Rapport Final PP2I GRP24

Samir ABDALLAH
Yamine BOUALIT
Adrien DIDON
Romain SAMBA

Mercredi 11 Janvier 2023

1 Introduction

Dans le cadre du premier semestre de première année à l'école Télécom Nancy, nous avions à concevoir une application en lien avec la gestion d'un jardin potager. Notre choix c'est tourné vers une application qui permettrait de placer au mieux des plantes dans un potager en fonction des synergies existantes entre les plantes. L'objectif de ce rapport est donc de résumer le travail qui a été effectué tout au long du projet et les choix techniques retenus. Seront abordés ici le fonctionnement en détail de l'application et les choix techniques retenus, les tests effectués et enfin la gestion de projet qui a été appliquée.

2 Conception de l'application web

La conception de cette application a logiquement nécessité une partie web de programmation. Dans cette partie de conception on peut y séparer deux aspects distincts qui néanmoins se complètent et s'assemblent : Le frontend et le backend.

Premièrement le backend, cela représente toute l'architecture de l'application. Pour cela nous avons implémenté ceci sur Python plus particulièrement avec le Framework Flask. Il a alors été nécessaire de créer différentes routes pour les différentes pages de l'application. Au total 6 routes différentes ont été utilisées pour l'application.

Tout d'abord, il y a la route de départ soit "" qui correspond à la page d'accueil. Il s'agit de la page de démarrage de l'application c'est pourquoi la route est représentée comme la racine. Il suffisait ici d'utiliser la fonction render_template() pour renvoyer une page html avec les composants de la page d'accueil, ce qui fait écho pour plus tard à la partie frontend.

Ensuite, à partir de cette page il est nécessaire de devoir s'identifier soit avec un compte que l'utilisateur possède déjà soit en créant un nouveau compte, pour cela il suffit de renseigner un nom d'utilisateur et un mot de passe (il a été choisi de ne pas demander d'autres informations comme l'e-mail ou autre car cela n'était pas utile pour l'utilisation de l'application). On peut noter qu'il a été choisi de limiter l'utilisation des fonctionnalités de l'application uniquement aux personnes possédant un compte dans un souci de simplicité et de conservation des résultats. Ce point est abordé dans la partie base de données.

La route d'inscription se divise en deux cas d'utilisations. On y distingue les deux requêtes que l'on peut appliquer sur une page c'est-à-dire GET et POST. Une simple structure conditionnelle sur la requête utilisée a été utilisée. En cas de requête GET un nouveau render_template() vient renvoyer une page html avec la page d'inscription qui contient un formulaire html.

Ce formulaire html permet de récupérer des informations en les demandant à l'utilisateur. Plusieurs types d'entrées sont disponibles, dans ce cas, un type text est utilisé pour le nom d'utilisateur et un type mot de passe pour le mot de passe, ce qui permet de cacher ce qu'écrit l'utilisateur. Ensuite, un bouton d'envoi permet de soumettre les données au serveur par l'intermédiaire d'une requête POST. Ainsi toujours sur la même route et grâce à la structure conditionnelle, l'application va récupérer les données du formulaire en passant par la

méthode request_get_form() qui fonctionne comme un dictionnaire python avec des valeurs associées.

Après avoir récupéré ces données que l'on stocke dans de simples variables python, les valeurs sont sauvegardées dans la base de données avec les requêtes nécessaires et l'utilisation de sqlite3 comme expliqué dans la partie base de données.

Une fois les données enregistrées, la méthode redirect() permet une redirection vers la route de connexion. Encore une fois une structure conditionnelle avec un formulaire html est implémentée sur la méthode GET, le nom d'utilisateur et le mot de passe y sont collectés. Puis la table utilisateurs de la base de données est parcourue avec une boucle for et les valeurs de la table sont comparées avec les informations du formulaire, en cas d'égalité simultanée pour le nom d'utilisateur et le mot de passe alors l'utilisateur est identifié et redirigé vers sa page utilisateur. En cas de non-égalité à la fin de la boucle for alors une page signalant un mot de passe ou un nom d'utilisateur incorrect apparaît et il faut revenir à la page de connexion ou d'inscription.

