Rapport de soutenance 3

Visualizer d'algorithme

Victor Tang B1
Sacha Layani B1
Olivier Bensemhoun B1
Jayson Vanmarcke B1

3 Juin 2024

Sommaire

1 Introduction

Bonjour et bienvenue dans cet ultime rapport de soutenance.

Nous allons vous présenter l'avancement final de notre travail concernant notre projet qui s'intitule : Visualizer d'algorithme.

Le but est de vous exposer clairement et simplement l'évolution de notre projet depuis notre dernière passation avec vous.

Pour rappel, notre visualizer va comparer « visuellement » différents algorithmes classiques tels que BFS, Dijkstra, A* (. . .).

Comme indiqué lors de la première soutenance, nous sommes quatre à constituer ce groupe qui s'intitule « Les Alchimistes ». Nous allons vous présenter ce projet et espérons que vous l'apprécierez ainsi que le travail commun déployé afin d'y parvenir et satisfaire vos attentes.

En ce qui concerne l'avancement du projet, il est terminé et contient même quelques implémentations supplémentaires.

Ce projet a été réalisé sous Rust avec l'aide de Cargo qui est un outils de gestion de projet, comme NuGet ou PIP. Ce projet a pour but de comprendre de manière visuelle, différents algorithmes étudiés en classe mais aussi quelques algorithmes qui ne sont pas au programme comme Counting Sort ou encore A*.

Nous vous en souhaitons bonne lecture !

2 Origines et Nature du Projet

2.1 Idée du Projet

Pour commencer, nous avions pour première idée commune de réaliser un projet sur les blockchains. Le choix de ce thème fut très rapide. Toutefois, après concertation, cette idée a engendré de nombreux débats au sein du groupe car à l'unanimité, nous sommes arrivés au constat que les blockchains étaient un sujet très vaste voir trop vaste pour deux d'entres nous. Mais, nous décidons de garder cette idée qui fera partie des deux retenues pour les présenter à notre professeur.

Dans un second temps, nous avons cherché et étudié d'autres thèmes pour finalement s'être entendu sur un sujet particulièrement intéressant : le visualizer d'algorithmes. Lors de nos recherches, nous avons été attirés par un site qui mettait en relief le processus du vizualizer d'algorithmes et voulions d'un commun accord le présenter comme second sujet.

Finalement, notre professeur Mr Boulay, a donné son accord pour le vizualizer d'algorithmes qui avait captivé tous les membres de notre groupe lors de sa découverte durant la phase de recherches.

2.2 Organisation

Nous avons choisi d'utiliser Github pour améliorer notre organisation et notre efficacité dans la réalisation de ce projet. Github est une plateforme collaborative de développement de logiciels qui permet de travailler en équipe sur un même projet.

En utilisant Github, nous avons travaillé ensemble en parallèle sur différentes parties de notre projet, tout en veillant à ce que chaque modification apportée au code soit suivie et synchronisée en temps réel. Cela a grandement amélioré notre collaboration et notre communication en tant qu'équipe.

En outre, Github nous a également aidé à gérer les conflits de version, en facilitant la fusion des différentes versions du code. Cette fonctionnalité a évité les pertes de temps et les erreurs potentielles qui auraient pu se produire en travaillant sur des versions de code différentes.

Enfin, nous avons utilisé Discord comme plateforme de communication pour notre projet, ce qui nous a permis de rester en contact facilement et efficacement tout au long du développement. Nous avons échanger des idées, discuter des problèmes et coordonner notre travail en temps réel grâce aux fonctionnalités de chat vocal et textuel de Discord. En utilisant cette plateforme, nous avons pu maintenir une communication fluide et régulière entre les membres de l'équipe, ce qui a grandement amélioré notre collaboration et notre organisation.

2.3 Formation du groupe

La formation du groupe s'est faite très simplement et rapidement. Dès le début du S3, nous avions connaissance de ceux qui partaient à l'international et de ceux qui restaient en France pour le S4.

Immédiatement, le groupe s'est instantanément formé sans même avoir la confirmation d'un projet en S4 et surtout s'il se ferait par groupe de 4. La seule évidence pour nous quatre, était d'être ensemble.

Après les examens du S3, chacun avait connaissance de sa classe en S4 et nous étions dans la même classe : B1.

A partir de ce moment là, nous commencions à réfléchir sur des idées de projet avec présentation de deux idées à notre professeur pour ensuite travailler activement sur le projet.

3 Etat de l'art

3.1 Introduction

Dans notre groupe, nous réalisons presque chaque jour des projets informatiques, qu'ils soient personnels ou scolaires. Le problème est que nous appliquons soit des concepts vus en cours, soit des notions découvertes grâce à nos propres recherches. En principe, c'est une approche correcte, mais il manque un élément essentiel : un visualizer d'algorithmes. En effet, ce dernier serait utile pour comprendre parfaitement le fonctionnement de la théorie algorithmique.

Ainsi, l'objectif de ce projet sera de créer un visualizer d'algorithmes permettant d'observer l'exécution des algorithmes de base.

3.2 Outils de Visualisation d'Algorithmes Existant

Actuellement, plusieurs outils de visualisation d'algorithmes sont déjà disponibles, se déclinant sous diverses formes :

— Les IDE de JetBrains, tels que PyCharm pour Python, intègrent des fonctionnalités de débogage avancées. Ils permettent de suivre l'exécution du code étape par étape, d'inspecter les variables, et de détecter les erreurs efficacement.

— Les sites web, comme Python Tutor, offrent une visualisation en ligne des programmes Python, Java, JavaScript, C, ou encore le C++. Ces sites permettent de suivre l'exécution du code Python étape par étape.

— D'autres sites, tels que Pathfinding-Visualizer, permettent, comme son nom l'indique, de visualiser des algorithmes de recherche de chemins, comme l'algorithme A*. Ils offrent aux utilisateurs la possibilité de créer des grilles, d'ajouter des obstacles, et de visualiser comment l'algorithme trouve le chemin optimal.

3.3 Choix Technologiques et Architecture du projet

Afin de réaliser ce projet, nous avons besoin d'utiliser le langage de programmation Rust, qui est un langage de programmation novateur reprenant certains points C vus au S3 ou encore du C# vus au S2 et même du Ocaml abordé en cours lors du S1.

Pour la réalisation de ce projet, nous avons tout d'abord décidé qu'il fallait trouver une configuration qui soit valide aussi bien aujourd'hui que demain. Ainsi, lorsque nous établirons des liens entre les algorithmes et l'interface graphique, cela sera moins compliqué que s'il y avait plusieurs fichiers cargo.toml.

3.4 Implémentation des Algorithmes

Dans le cadre de ce projet, nous avons décidé d'implémenter des algorithmes relevant du domaine du *Path Finding*,*Recherche*, englobant des opérations de recherche, d'insertion et de suppression. La liste des algorithmes que nous avons choisi d'implémenter comprend :

- Algorithmes de Dijkstra
- Algorithme A* Search
- BFS (*Breadth-First Search*), DFS (*Depth-First Search*), insertion et suppression (arbre binaire)
- BFS, DFS (graph)
- Algorithme Floyd-Warshall
- Algorithme de Ford-Bellman
- Algorithme des arbres couvrants minimum (*Minimum Spanning Trees*)
- Insertion sort
- Merge sort
- Counting sort

Cette sélection d'algorithmes couvre une variété de domaines et de fonctionnalités, contribuant ainsi à la diversité et à la richesse de notre projet.

4 Organisation et Répartition des tâches

4.1 Organisation du travail et répartition des tâches

Tâches	Victor Tang	Jayson Vanmarcke	Olivier Benshemoun	Sacha Layani
Algorithmes Dijkstra's		Responsable		
Algorithmes A* Search				Responsable
BFS,DFS, insertion et suppression (arbre binaire)				Responsable
BFS,DFS (graph)		Responsable		
Algorithme Floyd-Warshall				Responsable
Algorithme de Ford-Bellman			Responsable	
Algorithme Spanning Trees minimum			Responsable	
Insertion sort	Responsable			
Merge sort			Responsable	
Counting sort		Responsable		
Interface graphique	Responsable			
Site Web		Responsable		Responsable

4.2 Répartition du travail dans le temps

Légende : += Ebauche, ++= Avancé, +++= Terminés

Tâches	Soutenance 1	Soutenance 2	Soutenance 3
Algorithmes Dijkstra's	+	++	+++
Algorithmes A* Search	+	++	+++
BFS,DFS, insertion et suppression (arbre binaire)	++	+++	+++
BFS,DFS (graph)	++	++	+++
Algorithme Floyd-Warshall	+	++	+++
Algorithme de Ford-Bellman	+	++	+++
Algorithme Spanning Trees minimum	+	++	+++
Insertion sort	++	+++	+++
Merge sort	++	+++	+++
Counting sort	+	++	+++
Interface graphique	+	++	+++
Site Web		++	+++

5 Ressenti

5.1 Jayson Vanmarcke

5.1.1 Satisfactions

Dès le démarrage de ce projet, j'étais enthousiaste à l'idée de coder nos différents algorithmes préférés, étudiés en classe en langage Rust, afin de découvrir les différentes fonctionnalités de celui-ci. Une fois notre projet validé, je n'avais qu'une chose en tête : coder les algorithmes que j'avais choisis.

J'étais content car à part les méthodes BFS/DFS sur les graphs, les algorithmes Dijkstra et Counting Sort étaient inconnus pour moi. Je me suis alors intéressé aux algorithmes de recherche de plus court chemin et quand j'ai vu que notre S4 d'algorithmique avait une partie sur les plus courts chemins, j'ai immédiatement saisi l'occasion. Dès lors, j'ai regardé les mimos pour apprendre, découvrir et comprendre. Puis, j'ai réalisé Dijkstra grâce aux explications de la vidéo mais aussi aux recherches effectuées sur les plus courts chemins.

Pour Counting Sort, c'était particulier car je n'avais jamais entendu ce tri. En l'étudiant, j'ai vu que cet algorithme se basait sur le « comptage » pour trier ses valeurs et qu'il représentait l'un des algorithmes de tri le plus rapide. J'étais donc impatient de constater cette fameuse rapidité qui s'est vérifiée en observant la rapidité de l'exécution du code.

Pour conclure, je suis heureux d'avoir réalisé ce projet et coder ces différents algorithmes classiques que l'on manipule au quotidien.

5.1.2 Frustrations

Je n'éprouve pas spécifiquement de frustrations particulières sur ce projet. La seule petite contrariété que je peux relever et qui m'a contrarié, c'est que j'aurais voulu implémenter plus d'algorithmes dans les catégories que l'on possède mais que nous n'avons pas eu la chance de réaliser par manque de temps.

5.2 Victor Tang

5.2.1 Satisfactions

Depuis le projet de S2, la création d'interface graphique m'intéresse grandement et en tant que responsable de l'interface graphique de ce projet, j'ai pris énormément de plaisir à le réaliser surtout lorsque je parviens à implémenter une fonctionnalité nouvelle et qu'il y a un résultat graphique.

5.2.2 Frustrations

Mes plus grosses frustrations sont, la compréhension de GTK et son utilisation. En effet, j'avais mis plus de 30 heures pour comprendre comment partager un GtkNotebook entre plusieurs fonctions et fichiers. On peut d'ailleurs voir les commentaires de commit que j'ai fait sur ma branche. Les problèmes d'emprunts et les problèmes d'ownership sont également des obstacles conséquents car certains objets de GTK ne peuvent pas être clonés et où je suis obligé d'utiliser RefCell ou la combinaison d'Arc et Mutex qui ne sont bien évidemment pas vu en cours.

5.3 Olivier Bensemhoun

5.3.1 Satisfactions

J'ai eu le plaisir de passer du temps sur un tel projet. Dans un premier temps, par la découverte de nouveaux algorithmes et dans un second temps, leurs réalisation.

5.3.2 Frustrations

J'ai du réaliser le projet en Rust qui est un langage qui malgré l'ensemble des tp et la réalisation de ce projet, ne me passionne pas par sa syntaxe et la nécessité de connaitre toutes les macros ou utiliser une doc. Sachez que je n'aime pas ce langage !

5.4 Sacha Layani

5.4.1 Satisfactions

En tant que chef de groupe, j'ai été enthousiasmé par la réalisation de ce projet avec une équipe motivée et engagée qui ont su être à l'écoute des propositions données. En effet, Victor, Jayson et Olivier ont accueilli mes idées avec ouverture ce qui a renforcé notre dynamique de groupe. Je suis fier d'avoir accompagné de nouveau mon équipe à atteindre les objectifs fixés pour ce projet.

5.4.2 Frustrations

Je suis pleinement satisfait par la réalisation de ce projet.

6 Description détaillée des tâches

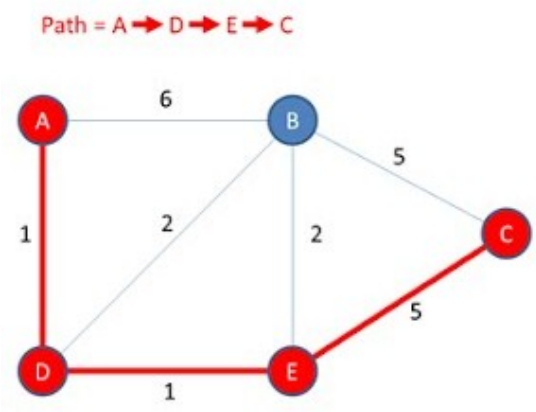
6.1 Algorithme Dijkstra's

L'algorithme de Dijkstra est un algorithme de recherche de chemin le plus court dans un graphe, c'est-à-dire un réseau de nœuds connectés par des arêtes ayant des poids ou des coûts associés. L'algorithme détermine le chemin le plus court entre un nœud de départ et tous les autres nœuds du graphe.

Cet algorithme est aussi surnommé algorithme glouton ou vorace car il utilise une approche gloutonne pour trouver le chemin optimal. Initialement, il attribue une distance infinie à tous les nœuds, sauf au nœud de départ qui a une distance de 0. Puis, l'algorithme sélectionne le nœud non visité avec la distance la plus faible, appelé le « nœud courant » à chaque répétition. Il met à jour les distances des nœuds voisins en les comparants à la distance actuelle du nœud courant, plus le poids de l'arête les reliant. Si la nouvelle distance est plus petite, elle est mise à jour. L'algorithme répète ce processus jusqu'à ce que tous les nœuds aient été visités ou que la distance minimale vers le nœud d'arrivée soit trouvée.

L'algorithme de Dijkstra assure de trouver le chemin le plus court dans un graphe sans arêtes de poids négatif, mais peut ne fonctionner correctement si de tels poids sont présents. Son utilisation est très répandue dans les applications de routage, dans les réseaux de télécommunications, la planification de trajets dans les systèmes de transports et d'autres domaines où la recherche de chemins optimaux est incontournable.

Il s'agit donc d'un algorithme très intéressant car soucieux de trouver le chemin le plus court, il sera peut-être plus optimisé que certains algorithmes déjà étudiés. La réponse à cette question sera prochainement apportée.



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

Afin de coder cet algorithme, il me fallait plusieurs informations. Tout d'abord, débutant le Rust, je n'ai aucune notion pour traiter et implémenter des graphs. Après de longues recherches et de compréhension, j'ai regardé le mimo (vidéo) sur l'algorithme de Dijsktra afin de comprendre son fonctionnement puis, j'ai étudié la fiche synthèse et la procédure donnée pour m'aider à coder cet algorithme. Une fois tous ces éléments en ma possession, je pouvais enfin coder mon algorithme. Ensuite, vient le codage de la fonction. Pour le codage, je me suis appuyé sur le mimo d'algorithmique S4 sur Dijkstra notamment du pseudo code pour écrire ma fonction. Cela

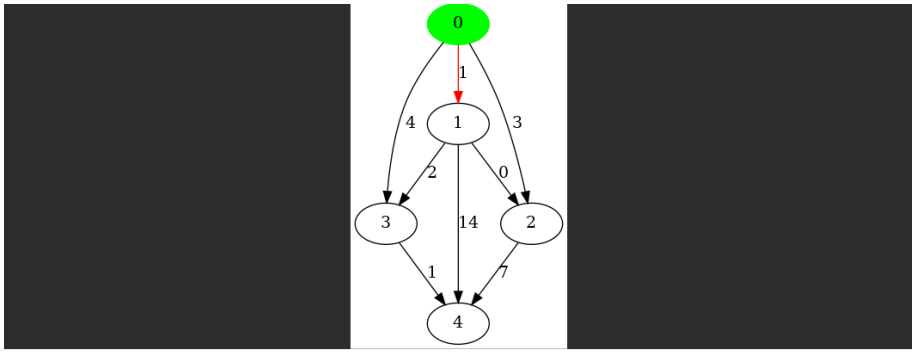
m'a beaucoup aidé pour compléter la fonction.

