HE2B ISIB

RAPPORT DE PROJETS

4IN1302 - Intelligence Artificielle

Alexandra DEGEEST

2023 - 2024

RESUME

Compte rendu portant sur les différents projets effectué ; les projets, les choix/décisions, ...

Antonin DE BREUCK

Le 24 mai 2024

Table des matières

Table de figures

Table des tableaux

1 Introduction

1.1 Explication du document

Ce document porte sur la réalisation de trois projets fait dans le cadre du cours d'Intelligence Artificiel.

Les 3 projets sont : les algorithmes de recherche, les algorithmes de jeux et programmation par contrainte.

1.2 Environnement d'implémentation

Tous les projets ont été développé en Python sur Visual Studio Code.

Mon choix s'est porté sur Python due à sa popularité pour sa richesse de son écosystème de bibliothèques et sa facilité d'utilisation.

2 Algorithmes de recherche

Sujet

L'objectif de ce projet est de développer un algorithme pour trouver le chemin optimal entre deux stations dans un réseau de métro. L'optimisation vise à minimiser le temps de trajet total, en prenant en compte non seulement les temps de déplacement entre les stations, mais aussi les temps de correspondance lorsque des changements de ligne sont nécessaires. Afin de savoir quel algorithme permet une meilleure optimisation, nous en testons plusieurs : BFS (Beam-first Search), Random Search, Greedy Search, Beam Search, A*.

Pour le bien de l'expérience, j'ai rajouté la future ligne 3 ainsi qu'une ligne fictive entre Simonis et Porte de Namur avec un coût abusif (150).