En cas de connexion de l'utilisateur, la question s'est posée de savoir comment transmettre à la route quel utilisateur s'est connecté. Plusieurs essais ont été réalisés notamment avec la bibliothèque Flask_login et la transmission de cookies. Mais finalement une idée plus simple a été retenue : transmettre le nom d'utilisateur directement via l'url et ainsi pouvoir retrouver l'utilisateur dans la base de données si besoin.

La page utilisateur permet de visualiser les précédents potagers que l'utilisateur a généré et contient un lien vers la génération d'un nouveau potager. Ainsi l'utilisateur est redirigé vers une autre route qui va premièrement demander le nombre de fruits ou de légumes à placer. Cette page a été la solution d'un problème rencontré.

En effet pour un formulaire html il est nécessaire de connaitre au préalable le nombre de paramètres que l'utilisateur devra renseigner or ici le nombre de fruits et légumes est variable. Ainsi nous avons essayé de créer un bouton pour ajouter au click une nouvelle section au formulaire or cela faisait intervenir des notions en dehors des attendus du projet tels que du JavaScript.

Finalement nous avons opté pour diviser cette requête d'informations en deux parties avec d'abord le nombre de fruits et légumes puis les plantes à placer

et leur quantité. Une fois les informations recueillies, les données sont stockées dans un tableau de couple avec le nom de la plante associé à son nombre de pieds. L'algorithme expliqué dans une autre partie va alors renvoyer un résultat (la programmation orientée objet dans l'algorithme a nécessité une adaptation du résultat pour pouvoir l'afficher en remplaçant chaque élément du potager par son nom). Le résultat s'affiche alors et peut ensuite être sauvegardé dans la base de données associée à l'utilisateur connecté. Pour permettre à l'utilisateur de s'y retrouver parmi ses plans, un titre est demandé pour chaque plan sauvegardé.

Ainsi la page de l'utilisateur répertorie les différents potager et leur nom. Grace à cela l'utilisateur peut consulter à tout moment l'emplacement des plantes dans ses différents potagers et faciliter leur suivi.

Pour le frontend, on peut noter l'utilisation du CSS avec des balises de feuilles de style. Une barre de navigation a été placée dans chaque page pour faciliter la navigation. De plus on peut noter lutilisation de balises div pour permettre de manipuler les différents objets de la page et ainsi obtenir la meilleure configuration. Sur la page d'accueil il a été décidé dy placer le nom de l'application et une description des fonctionnalités accompagnées d'une image d'illustration. La barre de navigation se compose dun logo et des boutons Se connecter et Sinscrire qui permettent de poursuivre la navigation. Les pages de connexion et d'inscription ont des feuilles de style similaires avec des emplacements pour les cadrans de remplissage d'informations.

Par la suite il est intéressant d'afficher le nombre de plantes souhaité. Le nombre de plantes que l'utilisateur renseigne va ensuite être utilisé pour créer une boucle dans le ficher html en faisant apparaitre la balise du formulaire autant de fois que nécessaire. La boucle utilisée provient de la syntaxe Jinja. On peut noter qu'une boucle similaire est utilisée pour l'affichage des différents fruits et légumes. En revanche, ici la boucle parcourt directement sur la base de données.

3 Conception de la base de données

La base de données est essentielle dans l'architecture d'une application web. Elle permet en effet de stocker les données nécessaires au bon fonctionnement de l'application, par exemple l'authentification de l'utilisateur. Ici l'application ne nécessite pas une base de données très complexe avec beaucoup de tables, cependant, il était envisageable de séparer en deux parties la base de données : une partie contenant les plantes, les terrains et les associations qui est davantage utile pour la partie algorithmique et une partie qui regroupe les données utilisateur permettant d'identifier ces derniers et de sauvegarder leurs plans. Pour éviter de trop complexifier le code à cause d'un conflit entre les deux potentielles bases de données, cette idée n'a pas été retenue et une seule base de données regroupant toutes les données nécessaires a été créée.