Jusqu'à maintenant, l'algorithme de Dijkstra était fonctionelle mais trop complexe pour la compréhension. J'ai donc décidé de le modifier légèrement afin de le rendre plus simple à comprendre. L'initialisation de la fonction n'est pas très complexe : tout d'abord, elle prend en paramètre une source à partir d'un sommet. Ceci est mon point de départ pour trouver le plus court chemin. Puis, il y a le graph sur lequel on va travailler.

Je commence par faire une fonction intermédiaire permmettant de choisir le sommet ayant la plus petite distance. Dès que cette fonction est réalisée, il faut faire Dijkstra. Je commence par initialiser deux vecteurs : un pour la distance et l'autre pour les prédécesseurs des sommets. Ensuite, je pars de ma source et je parcours ses fils, en regardant si la distance du fils n'est pas inférieur à la distance du père plus le coût de l'arc père-fils. Si la distance obtenue est plus petite, je modifie le vecteur distance et le vecteur père. Sinon, je ne fais rien. Je répète cette opération en boucle jusqu'à avoir parcourue tous les sommets accessibles du graph (certains ne sont pas accessibles et si c'est le cas, on aura comme valeurs : infini et -2).

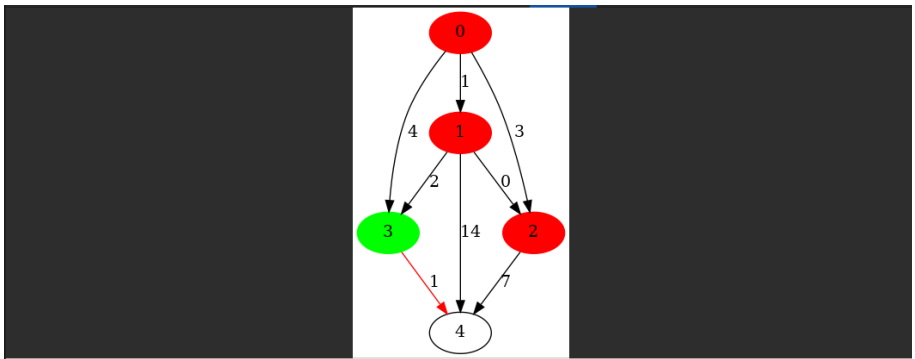
A la fin du parcours, mon programme renvoie les vecteurs distances et prédécesseurs.

Voici un résultat de l'application de Dijkstra sur l'interface graphique en prenant 0 comme source.



dist : [0, 1, inf, inf, inf]
p : [-1, 0, -2, -2, -2]

Cette seconde photo vous montre le résultat une fois l'algorithme appliqué sur le graph.

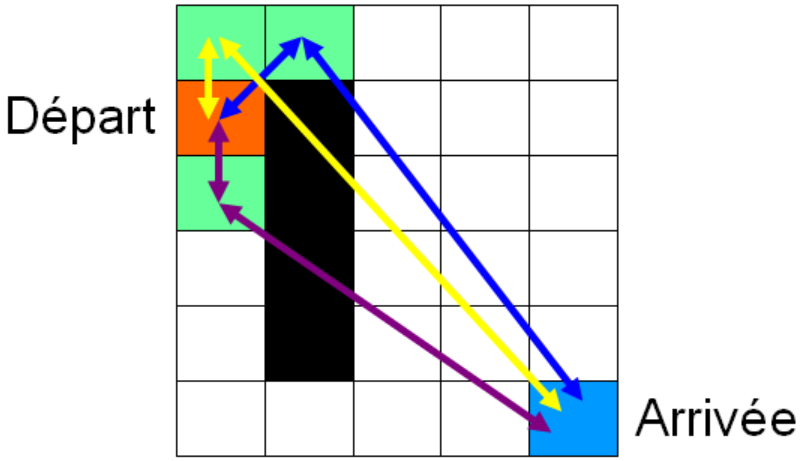


dist : [0, 1, 1, 3, 4]
p : [-1, 0, 1, 1, 3]

6.2 Algorithme A* Search

L'algorithme A* est utilisé pour trouver un chemin entre un nœud initial et un nœud final. Il repose sur une évaluation approximative de chaque nœud afin d'estimer le chemin optimal, puis explore les nœuds en suivant cette estimation. Cet algorithme est simple, ne nécessite pas de préparation préalable et limite l'utilisation de la mémoire.

L'algorithme A* est considéré comme le plus rapide pour trouver le chemin le plus court, surpassant l'algorithme de Dijkstra. En d'autres termes, intégrer cet algorithme dans notre projet serait une excellente manière de comparer le temps de traitement et le raisonnement entre l'algorithme A* et l'algorithme de Dijkstra.



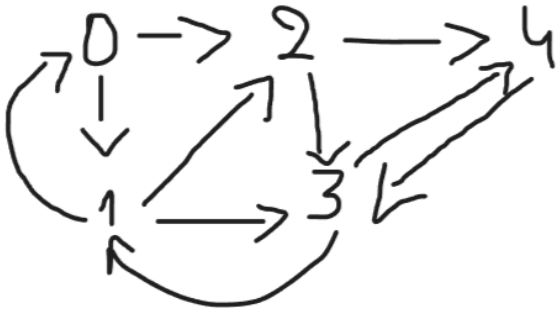
Étant donné que l'algorithme de recherche A* n'était pas abordé au cours de ma scolarité, j'ai dû me renseigner sur son fonctionnement. J'ai commencé par explorer plusieurs sites internet universitaires pour comprendre la théorie derrière cet algorithme. Malheureusement, la plupart de ces sites semblaient être conçus pour des étudiants déjà familiarisés avec le sujet, ce qui m'a laissé dans le flou puisque je n'avais jamais été exposé à ce concept auparavant.

J'ai alors eu l'idée de me tourner vers des vidéos explicatives, qui ont souvent l'avantage d'être plus claires et détaillées. Cette démarche s'est avérée fructueuse, car une vidéo en particulier, que j'ai dû visionner à plusieurs reprises, m'a permis de comprendre le fonctionnement de cet algorithme.

Tout d'abord, on commence par initialiser deux listes. La première est la liste des sommets à visiter (open_liste), et la deuxième est la liste des sommets déjà visités (closed_list). Ensuite, nous entamons le parcours du graphe. À chaque sommet, on calcule un "score" avec la formule suivante : score = coût déjà parcouru jusqu'à cet endroit + estimation du coût restant jusqu'à la destination. Une fois que les scores ont été calculés, on choisit le chemin le plus optimiste, celui ayant le coût le plus bas. Ainsi, notre algorithme continue à calculer et sélectionner les endroits jusqu'à ce qu'il atteigne sa destination ou qu'il n'y ait plus d'endroits à visiter.

En résumé, l'algorithme A* utilise une combinaison de la distance déjà parcourue et d'une estimation de la distance restante pour guider ses choix et trouver le meilleur chemin vers sa destination.

Voici un exemple :
Ce schéma représente un graphe. À présent, l'objectif est de le convertir en une



liste d'adjacence.

Pour cela, nous utilisons la méthode suivante :

```rust
fn main() {
    let _g: Graph = Graph::new( key: 0,  adjlists: vec![vec![0, 1, 1, 0, 0],
                                        vec![1, 0, 1, 1, 0],
                                        vec![0, 0, 0, 1, 1],
                                        vec![0, 1, 0, 0, 1],
                                        vec![0, 0, 0, 1, 0]],  order: 5);
    let start : i32  = 0;
    let end : i32  = 4;
    let result : Vec<i32>  = a_algorithm(_g, start, end);
    println!("The shortest path from {} to {} is : {:?}", start, end, result);
}
```

Voici le contenu de la fonction main(), qui est le point d'entrée de notre programme et qui exécutera le code avec l'exemple graphique.

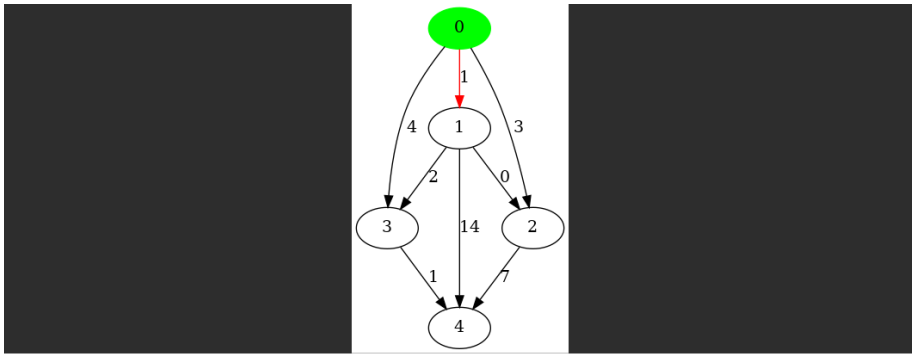
En observant, nous constatons que le graphe a été converti en une liste d'adjacence.

```
The shortest path from 0 to 4 is : [0, 2, 4]
```

Le résultat que j'obtiens est bien le chemin le plus cours, c'est à dire : 0 -> 2 -> 4.

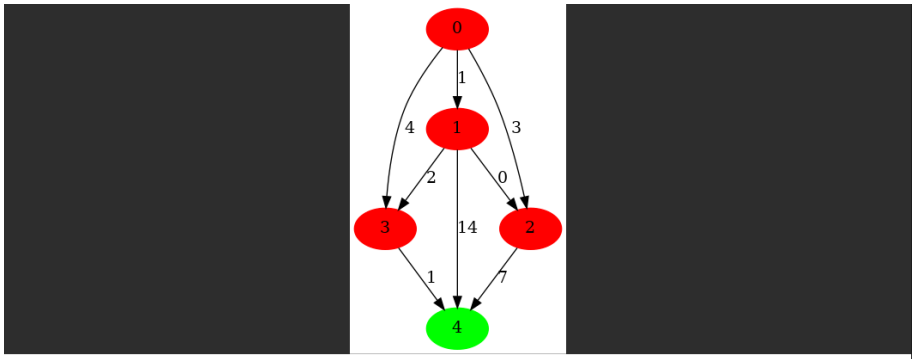
Voici un résultat de mon programme sur l'interface graphique en prenant 0 comme source :

Cette première photo vous montre le graph d'origine ainsi que les vecteurs.



dist : [0, 1, inf, inf, inf]
p : [-1, 0, -2, -2, -2]
H : [4, 3, inf, inf, inf]
F : [4, 4, inf, inf, inf]

Cette deuxième photo vous montre le résultat de cet algorithme sur le graph.

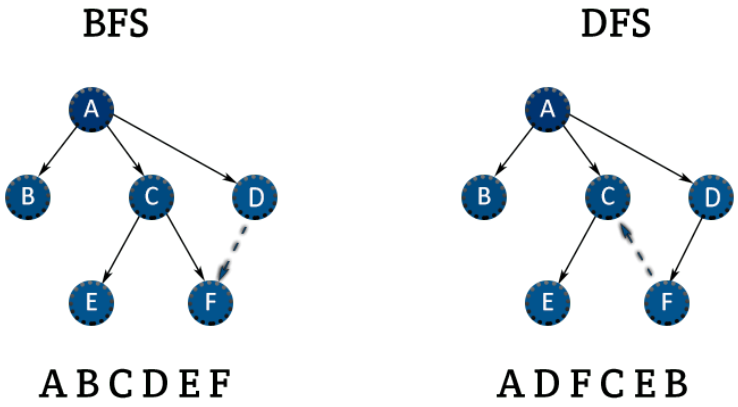


dist : [0, 1, 1, 3, 4]
p : [-1, 0, 1, 1, 3]
H : [4, 3, 2, 1, 0]
F : [4, 4, 3, 4, 4]

6.3 ABR (Arbres Binaire de Recherche)

Comme vu lors de notre S2, les principaux algorithmes à implémenter seront ceux de l'insertion, de la suppression, ainsi que de la recherche dans le contexte d'un arbre binaire de recherche. Néanmoins, étant donné que le projet consiste à réaliser un visualiseur d'algorithme, nous aimerions l'implémenter à la fois pour un BFS (parcours en largeur : niveau par niveau) et un DFS (parcours en profondeur : à compléter).

Cette partie vise à détailler, étape par étape, la manière d'insérer, de supprimer et de rechercher un élément dans un arbre binaire de recherche, tout en présentant visuellement ces étapes.



Ce qui a été le plus complexe n'a pas été les algorithmes en eux-mêmes, mais plutôt leur implémentation.

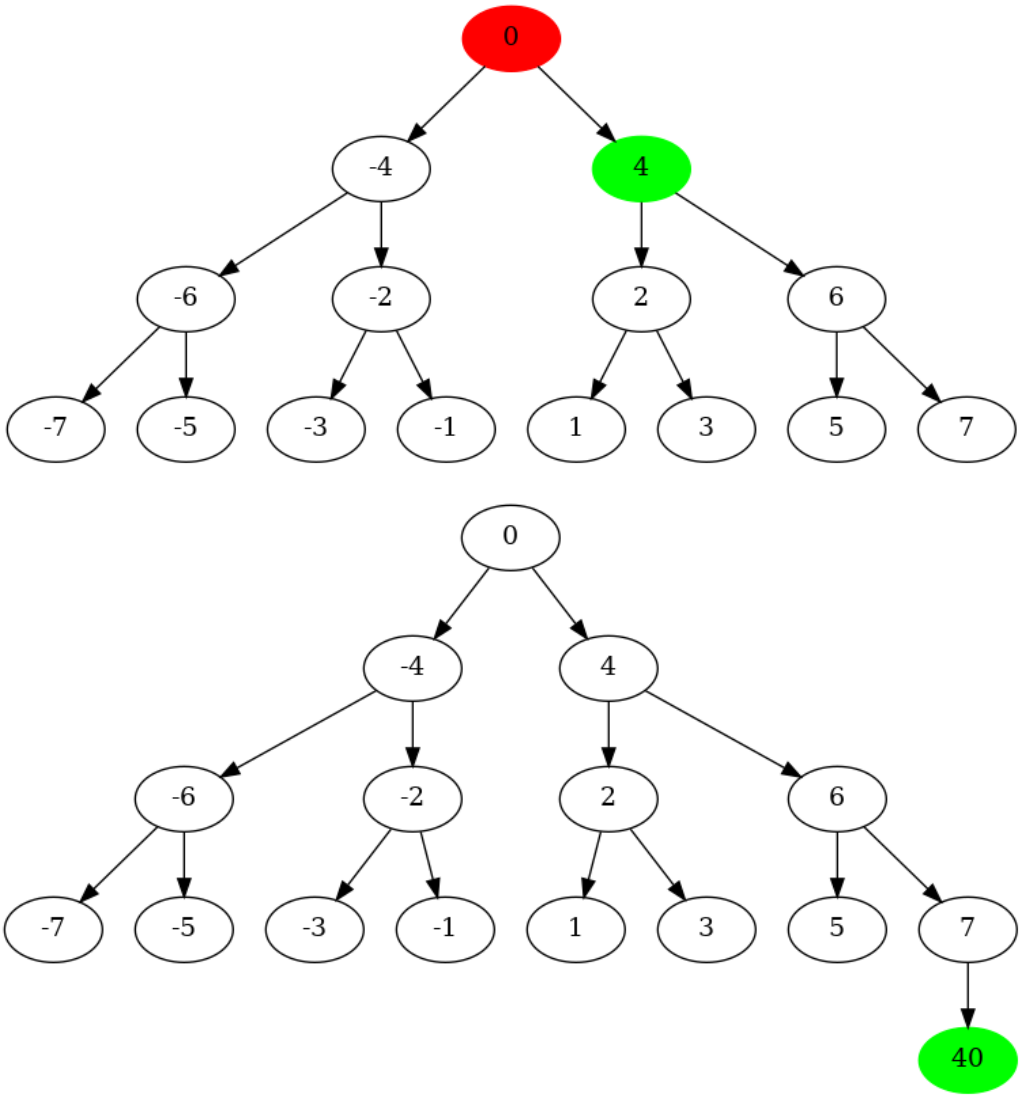
En effet, j'ai souhaité implémenter la même structure que celle étudiée en cours, à savoir :
- Une clé (key) pour chaque nœud
- Un tableau contenant les sous-arbres gauches (left)
- Un tableau contenant les sous-arbres droits (right)

Il est important de noter que Btree.left & Btree.right correspondent aux sous-arbres de la clé visitée.

6.3.1 Insertion

L'objectif de l'insertion est d'ajouter un nœud à l'arbre binaire de recherche. Pour ce faire, on parcourt l'arbre de la manière suivante :
- Si le nœud que l'on souhaite insérer est inférieur ou égal au nœud actuel, on se dirige vers la partie gauche de l'arbre.
- Sinon, on se dirige vers la droite.
- Si l'un des sous-arbres que l'on souhaite accéder est vide, cela signifie que c'est à cet emplacement que l'on doit insérer le nouveau nœud.

