



**Université Sultan Moulay Slimane
Faculté Polydisciplinaire Béni-Mellal
Département Informatique**

Projet de Fin d'Études de la Licence Fondamentale Sciences
Mathématiques et Informatiques

**Étude et Implémentation de l'algorithme de
plus court chemin A* en utilisant
le langage Python**

Réalisé par :
Abdellah Karani

Encadré par :
Monsieur Hicham Mouncif

Soutenu le 20/06/2025 devant le Jury :

Monsieur H. Mouncif
Monsieur Y. Madani
Monsieur M. Zouina
Monsieur H. Ouchitachene

Année Universitaire : 2024-2025

Remerciements

Tout d'abord, je remercie Dieu tout puissant qui m'a donné la force et la volonté pourachever ce modeste travail.

Ensuite, je tiens à exprimer toute ma gratitude à mon encadrant **Monsieur Hicham Mouncif** de m'avoir proposé ce sujet de fin d'étude, et de m'avoir suivi, guidé, conseillé et aidé durant toute la période du travail.

Je remercie également les membres du jury, M. Hicham Mouncif, M. Youness Madani, M. Mohammed Zouina et M. Hicham Ouchitachene, qui m'ont honoré en acceptant d'examiner mon travail.

Je remercie sincèrement les professeurs qui m'ont enseigné durant mon parcours universitaire à l'université « **Sultan Moulay Slimane** ».

Un grand merci à ma famille pour son indéfectible soutien, sa patience et sa complaisance. Ce travail vous est dédicacé. Je remercie aussi tous mes amis pour leur prévenance et leurs encouragements.

Abdellah Karani

Dédicaces

Tout d'abord, à mes parents, je souhaite exprimer ma gratitude pour Leur soutien inconditionnel et leurs encouragements constants ont été une source inestimable de force et de détermination pour poursuivre mes rêves. Leur confiance en moi a été le moteur essentiel. Leur présence et leurs paroles d'encouragement ont été un pilier solide sur lequel je me suis appuyé tout au long de cette aventure. Leur amour et leur soutien ont été les fondements de ma réussite et je leur suis profondément reconnaissant.

À mon professeur MONSIEUR HICHAM MOUNCIF, je suis aussi très reconnaissant pour son encadrement. Ses précieux conseils et orientations ont joué un rôle clé dans la réussite de ce projet. Sa connaissance et son expérience m'ont inspiré et ont contribué à ma croissance personnelle et professionnelle.

À toute ma famille, merci du fond du cœur pour votre amour et votre soutien constants. Vous êtes mes piliers et ma plus grande richesse.

Table des matières

Introduction générale	5
1 Analyse du Contexte et Cahier des Charges	6
1.1 Contexte du projet	6
1.2 Problématique	7
1.3 Objectifs du projet	7
1.4 Organisation du rapport	8
2 Fondements théoriques	10
2.1 Histoire des graphes	10
2.1.1 Origines mathématiques	10
2.1.2 Développements ultérieurs	11
2.1.3 Applications modernes	11
2.2 Notions de base sur les graphes	12
2.2.1 Qu'est-ce qu'un graphe ?	12
2.2.2 Les différents types de graphes	12
2.2.3 Représentation des graphes	19
2.3 Le principe de l'algorithme A*	23
2.3.1 Introduction à l'algorithme A*	23
2.3.2 Éléments fondamentaux	23
2.3.3 Concepts clés	24
2.3.4 Étapes de l'algorithme	25
2.3.5 Exemple pratique	26
2.4 Comparer A* avec d'autres algorithmes	28
2.4.1 Algorithme de Dijkstra	28
2.4.2 Recherche en largeur (BFS)	28
2.4.3 Recherche en profondeur (DFS)	29
2.4.4 Tableau résumé	29
2.4.5 Conclusion	29
3 Implémentation de l'algorithme A*	30
3.1 Développement en Python	30
3.1.1 Le langage Python	30
3.1.2 Choix des structures de données	31
3.2 Explication de l'implémentation	31
3.2.1 Représentation du graphe	31
3.2.2 Table heuristique	32
3.2.3 Fonction de calcul du coût	33

3.2.4	Fonction principale	33
3.2.5	Exemple d'utilisation	35
3.3	Performance de l'algorithme A*	36
4	Analyse et conception de l'application	37
4.1	Langage de modélisation	37
4.1.1	Description d'UML	37
4.1.2	Pourquoi UML	38
4.2	Diagrammes	39
4.2.1	Présentation	39
4.2.2	Analyse des besoins fonctionnels	40
4.2.3	Modélisation statique : Diagramme de classe	41
4.2.4	Modélisation dynamique : Diagramme de séquence	44
5	Architecture, visualisation et interface utilisateur	51
5.1	Introduction	51
5.2	Choix des bibliothèques graphiques	51
5.3	Architecture de l'application (fichiers modulaires)	52
5.3.1	Structure modulaire	53
5.3.2	Description des modules	53
5.3.3	Flux de données	54
5.4	Résultats et exemples visuels	54
5.4.1	Exemple réel d'un graphe	54
5.4.2	Lancement de l'application	55
Conclusion générale	63	
Annexe	65	

Introduction générale

L'algorithme A* (prononcé « A étoile ») est un algorithme de recherche du plus court chemin dans un graphe largement utilisé en informatique, et plus particulièrement dans le domaine de l'intelligence artificielle. Son rôle principal est de trouver le chemin le plus court entre un nœud initial et un nœud final dans un graphe, tout en optimisant les ressources et le temps de calcul. Cet algorithme combine les avantages de la recherche en largeur et de la recherche informée grâce à une fonction d'évaluation heuristique, ce qui lui permet d'être à la fois efficace et performant.

Le choix de ce sujet s'inscrit dans la volonté d'approfondir les concepts fondamentaux des algorithmes de recherche et leur application pratique dans le traitement de problèmes complexes. En effet, l'algorithme A* est un exemple emblématique d'algorithme de planification, utilisé dans de nombreux domaines tels que la robotique, la navigation, les jeux vidéo, et la modélisation de réseaux. Son importance réside dans sa capacité à gérer des graphes orientés avec des poids positifs tout en garantissant un résultat optimal lorsqu'une heuristique admissible est utilisée.

L'objectif du projet est d'étudier l'algorithme A* en théorie et de le programmer en Python pour résoudre le plus court chemin sur des graphes complexes, avec une interface visuelle montrant les graphes et les chemins trouvés.

En résumé, Ce projet vise à offrir une base théorique et une application pratique de l'algorithme A*, montrant son utilité et ses performances en recherche de chemin.

Chapitre 1

Analyse du Contexte et Cahier des Charges

1.1 Contexte du projet

Dans le monde moderne, chercher le chemin le plus court est un problème très courant. On le retrouve dans plusieurs situations : par exemple, lorsqu'on utilise une application GPS pour trouver l'itinéraire le plus rapide entre deux villes, lorsqu'un robot doit se déplacer dans une pièce sans toucher les obstacles, ou encore dans les jeux vidéo, où un personnage doit aller d'un endroit à un autre rapidement.

Pour résoudre ce genre de problème, on utilise ce qu'on appelle des **algorithmes de recherche de chemin**. Ces outils sont très importants en informatique. Ils permettent à un programme de choisir le meilleur chemin parmi plusieurs possibilités. L'un des algorithmes les plus connus est **l'algorithme A*** (on dit "A étoile"). Il est apprécié car il est à la fois **rapide** et **précis**. Il utilise une méthode intelligente pour éviter les chemins inutiles et trouver plus vite la solution.

Ce projet de fin d'études s'inscrit dans le cadre de ma formation en informatique. L'objectif est d'étudier en détail le fonctionnement de l'algorithme A*, de comprendre comment il prend ses décisions, et de voir dans quels domaines il peut être utilisé (cartes, robotique, jeux, etc.).

En développant ce projet, je peux appliquer mes connaissances théoriques sur les graphes, les heuristiques, et les structures de données. Je travaille aussi sur la **programmation en Python**, ce qui me permet d'améliorer mes compétences techniques en codage, en résolution de problèmes, et en organisation du code.

Ce travail est donc à la fois **pratique** et **pédagogique**, car il me permet

de mieux comprendre un outil fondamental de l'informatique tout en me préparant au monde professionnel.

1.2 Problématique

La recherche du **chemin le plus court** dans un graphe est un problème fondamental en informatique. Cette question simple en apparence devient complexe lorsqu'il y a de nombreux nœuds, des obstacles ou des contraintes à prendre en compte.

Ce type de problème apparaît dans plusieurs domaines :

- **En logistique** : pour optimiser les trajets de livraison et réduire les coûts.
- **Dans les jeux vidéo** : pour que les personnages trouvent un chemin réaliste et rapide.
- **En robotique** : pour planifier les déplacements d'un robot en évitant les obstacles.
- **Dans les systèmes GPS** : pour calculer l'itinéraire le plus rapide ou le plus court.

Les algorithmes classiques, comme **Dijkstra**, sont capables de trouver un chemin optimal. Cependant, ils peuvent être **lents** sur des grands graphes, car ils explorent beaucoup de nœuds, même ceux qui ne mènent pas rapidement à la solution.

Pour répondre à cette difficulté, il faut des méthodes plus **intelligentes**, capables d'orienter la recherche dans la bonne direction. C'est dans ce contexte que s'inscrit l'intérêt pour des algorithmes comme **A***, qui utilisent une estimation de la distance vers la cible pour éviter de perdre du temps sur des chemins inutiles.

Comprendre comment améliorer la recherche de chemin est donc une problématique centrale pour de nombreuses applications informatiques modernes.

1.3 Objectifs du projet

Ce projet a pour objectif principal de **comprendre**, **implémenter** et **visualiser** l'algorithme A* en utilisant le langage Python. Pour bien atteindre cet objectif, plusieurs étapes importantes sont prévues :

- **Étude théorique approfondie** : Étudier en détail le fonctionnement de l'algorithme A*, comprendre comment il combine le coût déjà parcouru et une estimation du coût restant (heuristique) pour choisir le

chemin le plus efficace. Cela comprend aussi les notions de base sur les graphes, comme les sommets (points), les arêtes (liens), et les poids (coûts associés aux déplacements).

- **Choix des bonnes structures de données** : Identifier les structures adaptées pour représenter le graphe et gérer les éléments nécessaires à l'algorithme, comme les listes d'adjacence pour représenter les connexions entre sommets, et les files de priorité pour sélectionner rapidement le prochain sommet à explorer. Ces choix sont essentiels pour améliorer la rapidité et l'efficacité du programme.
- **Développement en Python** : Programmer une version complète de l'algorithme A* capable de traiter différents types de graphes orientés avec des poids positifs. Le code devra être clair, fonctionnel, et respecter les bonnes pratiques de programmation.
- **Validation par des tests** : Tester l'algorithme sur plusieurs exemples simples pour vérifier qu'il trouve bien le chemin le plus court. Ces tests utiliseront des graphes petits et faciles à analyser, présentés sous forme de listes, pour s'assurer que l'algorithme fonctionne comme prévu.
- **Création d'une visualisation graphique** : Développement d'une interface visuelle qui montre le graphe, les sommets explorés, ainsi que le chemin optimal trouvé. Cette visualisation permettra de mieux comprendre le comportement de l'algorithme. Des bibliothèques Python comme NetworkX, Matplotlib ou Tkinter seront utilisées.
- **Organisation et modularité du code** : Structurer le projet en plusieurs fichiers et fonctions claires. Cette organisation facilitera la lecture, la maintenance et les futures améliorations du programme. Un code bien organisé est aussi plus facile à comprendre et à partager.

Ces objectifs permettront de mieux comprendre un algorithme fondamental en informatique tout en développant des compétences pratiques en programmation et en visualisation.

1.4 Organisation du rapport

Ce rapport est structuré de la manière suivante :

- **Chapitre 1 – Analyse du Contexte et Cahier des Charges** : Présentation du contexte général du projet, formulation de la problématique, définition des objectifs, et description de l'organisation du rapport.
- **Chapitre 2 – Fondements théoriques** : Introduction à l'histoire des graphes, explication des notions de base, description du fonctionnement de l'algorithme A*, et comparaison avec d'autres algorithmes

de recherche de chemin plus court.

- **Chapitre 3 – Implémentation de l'algorithme A*** : Détail du développement en Python, choix des structures de données, et explication complète de l'implémentation.
- **Chapitre 4 – Analyse et conception de l'application** : Étude des besoins fonctionnels, modélisation statique (diagramme de classes) et modélisation dynamique (diagramme de séquence).
- **Chapitre 5 – Architecture, Visualisation et interface utilisateur** : Choix des bibliothèques graphiques, présentation de l'architecture modulaire du programme et présentation des résultats à l'aide d'exemples visuels,
- **Conclusion générale** : Résumé des travaux réalisés, difficultés rencontrées, et pistes d'amélioration ou d'évolution future.

Chaque chapitre a pour but de présenter clairement les étapes du projet, depuis l'étude du problème jusqu'à sa résolution concrète avec une application Python interactive.

Chapitre 2

Fondements théoriques

2.1 Histoire des graphes

Les graphes, en tant que structure mathématique, ont une histoire riche qui remonte au XVIII^e siècle. Leur développement est étroitement lié à des problèmes concrets et à des avancées théoriques majeures en mathématiques et en informatique.

2.1.1 Origines mathématiques

L'histoire des graphes commence en 1736 avec le mathématicien Leonhard Euler. Il a résolu un problème célèbre : *le problème des 7 ponts de Königsberg*.

Voici ce problème simple :

- La ville avait 7 ponts
- Euler s'est demandé : peut-on traverser tous les ponts sans repasser deux fois sur le même ?
- Il a prouvé que c'était impossible

En résolvant ce problème, Euler a inventé sans le savoir les bases des graphes modernes :

- Les **sommets** (points) = les zones de la ville
- Les **arêtes** (lignes) = les ponts entre les zones

C'était la première fois qu'on utilisait cette façon de représenter les connexions entre les choses. Aujourd'hui, on utilise partout ces "graphes" pour modéliser des réseaux.

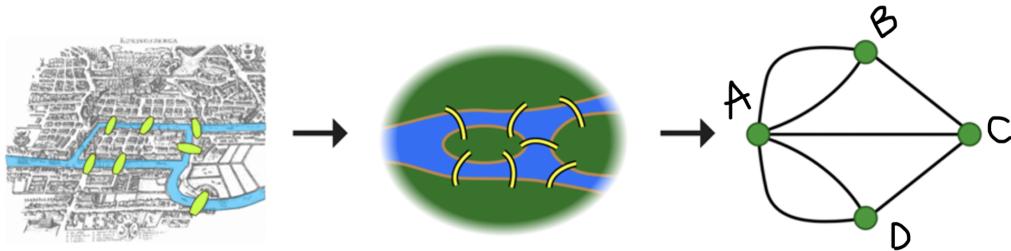


FIGURE 2.1 – Représentation historique du problème des sept ponts de Königsberg

2.1.2 Développements ultérieurs

Au XIX^e siècle, des mathématiciens comme Arthur Cayley ont contribué à formaliser la théorie des graphes :

- 1857 : Cayley étudie les arbres dans le cadre de ses travaux sur les structures chimiques
- 1878 : Sylvester introduit le terme « graphe » dans la littérature mathématique
- 1936 : Dénes Kőnig publie le premier traité complet sur la théorie des graphes

2.1.3 Applications modernes

Au XX^e siècle, les graphes sont devenus essentiels en informatique :

- Années 1950 : Utilisation dans les réseaux électriques et les systèmes de transport
- Années 1970 : Fondement des structures de données et algorithmes (comme l'algorithme de Dijkstra)
- XXI^e siècle : Central dans les réseaux sociaux, le machine learning et l'optimisation

« La théorie des graphes est un langage universel pour modéliser les relations entre objets, des molécules aux réseaux informatiques. »
– Béla Bollobás, théoricien des graphes

Cette évolution historique montre comment les graphes sont passés d'une curiosité mathématique à un outil fondamental en informatique, justifiant leur importance dans des domaines comme l'optimisation de chemin qui nous intéresse dans ce travail.