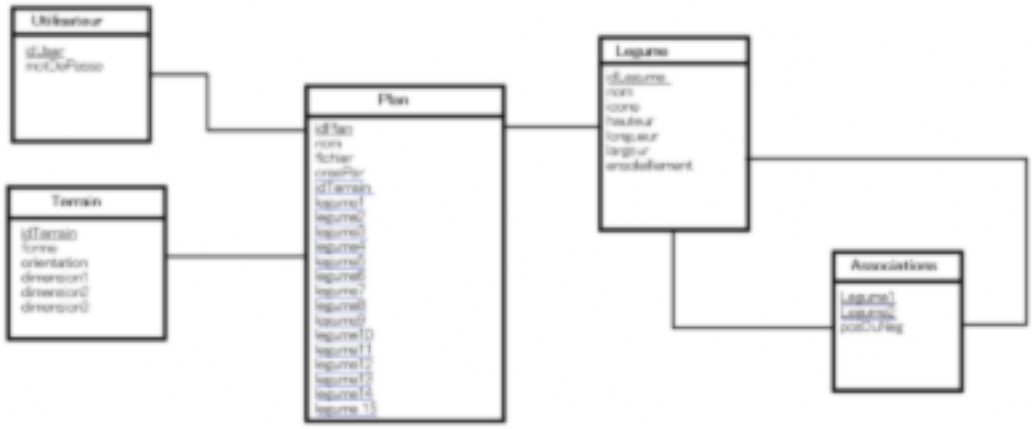
L'architecture de la base de données est relativement simple car il y a relativement peu de données différentes à stocker. Comme expliqué précédemment il faut stocker d'une part les données nécessaires au fonctionnement de l'algorithme, notamment les informations relatives aux plantes et aux associations entre celles-ci, ce qui est le principal enjeu de l'application. Pour cela, trois tables ont été prévues : une première nommée légumes qui regroupe toutes les informations relatives à la culture de la plante, la deuxième qui est issue d'une relation entre deux éléments de cette table qui permet de recenser les associations entre les plantes qui sont recommandées ou déconseillées, enfin, la dernière est relative aux informations du terrain. La seconde partie de la base de données contient les tables utilisateur qui stocke les identifiants de connexion de l'utilisateur, la table plans qui sauvegarde les plans créés par l'utilisateur et la table. Cette partie est reliée à la partie plutôt destinée à l'algorithmique entre la table plan et la table légumes puisqu'il est nécessaire pour générer le plan notamment pour l'affichage à l'utilisateur de connaître les plantes utilisées puisque l'algorithme identifie les plantes au moyen d'un entier qui doit ensuite être transformé pour que le nom de la plante soit affiché pour l'utilisateur. Le moyen le plus simple étant en effet de passer par la base de données, cela permet également de relier les deux parties de la base entre elles.

Si pour certaines tables, les champs permettant de qualifier un élément étaient clairs, sur certaines tables, il a fallu s'interroger sur les données nécessaires. Finalement il a été décidé de prévoir des champs supplémentaires pour éventuellement implémenter d'autres fonctions par exemple les dimensions des plantes pour par exemple améliorer la performance de l'algorithme.

Le principal enjeu pour déterminer les champs à récolter était pour les plantes, en effet, par souci de simplicité, il pouvait être possible d'utiliser comme clé primaire le nom de la plante mais cette idée a été abandonnée au profit d'un

identifiant qui correspond à un entier. Pour permettre l'affichage, sont stockés pour un légume une icône qui sert pour l'affichage et le nom de la plante. Finalement l'idée de l'icône a été abandonnée mais le champ a été conservé.

La table terrain a également été discutée car son utilité était moindre par rapport à l'algorithme car elle n'a fin de compte pas été utilisée par l'algorithme mais cela pourrait par la suite être incrémenté pour permettre de tenir compte de terrains dont la forme est moins conventionnelle et dont l'ensoleillement peut varier.