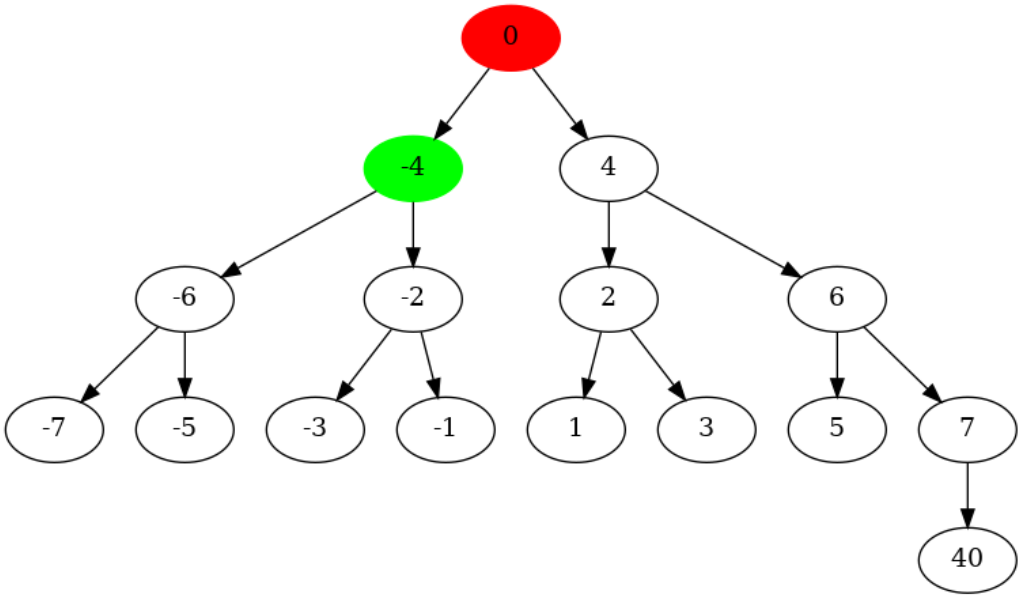
Voici un exemple d'insertion sur l'interface graphique. Nous souhaitons ajouter 40. La première image vous montre l'arbre d'origine et la deuxième vous montre l'arbre avec la valeur 40 ajoutée.

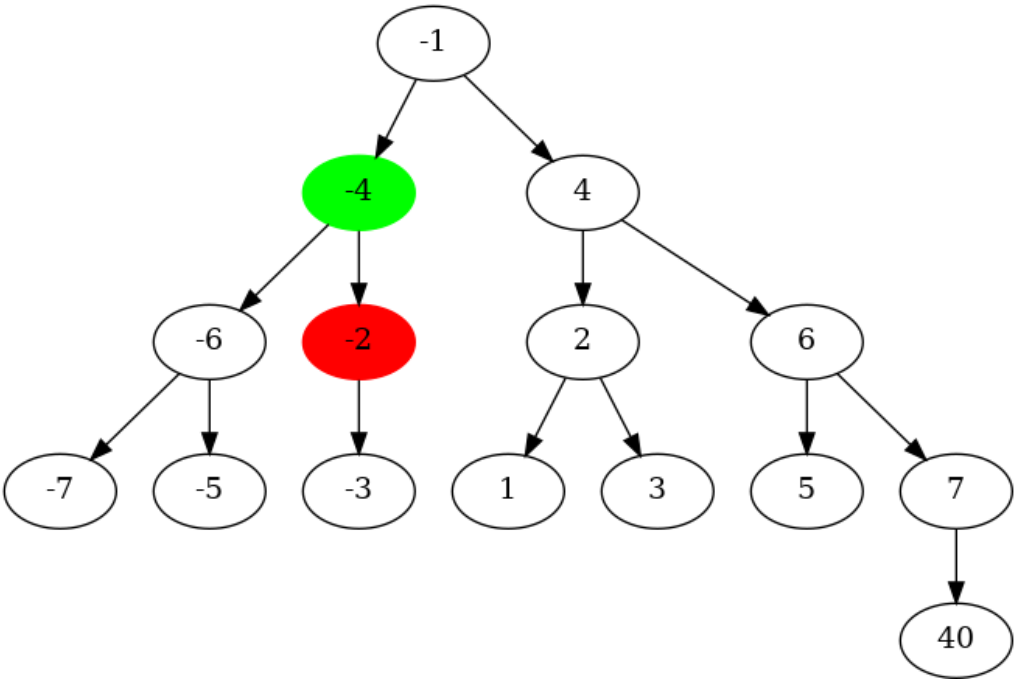


6.3.2 Suppression

L'objectif de la suppression est de supprimer un nœud dans un arbre binaire de recherche. La différence entre l'insertion et la suppression réside dans le cas de l'élément que l'on souhaite supprimer :
- Si c'est une feuille : on applique le même principe que pour l'insertion dans le sens de la recherche du nœud.
- Si ce n'est pas une feuille : on recherche le nœud. Une fois trouvé et lorsqu'on constate que ce n'est pas une feuille, on cherche le maximum du sous-arbre gauche et on l'échange avec l'élément que l'on souhaite supprimer. Ensuite, on le supprime comme une feuille.

Voici un exemple de supression sur l'interface graphique. Nous souhaitons enlever la racine soit 0. La première image vous montre l'arbre d'origine et le second vous montre l'arbre avec la valeur 0 supprimée.





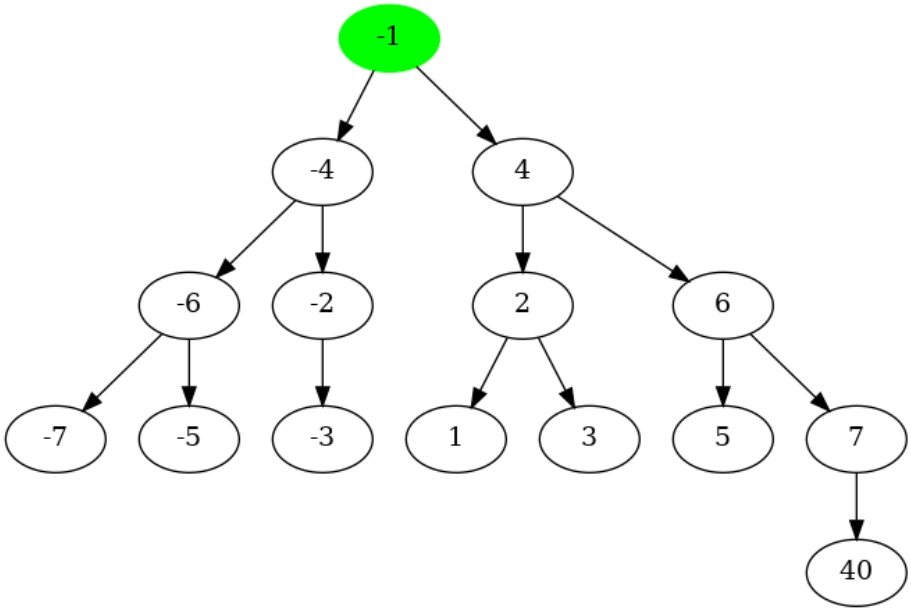
6.3.3 Parcours en profondeur

Pour afficher l'arbre après une insertion ou une suppression, nous avons actuellement la possibilité d'utiliser un parcours en largeur ou en profondeur.

Le parcours en profondeur consiste à lire chaque nœud en suivant une trajectoire du plus haut vers le plus bas, de gauche à droite. En d'autres termes, chaque nœud est noté dès sa première visite. Ce que je vous ai décrit correspond à ce que l'on appelle un parcours en profondeur de type préfixe. Il existe également d'autres types de parcours en profondeur : suffixe et infixe. Vous avez les 3 versions à votre disposition : à vous d'observez leur fonctionnalités.

Voici un exemple de parcours profondeur préfix sur l'interface graphique.

Le parcours se fera de la manière suivante : on parcours toute la branche gauche, ce qui nous donne : -1, -4, -6, -7, -5, -2, -3.

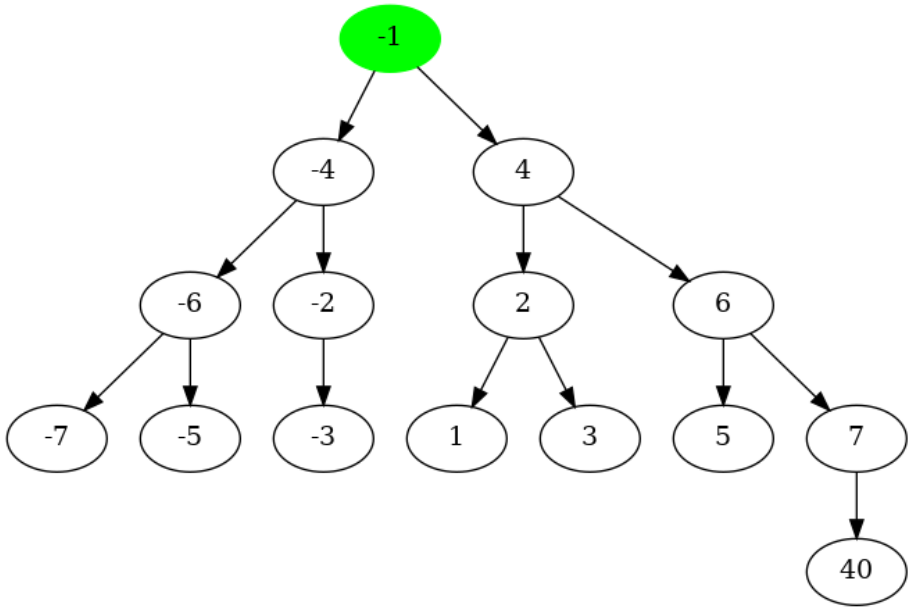


Ensuite, on fait la même chose à droite et on obtient : -1, -4, -6, -7, -5, -2, -3, 4, 2, 1, 3, 6, 5, 7, 40.

6.3.4 Parcours en largeur

Le parcours en largeur est une méthode qui consiste à explorer chaque niveau de l'arbre de manière progressive, de gauche à droite. Pour mieux comprendre cette approche, imaginez-vous lire un texte en français : vous le parcourez de gauche à droite, ligne par ligne. De la même manière, lors du parcours en largeur, nous notons chaque nœud de gauche à droite, niveau par niveau, comme si nous parcourions l'arbre de manière horizontale et ordonnée.

Voici un exemple de parcours largeur sur l'interface graphique.



En explorant par niveau, on obtient la séquence suivante : -1, -4, 4, -6, -2, 2, 6, -7, -5, -3, 1, 3, 5, 7, 40.

6.4 BFS, DFS (graph)

Le BFS est utilisé pour explorer de manière rapide, un graphe dans la largeur, niveau par niveau. Chaque itération augmentera la distance par rapport au nœud de départ. Son objectif est de déterminer rapidement s'il existe un chemin entre deux sommets. Il peut aussi définir le nombre de niveaux séparant deux sommets. Cet algorithme va ainsi explorer, niveau par niveau dans tous les chemins simultanément et non se concentrer chemin par chemin comme le DFS. Son fonctionnement se base sur une structure de données nommée « la queue ».

Le principe du DFS est d'établir du retour sur trace (backtracking) ou de faire des recherches complètes. Pour chaque chemin, il va explorer méticuleusement le chemin actuel. Puis, il va passer au chemin suivant pour être sûr d'avoir exploré toutes les possibilités de chaque chemin. Son fonctionnement se base sur une structure de données : la stack.

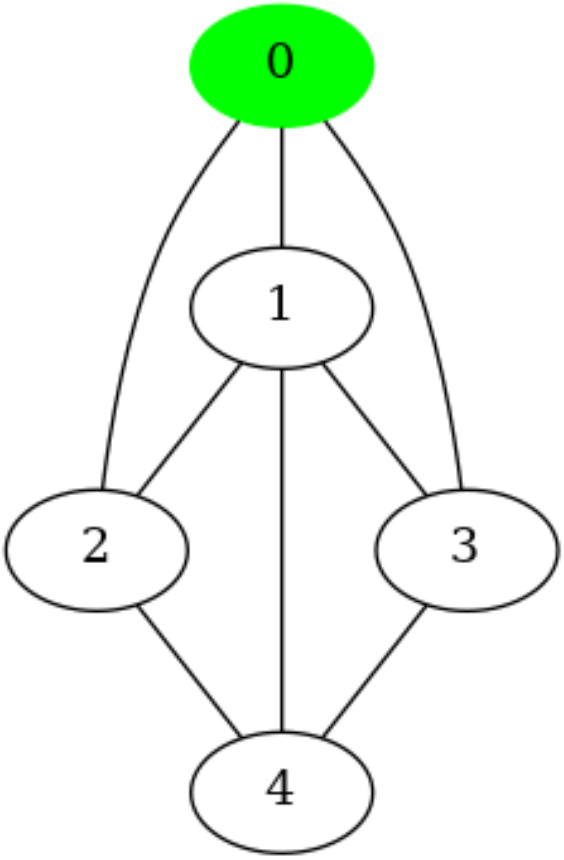
6.4.1 BFS

Afin de faire le parcours BFS, il fallait au préalable, se renseigner sur les méthodes disponibles. Fort heureusement, nous pouvons utiliser les queues comme en python grâce à la méthode suivante : VecDeqeque issue de la bibliothèque std : collections. Ensuite, il faut initialiser un graph pour utiliser le parcours. Le graph est une structure, comprenant des sommets et des arrêtes. J'ai utilisé une liste d'adjacence. Puis, j'ai une fonction addeges me permettant de tester ma fonction. Vient le codage du BFS.

Cette fonction prend deux paramètres : un graph et une source qui est la racine de notre graph. Comme en python, nous commençons par initialiser une queue et un vecteur. Dès lors, nous ajoutons dans notre queue la racine. A partir du moment où la queue n'est pas vide, on ajoute le sommet dans le vec final et on parcourt le graph en regardant si il possède des fils afin de les insérer (de gauche à droite) dans la queue. Et finalement, on reboucle jusqu'à parcourir le graph en entier. Dans l'ultime étape, nous retournons la liste de vecteurs, correspondant aux sommets parcourus en largeur.

Voici un exemple de parcours largeur d'un graph non orienté sur l'interface graphique.

En partant du sommet 0, on aura une séquence suivante : 0, 1, 2, 3 et 4.

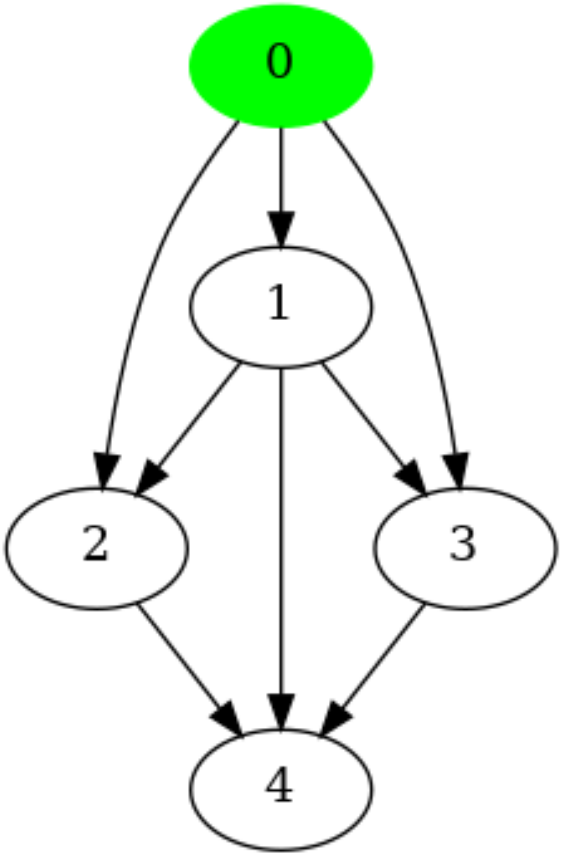


6.4.2 DFS

Afin de faire le parcours DFS, nous avons du nous renseigner sur les méthodes disponibles. Comme ce dernier n'a pas besoin de queue, son implémentation est plus simple. Il a fallu initialiser un graph pour utiliser le parcours. Le graph est une structure, comprenant des sommets et des arrêtes. J'ai utilisé une liste d'adjacence. Ensuite, j'ai une fonction addeges me permettant de tester ma fonction. Vient le codage du DFS.

Le DFS prend en paramètre un graph, une source et un Hashset permettant d'ajouter les sommets. Comme en python, on commence par insérer la source dans le Hashset. Puis, on initialise un vec contenant notre sommet source. Par la suite, nous regardons les fils possédés par notre racine source et dans le cas ou le Hashset ne contient pas ses fils, nous l'ajoutons dans le vec via une récursivité du fils et du Hashset. Pour finir, je retourne le vec, contenant tous les sommets du graph, parcouru en profondeur.

Voici un exemple de parcours profondeur d'un graph orienté sur l'interface graphique.