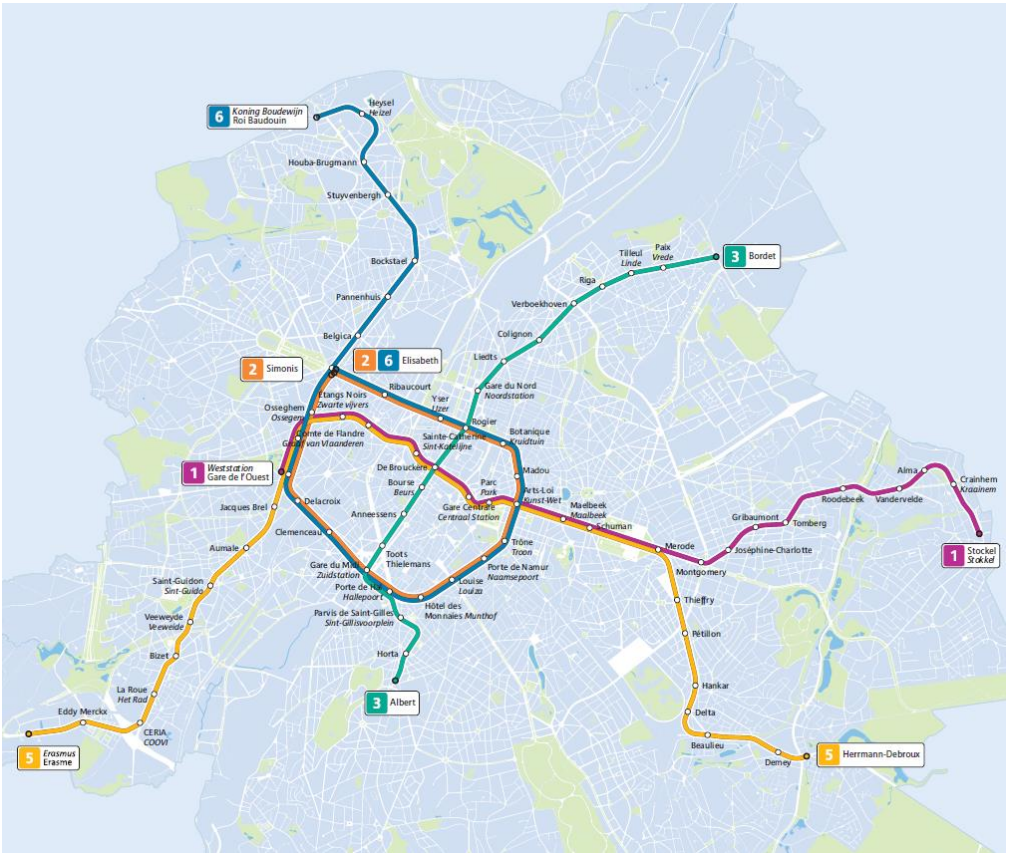

Figure 1 – Plan métro STIB avec la future ligne 3

2.1 Algorithme BFS (Breadth-first search)

2.1.1 Explication de l'implémentation

Nous avons choisi d'utiliser un algorithme de recherche en largeur (BFS, Breadth-First Search) avec des coûts associés pour chaque transition entre les stations. L'algorithme BFS est simple et efficace dans la recherche de chemins les plus courts dans les graphes.

Pour gérer les correspondances entre les lignes de métro, nous avons créé un graphe en incluant les numéros de lignes desservies pour chaque station et les coûts supplémentaires liés aux changements de ligne. Cette approche nous permet de calculer des temps de trajet réalistes qui incluent les temps de correspondance.

2.1.2 L'implémentation de l'algorithme

Initialisation : La file/queue est initialisée avec le point de départ.

Exploration des chemins :

- Tant que la file n'est pas vide, l'algorithme extrait le premier élément (chemin, coût actuel, ligne actuelle).
- Si la station actuelle est la destination, le chemin et le coût total sont retournés.
- Sinon, pour chaque voisin de la station actuelle, l'algorithme génère de nouveaux chemins en tenant compte des lignes de métro disponibles et ajoute les coûts de correspondance si la ligne change. Il ajoute ces nouveaux chemins à la fin de la file.

Gestion des correspondances : Si un changement de ligne est nécessaire, le coût de correspondance est ajouté au coût total. Les stations sont marquées comme visitées avec la ligne spécifique pour éviter de revenir sur nos pas et de tourner en boucle.

Retour des résultats : Si un chemin vers la destination est trouvé, il est retourné avec le coût total. Sinon, une indication d'absence de solution est retournée.

2.1.3 Package(s) utilisé(s)

Pour cette implémentation, nous avons utilisé le package de Python *collections* pour bénéficier de la structure de données *deque* qui est optimale pour l'implémentation de file/queue.

2.1.4 Vitesse

Python indique une exécution réelle de 0:00:01.150927.

2.1.5 Mémoire

Le chemin le plus court trouvé comprend 16 stations avec une moyenne de 3 de large.

$$O(b^m) = 3^{16} = 43.046.721 \quad (1)$$

2.2 Random Search

2.2.1 Explication de l'implémentation

L'algorithme de recherche aléatoire permet d'explorer le réseau de métro de manière non déterministe. Elle permet une exploration moins systématique mais peut offrir des solutions intéressantes en diversifiant les chemins explorés.

2.2.2 L'implémentation de l'algorithme

Initialisation : La file/queue est initialisée avec le point de départ.

Exploration des chemins :

- Tant que la file n'est pas vide, l'algorithme extrait le premier élément (chemin, coût actuel, ligne actuelle).
- Si la station actuelle est la destination, le chemin et le coût total sont retournés.
- Sinon, pour chaque voisin de la station actuelle, l'algorithme génère de nouveaux chemins en tenant compte des lignes de métro disponibles et ajoute les coûts de correspondance si la ligne change. Il ajoute ces nouveaux chemins de manière aléatoire dans la file.

Gestion des correspondances : Si un changement de ligne est nécessaire, le coût de correspondance est ajouté au coût total. Les stations sont marquées comme visitées avec la ligne spécifique pour éviter de revenir sur nos pas et de tourner en boucle.

Retour des résultats : Si un chemin vers la destination est trouvé, il est retourné avec le coût total. Sinon, une indication d'absence de solution est retournée.

2.2.3 Package(s) utilisé(s)

random : Utilisé pour insérer les nouveaux chemins à des positions aléatoires dans la file.

collections.deque : Package standard de Python qui est très pratique pour gérer une file.