Ci-dessus le schéma de la base de données de l'application

4 Conception de l'algorithme de résolution

Afin de répondre au problème qui est de générer un plan de potager évitant les associations défavorables entre les plantes dont l'utilisateur dispose il est nécessaire d'implémenter dans l'application un algorithme modélisant le problème et générant une solution à celui-ci. Celui-ci est exécuté du côté du serveur et est codé en Python. Son fonctionnement est le suivant : On représente tout d'abord le potager par une matrice qu'on choisi d'une taille telle que tous les légumes entrés par l'utilisateur puissent y apparaître. Afin d'obtenir un résultat le plus naturel possible à la fin, on remplir chaque ligne de la matrice par une unique espèce. On définit premièrement une fonction conflits dont le but est de repérer les mauvaises associations présentes dans le potager généré précédemment. Naturellement celle-ci prend en argument une matrice représentant un potager et revoie une liste contenant des doublets de doublets représentant des positions de légumes dans le potager. On définit ensuite une fonction intitulée correction_locale prenant en paramètre une matrice (représentant le potager) et un doublet (représentant la position d'un légume dans le potager) . Dans le cas où le doublet passé en argument est présent dans le liste renvoyé par conflits à laquelle on a passé le potager en argument, cette fonction échanger le légume dont les coordonnées ont été passées en argument avec tous les autres légumes présents afin de trouver une configuration dans laquelle le légume n'est plus cause d'une mauvaise association.

Enfin on définit une fonction intitulée correction_globale prenant en argument une matrice et qui : - Applique la fonction conflits en passant la matrice en argument. - Applique correction_locale pour chaque position renvoyée par conflits . - Applique la fonction conflits en passant la matrice obtenue en argument. - Ajoute dans une liste configurations cette matrice. - Recommence tant que conflits ne renvoie pas la liste vide.

Étant donné que l'on ne peut pas s 'assurer de la terminaison de cet algorithme on l'arrête au bout de 12 secondes de calcul et l'algorithme finit par retourner la configuration c pour laquelle la longueur de la liste conflits(c) est la plus courte.

5 Tests de l'application et complexité algorithmique

L'algorithme permettant de placer les différentes plantes est assez complexe. En effet, il faut parcourir un espace de dimension 2, ce qui nous garantit une complexité minimale en $\theta(n^2)$. Par ailleurs le nombre d'itérations de ce parcours pour trouver une matrice qui convient peut être très élevé, surtout s'il n'existe pas de solutions. La solution retenue a été de couper l'algorithme de recherche au bout d'un certain temps d'exécution (ici 12 secondes). Ce temps a été choisi après plusieurs essais qui ont permis de constater que l'algorithme mettait en moyenne 12secondes pour parcourir tous les cas différents. L'option backtracking qui sauvegarde tous les plans essayés puis une fois qu'il n'existe plus de combinaison qui n'a pas déjà été essayée, le programme s'arrête, cependant cela consommait énormément de mémoire, ce qui n'était pas raisonnable.

La fonction qui peut sembler la plus complexe est celle qui relève tous les conflits du potager. On retrouve 4 boucles for qui sont imbriquées entre elles. Si on se penche sur la fonction, on remarque que la dernière boucle for est exécutée trois fois au maximum puisqu'elle parcourt les voisins de la plante, tout comme l'avant dernière où chacune contient une opération. Cela signifie donc que comme ces deux boucles imbriquées sont exécutées n fois pour la longueur puis n fois pour la largeur, on obtient donc une complexité qui vaut $9n^2$, soit une complexité en $\theta(n^2)$.

En réalité la fonction la plus complexe est celle qui effectue la correction globale du potager visant à éliminer les conflits. Cela est dû au fait que les boucles for imbriquées impliquent a minima une complexité en $\theta(n^3)$.

```
def correction_globale(terrain: np.ndarray) -> np.ndarray:
    """Fonction visant à renvoyer un potager dans lequel il n'y a plus de conflits dans le positionnement des plantes"""
    active_config = terrain.copy()
    start = time.time()
    dt = 0
    configs = [[active_config,len(conflits(active_config))]]
    # On a deux conditions d'arrêt : plus de conflit ou l'algorithme tourne depuis plus de 12s (dans le cas où il n'existe
    while conflits(active_config) and dt < 12:
        active_config = terrain.copy()
        liste_conflits = conflits(active_config)
        # On fait en sorte de garder des conflits uniques pour ne pas avoir à resoudre conflits déjà résolus
        for conflitA in liste_conflits:
            for conflitB in liste_conflits:
                if (conflitA[1], conflitA[0]) == (conflitB[0], conflitB[1]):
                    liste_conflits.remove(active_config)
        # On recode chaque conflit repéré à l'aide de la correction locale
        for conflit in liste_conflits:
            if active_config[conflit[0]] != None:
                active_config = correction_locale(terrain=active_config, lock=conflit[0])
        configs.append([active_config,len(conflits(active_config))])
        dt = time.time() - start
    configs.sort(key=lambda x:x[1])
    return configs[0][0]
```