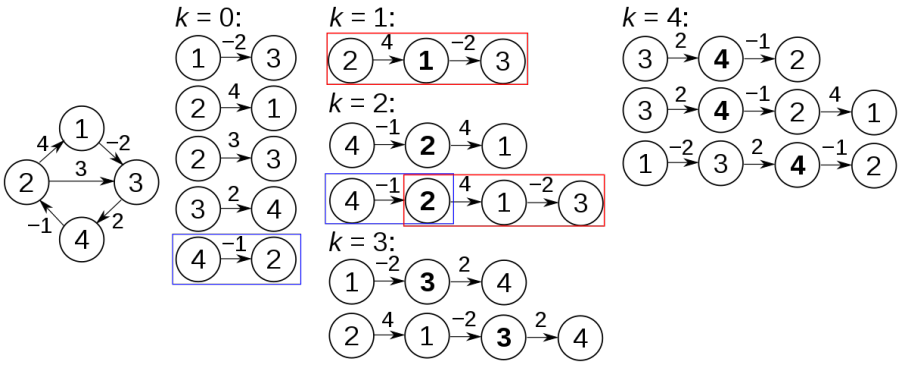
En partant du sommet 0, on aura la séquence suivante : 0, 1, 2, 4 et 3.

6.5 Algorithme Floyd-Warshall

L'algorithme de Floyd-Warshall est un algorithme de recherche de plus courts chemins dans un graphe pondéré, qu'il soit dirigé ou non dirigé, avec des poids positifs ou négatifs. Cette méthode utilise une approche dynamique pour calculer les distances les plus courtes entre tous les paires de nœuds dans le graphe.

L'idée principale réside dans la mise à jour itérative des distances entre les nœuds en explorant toutes les possibilités de chemins. L'algorithme maintient une matrice des distances entre les nœuds, ajustant continuellement les valeurs en considérant tous les nœuds intermédiaires possibles. La complexité temporelle de l'algorithme de Floyd-Warshall est relativement élevée, en O(V puissance 3), où V est le nombre de nœuds dans le graphe. Cependant, il a l'avantage de traiter efficacement les graphes avec des poids négatifs, contrairement à d'autres algorithmes.

En dépit de sa complexité, l'algorithme de Floyd-Warshall offre une solution complète et pratique pour trouver les plus courts chemins entre tous les points d'un graphe, faisant de lui un outil précieux dans divers domaines, tels que la planification de réseaux et l'optimisation des itinéraires.

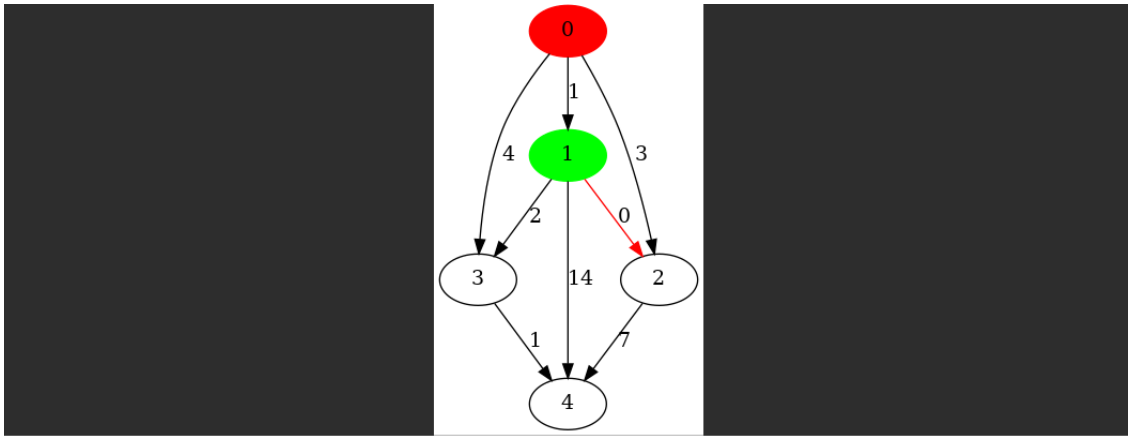


Son fonctionnement est basé sur la notion de programmation dynamique. Il maintient une matrice des distances minimales entre chaque paire de sommets. Initialement, cette matrice est remplie avec les poids des arêtes du graphe. Ensuite, l'algorithme explore toutes les paires de sommets (i, j) et vérifie si passer par un sommet intermédiaire k réduit la distance entre i et j. Si c'est le cas, la distance minimale entre i et j est mise à jour en utilisant la distance minimale entre i et k ajoutée à la distance minimale entre k et j. Cette mise à jour est répétée jusqu'à ce que toutes les paires de sommets aient été explorées et que la matrice des distances minimales soit stabilisée.

En fin de compte, la matrice résultante contient les distances les plus courtes entre tous les paires de sommets du graphe.

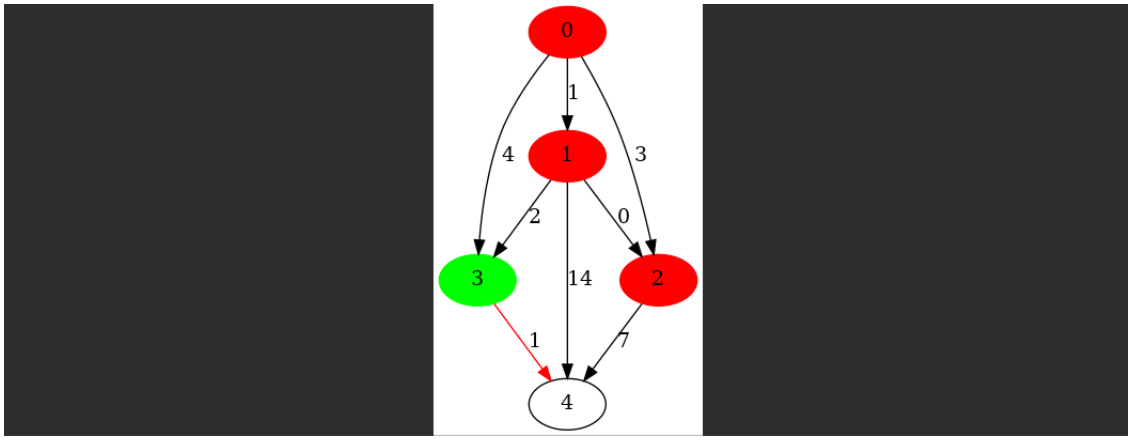
Voici un exemple d'éxécution de l'algorithme de Floyd-Warshall sur l'interface graphique.

Cette première photo représente le graph d'origine avec les initialisations nécessaires car l'algorithme de Floyd-Warshall cherche le plus court chemin d'un sommet vers tous les autres. C'est la raison pour laquelle, seulement une valeur est en infini contrairement aux autres algorithmes.



dist : [0, 1, 1, 4, inf]
p : [-1, 0, 1, 0, -2]

Cette deuxième image vous montre le résultat de cet algorithme sur ce graph, en renvoyant les vecteurs distances et prédécesseurs.

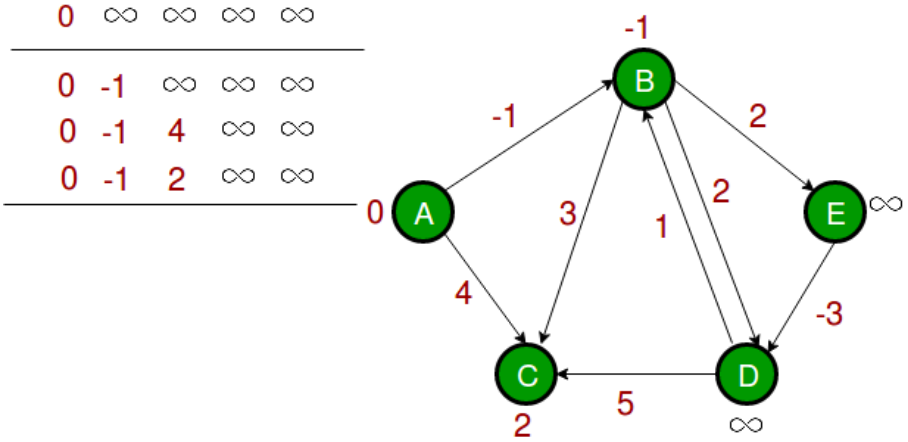


dist : [0, 1, 1, 3, 4]
p : [-1, 0, 1, 1, 3]

6.6 Algorithme de Ford-Bellman

L'Algorithme de Ford-Bellman est un algorithme de Maximum Flow. L'objectif de cet algorithme est de calculer la capacité de transfert entre deux nœuds d'un graphe. Pour le réaliser, je vais utiliser deux graphes et un dfs. Les deux graphes représentent un graphe sans ou avec les couts lié aux arrêtés.

Nous allons modifier le graphe avec les couts nul pour essayer d'atteindre le maximum possible avec un dfs. Il a une complexité de O(sommet * arcs) dans le pires des cas et O(arcs) dans le meilleur cas.



L'algorithme de Bellman-Ford est utilisé pour calculer les chemins de coût minimal dans les graphs. Il est nommé d'après Richard E. Bellman et Lester Ford, Jr. L'algorithme fonctionne en initialisant le coût de tous les sommets à l'infini, à l'exception du sommet source, qui est initialisé à 0.

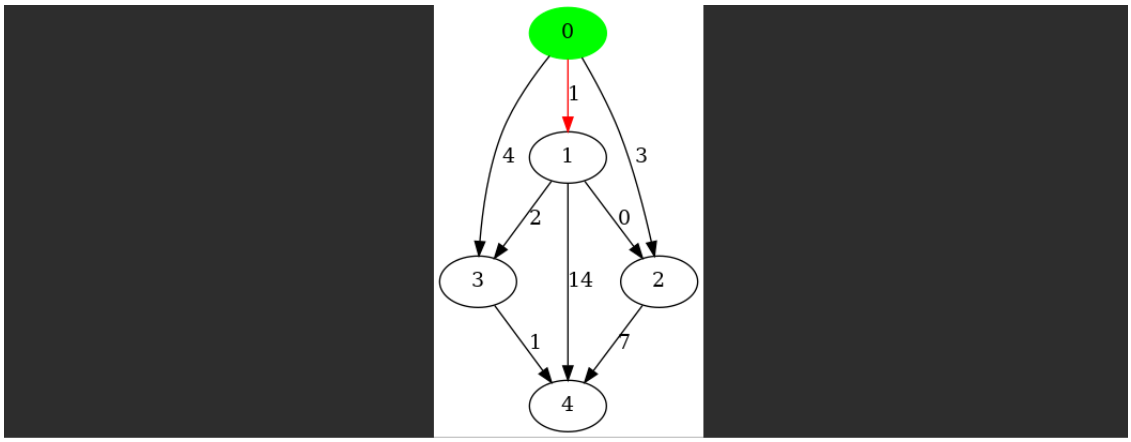
Ensuite, il effectue V-1 itérations, où V est le nombre de sommets dans le graph. Dans chaque itération, l'algorithme examine chaque arête (u, v) et met à jour le coût du sommet v si le coût du chemin de u à v via cette arête est inférieur au coût actuel du chemin de u à v. Si une mise à jour se produit pendant la dernière itération, cela indique qu'il existe un chemin négatif dans le graph, ce qui est impossible dans un graph sans cycle négatif.

Pour l'algorithme de Ford-Belleman, j'ai passé du temps pour cree un system de clock. Ceci m'a permis de realiser un system de clock pour de l'algorithme et d'en réaliser une seconde version. Pour le moment, mon programme prend le graph et un histogramme pour les coûts. Il rajoute les coûts dans le second graph pour simuler son utilisation.

Nous allons le réaliser pour tous les arcs en faisant un parcours DFS qui a pour objectif de parcourir le graph. J'ai réalisé des modifications dans l'algorithme pour l'optimiser en arrêtant le code si aucune valeur dans l'histogramme n'a été changée ce qui permet de faire moins V-1 parcours.

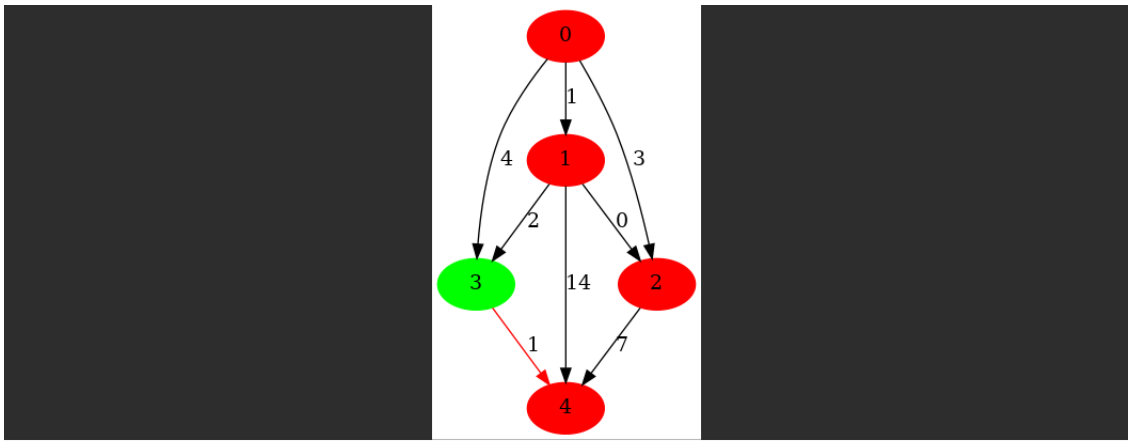
Voici un exemple d'éxécution de Bellman-Ford sur l'interface graphique.

Cette première photo montre le graph d'origine avant d'utiliser Bellman-Ford.



dist : [0, 1, inf, inf, inf]
p : [-1, 0, -2, -2, -2]

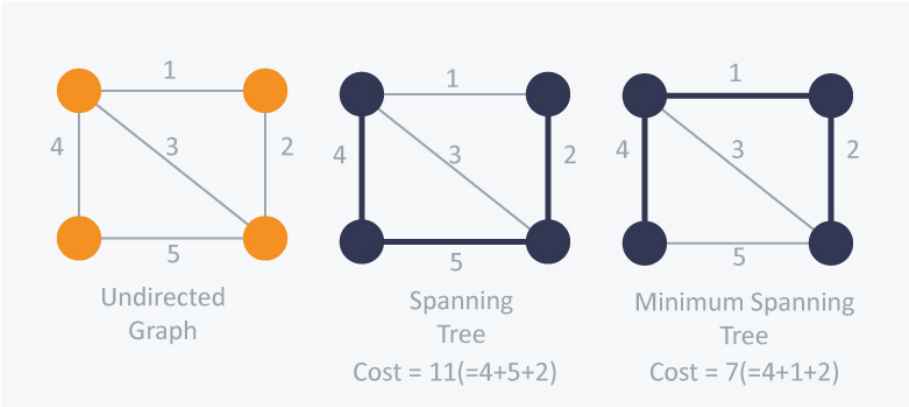
Cette deuxième photo montre le graph traité avec Bellman-Ford, avec l'obtention des vecteurs distances et prédécesseurs.



dist : [0, 1, 1, 3, 4]
p : [-1, 0, 1, 1, 3]

6.7 Algorithme Spanning Trees minimum

L'Algorithme Spanning Trees minimum est une famille d'algorithmes, nous allons nous utiliser celui de Prim's. Ce type d'algorithme est utilisé pour trouver le plus petit spanning Trees d'un graph. Il a une complexité de O((sommet * arcs)*sommet) dans le pire des cas et O(sommet) dans le meilleur cas.

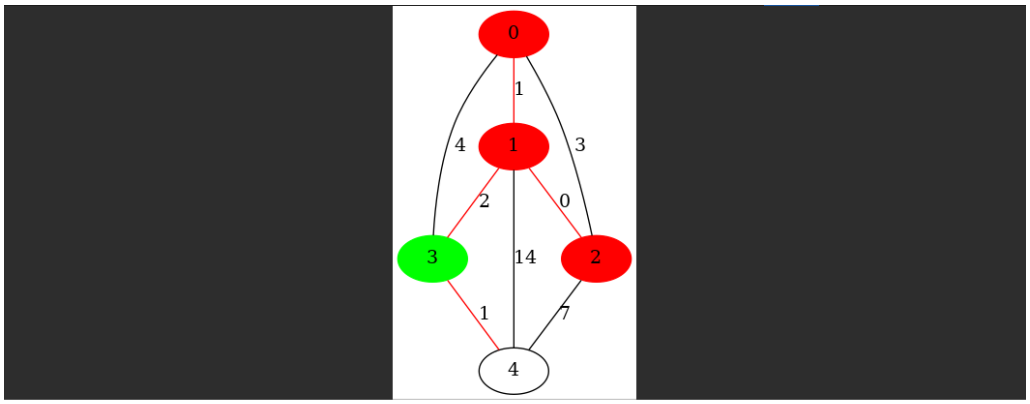


Pour l'algorithme de prim, j'ai réalisé de nombreuses recherches pour me permettre de comprendre le fonctionnement de cet algorithme. Suite aux différentes vidéos que j'ai visionnées, j'ai pu vous établir cette description. L'algorithme commence sur une arête du graph et l'inclut dans l'arbre sous- arbre que j'ai créé.