2.2 Notions de base sur les graphes

2.2.1 Qu'est-ce qu'un graphe ?

Un **graphe** est une structure mathématique utilisée pour modéliser des relations entre objets. Il est composé d'un ensemble de *sommets* (ou noeuds) et d'un ensemble d'*arêtes* (ou arcs), représentant les connexions ou interactions entre ces sommets.

Par exemple, on peut représenter les pays comme des sommets, et les routes internationales reliant ces pays comme des arêtes. De même, dans un réseau social, chaque utilisateur est un sommet, et chaque relation d'amitié ou de suivi est une arête entre deux sommets.

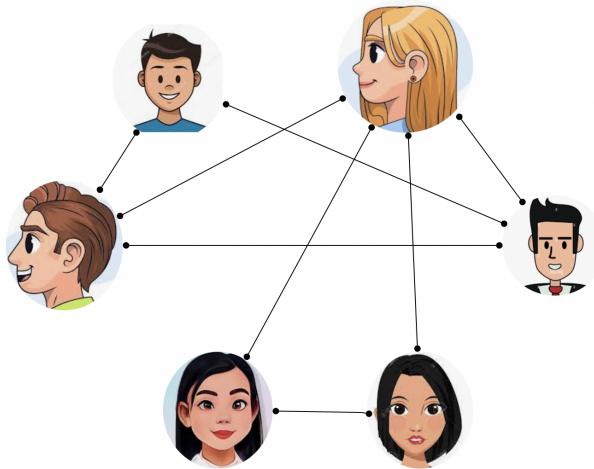


FIGURE 2.2 – Exemple de graphe représentant un réseau de personnes et leurs relations

En mathématiques, un graphe est représenté par $G = (V, E)$, où V est l'ensemble des sommets et E est l'ensemble des arêtes (ou relations) reliant ces sommets.

Exemple : Soit un graphe $G = (V, E)$ avec $V = \{A, B, C, D\}$ représentant quatre villes, et $E = \{(A, B), (A, C), (B, D), (C, D)\}$ représentant les routes qui relient ces villes.

2.2.2 Les différents types de graphes

On distingue plusieurs types de graphes, parmi lesquels on peut citer :

Graphe non orienté

Un **graphe non orienté** est une structure composée de plusieurs points appelés **sommets** ou **nœuds**, reliés par des liens appelés **arêtes**.

Dans ce type de graphe, les arêtes n'ont **pas de direction** précise. Cela signifie que si une arête relie deux sommets u et v , on peut se déplacer librement entre u et v dans les deux sens, sans distinction.

Autrement dit, l'arête représente une relation **symétrique** ou **bidirectionnelle** entre ces deux points. Par exemple, dans un réseau social, une relation d'amitié est souvent modélisée par un graphe non orienté car si u est ami avec v , alors v est aussi ami avec u .

Les arêtes sont représentées par des **paires non ordonnées** de sommets, notées $\{u, v\}$. Cela signifie que l'arête $\{u, v\}$ est la même que $\{v, u\}$, sans différence de sens.

Les graphes non orientés sont utilisés dans de nombreux domaines, comme :

- Les réseaux sociaux (relations d'amitié),
- Les réseaux électriques (connexion entre postes),
- Les réseaux informatiques où la communication est possible dans les deux sens,
- Les cartes où les routes sont à double sens.

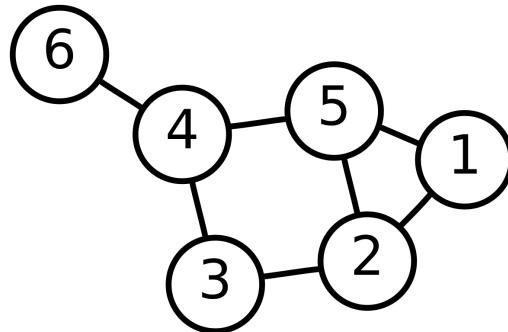


FIGURE 2.3 – Exemple d'un graphe non orienté

À partir de la figure ci-dessus, on peut décrire ce graphe non orienté de manière mathématique.

On note $G = (V, E)$, où :

- $V = \{1, 2, 3, 4, 5, 6\}$ est l'ensemble des sommets,
- $E = \{\{6, 4\}, \{4, 5\}, \{4, 3\}, \{5, 2\}, \{5, 1\}, \{2, 3\}, \{1, 2\}\}$ est l'ensemble des arêtes, où chaque paire non ordonnée $\{u, v\}$ représente une arête reliant les sommets u et v .

En résumé, un graphe non orienté montre simplement que la relation entre deux sommets est réciproque, sans notion de sens ou direction.

Graphe orienté (ou digraphe)

Un **graphe orienté**, aussi appelé *digraphe*, est une structure composée de plusieurs points appelés **sommets** ou **nœuds**, reliés entre eux par des liens appelés **arcs**.

Dans un graphe orienté, chaque arête a un **sens précis** : elle va d'un sommet u vers un sommet v . Cela signifie que l'on peut aller de u à v , mais pas forcément dans l'autre sens. On représente cette arête par une flèche allant de u vers v .

Par exemple, si on imagine un réseau de routes à sens unique entre des villes, chaque route est une arête orientée, car elle ne permet le passage que dans une seule direction.

Les graphes orientés sont très utilisés pour modéliser des systèmes où la direction est importante, comme :

- Les réseaux routiers avec des sens uniques,
- Les flux d'information ou de données,
- Les relations de dépendance dans un projet,
- Les transitions d'états dans une machine à états.

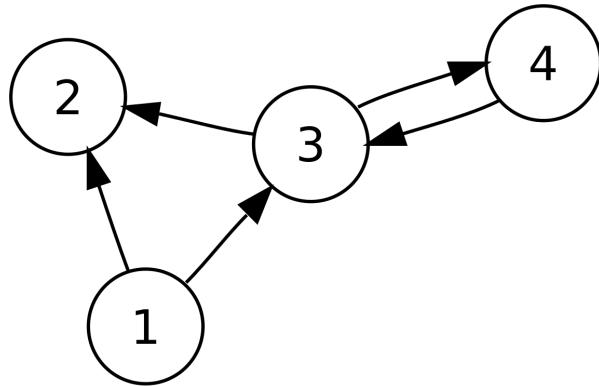


FIGURE 2.4 – Exemple d'un graphe orienté

À partir de la figure ci-dessus, on peut décrire ce graphe orienté de manière mathématique.

On note $G = (V, E)$ où :

- $V = \{1, 2, 3, 4\}$ est l'ensemble des sommets,
- $E = \{(1, 2), (1, 3), (3, 2), (3, 4), (4, 3)\}$ est l'ensemble des arcs, où chaque paire ordonnée (u, v) représente un arc allant du sommet u vers le sommet v .

En résumé, un graphe orienté montre que chaque lien a une direction précise, indiquant le sens dans lequel on peut aller d'un sommet à un autre.

Graphe pondéré

Un **graphe pondéré** est un graphe dans lequel chaque arête (ou arc) est associée à un **poids**, c'est-à-dire une valeur numérique qui représente une grandeur comme la distance, le coût, ou le temps.

Ces poids permettent de mieux modéliser des situations où il faut tenir compte de la quantité ou du prix pour aller d'un point à un autre.

Par exemple, dans un réseau routier, le poids peut être la distance en kilomètres entre deux villes, ou le temps nécessaire pour les relier.

Les graphes pondérés sont utilisés dans de nombreux domaines, notamment :

- La planification de trajets avec des coûts ou des distances,
- L'optimisation des réseaux de transport ou de communication,
- La gestion de ressources avec un coût associé aux connexions.

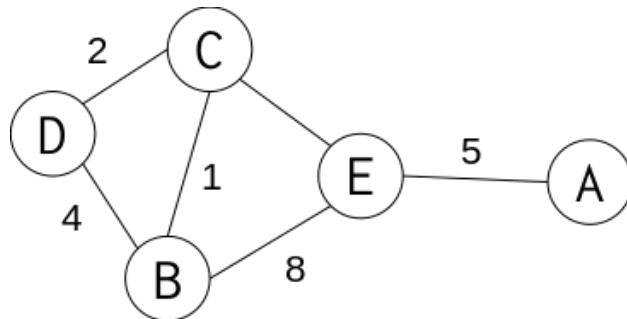


FIGURE 2.5 – Exemple d'un graphe pondéré

À partir de la figure ci-dessus, on peut décrire ce graphe pondéré de manière mathématique.

On note $G = (V, E)$ où :

- $V = \{A, B, C, D, E\}$ est l'ensemble des sommets,

- $E = \{(A, E, 5), (E, C, 2), (E, B, 8), (B, C, 1), (B, D, 4), (D, C, 2)\}$ est l'ensemble des arêtes pondérés, où chaque triplet (u, v, w) représente un arêt allant du sommet u vers le sommet v avec un poids w .

En résumé, un graphe pondéré permet d'ajouter une information importante sur chaque lien, ce qui rend la modélisation plus précise et utile pour résoudre des problèmes complexes.

Graphe simple

Un **graphe simple** est un graphe qui respecte deux règles importantes :

- Il ne contient pas de **boucles**, c'est-à-dire qu'aucune arête ne relie un sommet à lui-même.
- Il ne contient pas d'**arêtes multiples** entre deux sommets, ce qui signifie qu'il n'y a qu'une seule arête possible entre chaque paire de sommets.

Ces règles rendent le graphe plus simple à étudier et à utiliser, car il évite les cas complexes de liens redondants ou de connexions internes à un même point.

Les graphes simples sont souvent utilisés pour modéliser des relations claires et directes.

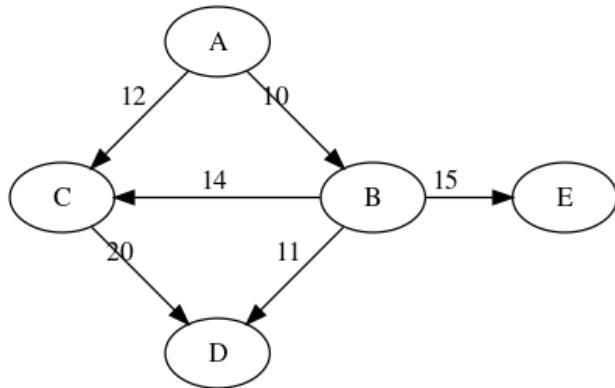


FIGURE 2.6 – Exemple d'un graphe simple

Graphe spécial (arbre)

Un arbre est un type particulier de graphe simple. Rappelons qu'un graphe simple ne contient ni boucle (c'est-à-dire pas d'arête reliant un sommet à lui-même), ni arêtes multiples entre deux mêmes sommets.

De plus, un arbre est un graphe simple qui est **connexe** (on peut aller d'un sommet à un autre par un chemin) et **sans cycle** (il n'existe pas de

chemin fermé). Cela signifie qu'un arbre avec n sommets possède exactement $n - 1$ arêtes.

Ainsi, un arbre est un graphe simple spécial, souvent utilisé en informatique pour représenter des structures hiérarchiques.

Il existe plusieurs types d'arbres, chacun avec ses propriétés propres. Parmi les plus connus, on trouve l'**arbre binaire**, dans lequel chaque nœud a au plus deux enfants : un fils gauche et un fils droit.

Les arbres binaires sont très utilisés en informatique, notamment pour les arbres de recherche, les tas, ou les arbres de syntaxe.

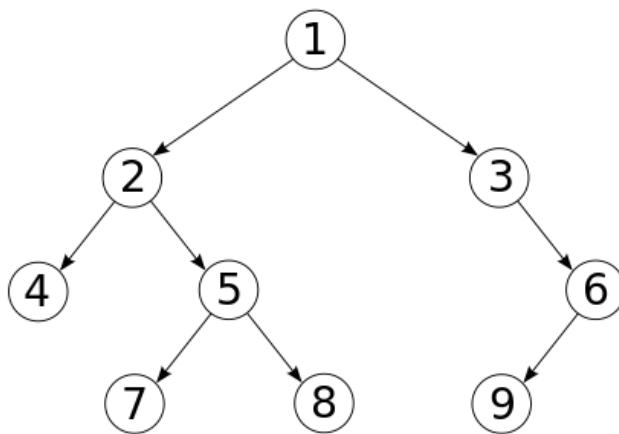


FIGURE 2.7 – Exemple d'un arbre binaire

Graphe complet

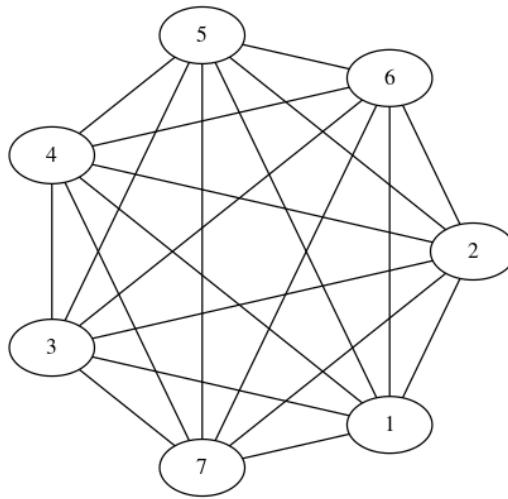
Un graphe complet est un graphe simple dans lequel chaque paire de sommets est reliée par exactement une arête. Autrement dit, tous les sommets sont connectés entre eux.

Si le graphe a n sommets, alors il contient toutes les arêtes possibles, soit $\frac{n(n-1)}{2}$ arêtes dans le cas d'un graphe non orienté.

Un graphe complet est souvent noté K_n où n est le nombre de sommets.

Par exemple, dans un graphe complet à 4 sommets K_4 , chaque sommet est relié aux 3 autres, ce qui donne un total de 6 arêtes.

Les graphes complets sont importants en théorie des graphes car ils représentent le cas extrême où le nombre d'arêtes est maximal.

FIGURE 2.8 – Exemple d'un graphe complet à 7 sommets K_7

Terminologie et propriétés des graphes

Pour mieux comprendre un graphe, voici quelques mots à connaître :

- **Sommet (ou nœud)** : Un point dans le graphe. Il peut représenter une ville, une personne, un objet, etc.
- **Arête (ou arc)** : C'est un lien entre deux sommets.
 - Dans un graphe non orienté, on parle d'une **arête**, représentée comme une simple ligne entre deux sommets.
 - Dans un graphe orienté, on parle d'un **arc**, représenté par une flèche indiquant une direction.
- **Degré d'un sommet** : C'est le nombre de lignes (arêtes) qui touchent un sommet.
 - Si le graphe n'est **pas orienté** :

$$\deg(v) = |\{ e \in E \mid v \in e \}|$$

Cela veut dire : combien de lignes touchent le sommet v .

- Si le graphe est **orienté** :

$$\deg^-(v) = |\{ (u, v) \in E \mid u \in V \}|$$

(Nombre d'arcs qui vont vers v — on dit *degré entrant*)

$$\deg^+(v) = |\{ (v, w) \in E \mid w \in V \}|$$

(Nombre d'arcs qui partent de v — on dit *degré sortant*)

- **Chemin** : Une suite de sommets reliés entre eux. On peut passer d'un sommet à un autre.
- **Cycle** : Un chemin qui commence et finit au même sommet sans repasser deux fois par la même arête.
- **Connexité** :
 - Un graphe est **connexe** s'il existe un chemin entre chaque paire de sommets.
 - Sinon, il est **non connexe**.