Par ailleurs, comme cette fonction fait appel à d'autres fonctions dans la complexité est souvent en $\theta(n^2)$. Il est donc essentiel de calculer cette complexité qui donne la complexité réelle de l'algorithme employé. La seconde boucle for contient la fonction correction_locale qui s'occupe de corriger un conflit une fois qu'il est détecté et dont le code est affiché ci-dessous.

```
def correction_locale(terrain: np.ndarray, lock: tuple) -> np.ndarray:
    """Fonction qui renvoie un potager dans lequel on a changé la place d'une plante dans une copie du potager initial pour qu'elle ne soit

    # On prend connaissance de la plante problématique
    planteA = terrain[lock]
    n, m = terrain.shape
    # Définition d'un booléen qui nous permettra de savoir si on a résolu le problème
    corrected = False
    i = 0
    # On va échanger la plante A avec chacune des autres plantes du potager afin de trouver une configuration dans laquelle la plante A ne
    while i < n and not corrected:
        j = 0
        while j < m and not corrected:
            # On récupère la plante B comme celle en position (i,j)
            planteB = terrain[i, j]
            # Il est inutile d'échanger deux plantes de la même espece
            if planteB != None and planteB.nom == planteA.nom:
                pass
            else:
                active_config = terrain.copy()
                # Ici on échange la plante A avec la plante B
                swap(active_config, lock, (i, j))
                # On vérifie que la nouvelle configuration ne génère pas de nouveaux conflits...
                if j not in voisins(active_config, i, j) and i not in voisins(active_config, lock[0], lock[1]):
                    # Si ce n'est pas le cas on a bien une configuration stable : on a résolu le problème
                    corrected = True
            j += 1
        i += 1
    # Pour des raisons diverses, le paramètre 'terrain' ne change pas lorsque l'on fait 'terrain = terrain', on se contente donc de renvoy
    return active_config
```

Le cas le plus défavorable se présente si on parcourt le potager et qu'il n'existe

aucune solution pour résoudre le conflit car les plantes sont toutes d'espèces différentes. Dans ce cas, l'opération d'échange qui cache en réalité une double affectation représente 2 opérations qui sont d'abord itérées n fois pour la boucle while intérieure puis encore n fois pour la boucle while extérieure. Ainsi on obtient une complexité de $2n^2$ soit en $\theta(n^2)$.

Cela signifie que la 2ème boucle for de la fonction contient déjà $2n^2$ opérations. Elle est exécutée n fois pour parcourir la liste des conflits. Cela donne donc un total de $2n^3$ opérations
La première boucle for contient une autre boucle for imbriquée. Dans cette boucle for, si le conflit est toujours résolu, on effectue une opération de suppression, n fois dans la première boucle puis n fois à nouveau dans la deuxième. Ainsi on obtient de nouveau une complexité en $\theta(n^2)$. Au total comme la boucle while peut être exécutée jusqu'à n fois (la longueur de la liste conflits), cela donne un total de $2n^4 + n^3$ opérations. On obtient donc une complexité en $\theta(n^4)$.

Au vu de la structure de l'algorithme présentée si dessous, il peut être cohérent d'étudier la complexité de la fonction permettant de générer un potager.

```python
# Modélisation d'un potager par une matrice
if __name__ == "__main__":
    from genere_potager import genere_potager
    from conflits import conflits
    from correction import correction_globale
    plantes = [("Concombres", 5,), ("Haricot", 3), ("Choux", 2)]
    n = 5
    terrain = genere_potager(plantes)
    print(terrain)
    print(conflits(terrain))
    # terrain = correction_locale(terrain=terrain, locA=(1,0))
    terrain = correction_globale(terrain=terrain)
    print(terrain)
    print(conflits(terrain))
```