2.2.4 Vitesse

A l'exécution, nous mesurons une attente de 3,10 secondes (montre en main).

Python indique une exécution réelle de 0:00:00.243907.

Attention, le temps est variable suite à la partie aléatoire dans la queue (reste de la grandeur de ~0.6 s pour l'exécution selon Python).

2.2.5 Mémoire

Pour l'algorithme de recherche aléatoire, la mémoire utilisée dépend également du nombre de chemins générés et de la profondeur explorée. Cependant, en raison de la nature aléatoire de l'insertion des chemins dans la file, la mémoire utilisée peut varier considérablement. Dans le pire des cas ça doit être exponentielle du même type que BFS.

Le chemin le plus court trouvé comprend 16 stations avec une moyenne de 3 de large.

$$O(b^m) = 3^{16} = 43.046.721 \quad (2)$$

2.3 Greedy Search

2.3.1 Explication de l'implémentation

Greedy Seach cherche à minimiser les coûts immédiats, grâce à une heuristique simple pour guider la recherche. La recherche Greedy n'est pas garantie de trouver le chemin optimal, mais elle est souvent efficace.

2.3.2 L'implémentation de l'algorithme

Initialisation : On initialise la file de priorité avec la station de départ et un ensemble pour suivre les stations visitées. Les heuristiques pour chaque station sont calculées en utilisant un BFS (Breadth-First Search) à partir de la station d'arrivée pour déterminer la distance en termes de nombre de stations.

Exploration des chemins :

- Tant que la file de priorité n'est pas vide, on extrait le chemin avec le coût heuristique le plus bas.
- Si la station actuelle est la destination, le chemin et le coût total sont retournés.
- Sinon, pour chaque station adjacente, on calcule le coût total en tenant compte du temps de trajet et du temps de transfert si nécessaire. Le chemin et le coût sont mis à jour et ajoutés à la file en fonction de l'heuristique (ordre croissant dans la file).

Heuristique : L'heuristique utilisée ici est la distance minimale en nombre de stations pour atteindre l'objectif, calculée par un BFS depuis la station d'arrivée.

2.3.3 Package(s) utilisé(s)

collections.deque : Pour implémenter une file de traitement efficace lors du calcul des heuristiques.

heapq : Pour gérer la file de priorité, permettant une insertion et une extraction rapide des chemins avec le coût heuristique le plus bas.

2.3.4 Vitesse

Python indique une exécution réelle de 0:00:00.042463.

2.3.5 Mémoire

La mémoire se calcule de la même manière que l'algorithme BFS car il y a un calcul d'heuristique avec BFS. Suite à ça, la mémoire maximum est équivalente. Après l'heuristique, ça prend moins de place.

Pour la mémoire maximum, tout comme le BFS, e chemin le plus court trouvé comprend 16 stations avec une moyenne de 3 de large.

$$O(b^m) = 3^{16} = 43.046.721 \quad (3)$$

Avec une implémentation du type DFS avec fonction de récursivité, on pourrait avoir une mémoire de maximum de $O(bm) = 3 * 20 = 60$ [1] qui est significativement mieux.

2.4 Beam Search

2.4.1 Explication de l'implémentation

Beam Search, entre le BFS et A* entre la largeur et la mémoire que ça prend, permet de conserver seulement un nombre de chemins ayant les heuristiques les plus faible. Très pratique quand le domaine est assez volumineux.

2.4.2 L'implémentation de l'algorithme

L'implémentation est identique que le Greedy Search a un seul changement près. Avant de voir l'exploration des chemins, on limite à un nombre de chemins voulu dans la queue (faisceau de 2 dans mon exemple).

2.4.3 Package(s) utilisé(s)

Les packages utilisés sont *collections.deque* et *heapq*, ils sont identiques à Greedy Search car c'est le même principe avec une taille limitée.

2.4.4 Vitesse.

Python indique une exécution réelle de 0:00:00.029999.