2.2.3 Représentation des graphes

Pour travailler avec un graphe en calcul réel, on a besoin d'une représentation, surtout en informatique où l'on doit représenter les graphes dans une structure de données. Parmi les méthodes utilisées pour représenter un graphe, on trouve la liste d'adjacence, la matrice d'adjacence et matrice d'incidence.

Liste d'adjacence

La liste d'adjacence est une structure de données qui associe à chaque sommet une liste contenant tous ses voisins (les sommets directement connectés par une arête). Cette représentation est particulièrement adaptée aux graphes peu denses, c'est-à-dire ceux ayant un nombre d'arêtes relativement faible par rapport au nombre de sommets.

Principe de la liste d'adjacence :

Pour chaque sommet $v \in V$, on associe une liste $L(v)$ contenant tous les sommets $u \in V$ tels que $(v, u) \in E$, où E est l'ensemble des arêtes du graphe. Autrement dit, $L(v) = \{u \mid (v, u) \in E\}$.

Avantages :

- **Mémoire optimisée** : la liste d'adjacence utilise un espace mémoire proportionnel à $O(n + m)$, où n est le nombre de sommets et m le nombre d'arêtes. Cela la rend plus efficace que la matrice d'adjacence pour les graphes peu denses.
- **Parcours rapide des voisins** : il est facile et rapide d'énumérer tous les voisins d'un sommet, ce qui facilite certains algorithmes comme le parcours en profondeur (DFS) ou en largeur (BFS).

Inconvénients :

- **Vérification plus lente de l'existence d'une arête** : pour savoir si une arête existe entre deux sommets donnés, il faut parcourir la liste des voisins du premier sommet, ce qui peut être lent si ce sommet a beaucoup de voisins.

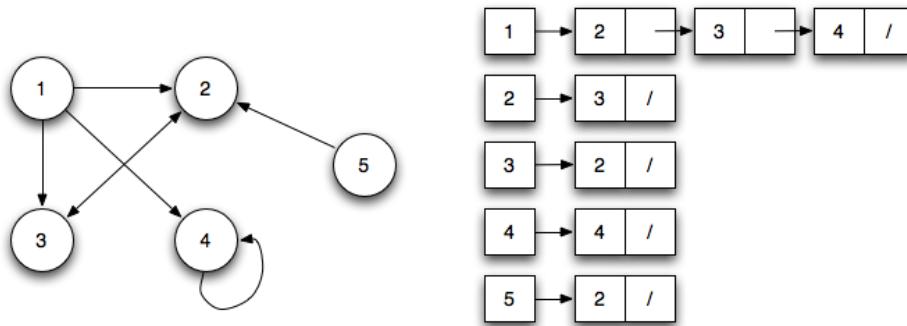


FIGURE 2.9 – Représentation d'un graphe avec une liste d'adjacence

Comme vous pouvez le voir sur cette figure, on a un graphe orienté. Pour le représenter, on a utilisé une liste d'adjacence. Elle contient les sommets 1, 2, 3, 4, 5, et chaque sommet est associé à une liste.

Matrice d'adjacence

La matrice d'adjacence est un tableau carré de taille $n \times n$, où n est le nombre de sommets du graphe. Chaque case (i, j) indique s'il y a une arête qui va du sommet i au sommet j , ou le poids de cette arête.

Principe de la matrice d'adjacence :**Cas non orienté :**

- La matrice est symétrique, c'est-à-dire que $M[i][j] = M[j][i]$.
- Si le sommet i est relié au sommet j , alors $M[i][j] = 1$ (ou le poids de l'arête).
- Sinon, $M[i][j] = 0$.

Cas orienté :

- Si l'arête va du sommet i au sommet j , alors $M[i][j] = 1$ (ou le poids de l'arête).
- Sinon, $M[i][j] = 0$.

Avantages :

- **Test d'existence d'une arête en temps constant** : vérifier si une arête existe entre deux sommets se fait en temps $O(1)$ en accédant directement à la cellule correspondante.
- **Adaptée aux graphes denses** : cette représentation est efficace lorsque le nombre d'arêtes est proche du nombre maximal possible, c'est-à-dire quand le graphe est dense.

Inconvénients :

- **Consommation mémoire élevée** : la matrice nécessite un espace mémoire en $O(n^2)$, ce qui peut devenir très coûteux pour les grands graphes peu denses.

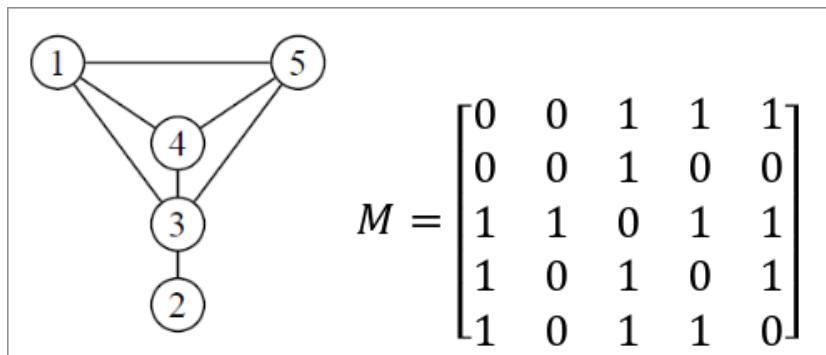


FIGURE 2.10 – Représentation d'un graphe avec une matrice d'adjacence

Comme vous pouvez le voir sur cette figure, on a un graphe non orienté. Pour le représenter, on utilise une matrice d'adjacence de taille 5×5 , car il y a 5 sommets. Chaque lien entre deux sommets est représenté par un 1 (s'il existe) ou un 0 (s'il n'existe pas). Par exemple, le sommet 1 est relié au sommet 5, alors la case $a_{5,1}$ vaut 1.

Matrice d'incidence

La matrice d'incidence est une manière de représenter un graphe avec un tableau de taille $n \times m$, où n est le nombre de sommets et m le nombre d'arêtes. Chaque ligne représente un sommet et chaque colonne une arête. La valeur dans chaque case indique si le sommet est lié à l'arête.

Principe de la matrice d'incidence :**Cas non orienté :**

- Si le sommet i est relié à l'arête j , alors $M[i][j] = 1$
- Sinon, $M[i][j] = 0$

Cas orienté :

- Si le sommet i est l'origine de l'arête j , alors $M[i][j] = -1$
- Si le sommet i est l'extrémité de l'arête j , alors $M[i][j] = 1$
- Sinon, $M[i][j] = 0$

Cette représentation est utile pour certains calculs en théorie des graphes ou algèbre.

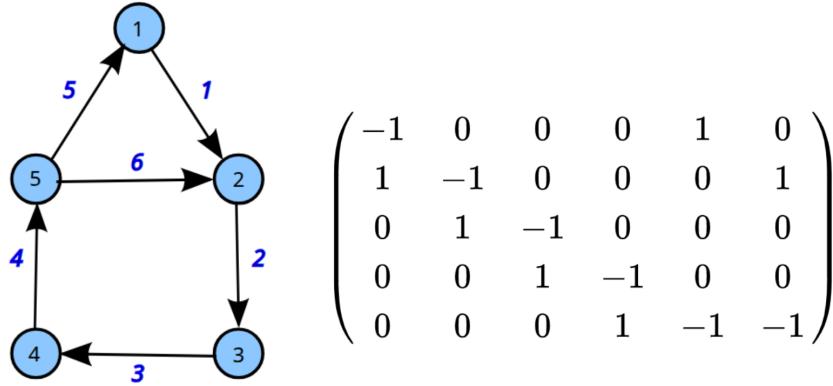


FIGURE 2.11 – Représentation d'un graphe avec une matrice d'incidence

Comme vous pouvez le voir, ici on a un graphe orienté et pondéré. On utilise une matrice d'incidence où chaque case peut prendre les valeurs 0, 1, ou -1 . Par exemple, pour l'arête $(1, 2)$ qui va du sommet 1 vers le sommet 2, la case $a_{1,2}$ vaut 1, et la case $a_{2,1}$ vaut -1 .

2.3 Le principe de l'algorithme A*

2.3.1 Introduction à l'algorithme A*

- Qu'est-ce que l'algorithme A* :

L'algorithme A* est une méthode de recherche utilisée pour trouver le chemin le plus court entre un point de départ et un point d'arrivée dans un graphe. Il combine les approches de recherche en largeur et en profondeur en utilisant une fonction d'évaluation pour guider la recherche de manière optimale.

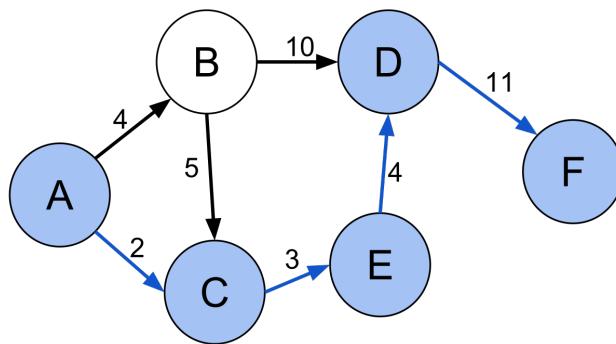


FIGURE 2.12 – Le chemin le plus court dans ce graphe

2.3.2 Éléments fondamentaux

- Principaux éléments :

Pour comprendre l'algorithme A*, il est important de connaître certains concepts fondamentaux utilisés dans cet algorithme. Parmi eux, on trouve :

- **Coût du chemin ($g(n)$)** : C'est le coût réel du déplacement d'un noeud à un autre, autrement dit, la somme des poids des arêtes entre les noeuds parcourus.
- **Heuristique ($h(n)$)** : C'est une estimation du coût entre un noeud donné et le noeud objectif. Attention, ce n'est qu'une estimation, et non un coût réel.

2.3.3 Concepts clés

- **Concepts clés de la recherche A* :**

L'algorithme A* utilise trois fonctions pour choisir le meilleur chemin :

- $g(n)$: C'est la distance réelle déjà parcourue depuis le point de départ jusqu'au noeud n .

Formule :

$$g(n_k) = \sum_{i=0}^{k-1} w(n_i, n_{i+1})$$

où $w(n_i, n_{i+1})$ est le poids (ou coût) entre les noeuds n_i et n_{i+1} .

- $h(n)$: C'est une estimation du coût qu'il reste entre le noeud n et l'objectif. Ce n'est pas une vraie valeur, juste une supposition.

Une bonne estimation ne doit jamais être plus grande que la réalité :

$$h(n) \leq h^*(n)$$

Dans une grille, il existe plusieurs méthodes pour calculer l'heuristique, parmi lesquelles :

- **Distance de Manhattan :**

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

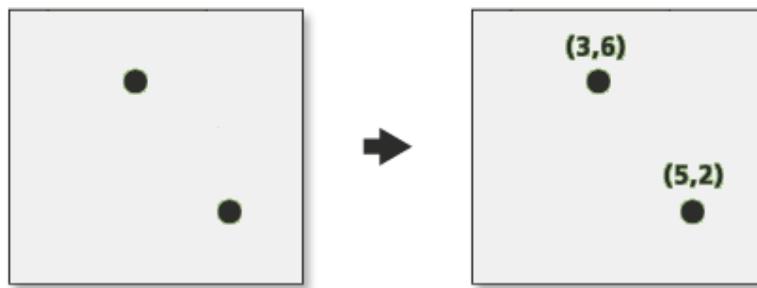
- **Distance euclidienne :**

$$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Note : chaque noeud dans la grille a des coordonnées (x, y) .

Dans les deux méthodes, (x_1, y_1) sont les coordonnées du noeud actuel, et (x_2, y_2) celles de l'objectif à atteindre.

- $f(n) = g(n) + h(n)$: C'est le coût total estimé pour aller du départ à l'objectif en passant par le noeud n .

FIGURE 2.13 – Chaque sommet possède des coordonnées x et y .

Dans ce cas, on peut utiliser les coordonnées (3, 6) pour calculer l'heuristique de ce sommet.

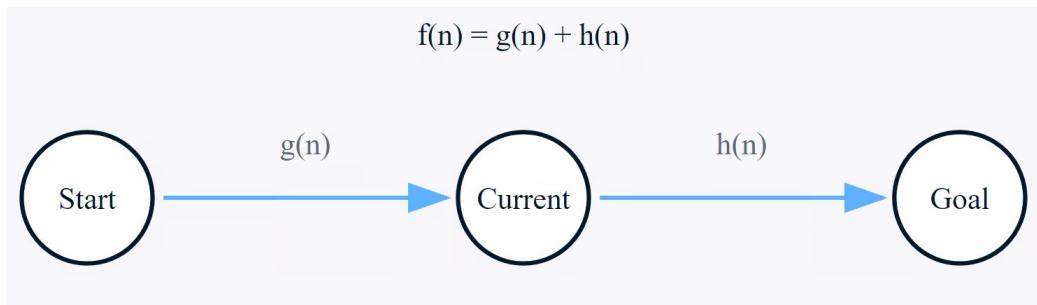


FIGURE 2.14 – Algorithme A* Fonction de coût

La fonction de coût total est toujours définie par :

$$f(n) = g(n) + h(n)$$

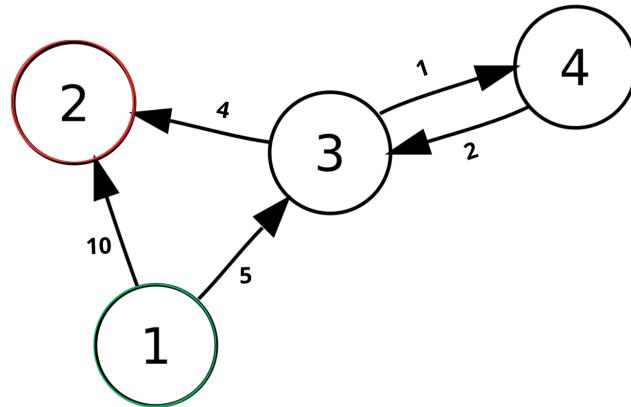
où $g(n)$ est le coût depuis le nœud de départ jusqu'à n , et $h(n)$ est l'heuristique estimée de n jusqu'au but.

2.3.4 Étapes de l'algorithme

- Étapes générales de l'algorithme A* :
 1. **Initialisation** : Choisir un sommet de départ.
 2. **Expansion** : Examiner tous les voisins du sommet sélectionné.
 3. **Calcul** : Calculer $f(n) = g(n) + h(n)$ pour chaque voisin.
 4. **Sélection** : Choisir le nœud avec le plus petit $f(n)$.
 5. **Répétition** : Répéter les étapes jusqu'à atteindre le sommet cible.

2.3.5 Exemple pratique

- **Simulation d'un exemple de graphe :** Voici le graphe dont on veut déterminer le chemin le plus court :



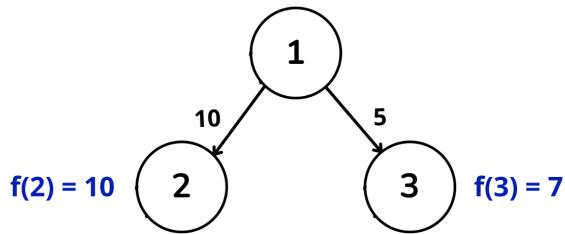
Le sommet de départ est 1 et le sommet d'arrivée est 2.