Ci-dessous la fonction permettant de générer un potager à partir des paramètres obtenus.

```python
def genere_potager(plantes: list) -> np.ndarray:
    """
    Fonction qui génère un potager en placant chaque espèce sur u
    Un exemple d'usage de cette fonction est :
        'genere_potager(plantes=[("Concombres", 5,), ("Radis", 3),
    """
    n = int(np.sqrt(sum([plante[1] for plante in plantes]))) + 1
    terrain = np.ndarray(shape=(n, n), dtype=Plante)
    plts = []
    for plante in plantes:
        for i in range(plante[1]):
            plts.append(Plante(plante[0]))

    def ligne_indice(k):
        return k // n, k % n

    for i in range(len(plts)):
        terrain[ligne_indice(i)] = plts[i]
    return terrain
```

On remarque la présence de 3 boucles for dont 2 imbriquées entre elles. Dans la boucle for imbriquée dans la première, une opération est effectuée et cette boucle est itérée 2 fois. La première boucle quant à elle est itérée n fois. Cela donne ainsi un total de 2n opérations.
La seconde boucle est également itérée n fois et contient une seule opération. Il y a donc au total n opérations effectuées, ce qui donne une complexité totale de $\theta(n)$.

La fonction de résolution de conflits étant très complexe, il a été décidé d'arrêter l'algorithme au bout de 12 secondes d'exécution. En effet, il s'agissait de la meilleure solution après de nombreux tests pour obtenir des résultats satisfaisants et garder un temps d'attente raisonnable pour l'utilisateur. Même avec des plantes qui ont quasiment toutes des conflits entre elles, l'algorithme renvoie une solution viable en une douzaine de secondes.

6 Gestion de projet

6.1 Vue d'ensemble et analyse post-mortem

Victimes de notre manque d'investissement sur la durée, nous pensions au début avoir le temps d'avancer chacun de notre côté, nous nous sommes très vite fait surprendre par d'autres obligations non anticipées. La stratégie de gestion de projet initialement prévue n'a pas été appliqué correctement et cela nous servira de leçon pour nos projets futurs. Pour faire un premier bilan nous pouvons reprendre les points soulignés dans notre première matrice SWOT.

Forces :
• Accord unanime de l'équipe sur l'idée du projet
• Compétences basiques en développement web, algorithmique et base de données

- Cet accord a persisté tout le long du projet, nous sommes fiers de notre idée.
- Nous avons exploité aux mieux les bases acquises en début d'année à travers le module CS54.

Faiblesses :
• Peu d'expérience en développement d'applications web
• Le projet manque de spécifications

- Cela s'est ressenti à travers des erreurs récurrentes au départ qui ont pu être corrigées au fil du temps.

- Nous n'avions pas de réelles attentes concernant le site en dehors de la fonctionnalité principale

Opportunités:
• Le projet s'inscrit complétement dans la continuité des compétences enseignées à l'école
• Existence dans le domaine du jeu vidéo d'algorithmes de génération de cartes avec contraintes

- Ce projet nous a en effet permis de solidifier et d'appliquer les compétences enseignées.

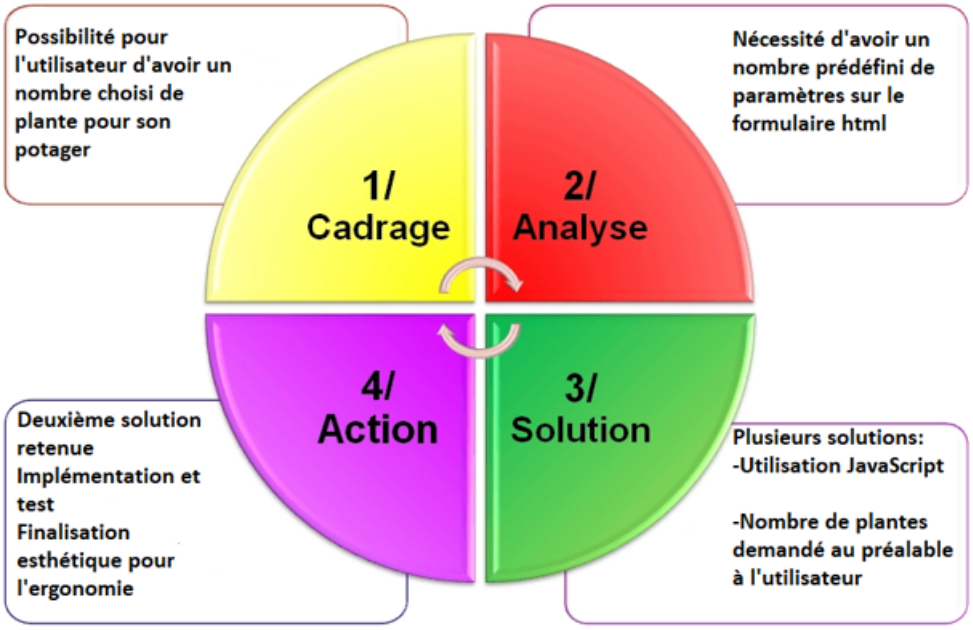
- Nous n'avons finalement pas exploité cette opportunité