2.4.5 Mémoire

Dans le cours, il est mis que la mémoire est constante de la taille de la largeur. Cependant je trouve ça anormal que ça ne prenne pas en compte la longueur des chemins. Après des recherches sur internet [2], je vois que de base Beam Search est de l'ordre de la taille de la largeur de du faisceau (B) multiplié par la profondeur (m).

$$O(B * m) = 2 * 20 = 40 \quad (4)$$

En réalité, comme je regarde les enfants de chaque chemin, je dois également les prendre en compte, ce qui donne :

$$O(B * m * w) = 2 * 20 * 3 = 120 \quad (5)$$

2.5 A*

2.5.1 Explication de l'implémentation

L'algorithme A* est optimale car il trouve le chemin le plus court dans un graphe tout en tenant compte des coûts de déplacement et d'une fonction heuristique qui estime le coût restant pour atteindre l'objectif.

2.5.2 L'implémentation de l'algorithme

Initialisation : On initialise la file de priorité avec la station de départ et un ensemble pour suivre les stations visitées. Les heuristiques pour chaque station sont calculées en utilisant un BFS (Breadth-First Search) à partir de la station d'arrivée pour déterminer la distance en termes de nombre de stations.

Exploration des chemins :

- Tant que la file de priorité n'est pas vide, on extrait le chemin avec le coût réel + heuristique (f = cost + h) le plus bas.
- Si la station actuelle est la destination, le chemin et le coût total sont retournés.
- Sinon pour chaque station adjacente, un nouveau chemin est créé avec son coût total et son coût heuristique.
- Les chemins terminant par une station déjà visitée sont supprimés s'ils ont un coût supérieur ou égal au chemin déjà trouvé.
- Pour chaque station adjacente, on calcule le coût total en tenant compte du temps de trajet et du temps de transfert si nécessaire.
- Les nouveaux chemins formant des boucles sont rejetés sinon ils sont ajoutés à la file par ordre croissant en fonction de la somme du cout réel et de l'heuristique.

Heuristique : L'heuristique utilisée ici est la distance minimale en nombre de stations pour atteindre l'objectif, calculée par un BFS depuis la station d'arrivée.

2.5.3 Package(s) utilisé(s)

Les packages utilisés sont *collections.deque* et *heapq*, ils sont identiques à Greedy Search et Beam Search car c'est le même principe de liste avec un tris selon l'heuristique. Le seul changement est que l'on rajoute le cout réel à l'heuristique mais toujours triées.

2.5.4 Vitesse

Python indique une exécution réelle de 0:00:00.105992.

2.5.5 Mémoire

La mémoire que prend A* dépend de la profondeur du chemin le plus court (m) et du nombre moyen de branches/enfants que les contiennent (b) [3].

$$O(m * b) = 20 * 2 = 40 \quad (6)$$

Sur internet, « m » est parfois mis « d » pour cet algorithme.

Comparaison (distance Manhattan)

Afin d'avoir une meilleure vision pour comparer les cinq algorithmes utilisés, voici un tableau comparatif pour un trajet entre Stuyvenbergh et Roodebeek avec métro bloqué entre Sainte-Catherine et De Brouckère. Pour Greedy Search et Beam Search, ils utilisent une heuristique BFS à partir du goal avec une distance de Manhattan (l'heuristique = le nombre stations d'écart).

Algorithme	**Nombre de stations trouvées**	**Durée du trajet trouvé**	**Vitesse d'exécution**	**Mémoire**
BFS (Breadth-first search)	16 stations	173 min	115,0927 * 10^{-2} s	$b^m = 43.046.721$
Random Search	16 stations	179 min	39,8824 * 10^{-2} s	Pire cas : $b^m = 43.046.721$
Greedy Search	16 stations	173 min	4,2463 * 10^{-2} s	$b^m = 3.486.784.401$
Beam Search	16 stations	173 min	2,9999 * 10^{-2} s	$Bmw = 120$
A*	21 stations	31 min	10,5992 * 10^{-2} s	$bm = 40$

Tableau 1 – Tableau comparatif des algorithmes de recherche

BFS (Breadth-first search)