Puis, à chaque étape, il ajoute une arête supplémentaire au minimum de poids qui relie un sommet de l'arbre à un sommet non inclus dans ce sous graph. Le processus se poursuit jusqu'à ce que tous les sommets soient inclus dans l'arbre. Après avoir parcouru de nombreux sites, j'ai effectué une seconde version du programme qui a pour objectif de vérifier ma compréhension de cet algorithme et de passer du temps pour comprendre comment transformer l'algorithme pour qu'il soit fonctionnel en exécutant le code en plusieurs fois pour nous permettre d'avoir une animation.

Pour cette nouvelle version du programme, J'ai réalisé un système de clock. Ce système utilise un ensemble de fonctions permettant d'appeler une fonction réalisant une itération par une itération.

Voici un exemple d'application de l'algorithme de Prim sur l'interface graphique.

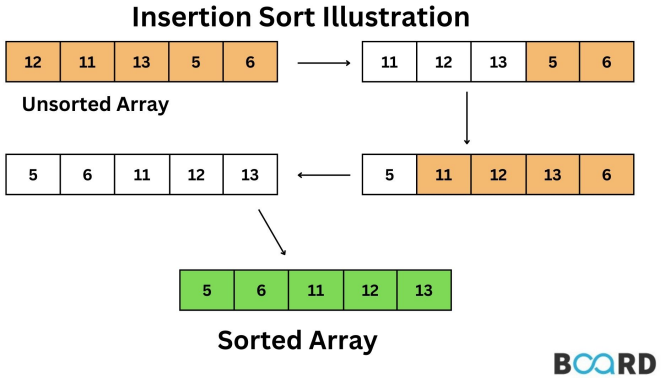


dist : [0, 1, 0, 2, 1]
p : [-1, 0, 1, 1, 3]

Comme vous pouvez le voir, nous avons un graph non orienté avec coûts. Après l'application de l'algorithme sur ce graph, nous obtenons un arbre de recouvremment minimal représenté par les arcs rouges. Nous avons ainsi le poids minimum qui est de 4. Nous obtenons aussi les vecteurs distances et prédécesseurs, car cet algo fonctionne comme Dijkstra.

6.8 Insert sort

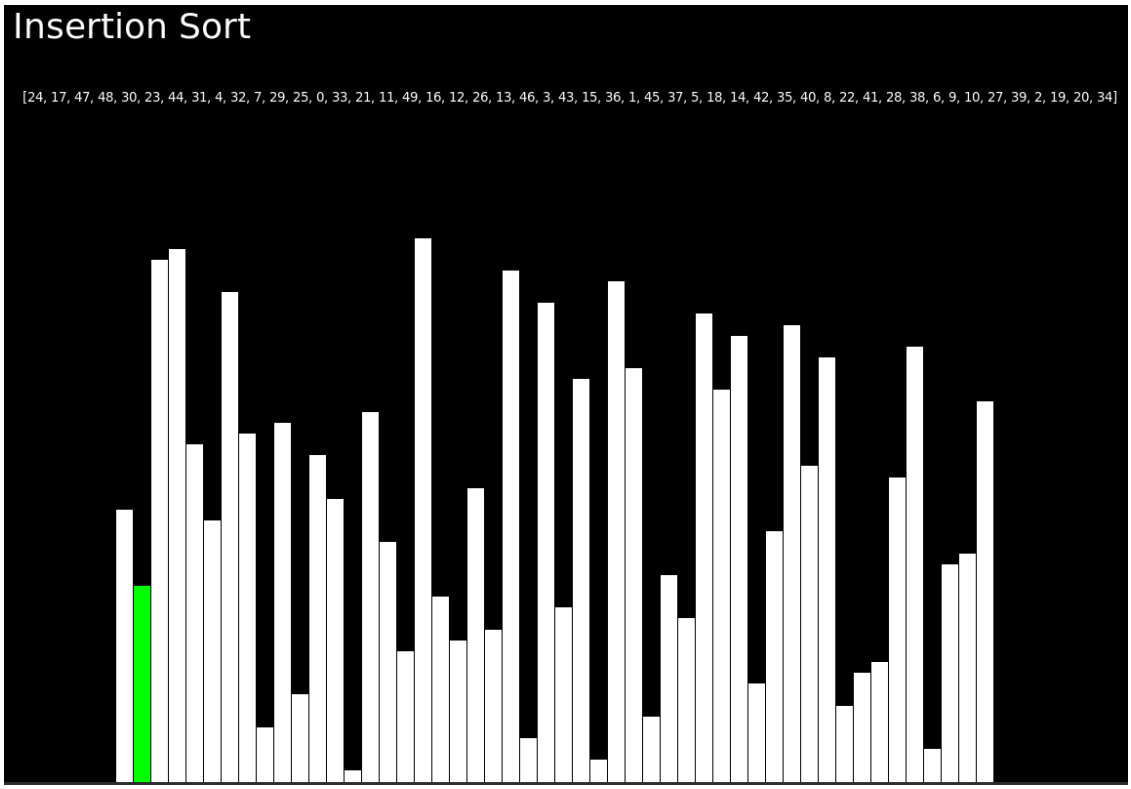
Le tri par insertion est l'un des premiers algorithme de tri vu en classe. il est utilisé pour trier les tableaux et est efficace pour trier les tableaux de petite taille ou des tableaux presque trié, le meilleur cas a une complexité de O(n), sa complexité moyenne est de $O(n^2)$ et le pire cas est de $O(n^2)$.



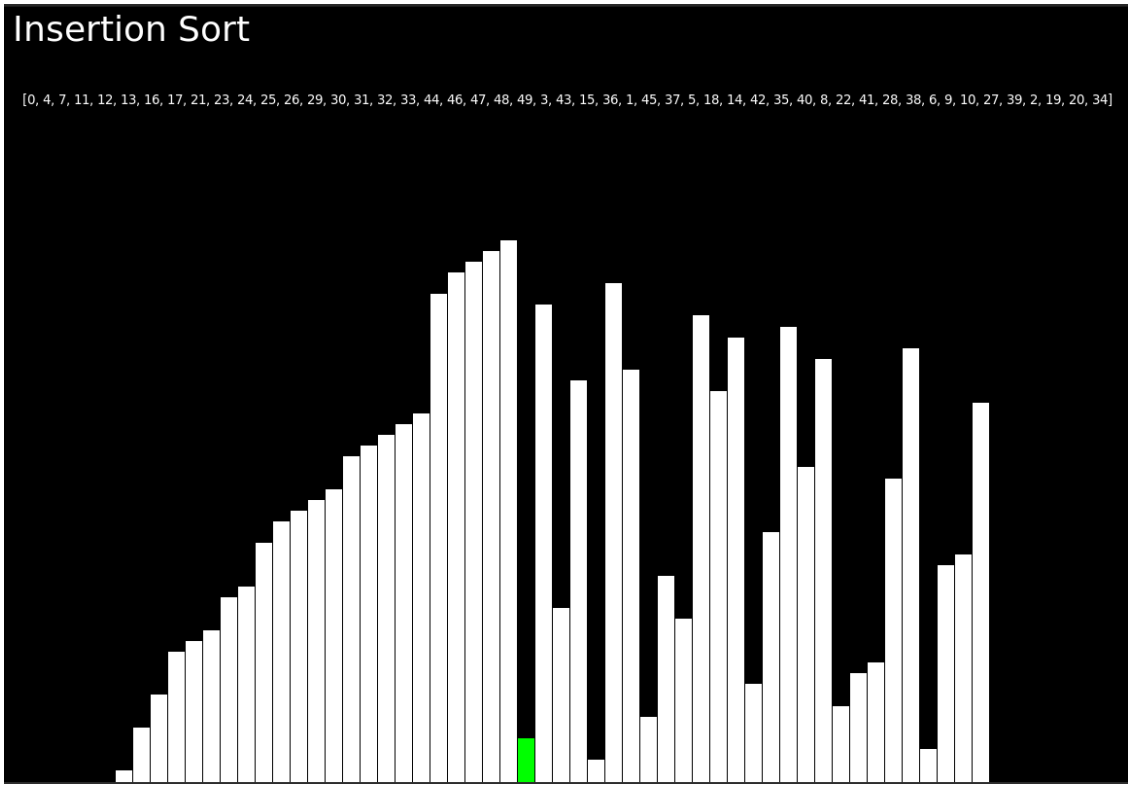
Son principe est très simple. On parcours tout le tableaux, et durant ce parcours ,si on tombe sur une valeur qui est mal placée dans le tableau, on décale cette valeur a gauche jusqu'à ce que le tableau soit triee entre la première valeur et là où etait la valeur.

Voici un exemple d'application de cet algorithme sur l'interface graphique.

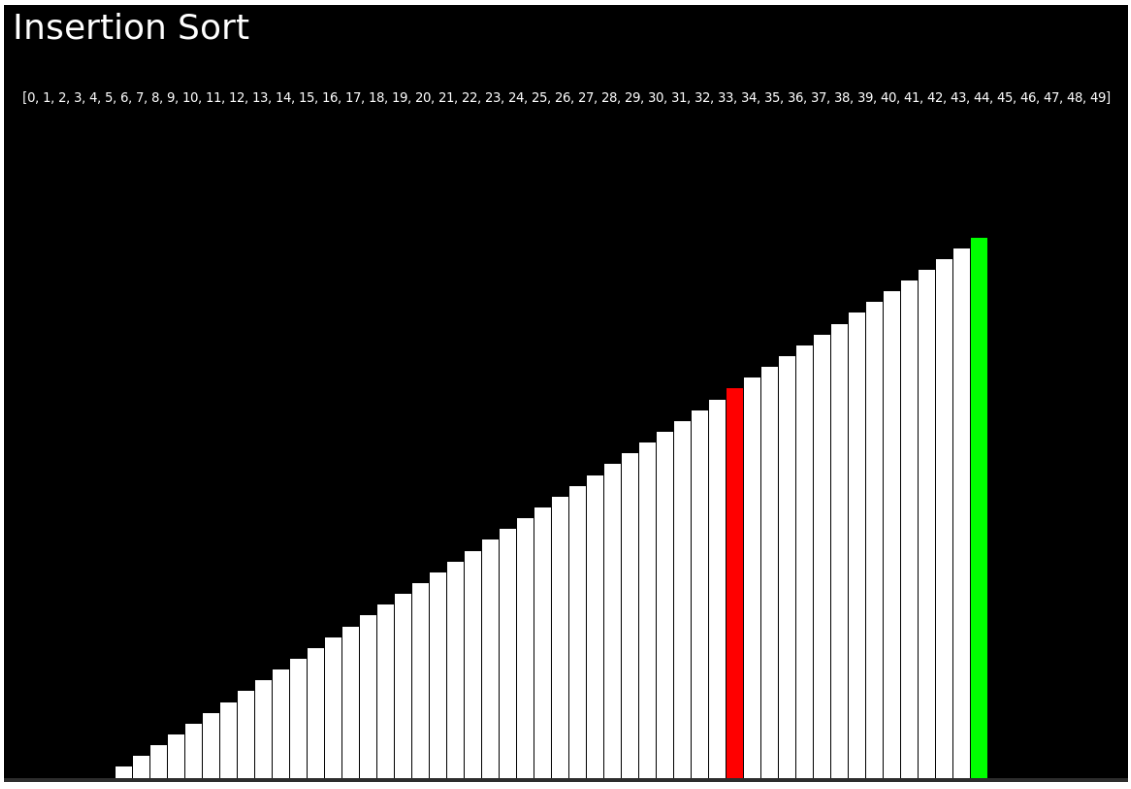
Cette première photo vous montre la liste d'origine.



Cette seconde image vous montre que chaque valeur est traitée au fur et à mesure que l'on progresse dans la liste.

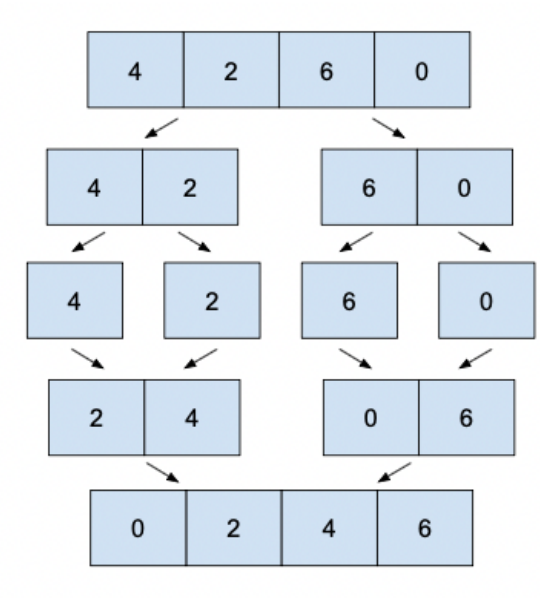


Puis, cette ultime image vous montre que la liste a bien été triée dans l'ordre croissant.



6.9 Merge sort

L'Algorithme de Merge sort a pour objectif de trier une liste. La difficulté pour cet algorithme est de créer un système qui va être capable de faire fonctionner chaque étape indépendamment puis les redécouper en sous étapes pour un affichage de la progression de l'algorithme visuel. Il a une complexité de O(nln(n)) dans le pire des cas et le meilleur des cas.

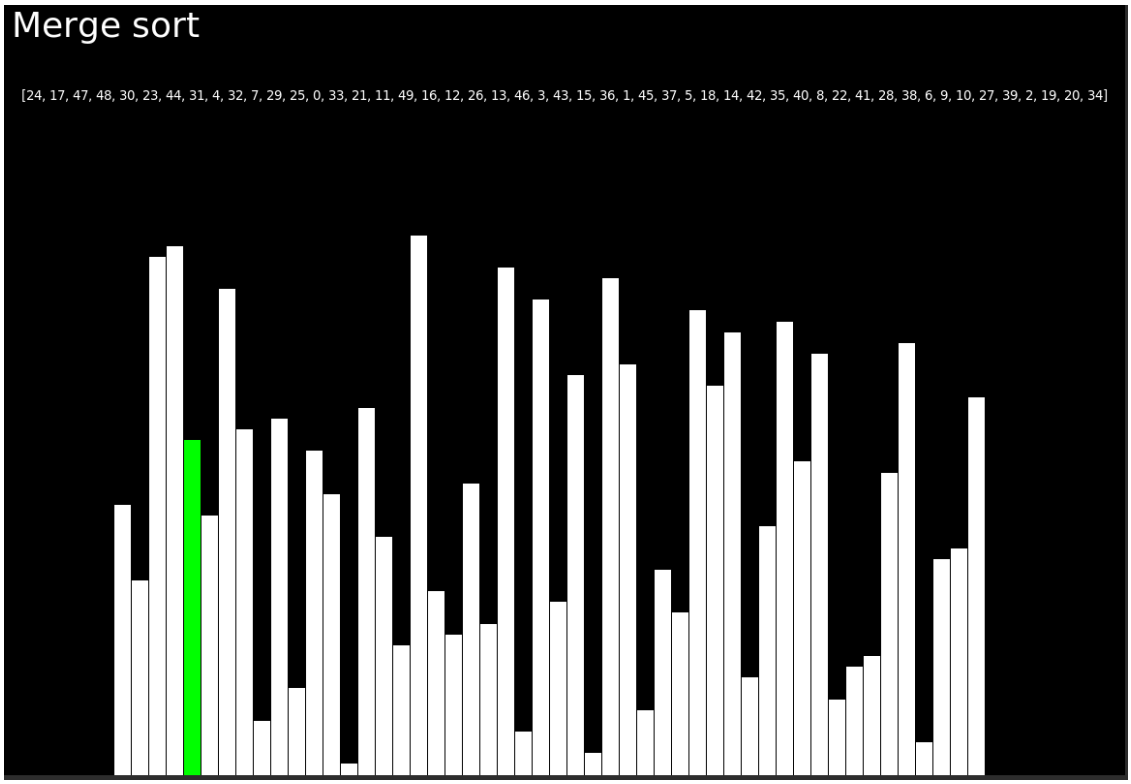


Pour l'algorithme du merge sort, j'ai effectué une seconde version qui se base sur le cours du S2. Cependant, pour le faire fonctionner avec des animations, j'ai découpé mon programme en différents sous programmes qui peuvent être appelés avec une liste et une liste de pointeurs. La version que j'ai créé utilise une liste de pointeurs comme marquer de différentes sous listes, après une autre fonction utiliser ses et la liste pour fusionner la sous-liste 2 pars deux pour réaliser le merge sort.

