'est le cas on passe au couple suivant. Sinon on regarde si un autre couple avant celui-ci n'a pas déjà pris ce casier. Si c'est le cas, c'est celui qui a le meilleur score qui le garde, et l'autre passe à son deuxième choix. Sinon, c'est qu'il n'y a plus de place, alors le couple passe à son deuxième choix. On valide le choix de Pierre-Paul puis celui de Théo-Antoine. Le choix de Thomas-Michel ne peut pas être validé car la seule place du casier 0 est déjà attribuée à Pierre-Paul. Comme on a $\sqrt{1+3^2} < \sqrt{1+4^2}$, c'est à Thomas-Michel que le casier est attribué. Pierre-Paul passe donc à son deuxième choix et on relance l'algorithme

	Pierre-Paul	Théo-Antoine	Thomas-Michel
	(1,4,0)	(2,4,1)	(1,3,0)
	(2,8,1)	(3,5,0)	(2,4,1)

- L'algorithme se finit quand on a parcouru toute liste puisque dans ce cas c'est que les choix sont tous valides. Dans notre exemple on retourne donc 1,1,0

Pierre-Paul	Théo-Antoine	Thomas-Michel
1	1	0

Pour le code python on a besoin d'une fonction `premier_meme` pour trouver le premier couple ayant le casier voulu.

```

1 def premier_meme(liste,i):
2     for j in range(len(liste)):
3         if liste[i]==liste[j] and j!=i:
4             return True,j
5     return False,0

```

Listing 13 – Fonction python `premier_meme`

Cet algorithme renvoie True et l'indice de la première apparition si il y en a une, False et 0 sinon. Ensuite il faut une fonction récursive qui applique le parcours de la liste jusqu'à pouvoir la parcourir en entier.

```

1 def rec_aux(liste,dispo,choix,fin):
2     dispo_cop = dispo[:]
3     for couple in range(len(liste)):
4         testeur = premier_meme(liste,couple) # Autre couple qui a le même casier
5         if dispo_cop[fin[couple]]>0: #Il reste de la place, il garde ce casier
6             dispo_cop[fin[couple]]-=1
7         elif testeur[0]: #Sinon, le plus faible score garde le casier
8             ach1 = liste[couple][choix[couple]][0]
9             vend1 = liste[couple][choix[couple]][1]
10            ach2 = liste[testeur[1]][choix[testeur[1]]][0]
11            vend2 = liste[testeur[1]][choix[testeur[1]]][1]
12            val1 = sqrt( ach1**2 + vend1**2)
13            val2 = sqrt( ach2**2 + vend2**2)
14            if val1>val2: # On teste le score
15                choix[couple]+=1
16                fin[couple] = liste[couple][choix[couple]][2]
17            else:
18                choix[testeur[1]]+=1
19                fin[testeur[1]] = liste[testeur[1]][choix[testeur[1]]][2]
20            return rec_aux(liste,dispo,choix,fin)
21        else: #Plus de place et personne n'a le casier: passe au choix suivant
22            choix[couple]+=1
23            fin[couple] = liste[couple][choix[couple]][2]
24            return rec_aux(liste,dispo,choix,fin)
25    return fin #On a fini une boucle donc l'attribution des casiers est valide

```

Listing 14 – Implémentation de l'algorithme récursif

On commence l'algorithme avec le premier choix de chacun des couples. L'appel à la fonction recursive se fait donc :

```

1 def backtrack_meilleur(liste,dispo):
2     choix_numero = [0 for i in range(len(liste))]
3     casiers = [liste[i][0][2] for i in range(len(liste))]
4     return rec_aux(liste,dispo,choix_numero,casiers)

```

Listing 15 – Implémentation de l'algorithme récursif

3 Tests et Performances

3.1 Tri Fusion

3.1.1 Tests

Pour tester l'algorithme, il faut dans un premier temps générer des listes de couples de réels. Pour cela, on utilise la fonction `uniform` du module `random`. Elle nous permet de générer un réel entre 0 et x aléatoirement, avec x la borne max choisie. On construit donc la fonction `gen_liste_aleat` qui va générer des listes de couples de réels aléatoires.

```
1 def gen_liste_aleat(n,x):
2     #liste (taille n) de couple de réel aléatoire entre 0 et x m
3     liste = [[random.uniform(0,x),random.uniform(0,x)] for i in range(n)]
4     return liste
```

Listing 16 – Fonction `gen_liste_aleat.py`

Dans un deuxième temps, il faut vérifier que le tri fonctionne. Pour ça, on utilise la fonction `sorted` de python, avec `key=norme`, ce qui donne le même résultat que notre algorithme de tri.

On va procéder à deux tests différents :

- un test dans les conditions réelles de notre application : un nombre de transactions et d'adresses de casiers cohérentes ainsi que des distances qui restent locales.
- un test dans des conditions plus générales de l'algorithme.