On suppose que la fonction heuristique pour chaque sommet est la suivante :

- $h(1) = 3$ (estimation du coût de 1 vers 2)
- $h(2) = 0$ (on est arrivé)
- $h(3) = 2$ (estimation du coût de 3 vers 2)
- $h(4) = 5$ (estimation du coût de 4 vers 2)

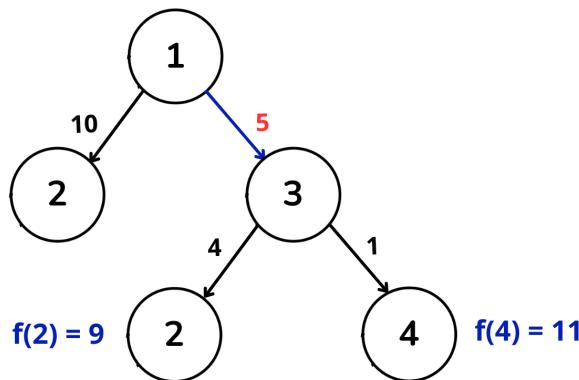
- **Étape 1 :** Nous sommes au sommet 1. On regarde ses voisins et on trouve qu'il existe les sommets 2 et 3.
- On calcule la fonction $f(n)$ pour chaque sommet voisin :

$$\begin{aligned}f(3) &= g(3) + h(3) = 5 + 2 = 7 \\f(2) &= g(2) + h(2) = 10 + 0 = 10\end{aligned}$$

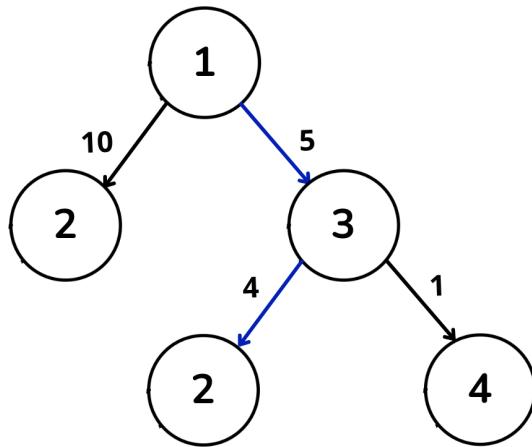


- **Étape 2 :** Le sommet ayant la plus petite valeur est le 3, donc on choisit le sommet 3 comme prochain sommet, ce qui fait partie du chemin le plus court.
- **Étape 3 :** De la même manière, on regarde les voisins du sommet 3, qui sont les sommets 2 et 4. On calcule la fonction $f(n)$ pour chaque nœud voisin :

$$\begin{aligned}f(4) &= g(4) + h(4) = 5 + 1 + 5 = 11 \\f(2) &= g(2) + h(2) = 5 + 4 + 0 = 9\end{aligned}$$



- **Étape 4 :** Le sommet ayant la plus petite valeur est le 2, donc on choisit le sommet 2 comme destination finale.



Donc, le chemin le plus court allant de 1 vers 2 est : $1 \rightarrow 3 \rightarrow 2$ avec un coût de 9.

2.4 Comparer A* avec d'autres algorithmes

L'algorithme A* sert à trouver le chemin le plus court dans un graphe. Il y a aussi d'autres façons de chercher un chemin, comme :

2.4.1 Algorithme de Dijkstra

- Il regarde tous les points autour du départ, puis ceux plus loin, en choisissant toujours le chemin le moins cher.
- Il trouve toujours le meilleur chemin.
- Mais il peut être lent parce qu'il visite beaucoup de points inutiles.
- Contrairement à A*, qui utilise une aide (heuristique) pour aller plus vite vers la cible.

2.4.2 Recherche en largeur (BFS)

- Il explore les voisins d'un point, puis leurs voisins, niveau par niveau.
- Il trouve le chemin avec le moins de pas, si toutes les routes ont le même coût.

- Mais il ne regarde pas le coût des routes, donc le chemin n'est pas toujours le moins cher.
- A* regarde à la fois le chemin déjà fait et ce qui reste à faire.

2.4.3 Recherche en profondeur (DFS)

- Il suit un chemin jusqu'au bout avant d'essayer un autre chemin.
- Il peut être rapide si la solution est proche.
- Mais parfois, il perd du temps sur de longs chemins inutiles.
- A* est plus intelligent car il choisit mieux où aller.

2.4.4 Tableau résumé

Algorithme	Trouve toujours le meilleur	Cherche partout	Rapide
Dijkstra	Oui	Oui	Moyen
BFS	Oui (sans poids)	Oui	Lent
DFS	Non	Non	Variable
A*	Oui (avec heuristique)	Oui	Rapide

2.4.5 Conclusion

En conclusion, on peut dire que A* est un mélange de Dijkstra et d'heuristiques. Il est rapide et trouve toujours le bon chemin, c'est pourquoi il est largement utilisé dans différents domaines, comme l'intelligence artificielle.

Chapitre 3

Implémentation de l'algorithme A*

Dans ce chapitre, j'ai détaillé l'implémentation pratique de l'algorithme A* en Python. J'ai expliqué mon choix technique pour l'implémentation de A* algorithm.

3.1 Développement en Python

3.1.1 Le langage Python

J'ai choisi Python pour implémenter l'algorithme A* parce que c'est le langage le plus adapté pour ce projet. Voici pourquoi :

- **Facile à comprendre** : La syntaxe de Python est simple. Par exemple, la représentation d'un graphe y est facile.
- **Idéal pour les débutants** : Je peux me concentrer sur l'algorithme plutôt que sur des détails techniques compliqués
- **Parfait pour les graphes** : En Python, on a plusieurs structures simples pour travailler avec les graphes, comme les dictionnaires, les listes, etc.
- **Rapide à tester** : Je peux faire des essais et voir les résultats tout de suite sans attendre
- **Beaucoup d'outils disponibles** : Si je veux améliorer le programme plus tard, il existe des bibliothèques utiles comme NetworkX pour les graphes, qui permettent de visualiser les graphes.

Avec Python, j'ai pu écrire l'algorithme A* en quelques lignes seulement. Le code reste facile à relire et à modifier si besoin. Par exemple, ajouter un nouveau noeud au graphe prend juste une ligne :

C'est pour toutes ces raisons que Python est l'un des meilleurs choix pour ce projet.

3.1.2 Choix des structures de données

J'ai utilisé principalement plusieurs structures de données, parmi lesquelles :

- **Dictionnaire** : Pour représenter le graphe sous forme d'une liste d'adjacence et la table heuristique.
- **Liste** : pour gérer les chemins à explorer et les nœuds visités
- **Tuple** : pour stocker les paires (sommet, coût)

3.2 Explication de l'implémentation

3.2.1 Représentation du graphe

Au niveau de Python, j'ai utilisé un dictionnaire pour la représentation du graphe. Voici le contenu de ce dictionnaire :

- Les clés du dictionnaire sont les nœuds ou les sommets.
- La valeur de chaque clé est une liste de voisins.
- Chaque liste contient des tuples qui représentent le sommet et leur coût.

Voici un exemple d'un graphe, que j'ai présenté sous forme d'une liste d'adjacence.

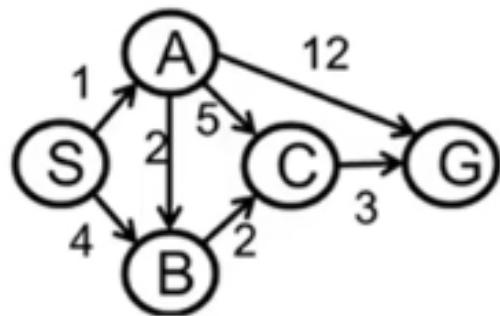


FIGURE 3.1 – Exemple d'un graphe

L'objectif est de déterminer le chemin le plus court dans ce graphe.

Voici le code :



```

1 graph = {
2     'S': [('A', 1), ('B', 4)],
3     'A': [('B', 2), ('C', 5), ('G', 12)],
4     'B': [('C', 2)],
5     'C': [('G', 3)]
6 }
```

FIGURE 3.2 – Représentation du graphe

Comme vous le voyez ici, dans le dictionnaire, chaque sommet est représenté par une clé et possède une liste (qui contient les voisins du sommet), et chaque liste contient des tuples.

3.2.2 Table heuristique

La table heuristique permet de stocker les valeurs heuristiques (estimations optimistes) pour chaque sommet :

Voici le code :



```

1 # heuristic table
2 H_table = {
3     'S': 7,
4     'A': 6,
5     'B': 4,
6     'C': 2,
7     'G': 0,
8 }
9
```

FIGURE 3.3 – Déclaration de la table heuristique

Comme vous pouvez le voir ici, ce dictionnaire permet de représenter la table heuristique, où chaque sommet possède une valeur estimée.

3.2.3 Fonction de calcul du coût

Cette fonction permet de calculer le coût total $f(n) = g(n) + h(n)$ d'un chemin, par exemple de S vers C.

Voici le code de la fonction :



```

1  def path_f_cost(path: list[tuple]) -> tuple:
2      g_cost = 0
3      for(node, cost) in path:
4          g_cost += cost
5      last_node = path[-1][0]
6      h_cost = H_table[last_node]
7      f_cost = g_cost + h_cost
8      return (f_cost, last_node)
9
10 path = [('S', 0), ('A', 1), ('C', 5)]
11 path_f_cost(path)

```

FIGURE 3.4 – Fonction qui calcule le coût total

Cette fonction prend en entrée :

- Un chemin (path), qui est une liste de tuples
- Exemple : `[('S', 0), ('A', 1), ('C', 5)]`
- Chaque tuple représente : (nœud, coût_pour_y_arriver)

Elle calcule trois choses :

- `g_cost` : le coût réel depuis le départ
- `h_cost` : l'estimation heuristique vers l'arrivée
- `f_cost` : la somme des deux (coût total)

Elle retourne :

- Un tuple avec (`f_cost`, `dernier_nœud`)

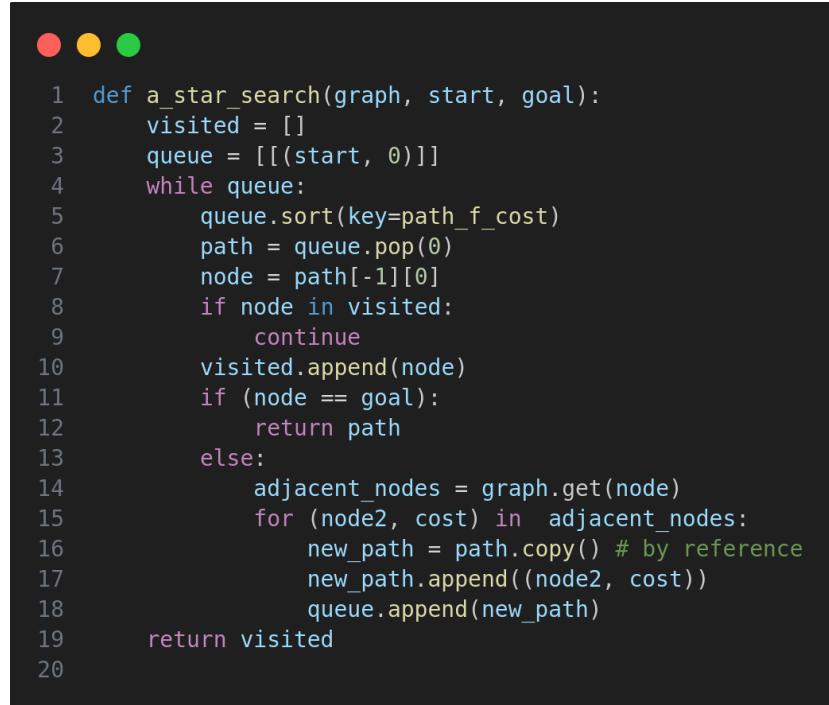
3.2.4 Fonction principale

Cette fonction permet de déterminer le chemin le plus court dans un graphe. Voici les étapes :

1. Utiliser une file de priorité qui contient les chemins (triée par priorité).
2. Extraire le chemin avec le plus petit coût f .

3. À partir du chemin extrait, extraire également le dernier sommet, car c'est le sommet sélectionné.
4. Attention : si un sommet est déjà visité, ne pas l'explorer à nouveau (c'est-à-dire ne pas pouvoir le sélectionner une seconde fois).

Voici le code de la fonction :



```

● ● ●
1 def a_star_search(graph, start, goal):
2     visited = []
3     queue = [[(start, 0)]]
4     while queue:
5         queue.sort(key=path_f_cost)
6         path = queue.pop(0)
7         node = path[-1][0]
8         if node in visited:
9             continue
10        visited.append(node)
11        if (node == goal):
12            return path
13        else:
14            adjacent_nodes = graph.get(node)
15            for (node2, cost) in adjacent_nodes:
16                new_path = path.copy() # by reference
17                new_path.append((node2, cost))
18                queue.append(new_path)
19    return visited
20

```

FIGURE 3.5 – Fonction principale de l'algorithme A*

Cette fonction prend en entrée :

- Un graphe (**graph**), représenté par un dictionnaire
- Un sommet de départ (**start**)
- Un sommet objectif (**goal**)

Elle réalise les étapes suivantes :

- Initialise une liste **visited** pour garder les sommets déjà explorés
- Initialise une file **queue** contenant un chemin commençant par le sommet de départ
- Tant que la file **queue** n'est pas vide :
 - Trie la file selon la fonction **path_f_cost** (coût total estimé)
 - Retire et récupère le chemin avec le plus petit coût
 - Récupère le dernier sommet du chemin
 - Si ce sommet a déjà été visité, passe au suivant

- Sinon, ajoute ce sommet à `visited`
- Si ce sommet est l'objectif, retourne le chemin
- Sinon, récupère ses voisins et ajoute de nouveaux chemins dans la file

Elle retourne :

- Le chemin le plus court trouvé entre `start` et `goal`, ou la liste des sommets visités si aucun chemin n'est trouvé

3.2.5 Exemple d'utilisation

On peut utiliser cet algorithme pour trouver le chemin le plus court dans le graphe déjà présenté.

Pour cela, on appelle la fonction principale en lui passant le graphe, le sommet de départ et le sommet d'arrivée.

Voici le code de l'appel de la fonction :

```

● ● ●
1 solution = a_star_search(graph=graph, start='S', goal='G')
2 print(solution)
3 print(f"Cost is ", path_f_cost(solution)[0])
4

```

FIGURE 3.6 – Appel de la fonction principale

Voici l'exécution de l'algorithme :

```

abdelah@abdelah-ThinkPad-L460:~/Documents/git_projects/a_star
● _algorithm/app$ python3 a_star_algo.py
[('S', 0), ('A', 1), ('B', 2), ('C', 2), ('G', 3)]
Cost is 8

```

FIGURE 3.7 – Exécution de l'algorithme

Comme vous pouvez le voir, le chemin le plus court est : S → A → B → C → G, avec un coût total de 8.

3.3 Performance de l'algorithme A*

La performance de l'algorithme dépend de plusieurs facteurs. Quand on parle de performance, on parle surtout de la rapidité à trouver le chemin le plus court, tout en gardant un résultat optimal.

Parmi les facteurs importants, on trouve :

- **La qualité de l'heuristique** : Plus l'heuristique est proche du vrai coût, plus l'algorithme est rapide.
- **La taille du graphe** : Si le graphe a beaucoup de sommets et d'arcs, le temps de calcul peut augmenter.
- **La structure du graphe** : Certains graphes (comme ceux avec beaucoup de chemins possibles) rendent la recherche plus longue.
- **Les coûts des arcs** : Si les coûts sont très différents, cela peut influencer l'ordre d'exploration des chemins.
- **Les ressources disponibles** : Plus il y a de mémoire et de puissance, plus l'algorithme peut être rapide.

Chapitre 4

Analyse et conception de l'application

4.1 Langage de modélisation

4.1.1 Description d'UML

UML, acronyme de Unified Modeling Language (Langage de Modélisation Unifié en français), est un langage de modélisation graphique utilisé dans le domaine du développement logiciel et de la conception orientée objet. Il permet de représenter visuellement les différents aspects d'un système, tels que sa structure, son comportement et les interactions entre ses composants.