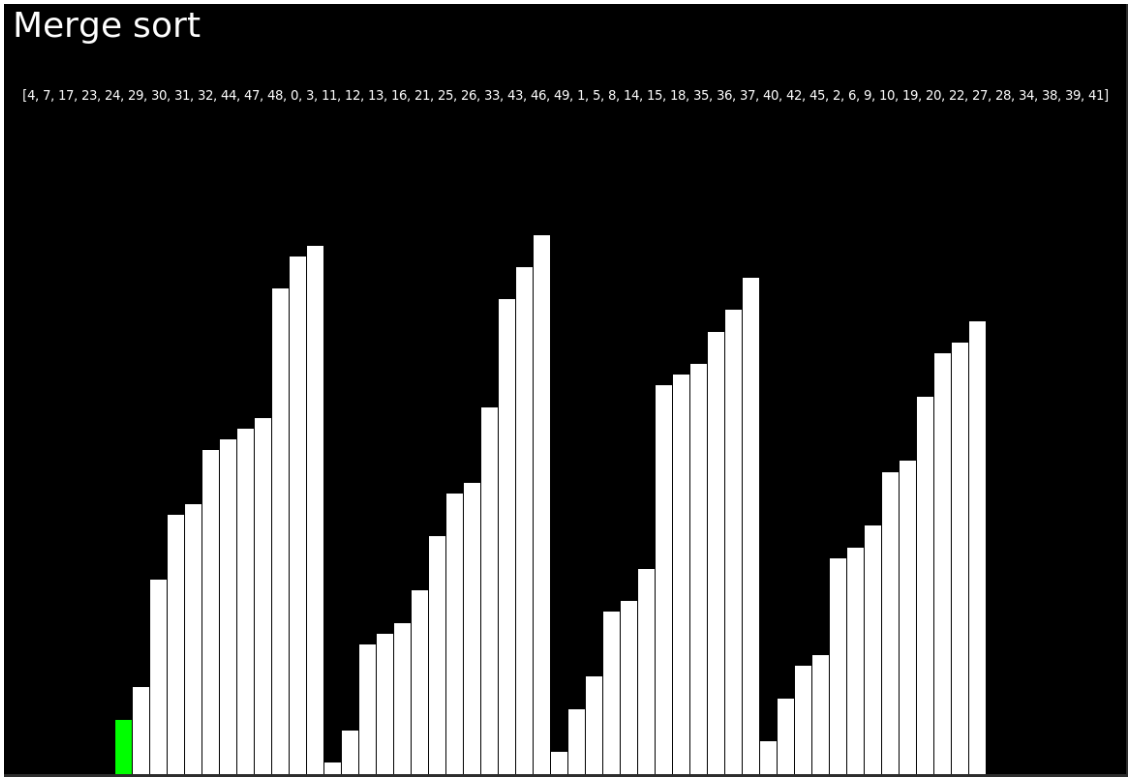
Cependant j'ai trouvé un certain nombre de problèmes pour réaliser une modification en place donc j'utilise une sous liste. L'une des solutions qui pourrait rendre mon algorithme plus efficace serait avec l'utilisation de liste chainé, mais qui ne sera pas le cas pour ce projet S4.

Voici un exemple d'application de cet algorithme sur l'interface graphique.

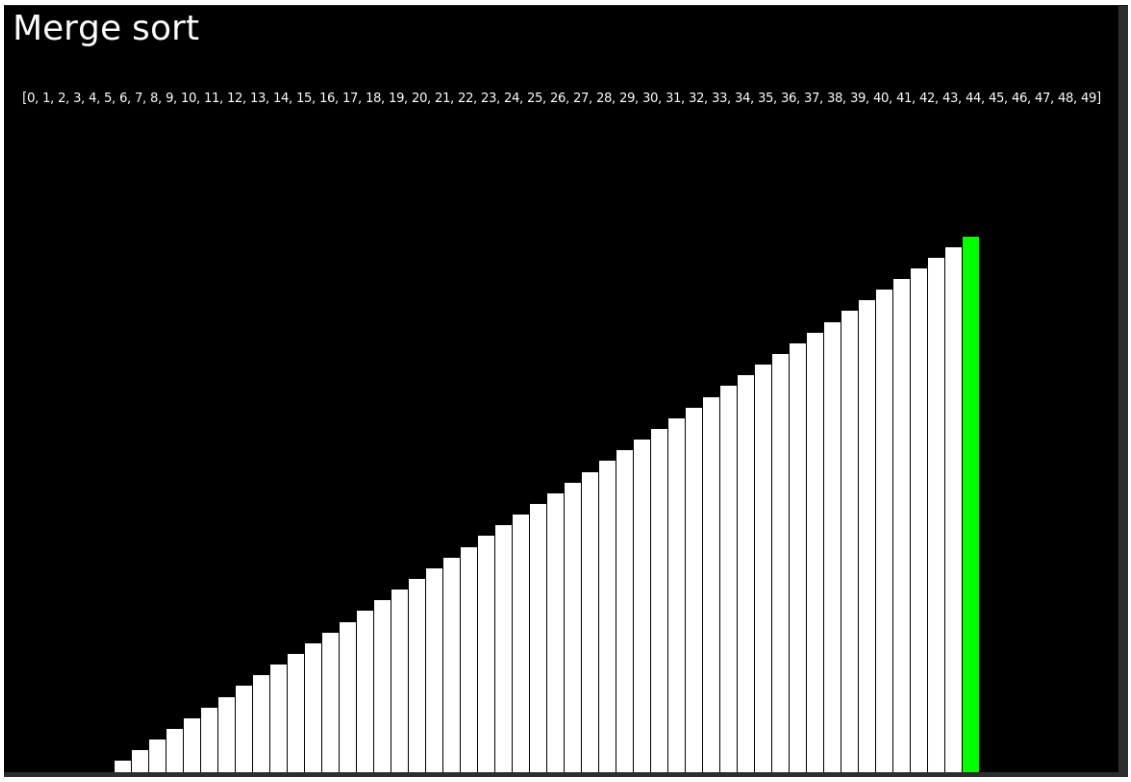
Cette première photo vous montre la liste d'origine.



Cette seconde image vous montre que le merge sort trie la liste en la divisant par 2 : on remarque donc que la première et la deuxième partie de la liste se trie.



Cette dernière image montre que la liste a été triée dans l'ordre croissant.

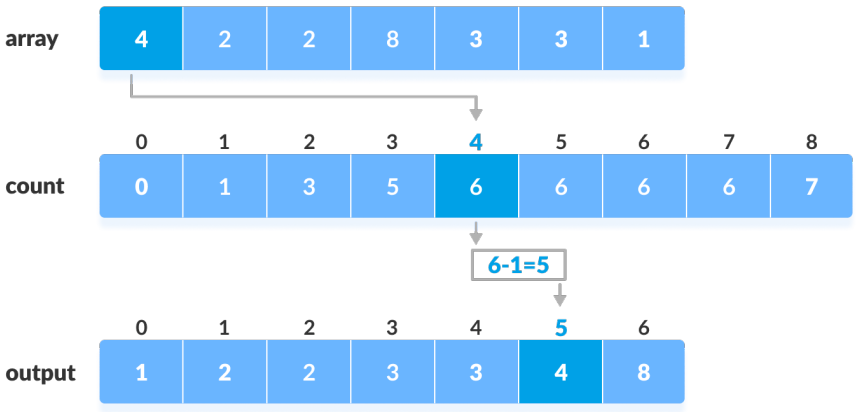


6.10 Counting sort

Le tri par dénombrement (counting sort) est l'un des algorithmes de tri le plus rapide, même s'il a quelques restrictions et défauts. Le tri s'exécute en un temps linéaire, mais uniquement sur des nombres entiers. La particularité du tri est qu'il est la base d'autres algorithmes de tri en temps linéaires, permettant de s'adapter aux besoins en temps et en mémoire.

Le principe de cet algorithme est le suivant : on parcourt notre tableau puis on compte le nombre de fois où chaque élément doit apparaître. Dès que l'on obtient le tableau T (T[i] correspond au nombre où i apparait dans le tableau), nous avons la possibilité de le parcourir dans le sens croissant ou décroissant, tout dépend tu tri effectué. On peut ainsi mettre dans le tableau trié T[i] fois l'élément i, allant du minimum jusqu'au maximum du tableau.

L'intérêt de cet algorithme est de voir si effectué, il est un algorithme très rapide par rapport a l'insert sort et au merge sort que l'on vous propose.



Cette fonction prend en paramètre un tableau mutable car on va modifier son contenu pour le trier dans l'ordre croissant et une valeur : l qui sera la valeur maximale de notre tableau. J'ai préféré mettre en paramètre cette valeur maximale plutôt que la calculée car cela facilite les tests et les modifications à apporter si besoin pour l'amélioration du code. Le tableau est modifié en place pour éviter de recréer un tableau supplémentaire où l'on insère nos valeurs finales. Ma première version de l'algorithme était comme cela mais j'ai voulu essayé en place. Cela n'a pas été simple car il y avait des changements de calculs mais j'ai réussi à le faire.

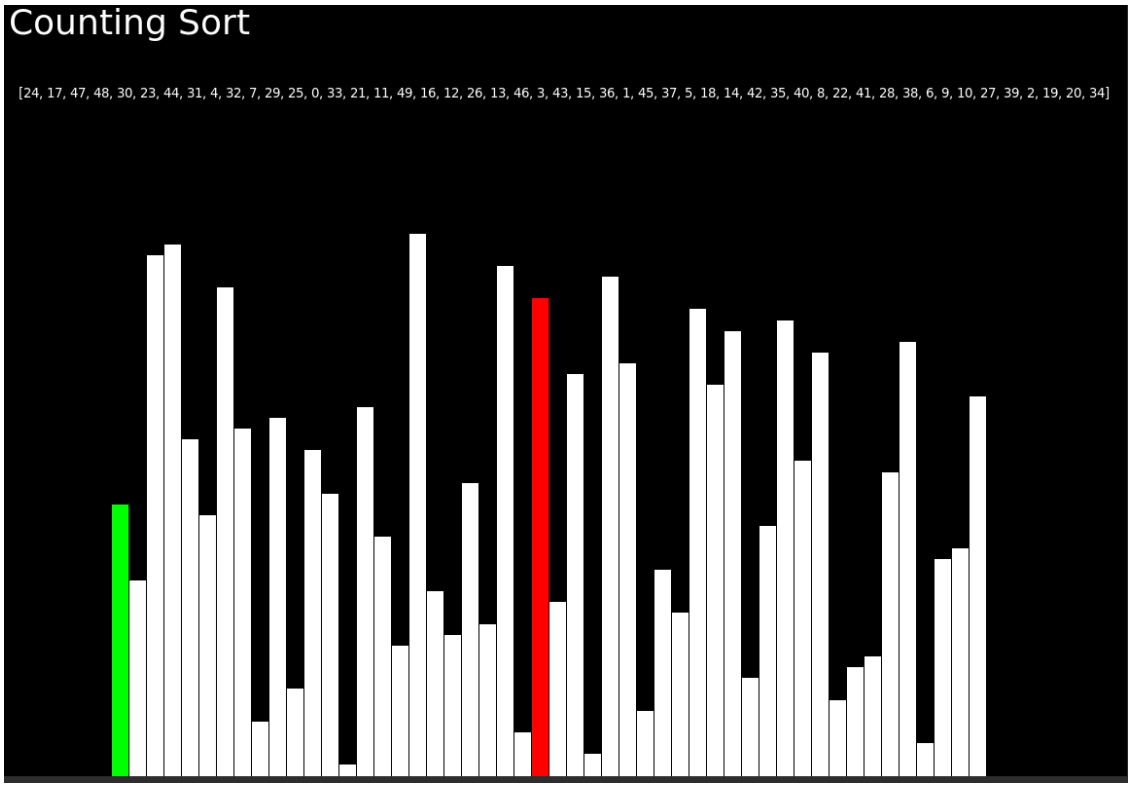
Je commence par créer un tableau (vec !) de type usize, de longueur valeur maximale du tableau + 1 contenant que des 0. Ce vec va permettre de stocker toutes les occurrences des différents éléments présents dans notre tableau d'origine. Vient une première boucle : celle-ci va parcourir notre tableau et stocker le nombre de chaque élément du tableau d'origine avec leur indice correspondant. Puis, une fois ces éléments en mains, grâce aux index proposés, de parcourir notre tableau d'occurrences et en fonction des résultats de notre première boucle, il est possible de trier notre tableau (si le 1er élément à 5 comme nb d'occurrence et le 2e 2, le 2e sera placé ainsi avant le 1er et ainsi de suite).

Après avoir codé ma fonction, il fallait naturellement la tester. J'ai donc fait une fonction main basique permettant de tester mon code. J'ai initialisé deux tableaux de longueurs et valeurs différentes puis j'ai testé.

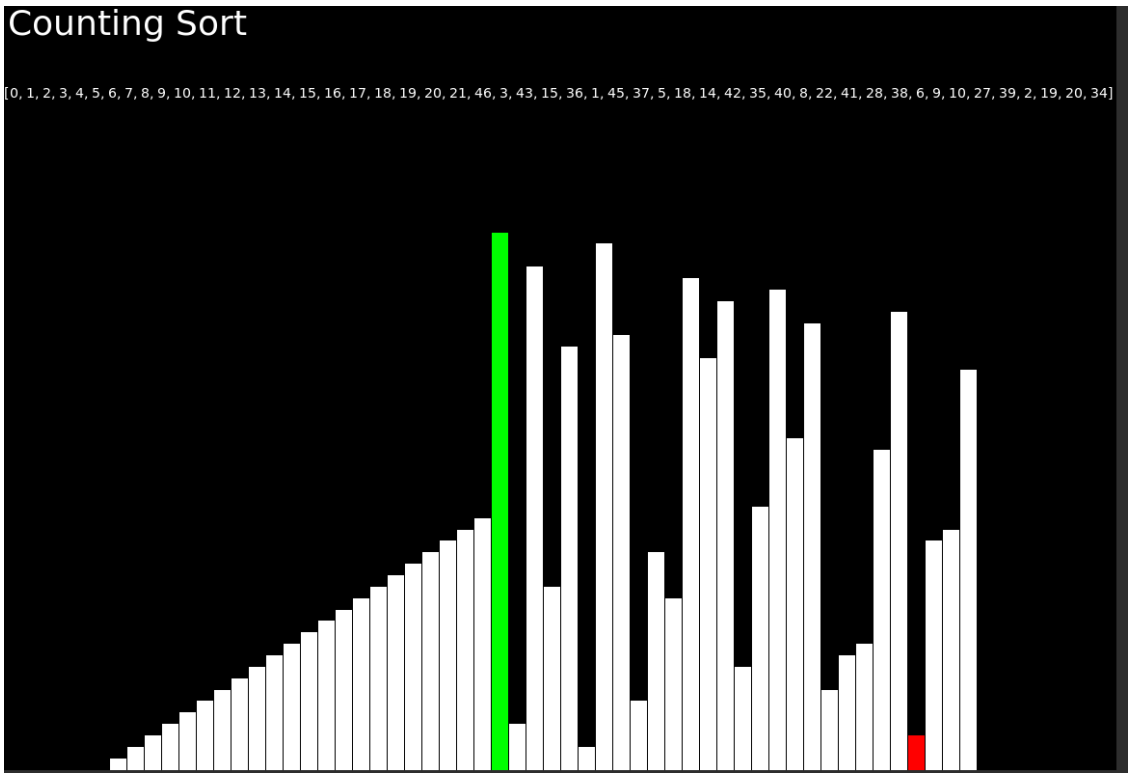
Le counting sort à un défaut majeur : il n'est possible de travailler qu'avec des nombres entiers positifs et qu'en temps linéaire. Par ailleurs, il est inefficace si la page de valeur à trier est très large. Toutefois, il fait partie des plus rapides algorithmes de tri basé sur des comparaisons. Quand nous donnons un tableau connu, nous avons la possibilité de se servir de cet algorithme pour avoir un temps d'exécution rapide et efficace. On pourrait modifier légèrement le fonctionnement de ce tri pour avoir une meilleur complexité en mémoire, en utilisant le Radix sort (tri par base), autre algorithme linéaire.

Voici un exemple d'application du Counting Sort sur l'interface graphique.