Chemin trouvé : ['Stuyvenbergh', 'Bockstael', 'Pannenhuis', 'Belgica', 'Simonis', 'Elisabeth', 'Ribaucourt', 'Yser', 'Rogier', 'Botanique', 'Madou', 'Arts-Loi', 'Maelbeek', 'Schuman', 'Merode', 'Montgomery', 'Joséphine-Charlotte', 'Gribaumont', 'Tomberg', 'Roodebeek']

➔ Plus court trajet mais un temps de trajet et de de calcul très long

Random Search

Chemin trouvé : ['Stuyvenbergh', 'Bockstael', 'Pannenhuis', 'Belgica', 'Simonis', 'Porte de Namur', 'Trône', 'Arts-Loi', 'Maelbeek', 'Schuman', 'Merode', 'Montgomery', 'Joséphine-Charlotte', 'Gribaumont', 'Tomberg', 'Roodebeek']

➔ Plus court trajet mais un temps de trajet vraiment très long et temps de calcul plus court Ça pourrait varier et avoir un temps de déplacement beaucoup plus long ou bien plus court.

Greedy Search et Beam Search

Chemin trouvé : ['Stuyvenbergh', 'Bockstael', 'Pannenhuis', 'Belgica', 'Simonis', 'Elisabeth', 'Ribaucourt', 'Yser', 'Rogier', 'Botanique', 'Madou', 'Arts-Loi', 'Maelbeek', 'Schuman', 'Merode', 'Montgomery', 'Joséphine-Charlotte', 'Gribaumont', 'Tomberg', 'Roodebeek']

➔ Plus court trajet mais un temps de trajet très long et temps de calcul plus court

A*

Chemin trouvé : ['Stuyvenbergh', 'Bockstael', 'Pannenhuis', 'Belgica', 'Simonis', 'Elisabeth', 'Ribaucourt', 'Yser', 'Rogier', 'De Brouckère', 'Gare Centrale', 'Parc', 'Arts-Loi', 'Maelbeek', 'Schuman', 'Merode', 'Montgomery', 'Joséphine-Charlotte', 'Gribaumont', 'Tomberg', 'Roodebeek']

➔ Plus long en station mais nettement plus cours en temps de trajet. Légèrement plus long en temps de calcul.

Conclusion

L'algorithme Beam-First Search calcul tout et prend un énorme temps et trouve le chemin ayant le moins de nœud mais pas forcément le moins coûteux en temps de trajet entre station.

L'algorithme Random Search est pratique pour de grand graphe mais dans mon cas, j'ai peu de chemins possibles, ce qui ne l'aide pas. Il va trouver peut-être plus rapidement un chemin mais il va peut-être trouver un chemin bien plus long aussi. (Ici le temps est légèrement plus long mais il pourrait viser d'autre chemin moins long en temps mais plus en nœud.)

L'algorithme Greedy Search est comme l'algorithme Beam-First Search dans ce projet car les deux doivent relier deux stations. L'avantage est de trouver plus facilement (moins de temps de calcul) le goal tout en continuant à prendre les heuristiques les plus faibles.

L'algorithme Beam Search est aussi pratique que le Greedy Search mais lui comporte le risque qu'il ne trouve jamais de station en preneur une heuristique faible qui ne mène à rien. Dans mon cas, un plan de métro, il n'y a pas une autre ligne proche du goal qui n'y est pas lié donc ça ne posera pas de soucis. L'avantage est qu'il avance plus vite en profondeur.

L'algorithme A* est rudement efficace car même s'il est plus lent, il prend en compte le coût réel et l'heuristique donc il est beaucoup plus juste et précis. On le voit bien ici avec un chemin ayant plus de nœud mais un temps de trajet vraiment moins long et optimisé !

➔ J'ai tenté de changer l'heuristique pour Greedy search et Beam search pour que la ligne fictive ait une heuristique plus grande mais ça n'a rien changé.