Dans le cas réel, on choisit 16 adresses de casiers et 50 personnes soit 800 couples à trier. Les distances restent inférieures à 10 km.

```
1 from test_algo import *
2 """
3 test de tri fusion dans le cas reel
4 """
5 t = [None for x in range(800)]
6 temps_exec = [None for x in range(800)]
7 correction = True
8
9 for i in range(0,800):
10     # generation de la liste aleatoire de taille i, de reel entre 0 et 10k
11     t[i] = gen_liste_aleat(i,10000)
12
13     # teste de correction
14     temp_correction =(tri(t[i])==trie_fusion_liste_distance(t[i]))
15     if temp_correction==False:
16         correction = False
17
18 #correction de l'algo
19 def test_cas_reel():
20     assert correction
```

Listing 17 – Fichier de test : `test_1.py`

Le code `test_1.py` permet de réaliser le test dans le cas réel. La correction est la variable importante : elle reste à vrai si aucune faute n'est commise, elle passe à faux si au moins un test est faux. La fonction `test_cas_reel` est invoqué par `pytest` avec la commande : `pytest -v test_1.py`

```

eleve@VM-TNCY:~/Bureau/working_dir/pvcc-grp_13/web_app$ pytest -v test_1.py
=====
platform linux -- Python 3.8.10, pytest-7.2.0, pluggy-1.0.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/eleve/Bureau/working_dir/pvcc-grp_13/web_app
collected 1 item

test_1.py::test_cas_reel PASSED [100%]

=====
1 passed in 0.75s =====

```

FIGURE 10 – Invoquation de `pytest` pour tester le tri fusion

Dans le cas général, on va aller jusqu'à des listes de 20 000 éléments (chaque liste générée a 10 éléments de plus que la précédente) et des distances de l'ordre de la centaine de kilomètres. De même on teste avec `pytest` (fichier `test_2.py`), le test dure 1 minute 30 environ et nous donne un résultat positif.

3.1.2 Performance

On récupère les temps d'exécution dans le cas réel et dans le cas général. On obtient :

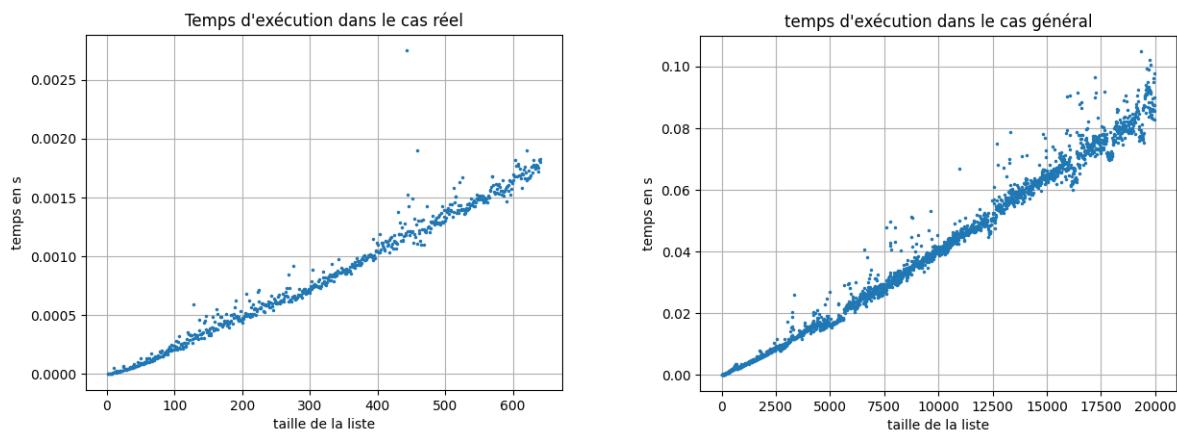


FIGURE 11 – Temps d'exécution en fonction de la taille de la liste

Les temps d'exécutions sont cohérents avec la complexité du tri fusion en $n \times \log(n)$. Cette complexité est optimale et conviendra pour une application à plus grande échelle. Par contre, il faut noter que la complexité spatiale est de l'ordre de n . Cela ne posera pas problème puisque la taille des listes n'augmentera pas, le nombre de point de vente dans une ville étant limité.

3.2 Backtracking

3.2.1 Tests

Nous allons utiliser une propriété intéressante : si pour chaque point de vente, il y a plus de casiers disponible que de transactions, chaque couple se voit attribuer le casier le plus proche. Il est important que l'algorithme fonctionne correctement dans ce cas particulier.

Afin de mener au mieux nos tests, nous avons modifié la fonction `traite_liste` pour obtenir la distance au casier le plus proche et la distance au casier attribué par l'algorithme. Également il nous fallait pouvoir accéder aux adresses et disponibilités des casiers ainsi qu'aux adresses des acheteur et vendeur sans passer par une base de données. La fonction modifiée est la suivante :

```

1 def traite_file_test_distance(file,dispo,liste_casier):
2     distance_par_couple = []
3     distance_min=0
4     for couple in file:
5         liste_distance=[]
6         for i in range(len(liste_casier)):
7             casier = liste_casier[i]
8             prem = get_distance(couple[0],casier)
9             deux = get_distance(couple[1],casier)
10            liste_distance.append([prem,deux,i])
11    liste_distance = trie_fusion_liste_distance(liste_distance)
12    distance_min+=liste_distance[0][0]+liste_distance[0][1] #somme des plus petites
distances parcouru
13    distance_par_couple.append(liste_distance)
14    casiers=backtrack_meilleur(distance_par_couple,dispo)
15    distance_final = 0
16    for i in range(len(casiers)):
17        for e in distance_par_couple[i]:#liste des distances pour le couple e
18            if e[2]==casiers[i]:
19                distance_final+= e[0]+e[1]#distance choisi par l'algo de backtracking
20    return distance_min,distance_final

```

Listing 18 – Fonction `traite_file` modifiée pour les tests

La fonction renvoie `distance_min` et `distance_final` :

- `distance_final` correspond à la somme des distances parcourues par chaque couple acheteur/vendeur
- `distance_min` correspond à la somme des distances minimales pour chaque couple.