En utilisant des diagrammes et des notations graphiques, UML facilite la communication et la compréhension mutuelle entre les développeurs, les concepteurs et les parties prenantes d'un projet. Il permet de documenter et de modéliser les systèmes d'information de manière précise et complète.

Grâce à son caractère universel et à son indépendance vis-à-vis des langages de programmation, UML peut être utilisé dans différents contextes et méthodologies de développement. Il est largement adopté dans l'industrie du logiciel pour améliorer la qualité et la gestion des projets.



FIGURE 4.1 – Unified Modeling Language

4.1.2 Pourquoi UML

Pour mon projet, qui est une application permettant de visualiser des graphes et de déterminer le chemin le plus court dans un graphe, j'ai choisi d'utiliser la modélisation pour mieux comprendre les différentes fonctionnalités de l'application et les organiser de manière efficace.

Grâce à la modélisation, je peux représenter clairement les différents aspects comme les graphes, les arcs, la grille, etc. Cela m'aide à réduire la complexité en éliminant les détails qui ne sont pas essentiels pour le fonctionnement global de l'application.

J'utilise aussi les modèles pour définir les limites de mon application et mettre en avant ce qui est vraiment important à comprendre ou à prévoir.

Je représente mes modèles sous forme de schémas ou de diagrammes, en combinant des éléments graphiques et du texte pour bien expliquer la signification de chaque symbole utilisé.

Pour cela, j'utilise différents formalismes comme les diagrammes de cas d'utilisation, de classes, et de séquence.

4.2 Diagrammes

4.2.1 Présentation

En UML, il existe deux grandes façons de représenter un système :

- La vue **statique**, qui montre la structure du système avec les *diagrammes de structure* (comme les diagrammes de classes).
 - La vue **dynamique**, qui montre le comportement du système avec les *diagrammes de comportement* (comme les diagrammes d'activités) et les *diagrammes d'interaction* (comme les diagrammes de séquence).

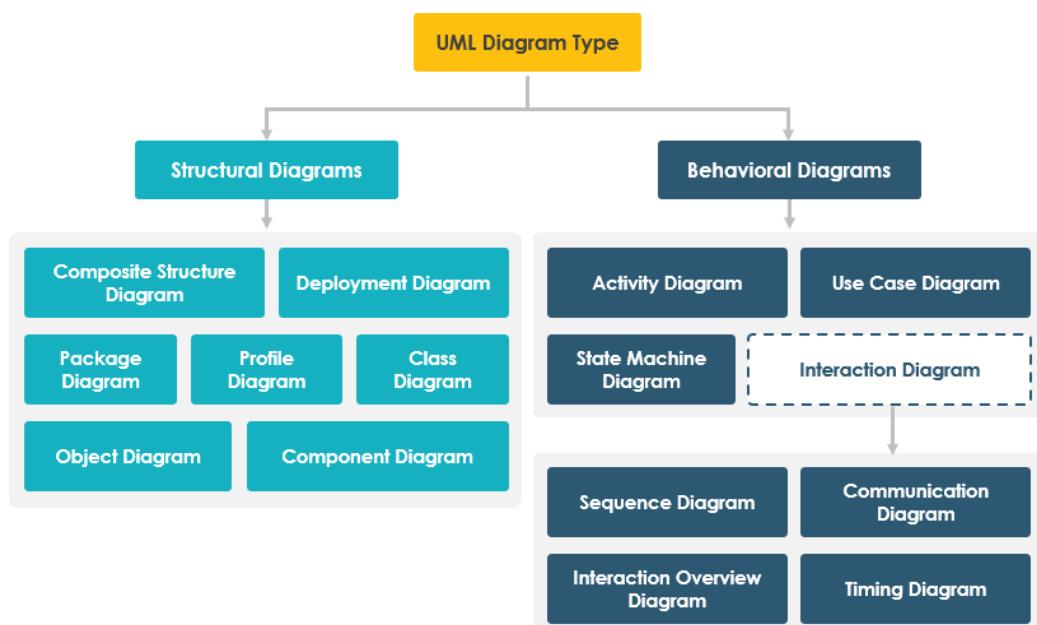


FIGURE 4.2 – les catégories des diagrammes uml

4.2.2 Analyse des besoins fonctionnels

On peut analyser les besoins fonctionnels à partir du diagramme de cas d'utilisation.

Définition du diagramme de cas d'utilisation

Le diagramme de cas d'utilisation sert à représenter ce que l'utilisateur peut faire avec l'application. Il montre les **fonctions principales** du système et comment **l'utilisateur interagit** avec ces fonctions. Ce diagramme ne montre pas *comment* les actions sont faites à l'intérieur, mais seulement *ce que* l'on peut faire.

Diagramme de cas d'utilisation

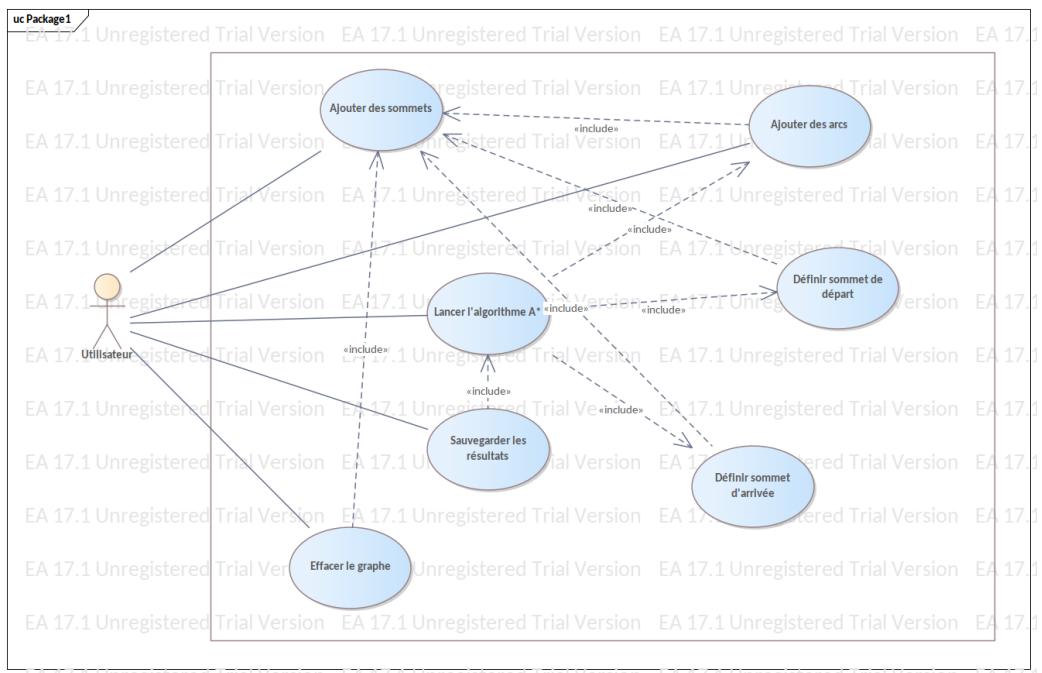


FIGURE 4.3 – Diagramme de cas d'utilisation

Description du diagramme :

Dans ce projet, l'utilisateur peut réaliser plusieurs actions sur un graphe à travers une interface :

- **Ajouter des sommets** : créer les points (ou sommets) du graphe.
- **Ajouter des arcs** : relier les sommets avec des liens dirigés (flèches), mais il faut d'abord ajouter des sommets.
- **Définir le sommet de départ** : choisir le sommet de départ du graphe. Le graphe doit déjà exister, c'est-à-dire contenir des sommets et des arcs.
- **Définir le sommet d'arrivée** : choisir le sommet d'arrivée du graphe. Le graphe doit déjà exister, c'est-à-dire contenir des sommets et des arcs.
- **Lancer l'algorithme A*** : démarrer la recherche du plus court chemin. Avant de faire cette action, l'utilisateur doit :
 - l'ajout des sommets,
 - l'ajout des arcs,
 - la définition du sommet de départ,
 - la définition du sommet d'arrivée.
- **Sauvegarder les résultats** : enregistrer les résultats trouvés par l'algorithme, tels que la table heuristique, le chemin le plus court, et le graphe sous forme d'une liste d'adjacence. Cette action se fait après le lancement de l'algorithme. Elle inclut le cas d'usage « lancer l'algorithme A* ».
- **Effacer le graphe** : effacer le graphe, c'est-à-dire supprimer tous les sommets, arcs et toutes les choses qui existent dans la grille (la place où l'utilisateur dessine).

4.2.3 Modélisation statique : Diagramme de classe

Définition du diagramme de classe

Le diagramme de classe permet de représenter la structure de notre application d'algorithme A*. Il montre les différentes classes (objets) du programme, leurs propriétés (attributs) et leurs actions (méthodes), ainsi que les relations entre ces classes.

Diagramme de classe

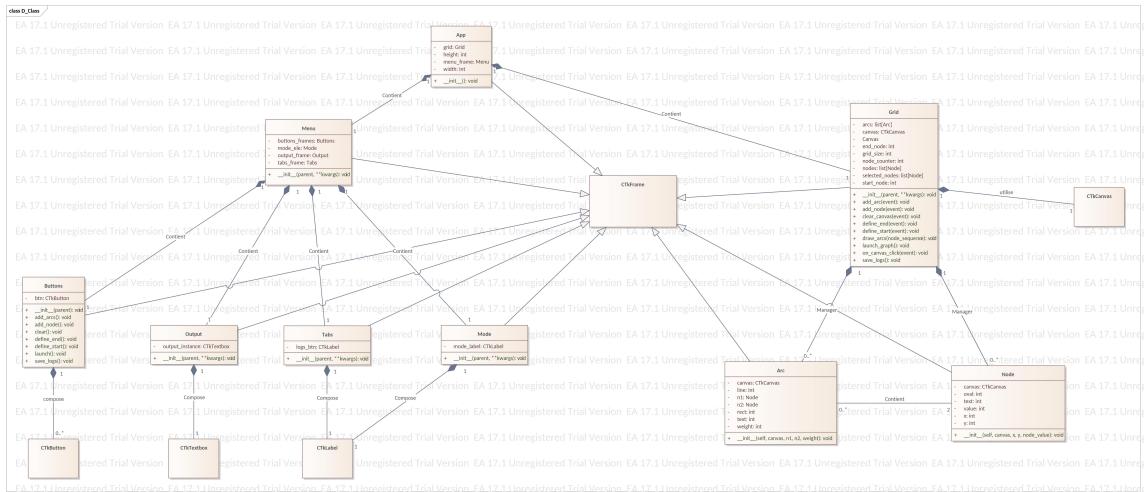


FIGURE 4.4 – Diagramme de classe

Description des classes principales de l'application

- **App** (héritée de CTk) Classe principale de l'application. Elle initialise l'interface globale, configure la taille de la fenêtre, et regroupe deux composants essentiels :
- **Menu** : barre de contrôle (boutons, onglets, etc.)
- **Grid** : zone de dessin interactive pour le graphe
- Méthodes :
 - `__init__()` : initialise tous les composants
 - `mainloop()` : lance l'application
- **Menu** (hérite de CTkFrame) : Conteneur pour les éléments d'interaction utilisateur.
- **Mode** : sélection du mode d'interaction
- **Buttons** : boutons de contrôle (ajout, lancement, etc.)
- **Tabs** : gestion des onglets (comme les logs)
- **Output** : zone d'affichage des messages
- **Buttons** (hérite de CTkFrame) : Regroupe tous les boutons de commande de l'utilisateur. Méthodes :
 - `add_node()`, `add_arcs()` : ajouter des éléments au graphe
 - `define_start()`, `define_end()` : définir le chemin A*
 - `launch()` : exécuter l'algorithme A*
 - `clear()`, `save_logs()` : effacer ou sauvegarder

- **Output** (hérite de CTkFrame) : Gère l'affichage des résultats (Par exemple, afficher le chemin le plus court...).
- **Mode** (hérite de CTkFrame) : Permet d'afficher le mode actuel de l'utilisateur (ex. : ajout de noeud, suppression, lancement...).
- **Tabs** (hérite de CTkFrame) : Juste un label pour donner un aperçu des logs.
- **Grid** (hérite de CTkFrame) : C'est la zone graphique qui permet à l'utilisateur de créer des sommets et des arcs etc.
 - Gère les événements souris (clic, dessin, sélection)
 - Stocke les sommets, arcs, sommet de départ/arrivée
 - Permet le lancement de l'algorithme A*
 - Dessine les arcs et les sommets

Méthodes : `add_node()`, `add_arc()`, `define_start()`, `launch_graph()`, etc.
- **Node** : Représente un sommet dans le graphe avec :
 - Coordonnées x, y
 - Valeur (identifiant unique)
 - Eléments graphiques ovale(circle) et texte
- **Arc** : Lien entre deux sommets, contient :
 - n1, n2 : les sommets reliés
 - weight : le poids de l'arc
 - Eléments graphiques : ligne, rectangle, texte

Relations entre les classes

- **App** contient → **Menu** et **Grid** (*App est la classe principale qui inclut les composants Menu et Grid pour gérer l'interface et la logique.*)
- **Menu** compose → **Mode**, **Buttons**, **Tabs**, **Output** (*Menu regroupe les sous-composants permettant la gestion des modes, des actions, des onglets et de l'affichage des résultats.*)
- **Grid** gère → plusieurs **Node** et **Arc** (*Grid est responsable de la gestion, du dessin et de la manipulation des nœuds et arcs du graphe.*)
- **Arc** relie → deux instances de **Node** (n1 et n2) (*Arc représente une connexion entre deux nœuds dans le graphe, avec un poids associé.*)
- Toutes les classes visuelles (**Menu**, **Buttons**, **Output**, **Mode**, **Tabs**, **Grid**) héritent de **CTkFrame** (*car tous les éléments de l'application sont sous forme de frame.*)
- **App** hérite → de **CTk** (*Classe principale de l'application héritant de la fenêtre principale.*)
- **Buttons** utilise → **CTkButton** pour créer un bouton.

- **Output** utilise → CTkTextbox pour créer une zone de texte (comme un champ de saisie).
- **Tabs** et **Mode** utilisent → CTkLabel pour créer un label.

4.2.4 Modélisation dynamique : Diagramme de séquence

Définition du diagramme de séquence

Un diagramme de séquence est un diagramme UML qui représente la séquence de messages entre les objets au cours d'une interaction. Un diagramme de séquence comprend un groupe d'objets, représentés par des lignes de vie, et les messages que ces objets échangent lors de l'interaction. Les diagrammes de séquence représentent la séquence de messages transmis entre des objets. Ils peuvent également représenter les structures de contrôle entre des objets.

Par exemple, dans notre application, l'utilisateur commence par sélectionner le mode d'ajout de sommets. Ensuite, il peut ajouter des sommets d'un graphe. Dans l'application, l'utilisateur est un objet, et la grille (placement qui contient les sommets) est aussi un objet qui fait partie du diagramme de séquence.