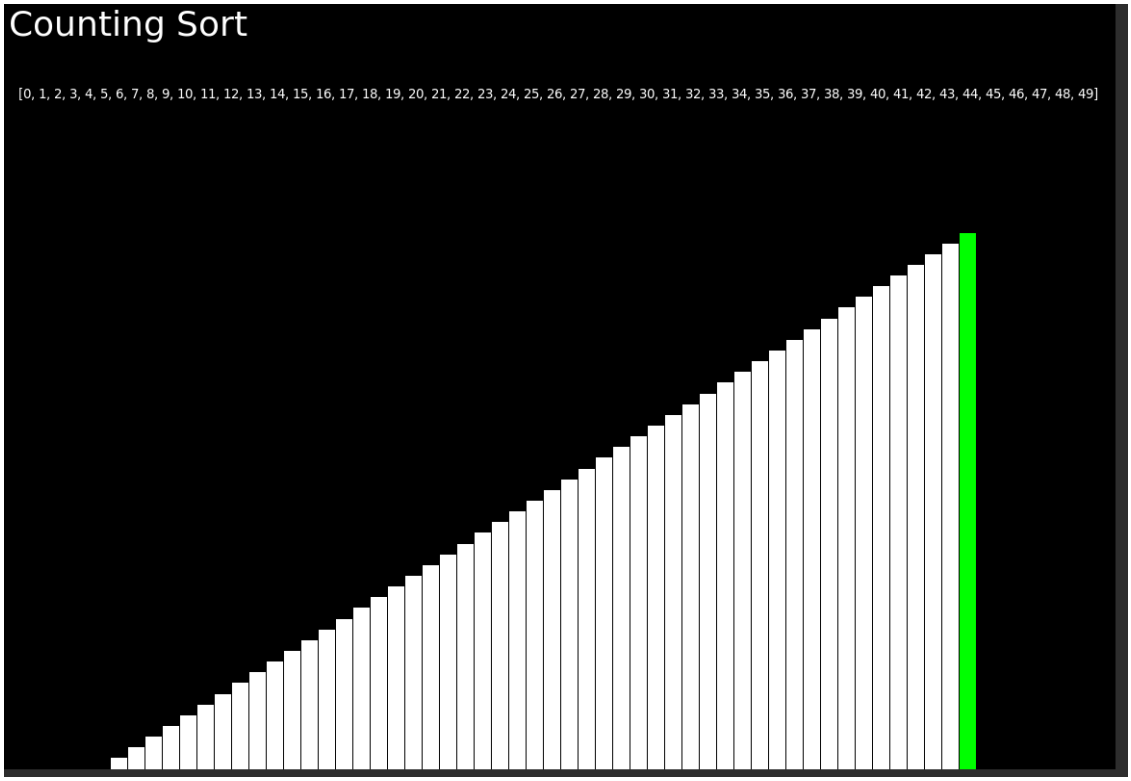
Cette première photo montre la liste dans son état d'origine.



Cette deuxième photo montre que la liste se trie au fur et à mesure que l'on avance dans liste en fonction du "comptage".



Cette dernière photo montre que la liste a bien été triée.



6.11 Interface graphique

L'interface graphique est implémentée avec gtk-rs 0.14.3, qui est une bibliothèque disponible sur crates.io. J'ai rencontré plusieurs problèmes concernant gtk : la documentation propose peu d'exemples, est incomplète, c'est à dire, qu'elle ne montre pas toutes les fonctions qui peuvent être utilisées sur un struct. De plus, à cause d'un manque de connaissance sur les struct comme Arc ou Mutex, il fut très compliqué d'avoir une variable qui est "globale" donc accessible partout, surtout avec les fonctions de gtk, où encore les problèmes d'emprunts.

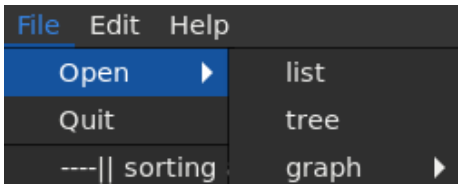
L'interface se décompose en plusieurs parties.

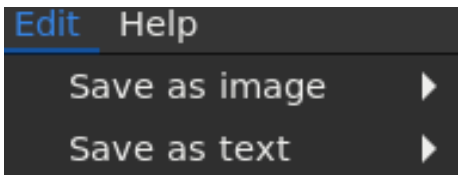
La partie haute c'est à dire la barre de Menu et la partie basse qui représente celle qui contient la partie la plus importante.

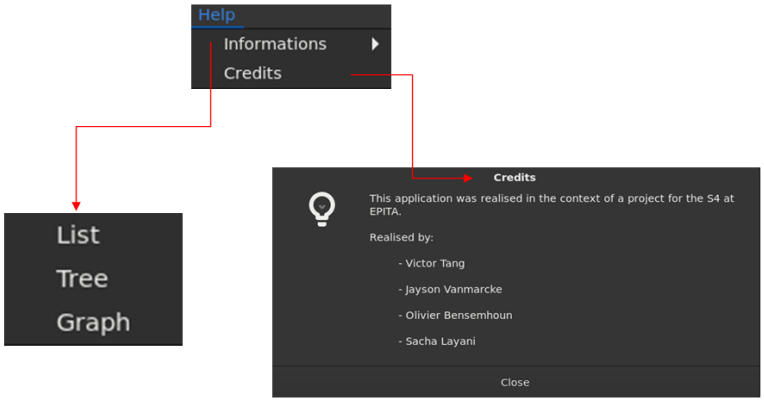
Pour la partie haute :

La barre du Menu est composée de 3 éléments : File, Edit et Help.

— File : ouvrir un fichier qui contient un tableau, un arbre binaire ou un graph.
— Edit : enregistrer un tableau/arbre/graph sous forme d'image ou en fichier txt ou dot.
— Help : un tutoriel pour utiliser l'application et les crédits.



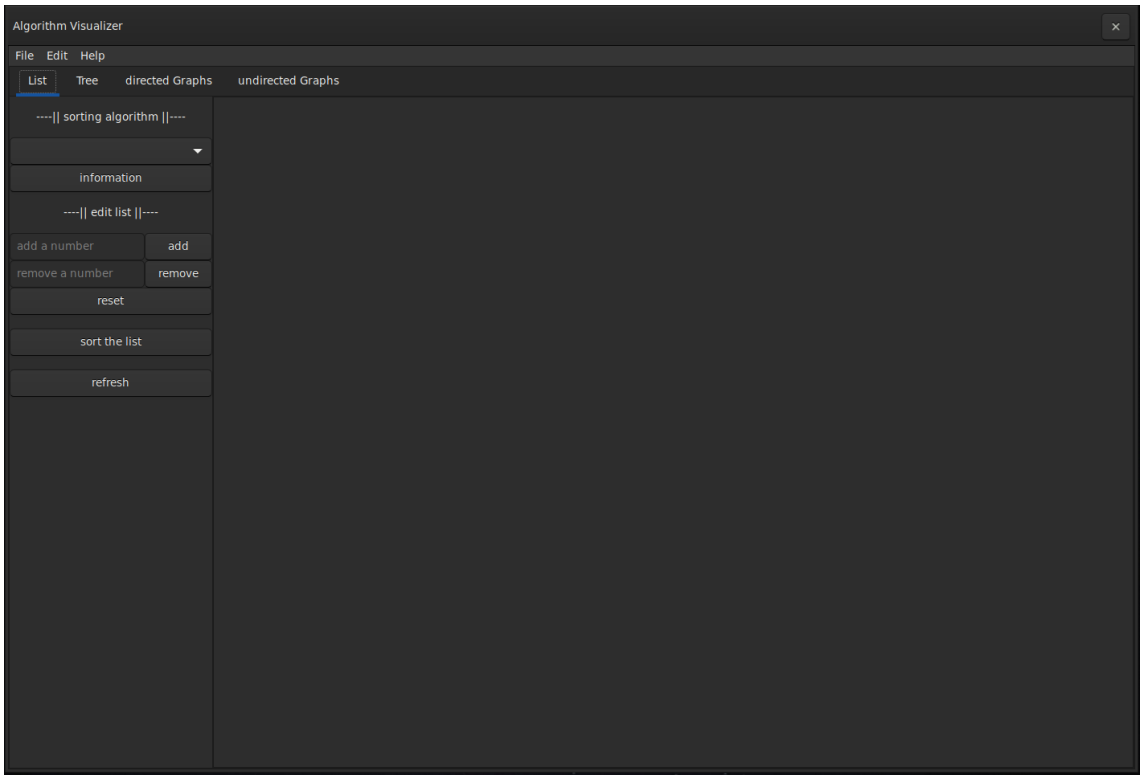




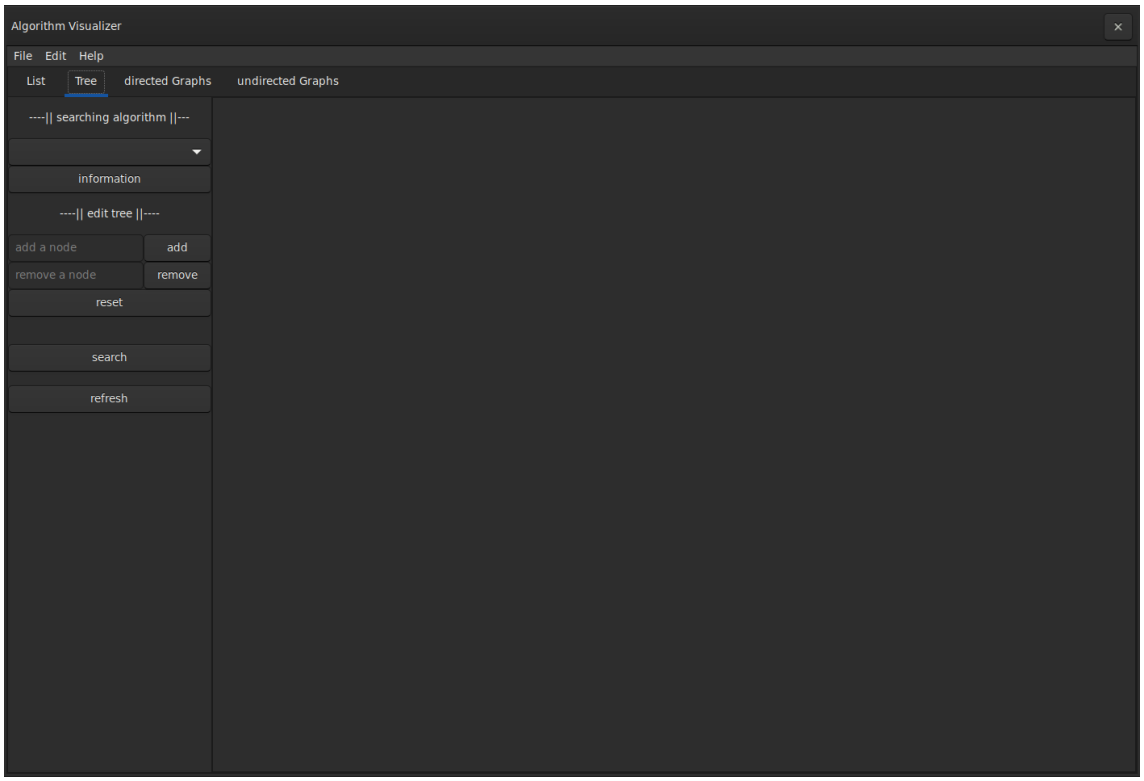
Pour la partie basse :

Elle contient 4 onglets où chaque onglet permet de faire des opérations sur les tableaux, arbres et graphes.

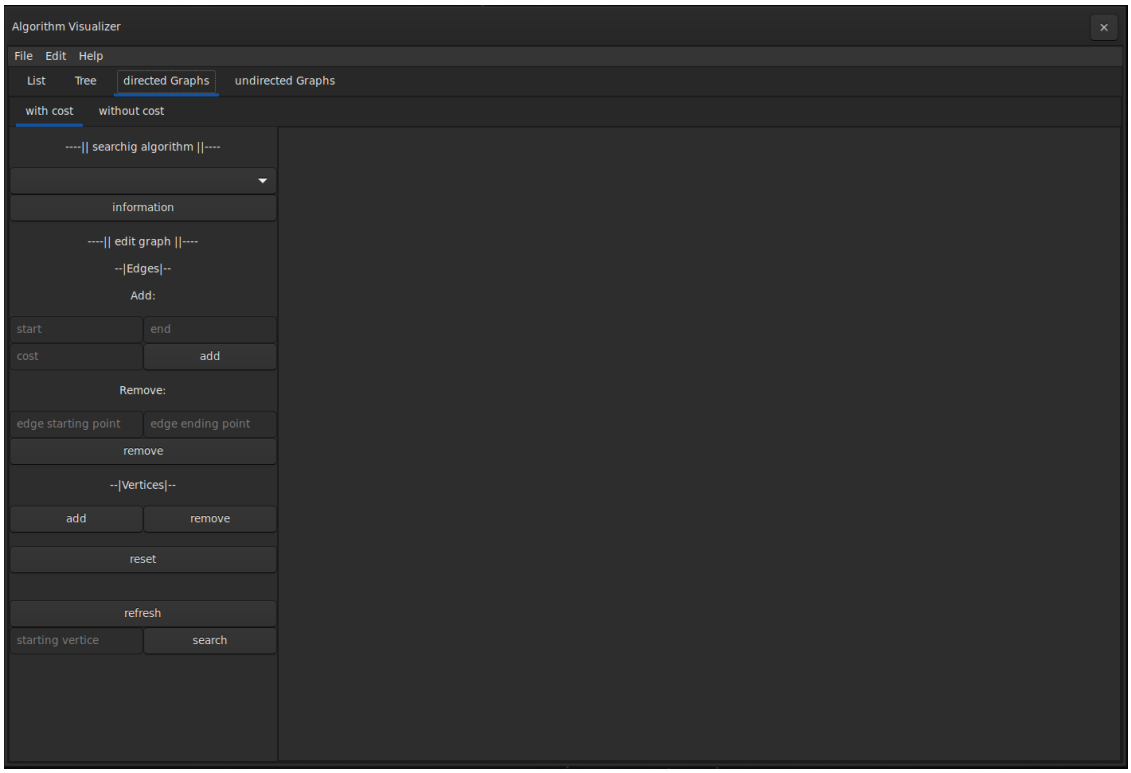
- List : onglet pour faire des opérations sur des tableaux.
  - Add : ajouter une valeur en fin de tableau.
  - Remove : retirer le premier élément rencontré dans le tableau.
  - Reset : enlever tous les éléments du tableau.
  - Sort the list : trie la list en fonction de l'algorithme choisi dans le menu déroulant.
  - Refresh : affiche l'état actuelle de la liste.
  - Information : ouvre un pop-up qui explique le principe de l'algorithme.



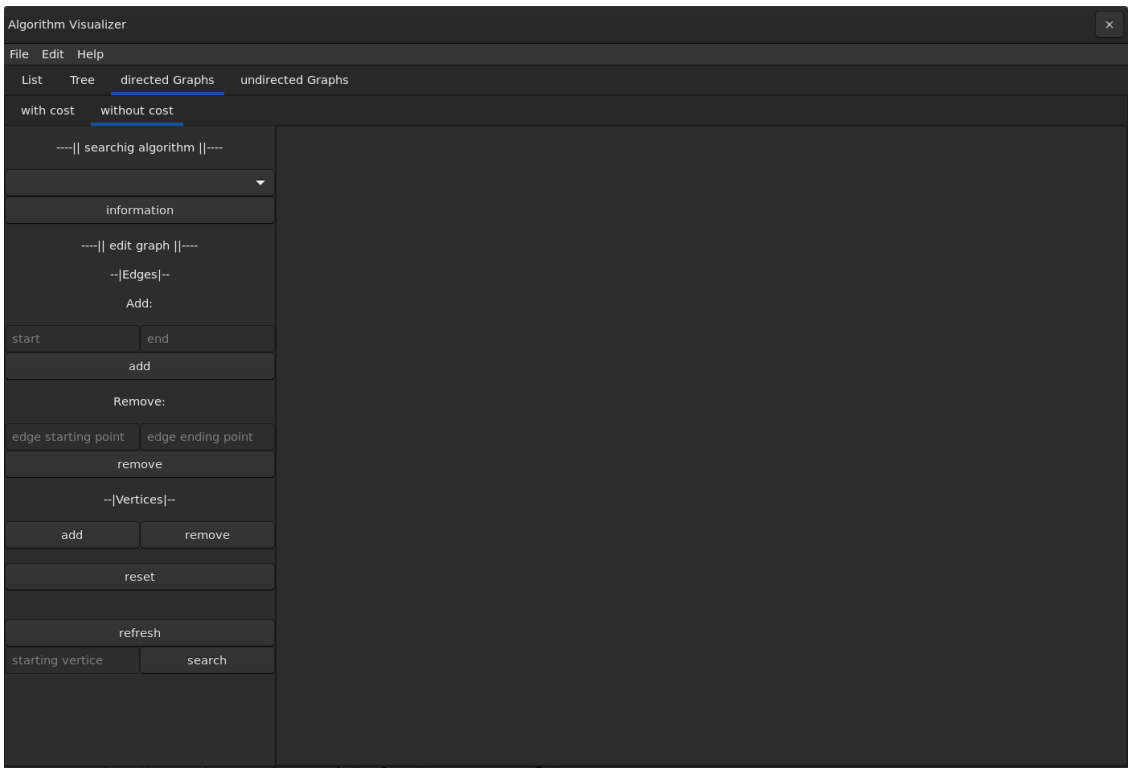
— Tree : onglet pour faire des opérations sur des arbres binaires.
  — Add : ajouter une valeur dans l'arbre.
  — Remove : retirer une valeur dans l'arbre.
  — Reset : enlève tout les noeuds de l'arbre.
  — Search : rechercher la valeur entrée avec l'algorithme choisi.
  — Refresh : supprime toutes les pages et affiche l'état actuelle de l'arbre.
  — Information : ouvre un pop-up qui explique le principe de l'algorithme.