Pour le premier test, on va vérifier que ces deux distances sont égales.

Pour effectuer les tests de la fonction il nous faut des adresses pour les acheteurs, les vendeurs, mais aussi pour les casiers. Pour les créer, nous avons déterminé les coordonnées gps du centre de Nancy et choisi des points gps au hasard dans un périmètre donné :

```

1 lat = 48.692054 #latitude d'un point central de Nancy
2 lon = 6.184417 #longitude d'un point central de Nancy
3 periometre = 0.05 #distance des utilisateurs au point gps {lat,lon}
4 point = [lat+periometre*(2*(random.random()-1)),lon+periometre*(2*(random.random()-1))]

```

Listing 19 – création de points gps autour du centre de Nancy

Cette méthode pose problème car elle peut générer des points non-accessible par un itinéraire Google map : il faut donc vérifier que le point créé permet de déterminer un itinéraire. On fait donc appel à `get_distance` en levant une erreur en cas d'échec. En prévision de tous les tests on génère 1000 points gps, stockés dans le fichier `coord_gps_nancy.txt`. Ils sont accessibles en utilisant la fonction `get_coord_gps`.

On peut désormais utiliser ces points gps pour tester une première fois la fonction sur le cas particulier décrit plus haut. On définit le nombre de transactions à 10, la limite pour la file d'attente de l'algorithme, et on incrémenté le nombre de casiers tout en s'assurant que chaque adresse de casier a plus de disponibilités que de transactions. Ce test est effectué dans le fichier `test_3.py` par `pytest` et prend 150 secondes environ.

Pour aller plus loin dans les tests, on va comparer la `distance_final` et la `distance_min` pour vérifier que notre algorithme choisit des solutions optimales : On va pouvoir mesurer l'écart à la distance minimal pour chaque couple :

```

1 def delta_distance(k):
2     #k est le nombre de delta différent

```

```

3     distances_finales=[]
4     distances_min=[]
5     gps_data = get_coord_gps()
6     for l in range(k):
7         casier =[random.choice(gps_data) for i in range(5)] #les adresses des casiers
8         couple =[[random.choice(gps_data),random.choice(gps_data)] for i in range(10)] #
9         un couple = une transaction acheteur vendeur , 10 transactions
10        dispo = [2 for i in range(5)]
11        temp_test = traite_file_test_distance(couple,dispo,casier)#temp test corr
12        distances_finales.append(temp_test[1])
13        distances_min.append(temp_test[0])
14    return  distances_finales,distances_min

```

Listing 20 – `test_4.py` permet de générer les courbes de `distance_final` et `distance_min`

Le code est similaire à celui de `test_3.py`, mais on teste pour 10 transactions et 5 casiers repartis sur 2 emplacements à chaque fois, et ce k fois. Chaque itération prend en moyenne 1 minute, ce qui est tout à fait acceptable pour notre application puisque cette fonction ne doit pas empêcher les utilisateurs de parcourir le site, c'est une tâche de fond. On obtient les écarts de distance suivant :

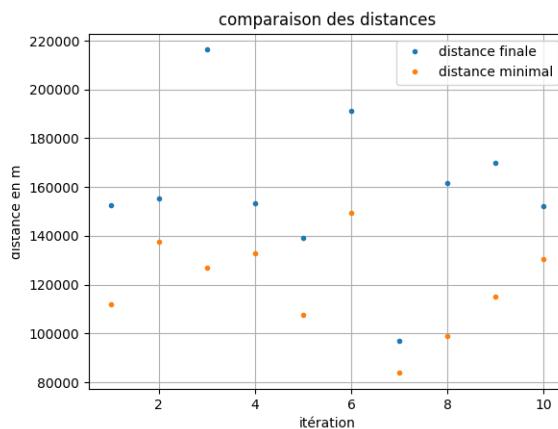


FIGURE 12 – `distance_final` et `distance_min`

Les distances sont un peu en dessous de 10 km, ce qui est à la limite des zones ciblées par notre application, et les casiers ne sont pas placés de manière stratégique, ce qui explique les grandes distances.

3.2.2 Performance

Nous allons observer le comportement de l'algorithme en fonction du nombre de casiers et de personnes puis comparer la complexité obtenue avec la théorique après l'avoir définie.