Diagramme de séquence

— Phase 1 : Construction du graphe

Acteur principal : Utilisateur

Objectif : Créer et placer des sommets sur la grille pour construire la structure de base du graphe

Scénario :

- L'utilisateur clique sur le bouton "Add Nodes" pour activer le mode d'ajout de nœuds
- L'interface active le mode d'ajout et attend les interactions de l'utilisateur
- L'utilisateur clique sur différentes positions du canvas pour placer les sommets
- Pour chaque clic, le système génère un événement add_node qui déclenche la création d'un nouveau sommet
- Chaque sommet créé est automatiquement ajouté à la structure de données du graphe
- Les sommets sont affichés visuellement sur le canvas aux positions sélectionnées

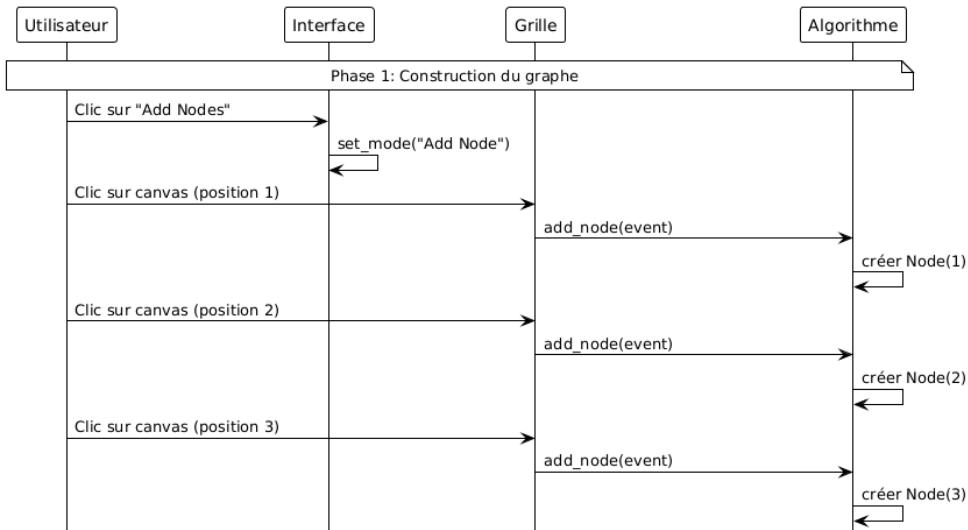


FIGURE 4.5 – Diagramme de séquence – Phase de construction du graphe

— Phase 2 : Ajout des arcs

Acteur principal : Utilisateur

Objectif : Connecter les sommets existants avec des arcs pondérés pour former le graphe complet

Scénario :

- L'utilisateur clique sur "Add Arcs" pour activer le mode de création d'arêtes
- L'interface passe en mode sélection d'arcs et attend les sélections de sommets
- L'utilisateur sélectionne le premier sommet puis le second sommet à connecter
- Le système affiche une boîte de dialogue "Entrer poids" pour saisir le poids de l'arc
- L'utilisateur saisit la valeur du poids et valide
- Un arc pondéré est créé entre les deux nœuds sélectionnés
- L'arc est ajouté à la structure du graphe et affiché visuellement

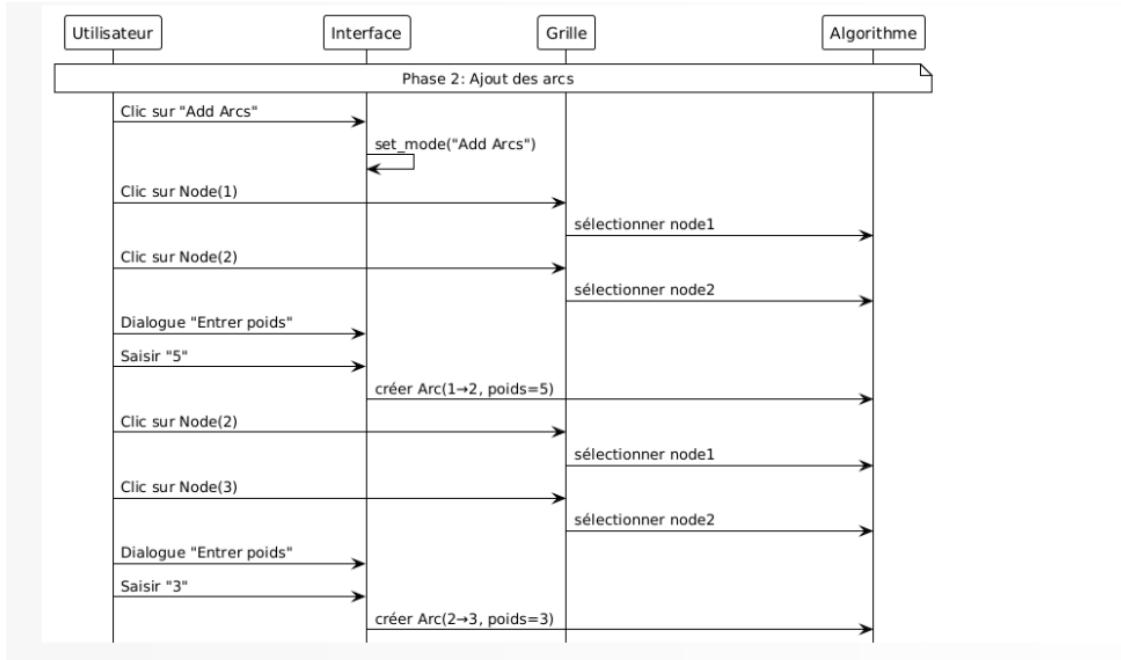


FIGURE 4.6 – Diagramme de séquence – Phase d'ajout des arcs

— Phase 3 : Définition des sommets de début et de fin

Acteur principal : Utilisateur

Objectif : Définir les sommets source et destination pour l'algorithme
Scénario :

- L'utilisateur clique sur "Define Start" pour définir le sommet de départ
- L'interface active le mode de définition du point de départ
- L'utilisateur sélectionne un sommet qui devient le sommet de départ
- Le système colore le sommet sélectionné en vert pour indiquer le point de départ
- L'utilisateur clique sur "Define End" pour définir le sommet d'arrivée
- L'utilisateur sélectionne un sommet qui devient le sommet de destination
- Le système colore le sommet sélectionné en rouge pour indiquer le point d'arrivée

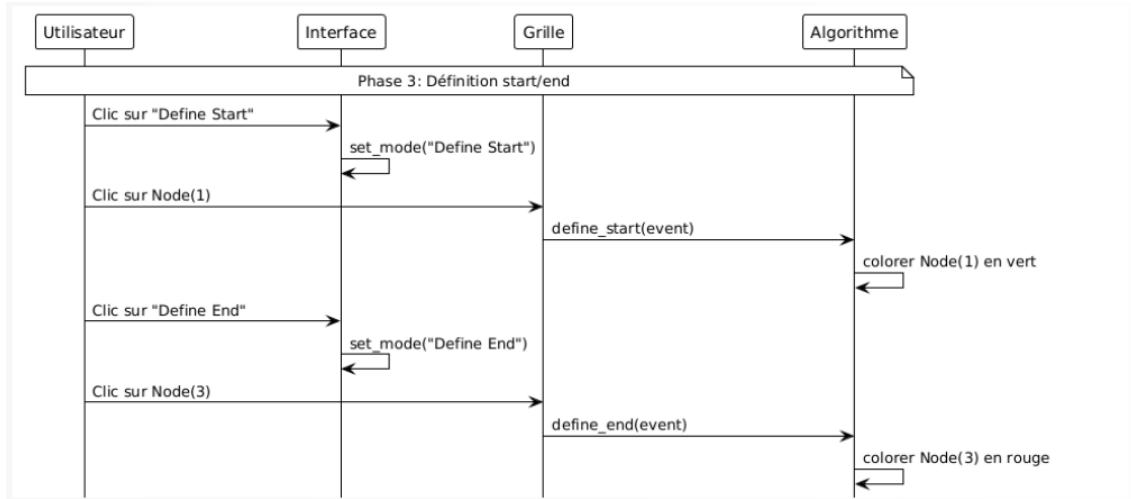


FIGURE 4.7 – Diagramme de séquence – Phase de définition des sommets de début et de fin

— Phase 4 : Exécution de l'algorithme

Acteur principal : Utilisateur

Objectif : Lancer l'algorithme de recherche du plus court chemin et afficher le résultat optimal

Scénario :

- L'utilisateur clique sur "Launch" pour lancer l'algorithme de recherche de chemin optimal
- L'interface active le mode d'exécution
- Le système valide la configuration (présence de sommets de départ et d'arrivée)
- L'algorithme initialise une table heuristique pour stocker les distances et coûts
- Le système construit la liste d'adjacence à partir des sommets et des arcs créés
- L'algorithme applique une méthode de recherche du plus court chemin (A*)
- Le système calcule le chemin optimal entre le sommet de départ et d'arrivée
- L'algorithme retourne la séquence de sommets constituant le chemin optimal
- Le système colore le chemin trouvé en rouge sur l'affichage
- Les résultats (distance totale, séquence de nœuds) sont affichés dans la zone Output

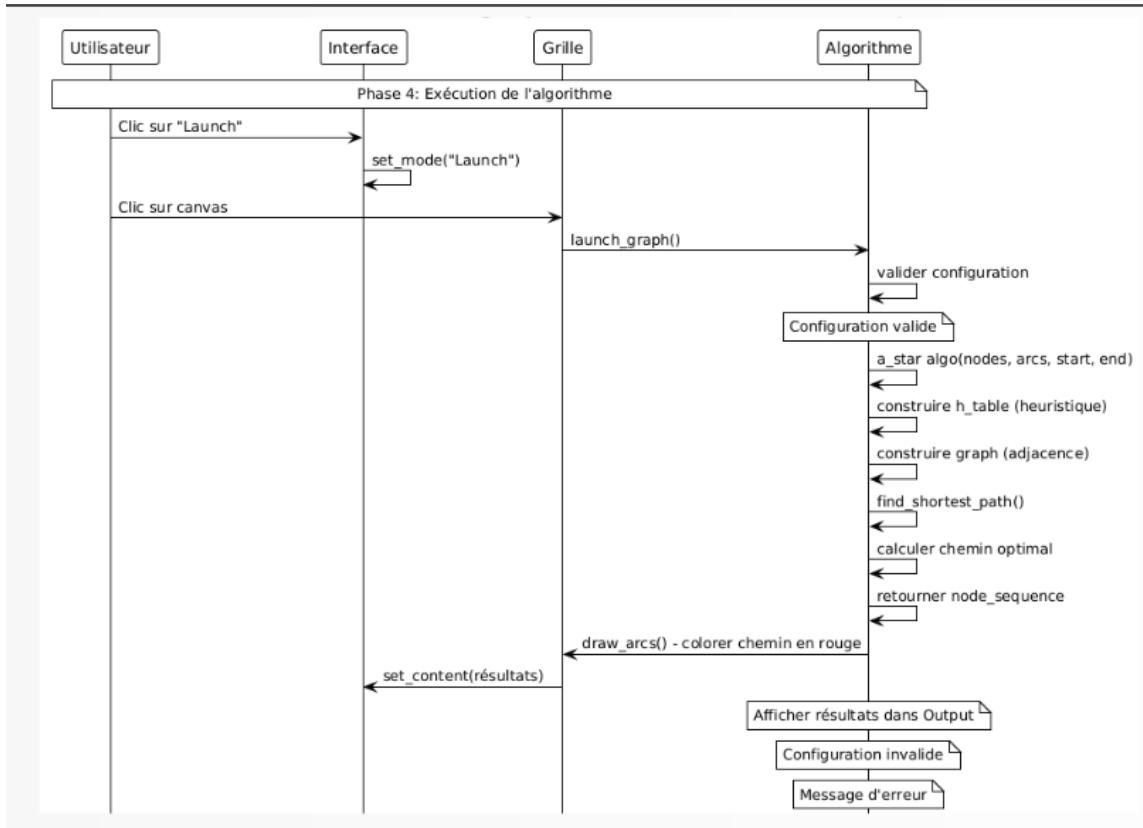


FIGURE 4.8 – Diagramme de séquence – Phase d'exécution de l'algorithme

— Phase 5 : Sauvegarde

Acteur principal : Utilisateur

Objectif : Sauvegarder la configuration du graphe et les résultats obtenus pour une utilisation ultérieure

Scénario :

- L'utilisateur clique sur "Save Graph" pour sauvegarder la configuration actuelle
- L'interface active le mode de sauvegarde des données
- Le système génère un fichier de sauvegarde contenant la structure du graphe
- Une boîte de dialogue "Sauvegarder fichier" permet à l'utilisateur de choisir l'emplacement
- L'utilisateur sélectionne l'emplacement et confirme la sauvegarde
- Le système écrit les résultats dans un fichier pour une utilisation ultérieure

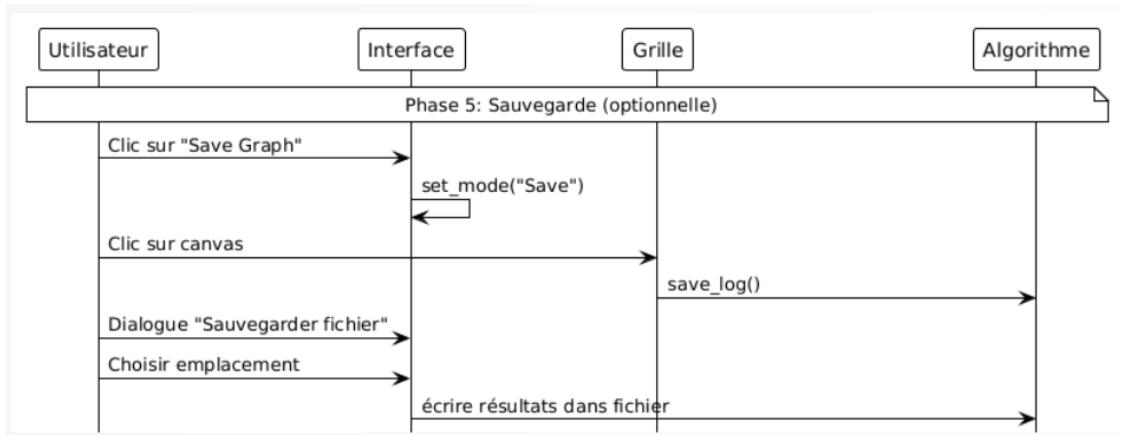


FIGURE 4.9 – Diagramme de séquence – Phase de Sauvegarde

— Phase 6 : Clear Canvas

Acteur principal : Utilisateur

Objectif : Effacer complètement le graphe actuel pour préparer un nouveau départ

Scénario :

- L'utilisateur clique sur "Clear Canvas" pour supprimer tous les éléments du graphe
- L'interface active le mode de nettoyage du canevas
- Le système demande à la grille d'effacer tous les noeuds et arcs affichés
- La grille appelle l'algorithme pour vider la liste d'adjacence
- Tous les noeuds, arcs, compteurs, ainsi que les sommets de début et de fin sont réinitialisés
- Le canevas devient vide et prêt à recevoir un nouveau graphe

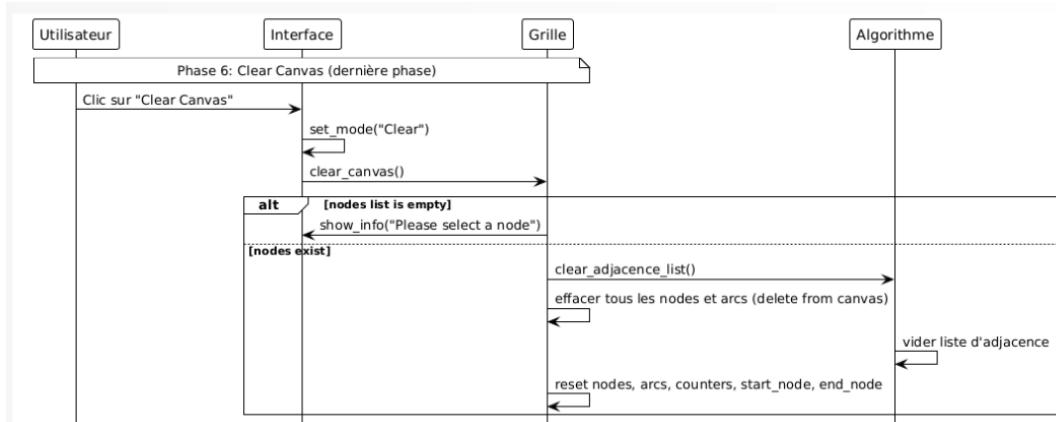


FIGURE 4.10 – Diagramme de séquence – Phase de Clear Canvas

Fin du scénario : Le chemin optimal est trouvé, visualisé sur le graphe.