3 Algorithmes de jeu

3.1 Sujet

Le jeu choisi est un Puissance 8. Comme le Puissance 4, il faut aligner des pions horizontalement, verticalement et/ou diagonalement. Dans cette version, il ne faut pas aligner quatre pions mais huit pour réussir la partie. La grille ne fait donc pas 6x7 mais 10x13 pour permettre la réalisation de l'alignement des huit palets.

3.2 Fonction d'évaluation

La fonction d'évaluation regarde pour une coordonnée combien de coup peuvent être réalisé en regardant dans toutes les directions jusqu'au bout de la grille tant que c'est vide ou le joueur qui joue.

Il suffit donc de boucler cette fonction pour chaque case et nous avons le nombre de chance de réussite par case. Nous pouvons limiter aux pions les plus bas (vu qu'ils tombent par gravité sur ceux les plus en dessous). Il risque cependant d'avoir des doublons, détectant une possibilité de réussite horizontale et évaluant les réussites du pion à côté, trouvant également la même réussite sans le savoir.

3.3 Profondeur utilisée

J'ai mis une profondeur de 3 qui est assez correcte courte mais l'IA n'est vraiment pas intelligente. Je pense qu'il y a un souci avec l'heuristique qui n'est probablement pas assez intelligente. Elle ne renvoie pas toujours une valeur intéressante à utiliser/exploiter par rapport à d'autres emplacements.

A titre d'information :

Profondeur	**Durée d'attente**
1	Instantané
2	Quelques secondes à instantané
3	~10 s
4	~30 s
5	>1 min

Tableau 2 – Durée d'attente par profondeur (choisie au départ)

3.4 Optimisation mise en place

Intégré dans la récursivité de la fonction min-max, il y a une optimisation alpha-beta. Il y a peut-être d'autres optimisations qui pourraient être intéressante à utiliser mais nous n'en avons pas vu d'autres en classe.

3.5 Flexibilité de l'algorithme

L'algorithme recalcul toujours en fonction de la grille actuelle. Il reste cohérent dans ce que lui propose et change parfois de stratégie quand le joueur interrompt dans sa lancée.

4 Programmation par contrainte

4.1 Sujet

Création d'un planning cohérent de mon agenda du mois d'avril 2024 en tenant compte des priorités et des contraintes des différentes activités. C'est un problème courant de planification et d'organisation en fonction de contraintes de non-chevauchement et de priorité.

4.2 Contenu

Ayant énormément d'activités, c'est souvent compliqué de trouver des solutions pour que toutes mes activités concordent.

C'est pourquoi j'ai pensé utiliser l'IA avec une programmation par contrainte afin de me faire un agenda/planning cohérent pour le mois d'avril 2024.

J'ai choisi ce mois due à la charge élevée et avec parfois plusieurs activités le même jour. Pour ce faire, l'algorithme doit pouvoir m'assigner un maximum d'activités sans en avoir au même moment (chevauchement).