Dans un premier temps, afin de mesurer et d'évaluer le temps d'exécution pour un nombre de casiers croissant nous allons utiliser la fonction `process_time` du module `time`. Elle va nous permettre d'obtenir le temps qu'a passé le processeur sur notre algorithme :

```

1 start = time.process_time()
2 temp_test = traite_file_test_distance(couple,dispo,casier)
3 end = time.process_time()-start

```

Listing 21 – Mesure du temps d'exécution de `traite_file_test_distance`

Le code pour la mesure du temps se trouve dans `test_5.py`. Il donne :

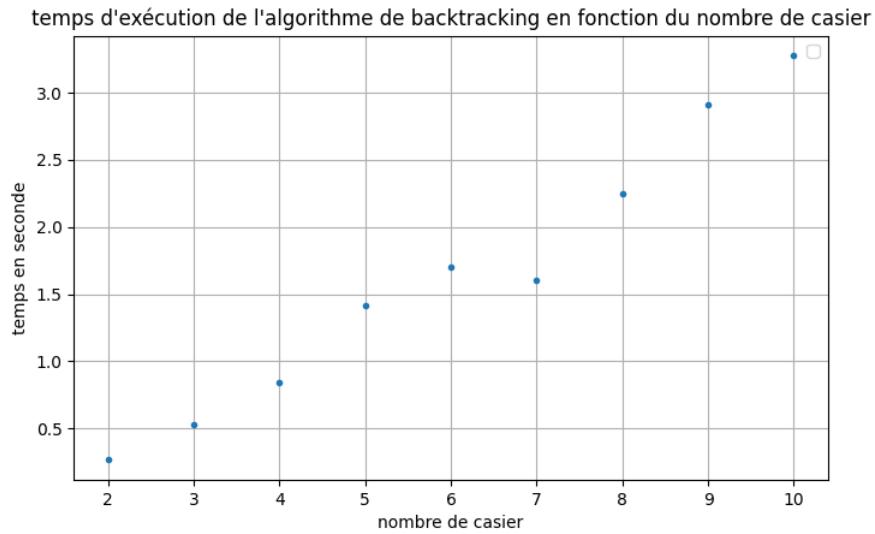


FIGURE 13 – Temps d'exécution processeur du backtracking en fonction du nombre de casiers

Cependant, le temps total d'exécution, en comptant le temps d'attente à l'API peut être mesuré à l'aide de `perf_counter` du module `time` :

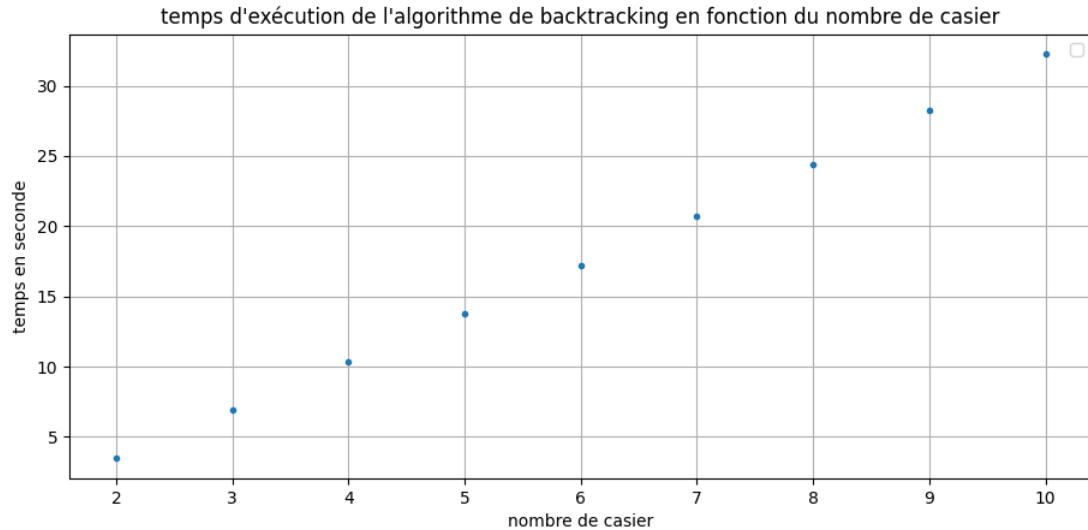


FIGURE 14 – temps d'exécution total du backtracking en fonction du nombre de casiers

La partie lente de l'algorithme provient de l'API Google Maps, qui multiplie par 10 le temps d'exécution.

On peut aussi mesurer le temps moyen pour des valeurs proches de notre utilisation, soit 9 casiers et 10 transactions :

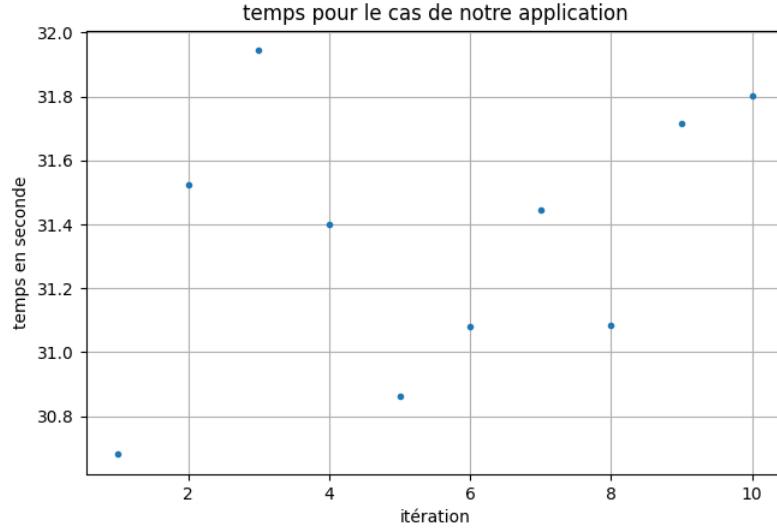


FIGURE 15 – temps d'exécution total du backtracking dans le cas de notre application

Il nous faut ensuite déterminer la complexité théorique de notre algorithme de backtracking afin de la comparer à celle observée. Dans le meilleur des cas on parcourt une première fois la liste et tous les casiers choisis sont valides, la complexité est de n . Dans le pire des cas, à chaque passage le choix du dernier couple n'est pas valide. On rappelle donc la fonction juste avant la fin du parcours de la liste. On suppose aussi que toutes les personnes obtiennent leur dernier choix, ce qui donne $n \times n \times \text{nombre_de_casiers}$, donc de l'ordre de n^2 . Aussi l'appel à `premier_meme` est de complexité n dans le pire des cas avec le parcours de la liste. Au final la fonction est de complexité n^3 dans le pire des cas.