Chapitre 5

Architecture, visualisation et interface utilisateur

5.1 Introduction

Dans cette phase de développement, j'ai passé de l'analyse et de la conception du projet à sa réalisation technique.

L'objectif est de traduire les diagrammes UML et les spécifications en code informatique, afin de créer une application qui répondra aux besoins préalablement définis. L'objectif de la phase d'implémentation est d'aboutir à un produit final, exploitable par les utilisateurs.

Dans ce chapitre, consacré à la réalisation et à la mise en œuvre de mon application permettant de visualiser un graphe et le chemin le plus court, j'ai présenté les principales interfaces de l'application.

5.2 Choix des bibliothèques graphiques

Pour ce projet, j'ai choisi d'utiliser la bibliothèque **CustomTkinter**, qui est une extension de **Tkinter**, la bibliothèque graphique standard de Python.

- **Tkinter** (Tk interface) est un module intégré à Python. Il permet de créer des interfaces graphiques simples avec :
 - des fenêtres,
 - des frames,
 - des widgets (boutons, zones de texte, cases à cocher, etc.),
 - la gestion des événements (clavier, souris, etc.).

Tkinter est disponible sur Windows et la plupart des systèmes Unix, ce qui rend les interfaces portables.

- **CustomTkinter** est une bibliothèque basée sur Tkinter. Elle propose des éléments déjà stylisés et modernes, ce qui permet de créer une interface plus esthétique sans beaucoup d'efforts.
- **Pourquoi ce choix de Tkinter ?**
 - Facile à comprendre (le système de placement ressemble à *grid* en HTML),
 - Compatible avec tous les systèmes d'exploitation,
 - Possède des sous-bibliothèques comme CustomTkinter qui améliorent le design.

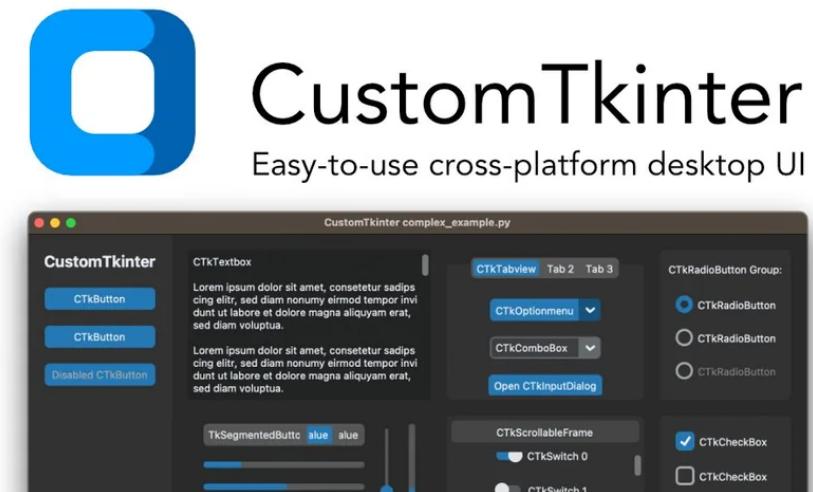


FIGURE 5.1 – La bibliothèque CustomTkinter

5.3 Architecture de l'application (fichiers modulaires)

Mon application est organisée en fichiers modulaires pour mieux structurer le code et faciliter sa maintenance. Voici la description de chaque module :

5.3.1 Structure modulaire

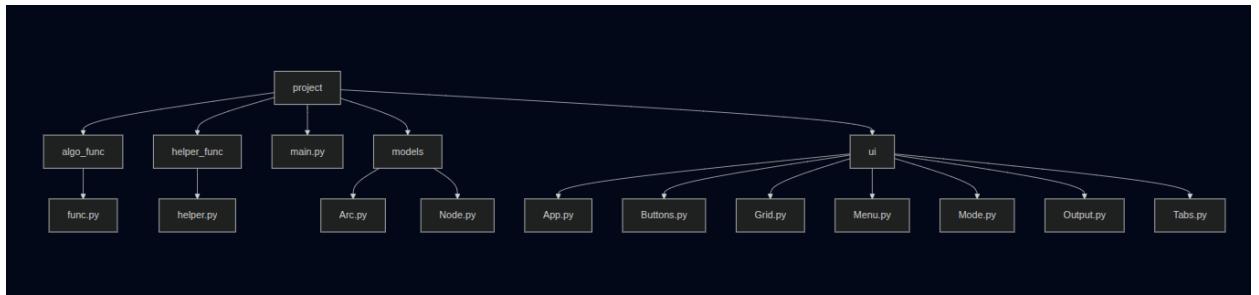


FIGURE 5.2 – La structure des fichiers du projet

Pourquoi ce système est bien organisé :

- Les tâches sont bien séparées : chaque partie du code a une fonction précise.
- Facile à modifier : on peut changer une partie sans toucher au reste.
- Évolution simple : on peut ajouter des fonctionnalités sans tout refaire.
- Réutilisable : certaines parties du code peuvent être utilisées dans d'autres projets.

5.3.2 Description des modules

- **algo_func/** : Contient la logique métier de l'algorithme de recherche de chemin
 - **func.py** : Implémente l'algorithme A* avec calcul des coûts et heuristiques
- **helper_func/** : Fournit des fonctions utilitaires
 - **helper.py** : Contient des fonctions auxiliaires pour le traitement des données
- **models/** : Définit les structures de données principales
 - **Arc.py** : Modélise les connexions entre sommets avec leurs poids
 - **Node.py** : Représente les sommets du graphe avec leurs positions
- **ui/** : Gère l'interface graphique (CustomTkinter)
 - **App.py** : Crée la fenêtre principale et organise les layouts
 - **Buttons.py** : Gère les boutons d'action (ajout sommet/arc, etc.)
 - **Grid.py** : Affiche la grille et gère les interactions dessus
 - **Menu.py** : Organise le panneau de contrôle latéral
 - **Mode.py** : Affiche le mode courant (ajout, suppression, etc.)
 - **Output.py** : Affiche les résultats textuels des algorithmes

- `Tabs.py` : Gère les onglets de navigation entre les vues
- `main.py` : Point d'entrée qui lance l'application

5.3.3 Flux de données

Le programme fonctionne étape par étape, comme une chaîne :

- **L'utilisateur agit** : Il clique sur un bouton ou utilise la grille (ex : pour définir un sommet de départ, une arrivée, etc.).
- **Le programme réagit** : Il interprète la demande de l'utilisateur et exécute l'action correspondante.
- **Les données sont mises à jour** : Les éléments comme les sommets (*Node*) ou les arcs (*Arc*) sont modifiés selon l'action effectuée.
- **L'algorithme A* démarre** : Lorsqu'on demande de chercher un chemin, le programme lance l'algorithme A* pour trouver le meilleur chemin.
- **Le résultat s'affiche** : Le chemin trouvé est affiché dans une zone dédiée à l'écran.

5.4 Résultats et exemples visuels

Dans cette section, je veux présenter l'affichage des graphes, la visualisation du chemin optimal, ainsi que différents cas d'utilisation.

5.4.1 Exemple réel d'un graphe

Voici le graphe sur lequel je veux travailler. Je vais modéliser les sommets dans mon application sous forme de nombres, pas de caractères, comme vous le voyez dans le graphe.

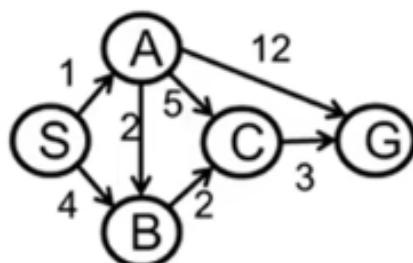


FIGURE 5.3 – Exemple d'un graphe

Le chemin le plus court dans ce graphe est : S → A → B → C → G, dans notre application : 1 → 2 → 3 → 4 → 5.

5.4.2 Lancement de l'application

Première vue de l'application

Lorsqu'une application est lancée, la grille est vide et le mode est nul.

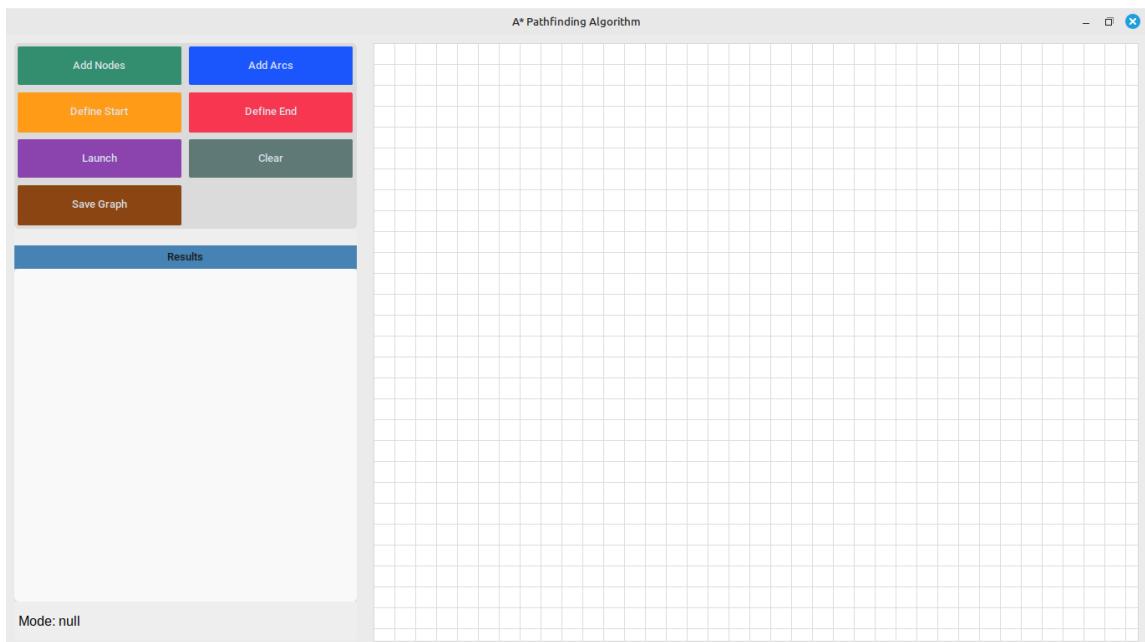


FIGURE 5.4 – First View

Sélection du mode d'ajout des sommets

L'utilisateur sélectionne le mode d'ajout de sommets, puis clique sur la grille pour ajouter des sommets.

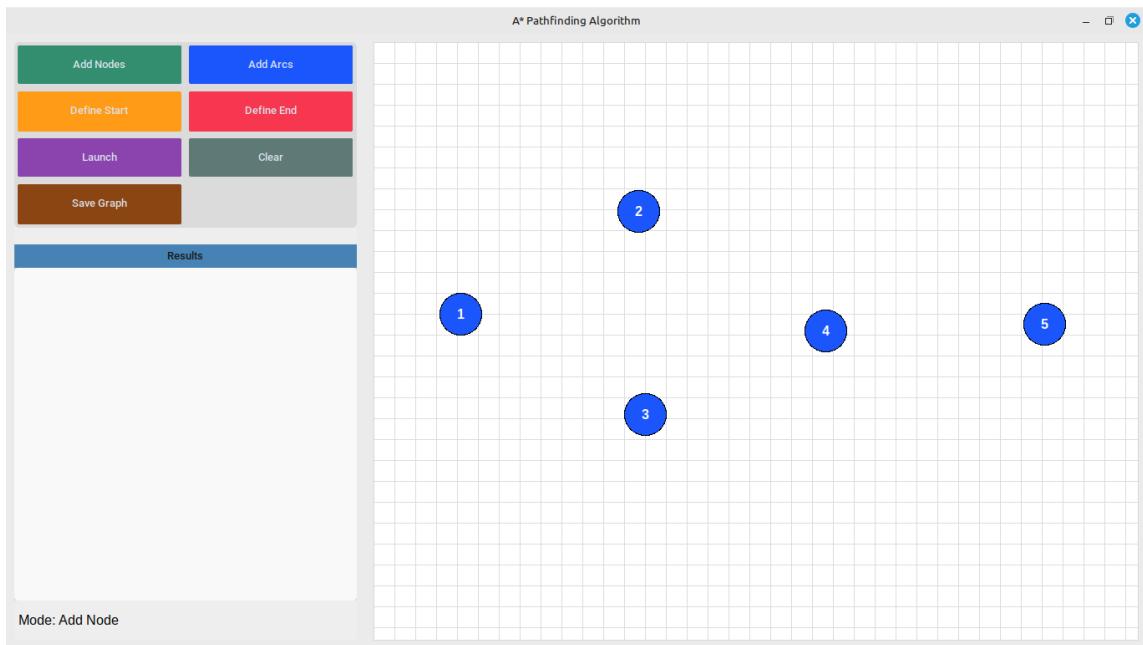


FIGURE 5.5 – Ajout des sommets

Sélection du mode d'ajout des arcs

L'utilisateur sélectionne le mode d'ajout des arcs, clique sur deux sommets, puis l'application demande de saisir un poids.

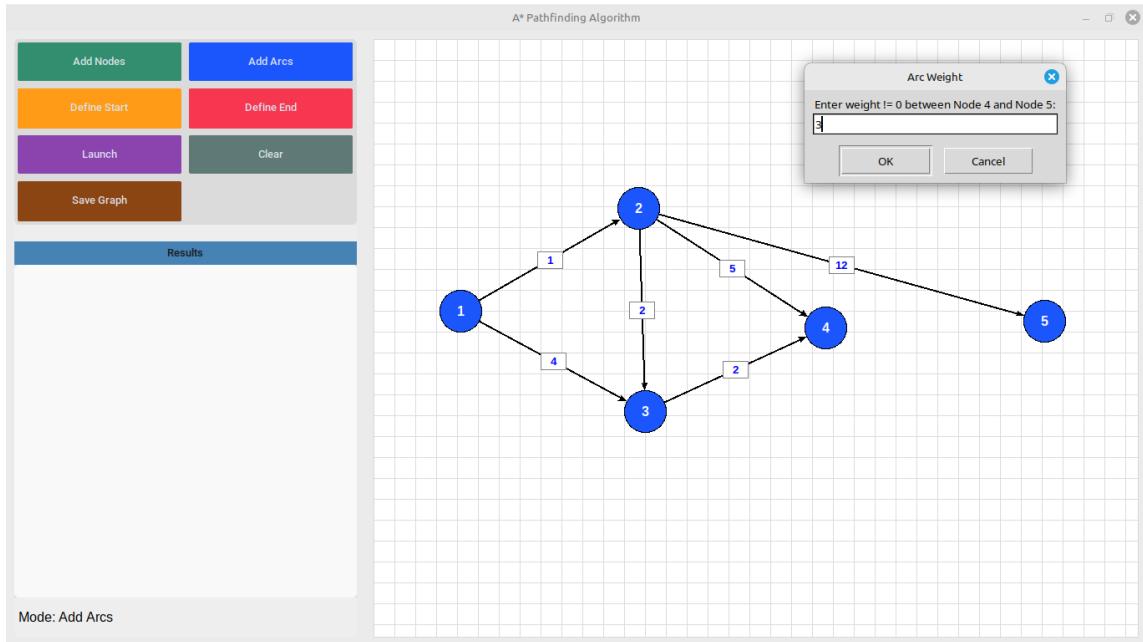


FIGURE 5.6 – Ajout des arcs

Définition du sommet de départ

L'utilisateur sélectionne le mode pour définir le départ, puis choisit un sommet comme sommet de départ.