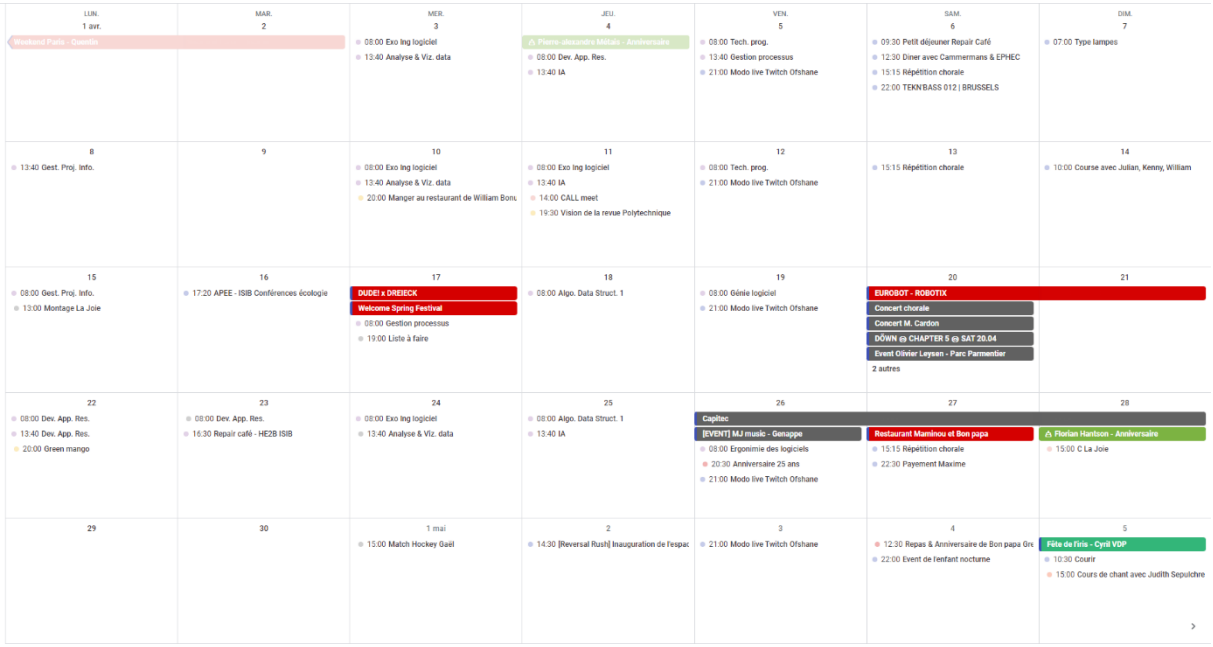

Figure 2 – Mon agenda d'avril 2024

4.3 Package(s)/Librairie(s) utilisé(s)

Le principal package utilisé pour l'implémentation est *docplex.cp.model* [4] de la bibliothèque DOcplex d'IBM. DOcplex est un package Python pour la programmation par contraintes et la programmation mathématique, qui fait partie de l'IBM Decision Optimization suite. Il permet de créer, manipuler et résoudre des modèles d'optimisation avec le solveur IBM ILOG CPLEX Optimization Studio (à télécharger [5] et installer lors de l'utilisation du code Python).

La deuxième librairie est *datetime* permettant de gérer les dates. Elle sert à pouvoir mettre les dates des événements (début et fin de chaque activités) en minutes et de faire l'inverse quand l'algorithme a trouvé une solution, pour afficher de manière lisible le programme du mois.

4.4 Algorithme(s)

L'algorithme utilisé est basé sur la programmation par contraintes (CP). Les étapes de l'algorithme incluent :

1. Modélisation des activités : Chaque activité est modélisée comme une variable d'intervalle avec des contraintes sur les heures de début et de fin.
2. Définition des contraintes :
   - Non-chevauchement : Aucune activité ne doit se chevaucher avec une autre.
   - Priorité : Certaines activités ont une priorité plus élevée et peuvent exclure d'autres activités.
   - Exclusivité : Certaines activités optionnelles s'excluent mutuellement (à la suite d'un des points précédents).
3. Optimisation : L'algorithme maximise la présence des activités optionnelles tout en respectant les contraintes de non-chevauchement et de priorité.
4. Résolution : Le modèle est résolu en utilisant le solveur CP d'IBM, qui trouve une solution satisfaisant toutes les contraintes.

4.5 Contraintes

Les principales contraintes du modèle incluent :

1. Non-chevauchement : Les activités ne doivent pas se chevaucher dans le temps.
2. Priorité maximale : L'activité "Concours de robotique" a une priorité maximale et exclut toutes les autres activités le 20 et 21 avril.
3. Exclusivité des activités optionnelles : Les activités optionnelles ne peuvent pas avoir lieu en même temps mais doivent être maximisé le plus possible dans l'agenda.

4.6 Domaine de définition

Le domaine de définition du problème est la date et l'heure avec la durée de chaque activité.