4 Gestion de projet

4.1 Cahier des charges fonctionnel

Les différentes parties de ce cahier des charges :

- Groupe d'expression du besoin
- Historique des modifications et révisions de ce document
- Résultats et changements attendus
- Parties prenantes
- Contraintes de planning

4.1.1 Auteurs de ce cahier des charges / groupe d'expression du besoin

Nom / email	Qualité / rôle
Maxence BEKHEDDA / maxence.bekhedda@telecommancy.eu	Membre de l'équipe projet
Léo GERMAIN / leo.germain@telecommancy.eu	Membre de l'équipe projet
Thomas GIANELLI / thomas.gianelli@telecommancy.eu	Membre de l'équipe projet
Théo WALTER / theo.walter@telecommancy.eu	Membre de l'équipe projet
Olivier FESTOR / olivier.festor@telecommancy.eu	
Anne-Claire HEURTEL / anne-claire@telecommancy.eu	
Gérald OSTER / gerald.oster@telecommancy.eu	Commanditaires

4.1.2 Historique des modifications et révisions de ce document

n° de version	Date	Description et circonstances de la modification
V1	19/10/2022	1ère version du cahier des charges en vue de la 1ère soutenance
V2	10/11/2022	Utilisation concrète des outils de gestion de projet

4.1.3 Résultats et changements attendus

Livrables :

- **Premier rapport**, écrit en LaTex, en trois sections : l'état de l'art, le concept et la gestion de projet.
- **Présentation pour la première soutenance** sur l'état de l'art, le concept et la gestion de projet.
- **Second rapport** avec une section pour l'implémentation de la base de données, du serveur web et des algorithmes de traitement, une section consacrée aux tests et aux performances puis une dernière section pour la gestion de projet.
- **Présentation pour la seconde soutenance** sur l'application et la gestion de projet puis démonstration des fonctions.

4.1.4 Parties prenantes

L'objectif de cette partie est de recenser exhaustivement tous les acteurs concernés par le projet et ses conséquences.

Commanditaires :

- initiateurs du projet : Olivier FESTOR, Anne-Claire HEURTEL, Gérald OSTER

Équipe de réalisation :

- Maxence BEKHEDDA
- Léo GERMAIN
- Thomas GIANELLI
- Théo WALTER

Autres parties prenantes :

- utilisateur finaux : particulier souhaitant vendre ses fruits et légumes en circuit-court, toute personne souhaitant acheter des fruits et légumes produit par des particuliers à proximité.
- soutiens et opposants au projet : les autres groupes formés d'élèves de 1A créant des applications web répondant au même besoin
- personnes-ressources et experts : Christophe BOUTHIER, Olivier FESTOR, Anne-Claire HEURTEL, Gérald OSTER

4.1.5 Contraintes de planning

- 1er rapport : 20/10/2022 à 18h
- 1ère soutenance : 22/10/2022 à 9h
- Date de rendu du projet : 11/01/2023 à 12h
- Date de soutenance : 19/01/2023 à 16h

4.2 To-Do List

4.2.1 To-Do List en vue de la première soutenance

Tâches	Priorité (0-4)	Date de début	Date de fin	Charge de travail	Livrables	Progrès	Dernière mise à jour
État de l'art	4	10/10/22	18/10/22	8h	Chercher et relever quelques exemples d'applications web de vente de fruits et légumes en circuit court existantes afin de les mettre en perspective avec notre application	100%	17/10/22
Gérer la gestion du projet	2	10/10/22	18/10/22	6h	Rédiger des comptes-rendus pour chaque réunion. Définir la démarche avec une to-do list et un cahier des charges.	100%	17/10/22
Expliquer le concept de l'application et son aspect innovant	2	14/10/22	18/10/22	2h	Expliquer avec détails le concept de l'application. Puis, à partir de l'état de l'art, énoncer ses caractéristiques innovantes.	100%	17/10/22
Rédiger le rapport	2	16/10/22	18/10/22	8h	Rédiger un rapport détaillé reprenant tous les points majeurs attendus de notre démarche : - L'état de l'art - La gestion du projet - L'explication du concept	80%	17/10/22
Réaliser la présentation	2	16/10/22	18/10/22	4h	Créer un support de présentation concis	60%	17/10/22
Préparation finale en vu de la soutenance	2	16/10/22	18/10/22	2h	Organisation des temps de parole et préparatifs en vue de la soutenance	50%	17/10/22