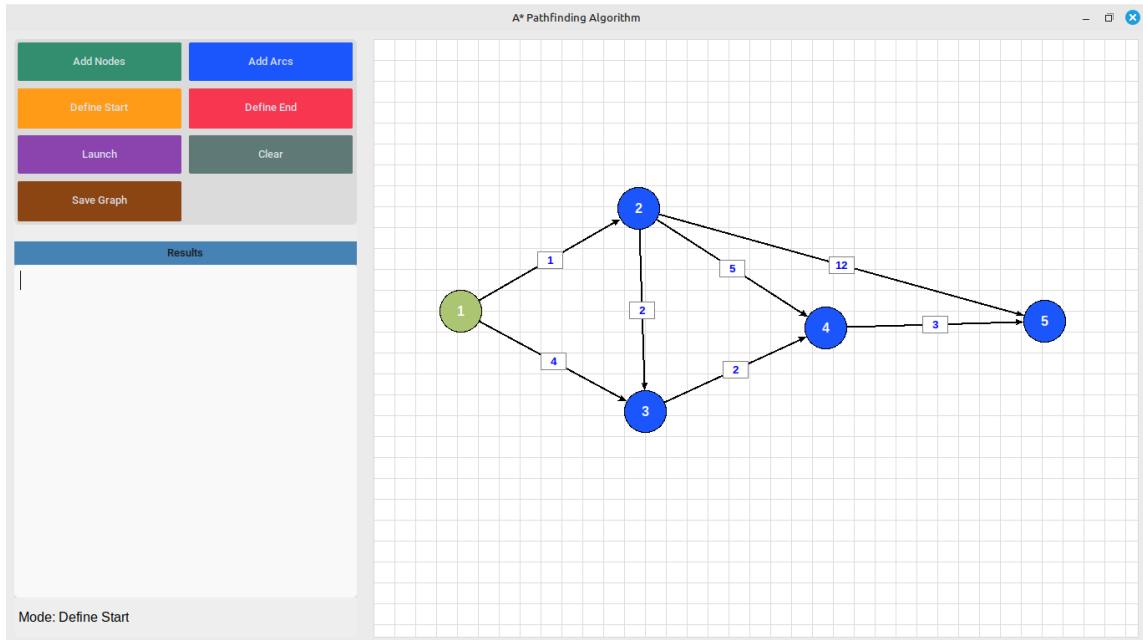


FIGURE 5.7 – Définir départ

Définition du sommet d'arrivée

L'utilisateur sélectionne le mode pour définir l'arrivée, puis choisit un sommet comme sommet d'arrivée.

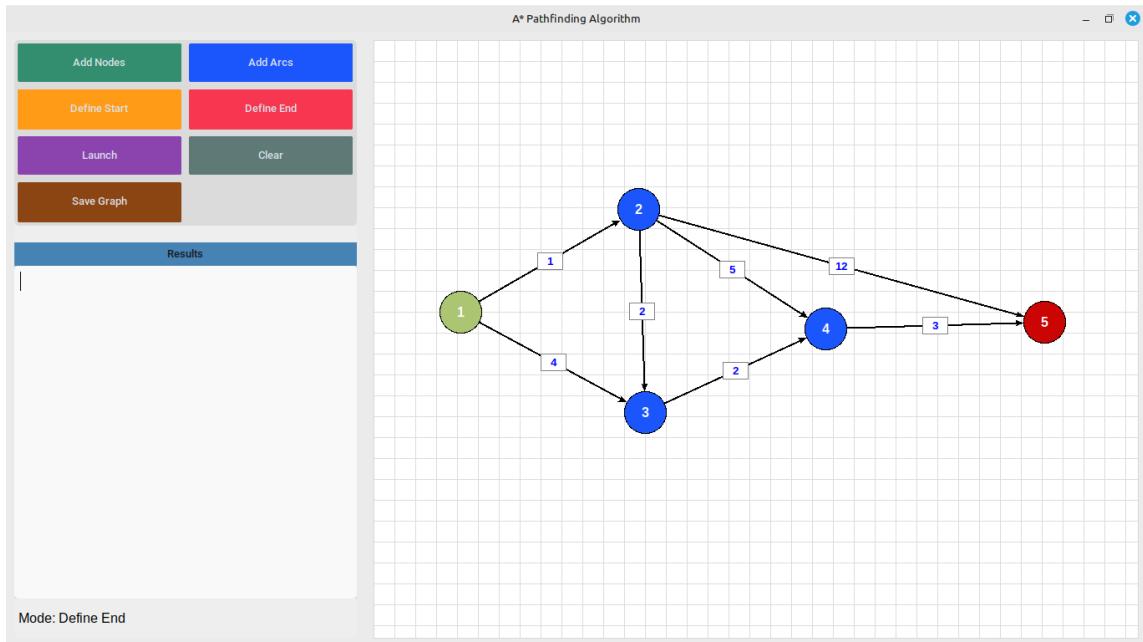


FIGURE 5.8 – Définir arrivée

Exécution de l'algorithme

L'utilisateur clique sur "Luanch", et l'algorithme commence à déterminer le plus court chemin. Des logs s'affichent ensuite dans la section des résultats.

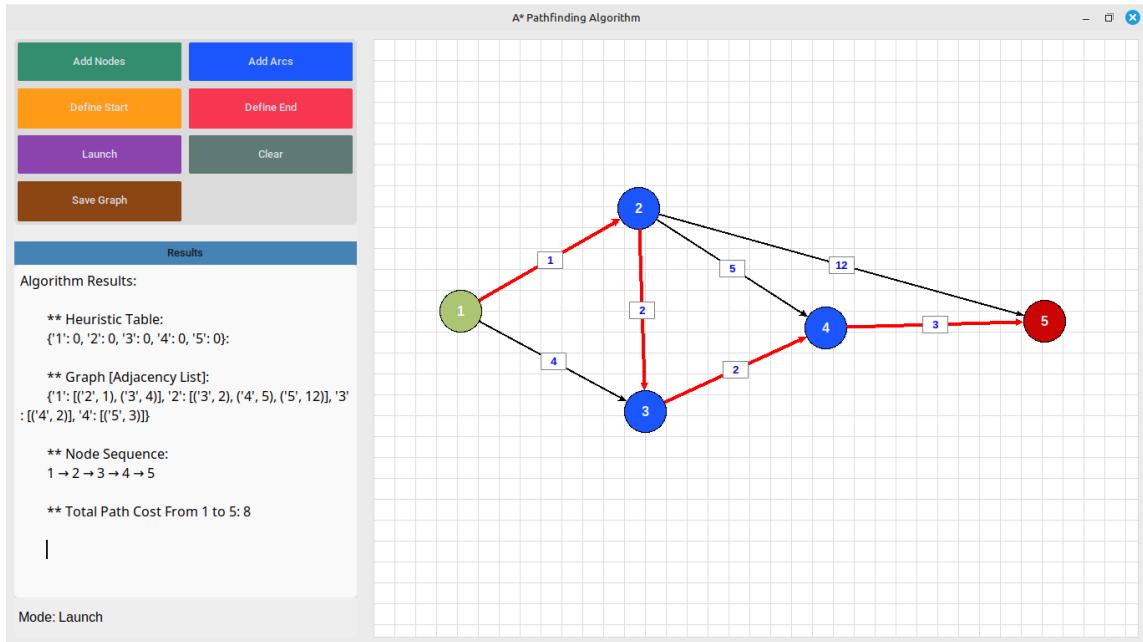


FIGURE 5.9 – Exécution de l'algorithme

Sauvegarde du graphe

L'utilisateur peut sauvegarder les journaux s'il le souhaite.

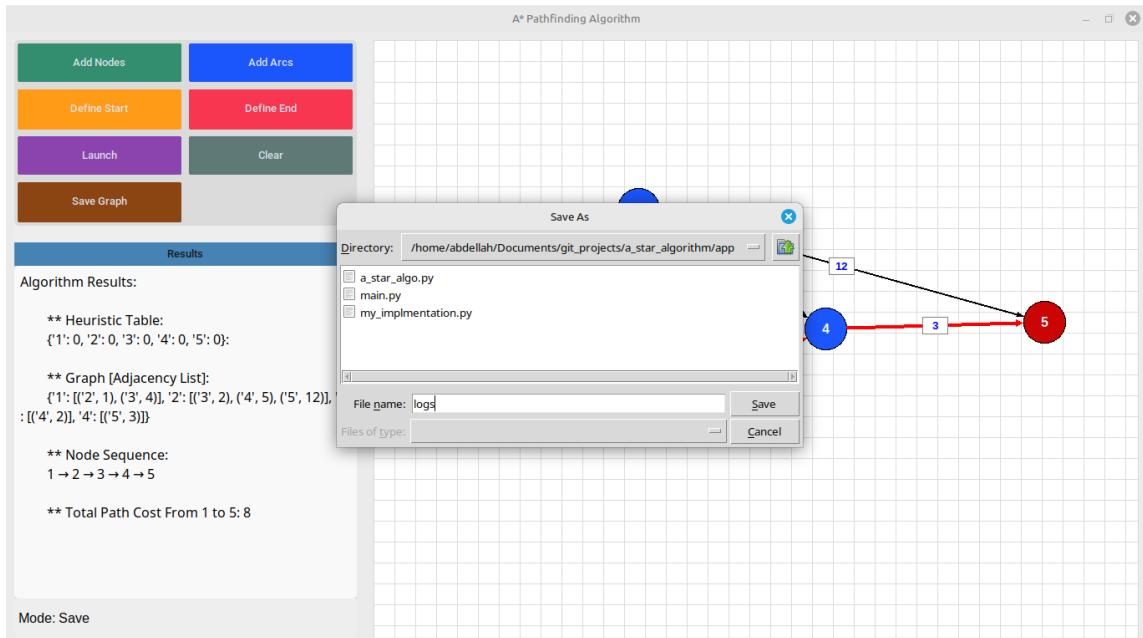


FIGURE 5.10 – Sauvegarde des journaux

Effacement du graphe

L'utilisateur peut effacer tout le contenu de la grille ainsi que les journaux en cliquant sur "Clear".

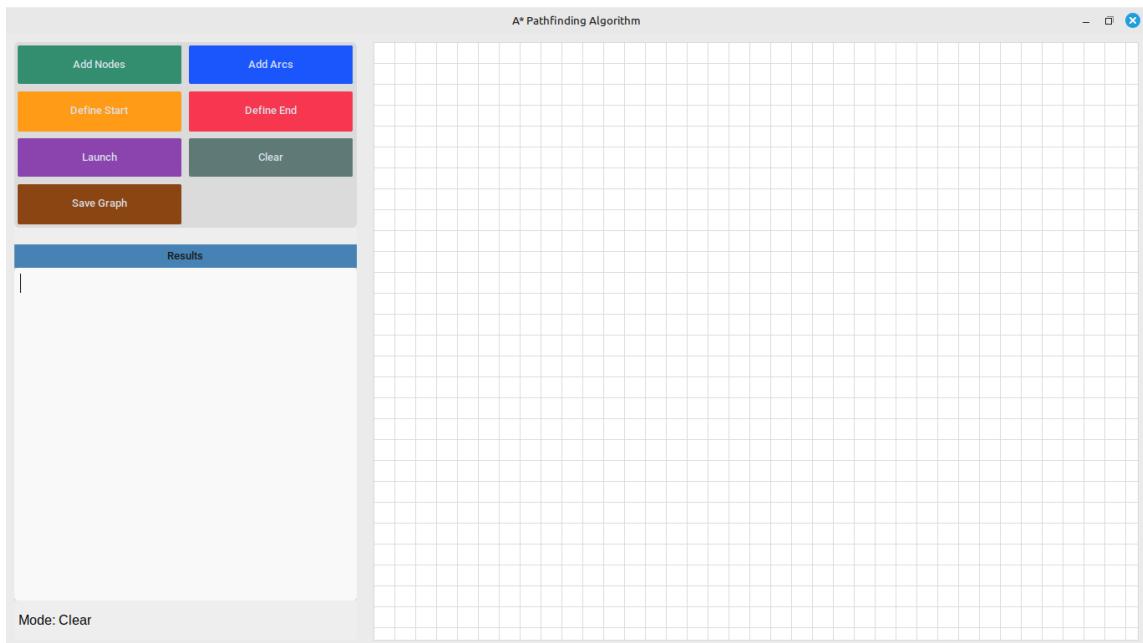


FIGURE 5.11 – Effacement du graphe

Conclusion générale

Mon projet de fin d'études porte sur l'étude et la programmation en Python de l'algorithme A*. Cet algorithme, largement utilisé en intelligence artificielle, permet de rechercher le plus court chemin dans un graphe entre un point de départ et un point d'arrivée. Grâce à sa simplicité et son efficacité, il constitue un bon exemple d'algorithme de planification.

Dans ce travail, j'ai pour objectif de comprendre en profondeur le fonctionnement de l'algorithme A*, d'étudier la théorie des graphes, et d'identifier les structures de données les plus adaptées à son implémentation, comme la liste d'adjacence et la file de priorité. Je développe une solution en Python capable de traiter des graphes orientés et pondérés positivement.

Je conçois une architecture modulaire pour organiser le code, et j'intègre une interface graphique permettant de visualiser les graphes et les chemins calculés. Pour cela, j'utilise Tkinter.

Je suis convaincu que ce projet est une excellente opportunité pour mettre en pratique mes connaissances en algorithmique, en programmation Python et en conception d'interfaces.

Améliorations à venir

Dans le cadre de l'évolution de cette application, plusieurs améliorations et fonctionnalités sont envisagées afin d'enrichir l'expérience utilisateur et d'augmenter ses capacités :

- **Fonctionnalité de déplacement des nœuds et des arcs** : Actuellement, les nœuds et arcs sont statiques après leur création. Il est prévu d'ajouter une interface intuitive permettant à l'utilisateur de sélectionner et déplacer librement les nœuds ainsi que leurs arcs associés, pour faciliter la modification et la personnalisation du graphe.
- **Fonctionnalité de suppression des nœuds** : Il sera également possible de supprimer un ou plusieurs nœuds ainsi que leurs arcs connectés. Cette fonctionnalité permettra une gestion plus flexible du graphe, notamment pour corriger des erreurs ou simplifier la structure du graphe lors de la modélisation.
- **Développement d'un site web de présentation** : Un site web sera créé pour présenter l'application, ses objectifs, son mode d'utilisation et ses résultats. Ce site aura pour vocation de fournir une documentation claire et accessible, des tutoriels, ainsi que des exemples d'utilisation. Il pourra aussi servir de plateforme pour recueillir les retours des utilisateurs et partager les mises à jour du projet.

Ces perspectives visent à rendre l'application plus complète, ergonomique et accessible, tout en facilitant son adoption dans différents contextes d'étude et d'enseignement.

Annexe

Dans le cadre de ce projet, j'ai utilisé plusieurs outils et ressources pour le développement et la compréhension de l'algorithme A*.

Outils utilisés

- **VS Code** : Environnement de développement intégré (IDE) pour écrire et tester le code Python.
- **Tkinter** : Bibliothèque Python utilisée pour créer l'interface graphique de visualisation des graphes.
- **Trello** : Outil de gestion de projet pour organiser les tâches et le suivi de l'avancement.

Ressources consultées

Pour approfondir mes connaissances et assurer la qualité de l'implémentation, j'ai consulté plusieurs sources fiables et didactiques :

- Tutoriel vidéo complet sur l'algorithme A* : <https://youtu.be/j7vtAAMKV4A?si=2-Tb8G8rs5L7oWbZ>
- Article explicatif détaillé en français : <https://datascientest.com/lalgorithme-a-a-star-algorithm>
- Tutoriel avec exemples pratiques en Python : <https://www.datacamp.com/tutorial/a-star-algorithm>
- Blog pédagogique sur les algorithmes de pathfinding : <https://www.redblobgames.com/blog/>