4.7 Résultat

Voici le résultat obtenu :

```
! ---------------------------------------------------- CP Optimizer 22.1.1.0 --
! Maximization problem - 16 variables, 6 constraints
! Presolve      : 1 extractable eliminated
! Initial process time : 0.01s (0.01s extraction + 0.00s propagation)
!  . Log search space  : 5.0 (before), 5.0 (after)
!  . Memory usage      : 442.2 kB (before), 442.2 kB (after)
! Using parallel search with 12 workers.
! ---------------------------------------------------------------------------
!          Best Branches  Non-fixed    W       Branch decision
                       0         16               -
+ New bound is 11
! Using iterative diving.
! Using temporal relaxation.
*           11        1  0.04s        1      (gap is 0.00%)
! ---------------------------------------------------------------------------
! Search completed, 1 solution found.
! Best objective         : 11 (optimal - effective tol. is 0)
! Best bound             : 11
! ---------------------------------------------------------------------------
! Number of branches     : 158922
! Number of fails        : 14565
! Total memory usage     : 5.0 MB (4.9 MB CP Optimizer + 0.0 MB Concert)
! Time spent in solve    : 0.04s (0.03s engine + 0.01s extraction)
! Search speed (br. / s) : 5297400.0
! ---------------------------------------------------------------------------
Exo ingé. logiciel: Wednesday 03 April 2024 at 08:00 - Wednesday 03 April 2024 at 12:30
Analyse & visualisation de data: Wednesday 03 April 2024 at 13:40 - Wednesday 03 April 2024 at 18:00
Développement d'app réseau: Thursday 04 April 2024 at 08:00 - Thursday 04 April 2024 at 12:30
IA: Thursday 04 April 2024 at 13:40 - Thursday 04 April 2024 at 18:00
Chorale 6: Saturday 06 April 2024 at 15:15 - Saturday 06 April 2024 at 17:30
Chorale 13: Saturday 13 April 2024 at 15:15 - Saturday 13 April 2024 at 17:30
Concours de robotique: Saturday 20 April 2024 at 00:00 - Sunday 21 April 2024 at 23:59
Ergonomie des logiciels: Friday 26 April 2024 at 08:00 - Friday 26 April 2024 at 12:30
Anniversaire 25 ans Florian: Friday 26 April 2024 at 20:30 - Saturday 27 April 2024 at 05:00
Chorale 27: Saturday 27 April 2024 at 15:15 - Saturday 27 April 2024 at 17:30
Repas de famille 27 avril: Saturday 27 April 2024 at 19:30 - Saturday 27 April 2024 at 23:00
```

Figure 3 – Résultat de l'algorithme de contrainte

5 Ressources

[1] Keramat, «Why is the space-complexity of greedy best-first search is O(bm),» 24 Avril 2022. [En ligne]. Available: https://ai.stackexchange.com/questions/17971/why-is-the-space-complexity-of-greedy-best-first-search-is-mathcalobm. [Accès le 27 Mai 2024].

[2] «Define Beam Search,» [En ligne]. Available: https://www.javatpoint.com/define-beam-search. [Accès le 28 Mai 2024].

[3] G. P. Graham Cox, «A* Pathfinding Algorithm,» 18 Mars 2024. [En ligne]. Available: https://www.baeldung.com/cs/a-star-algorithm#:~:text=The%20space%20complexity%20of%20standard,and%20are%20never%20going%20to.. [Accès le 28 Mai 2024].

[4] A. Chabrier, «Scheduling in Python with Constraint Programming,» 6 Novembre 2019. [En ligne]. Available: https://medium.com/@AlainChabrier/scheduling-with-constraint-programming-35a23839e25c. [Accès le 9 Mai 2024].

[5] IBM, «IBM ILOG CPLEX Optimization Studio,» [En ligne]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio. [Accès le 25 Mai 2024].

[6] C. E. L. R. L. R. C. S. Thomas H. Cormen, «Introduction to Algorithms [Fourth Edition],» 2022. [En ligne]. Available: https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf. [Accès le 28 Mai 2024].

[7] S. R. e. P. Norvig, «Artificial Intelligence: A Modern Approach - Chapitre 4, Informed search and exploration,» 22 Août 2022. [En ligne]. Available: https://aima.cs.berkeley.edu/4th-ed/pdfs/newchap04.pdf. [Accès le 27 Mai 2024].