4.2.2 To-Do List en vue des vacances de la Toussaint

Tâches	Priorité (0-4)	Date de début	Date de fin	Qui ?	Livrables	Progrès	Dernière mise à jour
Création de la base de donnée	2	31/10/22	06/11/22	Léo	Création des schémas relationnels et de la base de données en SQL	100%	06/11/22
Page de connexion	2	31/10/22	06/11/22	Thomas	Page de connexion en Flask avec utilisation de la base de données temporaire	100%	06/11/22
Page d'accueil	2	31/10/22	06/11/22	Théo	Page d'accueil en Flask avec utilisation de la base de données temporaire	100%	17/10/22
Révisons des outils de gestion de projet	2	31/10/22	06/11/22	Maxence	Rédiger et utiliser de nouveaux outils de gestion de projet tels que : - Les ToDo listes - Le cahier des charges fonctionnel	80%	17/10/22

4.2.3 Utilisation des To-Do List

Les To-Do List sont des outils nous permettant de nous organiser dans la répartition des tâches, de nous fixer des deadlines et d'avoir connaissance de l'avancée de chacun dans ses objectifs.

Cependant, la plupart de ces informations sont fixées pendant les réunions et existent par conséquent déjà dans les compte-rendus qui seront présentés par la suite. Afin d'éviter la multiplication inutile d'outils de gestion de projet et de gagner du temps, nous nous sommes par la suite contentés des informations contenues dans les comptes-rendus de réunion.

4.3 Compte-rendus de réunion

4.3.1 Modèle type de compte-rendu

Motif / type de réunion : Réunion de chantier	Lieu : Telecom Nancy (1.33)
Présent(s) (retard/excusés/non excusés) : Équipe-projet : BEKHEDDA Maxence GERMAIN Léo GIANELLI Thomas WALTER Théo	Date / heure de début : 10/12/2022 à 10h30 Durée : 40min

Ordre du jour

1. Retour sur le travail effectué par chacun depuis la dernière réunion.
2. Difficultés rencontrées.
3. Listing et répartition des tâches restantes.

Informations échangées

- Les objectifs fixés lors de la dernière réunion ont été accomplis sans rencontrer de difficultés particulières.
- Plusieurs fonctionnalités (notamment l'onglet "profil") réutilisent le code de l'onglet "publications". Il est donc nécessaire de le faire en priorité afin de pouvoir avancer.
- Le style n'est pas une priorité, il vaut mieux coder quelque chose de fonctionnel permettant de débloquer de nouvelles fonctionnalités et faisant avancer le projet. On pourra s'occuper du style dans un second temps.
- La date de rendu est le 6 janvier 2023. Cette date étant proche, il faut concentrer nos forces sur l'écriture code. La rédaction du rapport et la conception du support de présentation pour la soutenance viendront dans un second temps.

Remarques / Questions

- Remarque 1 : Pour nous en sortir dans cette période de "dernière ligne droite" avant le rendu du projet, cela pourrait être une bonne idée de lister les objectifs que nous souhaitons nous fixer d'ici le 6 janvier.
- Question 2 : Afin d'intégrer de l'algorithme dans notre projet, nous souhaitons créer un programme permettant de calculer les casiers les plus proches de l'acheteur et du producteur qui sont disponibles. Comment mettre cela en place ?
- Question 3 : Dans la même optique d'ajouter de l'algorithme au projet, qu'en est-il de l'idée d'ajouter un algorithme de recommandation en fonction des récents achats et de la localisation, suggérant ainsi des publications plus probable d'être utile à l'utilisateur ?

Décisions

- Réponse 1 : Nous allons effectuer un diagramme de Gantt, pour lequel un listing de l'ensemble des objectifs du début à la fin du projet sera nécessaire. Cela nous permettra de faire d'une pierre deux coups.
- Réponse 2 : Un des membres de l'équipe projet sait utiliser les API de Google. Cet outil nous permet d'accéder aux localisations et ainsi d'évaluer des distances.
- Réponse 3 : Cela ne constitue pas une priorité sachant que le temps restant est restreint. Cependant, nous maintenons que cela est une bonne idée qui pourrait être mise en oeuvre si nous en trouvons le temps.

Actions à suivre / Todo list

Description	Responsable	Délai	Livrable	Validé par
Utilisation des API de Google pour intégrer le calcul des distances au choix des casiers	BEKHEDDA Maxence WALTER Théo	28/12	Code fonctionnel	L'équipe projet
Code de l'onglet "publication" et avancée de la partie Gestion de Projet	GERMAIN Léo	28/12	Code fonctionnel et publication de nouveaux outils de GdP	L'équipe projet
Code de l'onglet "profil" à partir de "publication" et lien entre ces onglets et la BD	GIANELLI Thomas	28/12	Code fonctionnel	L'équipe projet

4.3.2 Utilisation des compte-rendus

Afin d'éviter l'effet tunnel et d'avancer de manière régulière et coordinée, nous avons fait en sorte de nous réunir de manière régulière, environ 2 fois par mois. Chacune de ces réunions a fait l'objet d'un compte-rendu dans un court délai. Ces compte-rendus ont plusieurs intérêts :

- Garder en mémoire la date de la réunion et l'état du projet à ce moment-là.
- Garder en mémoire les informations échangées, les questionnements et leurs réponses associées, ainsi que les blocages.
- Poser une répartition claire des objectifs de chacun, qu'il est possible de consulter tout au long du projet.

Afin de jalonner notre projet le plus efficacement possible, nous avons choisi de nous fixer chacun des objectifs à réaliser pour la prochaine réunion.