Menaces :
• Il nous sera difficile d'obtenir des retours externes sur l'utilisation de notre application
• Contrainte temporelle : notre marge de manœuvre diminuera fortement au cours du temps

Comme prévu il a été compliqué d'avoir des retours sur notre application, faute de délais nous n'avons pas pu consulter nos professeurs en revanche nous avons lors des vacances pu envisager une prise en main de nos proches.

6.2 Résolution de problèmes

Le développement d'une application comme celle-ci conduit comme tout projet à des problèmes. Il est donc important, pour garantir l'évolution du projet, d'identifier et de résoudre au plus vite ces problèmes.
La méthode de conception de notre application était telle qu'une répartition équitable du travail a été faite. Ainsi la plupart du développement s'est réalisé en travail personnel et individuel. Il était donc très facile de ne pas faire part de ses problèmes et ainsi tomber dans l'effet tunnel. Nous avons donc décidé dès le début du projet de mettre en place une méthode de résolution de problème. Ainsi lorsque l'un de nous rencontrait un problème il n'hésitait pas à en parler. La méthode utilisé est cycle reposant sur 4 étapes. Voici un exemple de cycle réalisé sur un des différents problème rencontré. Ici, il s'agit d'une réflexion sur comment faire varier le nombre de plantes que l'utilisateur peut entrer.



Ci-dessus le cycle relatif à la résolution de problèmes pour relever le nombre de plantes à placer

On peut voir à travers ces 4 étapes, 4 étapes différentes de la résolution de ce problème. Premièrement le cadrage, correspond au repérage du problème par le

développeur de cette partie. Une fois en avoir fait part aux autres, une réunion est organisée. On arrive alors à la phase d'analyse où le but est d'expliquer la cause du problème et éventuellement trouver une solution évidente. Une fois le problème précisément identifié, une réflexion commune avec un brainstorming se met en place pour trouver une ou plusieurs solutions. Pour chaque solution trouvée une étude de faisabilité est réalisée. On vérifie qu'elle permet bien de résoudre le problème et que cela amène à un objectif SMART (Spécifique Mesurable Accepté Réaliste Temps). Pour l'exemple donné plus haut, la solution utilisant du JavaScript était bien spécifique et mesurable étant donné qu'elle permettait bien de résoudre le problème et que l'on pouvait directement vérifier sa fonctionnalité. Cependant, ce n'était pas réaliste étant donné que c'est un langage qu'aucun de nous ne connait et que le délai imposé était trop court pour nous permettre d'être au point dessus. La deuxième solution quant à elle était un objectif SMART et permettait de résoudre le problème. Il s'agissait d'indiquer au préalable le nombre de plantes différentes avant de sélectionner les espèces. Cette solution pouvant être réalisée en Python, l'objectif devenait donc réalisable

7 Conclusion

Pour conclure, nous avions une idée d'application assez précise au début du projet, en revanche une gestion de projet quelque peu chaotique nous a forcé à revoir certaines ambitions à la baisse. Nous avons pour pallier cela décidé de se concentrer sur les fondamentaux de cette application afin que ceux-ci fonctionnent normalement. Comme n'importe quel projet de développement, nous nous sommes heurtés à des problèmes de dysfonctionnements ou des difficultés à implémenter une fonction avec les outils dont nous disposions. En revanche, à partir de séances de brainstorming et de travail en groupe, nous avons su venir à bout de ses difficultés. Si pour obtenir une application utilisable dans les temps certaines fonctionnalités ont été sacrifiées au niveau de l'algorithme, tout a été mis en œuvre notamment au niveau de la base de données pour implémenter par la suite une version plus poussée de l'algorithme incluant de nouveaux facteurs.