4.4 Matrice RACI

Tâches	Maxence	Léo	Thomas	Théo	MOA
Etat de l'art	R	C	R	R	AC
Rédaction des comptes-rendus	C	R	C	C	AI
Conception du logo	R	C	C	R	AI
Etablir et actualiser le cahier des charges	R	R	C	C	AC
Etablir les to-do list	R	R	C	C	AI
Rédaction des rapports et des supports de présentation	R	R	R	R	AC
Création de la base de donnée	R	R	C	C	AI
Création de la structure initiale du projet	C	C	C	R	AI
Gestion de la documentation	C	C	C	R	AI
Création de la session	R	C	R	C	AI
Création de la page d'authentification	R	C	R	C	AI
Création de l'onglet "publications"	R	R	R	C	AI
Gestion des profils	R	C	R	R	AI
Design du layout	R	C	R	R	AI
Design du style	R	C	R	C	AI
Rédaction de la matrice RACI	C	R	C	C	AI
Création de l'onglet "lockers"	C	C	C	R	AI
Gestion navigation entre les profils	C	C	R	C	AI
Algorithme de placement des casiers	R	C	C	C	AI
Gestion des likes	R	C	R	C	AI
Création de la page de paramètres	R	C	R	C	AI
Création du diagramme de Gantt	C	R	C	C	AI
Gestion de la fonctionnalité "suivre"	C	C	R	C	AI
Création de la page "vente"	R	C	R	C	AI

R=réalise, A=autorité, C=consulté, I=informé

La matrice RACI est un outil qui nous permet de prendre en note les différentes tâches réalisées par les différents membres de l'équipe projet. Elle permet de mettre en lumière ce qui a nécessité une coopération entre les différents membres et ce qui a été réalisé par une seule personne.

On observe sur le diagramme que les tâches ont été réparties de manière relativement homogène, chaque membre a pu contribuer efficacement au projet, et ce de plusieurs manières. On observe cependant des tendances liées aux profils des différents membres : l'aisance de certains membres avec l'algorithme leur a permis de davantage se concentrer sur les différentes fonctionnalités de l'application Web, tandis que d'autres se sont concentrées sur des outils vus en cours, à travers les outils de gestion de projet et les langages vus en CS54.

Durant tout le projet, chaque avancée par un des membres a été publiée et mise à disposition de tous les membres de l'équipe. Il a été important pour nous que chacun valide les modifications au fur et à mesure afin d'être familier avec les ajouts, et d'éviter de devoir retourner sur des défauts de conception bien plus tard. Cela nous a été permis avec le projet GitLab, et c'est pourquoi chaque membre de l'équipe projet est à minima consulté quelle que soit la tâche. De même, les commanditaires sont informés de l'avancée du projet au fur et à mesure par les différents commits. Ils ont été consultés dans le cadre de la première soutenance.

4.5 Diagramme de Gantt

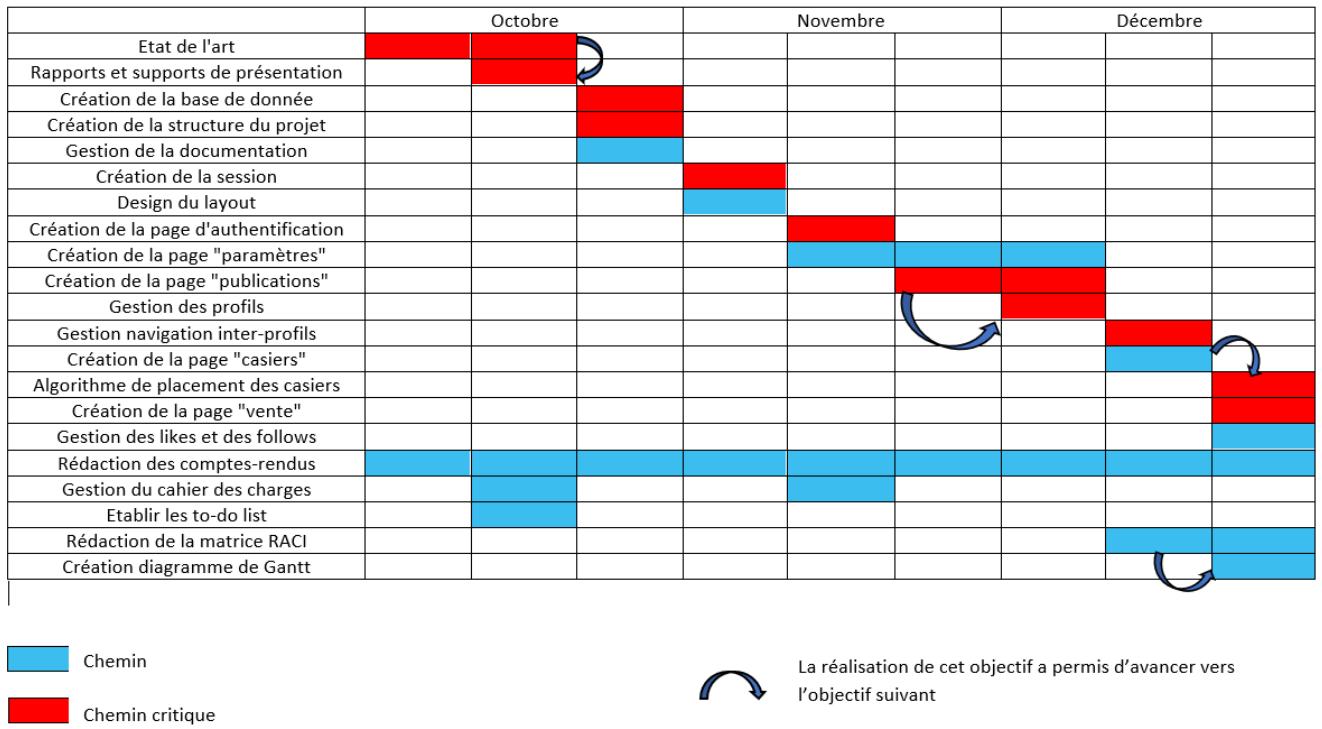


FIGURE 16 – Diagramme de Gantt

Pour réaliser ce diagramme de Gantt, nous avons repris au fur et à mesure les différents objectifs fixés puis réalisés ce que nous avons relevé en réunion. Au cours de ces réunions apparaissait de manière souvent évidente que l'accomplissement d'une ou plusieurs de ces tâches étaient essentielles afin de pouvoir poursuivre nos avancées. Ces tâches constituent le chemin critique (en rouge dans le diagramme).

Nous avons fait le choix de ne pas afficher la durée requise pour chaque objectif en jour/homme, car ne travaillant pas à plein temps ni de manière égale chaque jour sur le projet, ces données ne seraient pas pertinentes.

Alors que le projet touche à sa fin, il apparaît que deux périodes ont été particulièrement denses durant ces trois mois :

- La première, mi-octobre, correspond à l'étape de définition du projet en vue de la première soutenance, que nous avons réalisée dans un délai restreint.
- La seconde, durant le mois de décembre, correspond à la dernière ligne droite du projet, à un peu moins d'un mois du rendu final des livrables. Bien que le projet prenait forme, il restait une certaine quantité de travail à abattre afin d'aboutir à l'objectif que nous nous étions fixés au début du projet.

4.6 Post-mortem

Afin de clore ce projet et d'en tirer des conclusions, à la fois personnelles et collectives, nous avons fait une ultime réunion de post-mortem. Pour cela, nous avons chacun répondu à des questions classiques de post-mortem. Voici un résumé des réponses fournies par l'équipe projet :

- Êtes-vous satisfaits du résultat final de ce projet et de la manière de l'atteindre ?

Nous sommes globalement satisfaits, le résultat correspond aux objectifs que nous nous étions fixés au départ. De plus, un travail régulier durant ces trois mois nous ont permis d'avancer sereinement, sans le sentiment d'être débordé ou de devoir se précipiter.

- Quels sont les points à surveiller selon vous ?

Nous avons parfois fait les frais de l'effet tunnel, que nous aurions pu éviter par une meilleure communication. Il aurait été utile d'augmenter le nombre de jalons et de réunions, afin de mieux cadrer le travail de chacun.

- Quelle a été la partie la plus gratifiante dans ce projet selon vous ?

Globalement, les différents membres de l'équipe ont apprécié le gain d'expérience apporté par ce projet : que ce soit dans la gestion des imprévus ou dans l'utilisation des outils de code vus en cours. Ce projet nous a forcés à effectuer des recherches et à faire preuve de ressources pour répondre à nos attentes de départ. Cela ayant porté ses fruits pour arriver à ce résultat, cela représente une grande satisfaction pour tous.

- Quelles méthodes ou processus ont particulièrement bien fonctionné ?

L'usage des to-do list, qui étaient établies lors de chaque réunion, s'est révélé très utile pour tous, jalonnant rigoureusement et clairement le projet. De plus, la coopération - bien qu'à distance - s'est retrouvée facilitée par l'historique des commits sur GitLab, chaque membre publant son travail régulièrement en l'annotant et le nommant avec soin.

- Si vous pouviez modifier une chose de ce projet, que changeriez-vous ?

Le style de notre application est différent de ce que nous avions prévu au départ. Il reste satisfaisant et il est intuitif, comme nous le souhaitions tout particulièrement depuis le début. C'est pourquoi, si nous pouvions modifier une chose, ce serait le CSS.

5 Conclusion

Pour conclure, nous pouvons dire que ce projet nous a beaucoup apporté et que nous sommes satisfaits et fiers de notre travail. Nous avons en effet pu réaliser la majorité des fonctionnalités que nous avions l'ambition d'implémenter. Néanmoins, nous avons pu expérimenter le paradoxe de la gestion de projet. Nous avons des idées d'améliorations et nous savons précisément dans quelle direction nous voudrions faire évoluer notre projet ; ce qui n'était pas le cas il y a trois mois de ça. Avec l'expérience, nous sommes capable d'implémenter de nouvelles fonctionnalités mais le temps nous manque. Bien que des idées originales qui n'étaient pas initialement prévues, telles que notre algorithme de choix de casier, viennent enrichir notre projet, nous n'avons pu ajouter certaines fonctionnalités comme les commentaires ou les notations car elles étaient plus difficiles ou plus chronophage que prévues.

Tout de même, nous pensons que notre application est fidèle à l'idée de départ. Dans le contexte actuel, nous sommes d'avis qu'avec quelques améliorations, notre application est une innovation qui pourrait favoriser les circuits courts et dynamiser l'échange entre les acheteurs et les producteurs particuliers et professionnels.