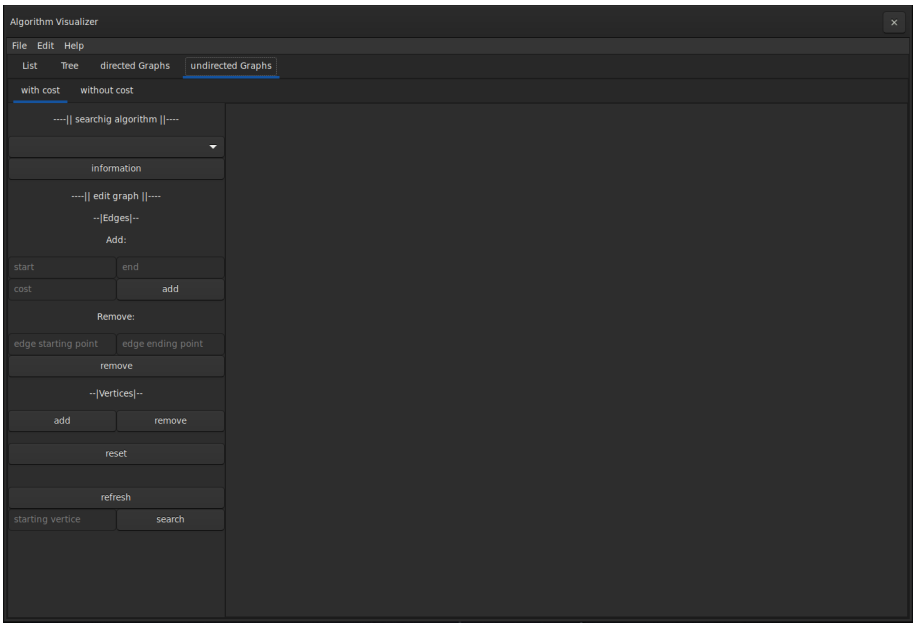
— Graph : onglets pour faire des recherches/opérations sur les graphes.
  — Orienté
    — Avec coûts
      — Add Vertice : ajoute un noeud dans le graph.
      — Remove Vertice : retire un noeud du graph et toutes les arêtes connectées.
      — Add Edge : ajoute une arête avec coût dans le graph.
      — Remove Edge : retire une arête du graph.
      — Reset : enlève tous les noeuds du graph.
      — Refresh : enlève toutes les images et affiche l'état actuel du graph.
      — Search : exécute l'algorithme choisi sur le graph.
      — Information : ouvre un pop-up qui explique le principe de l'algorithme.



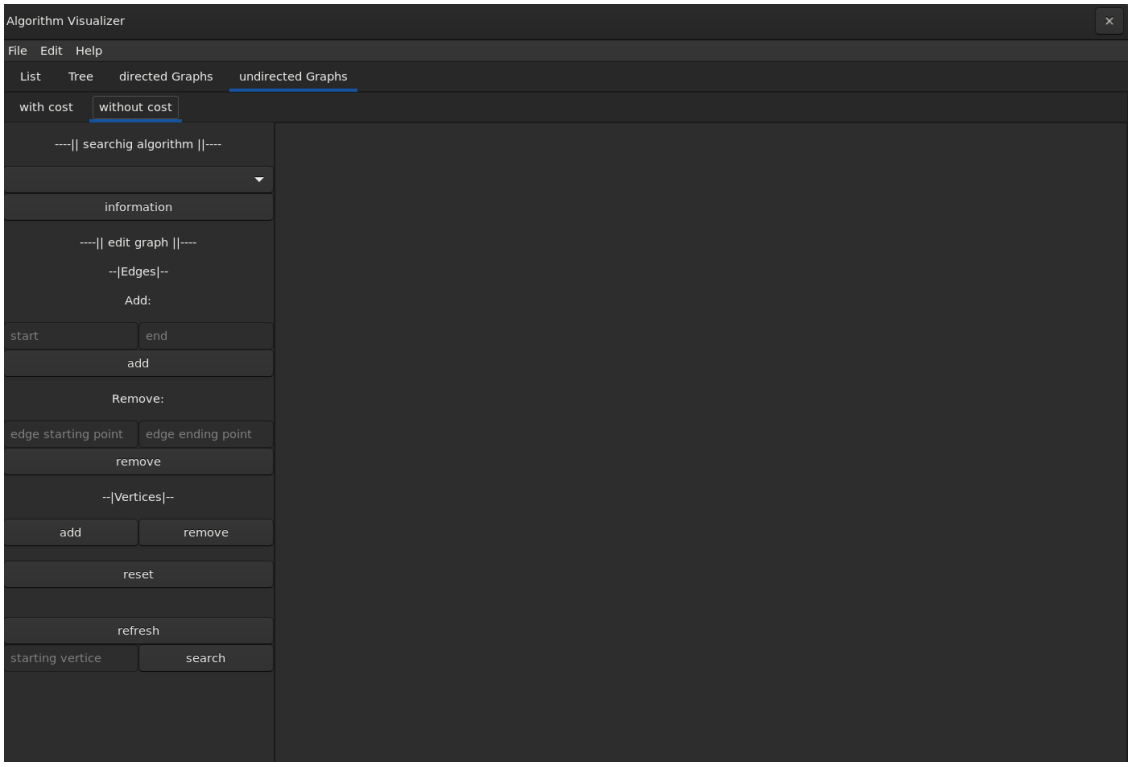
    — Sans coûts
      — Add Vertice : ajoute un noeud dans le graph.
      — Remove Vertice : retire un noeud du graph et toutes les arêtes connectées.
      — Add Edge : ajoute une arête dans le graph.
      — Remove Edge : retire une arête du graph.
      — Reset : enlève tous les noeuds du graph.
      — Refresh : enlève toutes les images et affiche l'état actuel du graph.
      — Search : exécute l'algorithme choisi sur le graph.
      — Information : ouvre un pop-up qui explique le principe de l'algorithme.



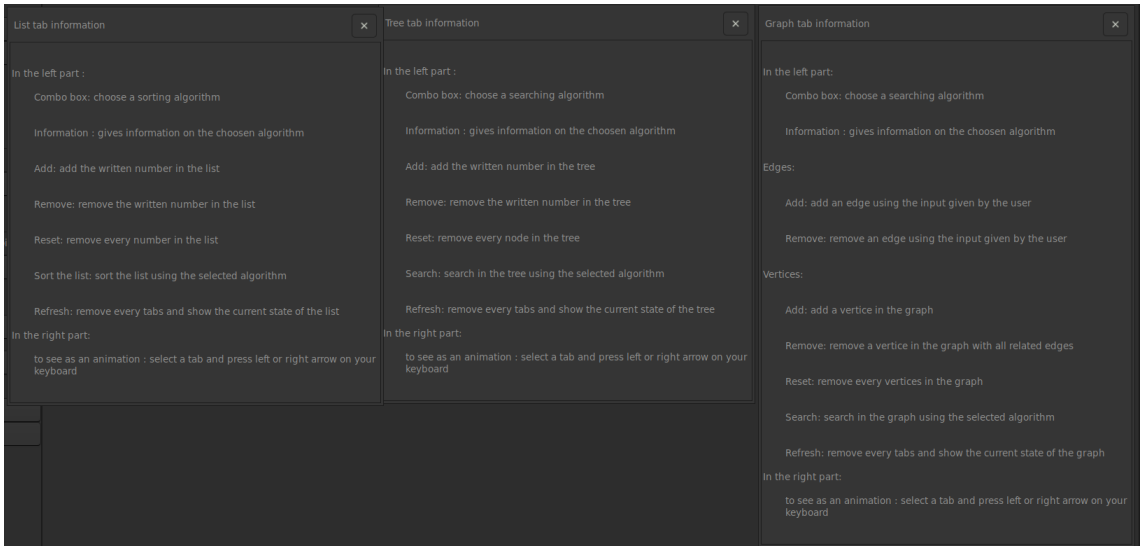
— Non-orienté
  — Avec coûts
    — Add Vertice : ajoute un noeud dans le graph.
    — Remove Vertice : retire un noeud du graph et toutes les arêtes connectées.
    — Add Edge : ajoute une arête avec coût dans le graph.
    — Remove Edge : retire une arête du graph.
    — Reset : enlève tous les noeuds du graph.
    — Refresh : enlève toutes les images et affiche l'état actuelle du graph.
    — Search : exécute l'algorithme choisi sur le graph.
    — Information : ouvre un pop-up qui explique le principe de l'algorithme.



— Sans coûts
  — Add Vertice : ajoute un noeud dans le graph.
  — Remove Vertice : retire un noeud du graphe et toutes les arêtes connectées.
  — Add Edge : ajoute une arête dans le graph.
  — Remove Edge : retire une arête du graph.
  — Reset : enlève tous les noeuds du graph.
  — Refresh : enlève toutes les images et affiche l'etat actuel du graph.
  — Search : exécute l'algorithme choisi sur le graph.
  — Information : ouvre un pop-up qui explique le principe de l'algorithme.



Voici une image sur les informations que vous pouvez trouver :



6.11.1 Mise à jour de qualite de vie

Depuis la première soutenance, une multitude de petites implémentations ont été faites pour une meilleure expérience lors de l'utilisation de l'interface.

Vous trouverez ci-dessous la liste des implémentations faites et leurs descriptions :

- List
  - Universel : l'affichage fonctionne aussi avec les nombres négatifs et zéro.
  - Affichage adaptatif : l'affichage de l'état de la liste s'adapte en fonction de la longueur de celle-ci et change la police afin de l'afficher complètement.
  - Refresh : ajout d'un bouton refresh pour enlever toutes les pages et affiche la page actuelle.
  - Pages : les pages ont le nom de l'opération faite.
  - Etat actuelle : affiche la dernière page après chaque opération.
  - Information : ouvre un pop-up qui explique le principe de l'algorithme.

- Tree
  - DFS : ajout des 3 parcours en profondeur (préfixe, infixe, suffixe).
  - Posisiton : affiche l'actuelle position dans l'arbre.
  - Supression : affiche les étapes de supression après la recherche.
  - Pages : les pages ont le nom de l'opération faite.
  - Etat actuel : affiche la dernière page après chaque opération.
  - Information : ouvre un pop-up qui explique le principe de l'algorithme.

- Graphs
  - Etat actuel : affiche la dernière page après chaque opération.
  - Resultat : affiche l'etat des vecteurs de père et de distance durant l'exécution.
  - Affichage adaptatif : l'affichage des vecteurs s'adapte en fonction de la longueur de celle-ci et change la police afin de l'afficher complètement.
  - Refresh : ajout d'un bouton refresh pour enlever toutes les pages et affiche la page actuelle.
  - Pages : les pages ont le nom de l'opération faite.
  - Information : ouvre un pop-up qui explique le principe de l'algorithme.
  - Dot : affiche les noeuds deja visités en rouge et le noeud actuel en vert.
  - Dot : affiche les arêtes en cours de traitement en rouge pour une meilleure compréhension de l'algorithme.
  - Flexibilité d'orientation : un graph orienté peut être ouvert en tant que graphe non orienté et vice-versa.
  - Prim : affiche l'arbre de recouvrement de poids minimum avec des arêtes rouge durant l'exécution.
  - Flexibilité des coûts : un graphe contenant des coûts peut-être ouvert en tant que graph sans coûts.

— Menu
  — Menu déroulant : utilisation des menus déroulant pour éviter de surcharger l'interface de boutons.

— Débogage
  — Messages d'erreurs : les messages d'erreurs ne doivent plus être fermés pour continuer d'utiliser l'application.
  — Meilleurs formats : les messages d'erreurs sont plus esthétiques.

— Fenêtre d'application
  — Thème sombre : l'application a maintenant un thème sombre par défaut.
  — Amélioration visuel : ajout d'un nom et d'un bouton fermé sur l'application.

6.12 Site Internet

Dans le cadre de notre deuxième projet, nous avons développé deux sites web distincts. Le premier est dédié à la présentation du projet en lui-même, tandis que le second est consacré à notre groupe, les Alchimistes. Ce dernier a été créé au cours du deuxième semestre de l'année précédente, avec Olivier. Nous avons donc pensé qu'il serait intéressant de mettre en place un site web présentant les différents projets que nous avons réalisés.

Pour ce faire, nous avons utilisé les langages de programmation suivants :
— HTML / CSS
— JavaScript
— Tailwind CSS
Nous aurions souhaité intégrer le langage PHP dans sa version 8.0, qui corrige de nombreux défauts des versions précédentes souvent reprochés. Nous aurions également aimé utiliser le framework Laravel, ce qui nous aurait permis de gérer une petite base de données pour les messages et les contacts. Cependant, nous avons rencontré un obstacle majeur : GitHub ne permet pas de stocker et d'exécuter du code PHP.

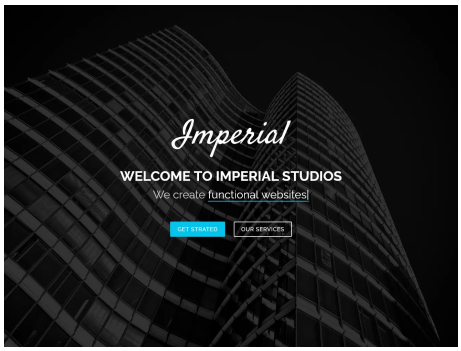
Ce site internet sera la vitrine de notre projet. Il est donc indispensable qu'il possède un beau design et qu'il soit simple et fluide au niveau de la navigation.

J'ai apprécié être en charge de la réalisation de ce nouveau site internet pour le projet du S4, car j'ai aussi effectué la création de site pour les projets du S2 et S3.

Pour réaliser le site de ce projet, j'ai pris la décision d'utilisé un Template : Bootstrap afin d'avoir une base sur laquelle travailler. Ce Template Imperial est un modèle d'une page Bootstrap moderne et créative, idéal pour les agences de création, les studios, les agences de design numérique ou d'autres entreprises similaires.

L'entête est livré avec une partie héros en plein écran. Imperial est également livré avec un menu mobile moderne hors toile pour une meilleure navigation et une meilleure expérience utilisateur. Bien évidemment, j'ai changé plusieurs lignes de codes afin d'obtenir un résultat optimal.

J'ai pris la décision d'utiliser le même Template que pour la création du site du S3 car selon moi, il est particulièrement riche en ressources avec de beaux designs en corrélation avec notre projet.



Commençons par parler de la conception graphique du site. La charte graphique doit être entièrement définie. Pour ce faire, nous avons prédéfini les logos, les couleurs et la typographie que nous appliquerons sur un style graphique accessible mais moderne.

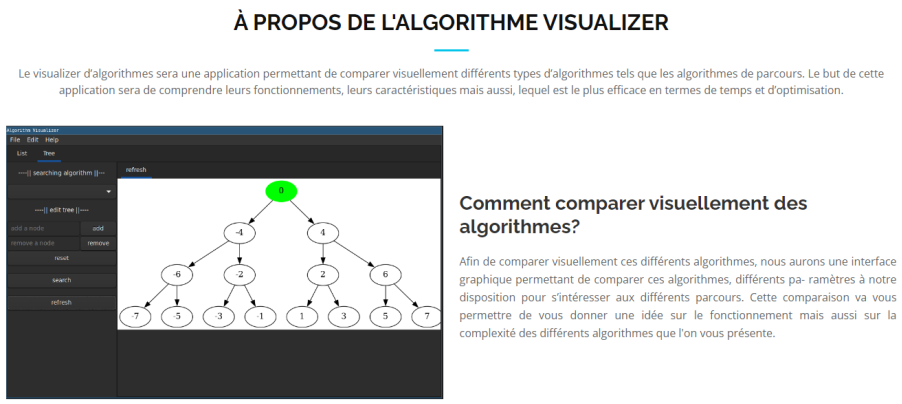
Le Logo est symbolisé par le nom de notre groupe. Il est rouge avec quelques effets en blanc car le rouge reflète la force, l'énergie et la passion et surtout cette couleur attire l'attention.

LES ALCHIMISTES

Ensuite, parlons des spécifications techniques. Le site est développé en partie sur une seule page appelée « one page » ou seule la partie présentation du projet renvoie pour chaque thème vers une autre page complète et détaillée.

Ce type de site est totalement adapté car il y a peu de contenu et il présente l'avantage de mieux contrôler la navigation des internautes en proposant une expérience utilisateur adaptée. Cela permet l'amélioration de la conversion du site web favorisant ainsi le référencement.

Laissez-moi vous montrer quelques captures écrans de notre site internet :



RÉSUMÉ

Notre outil d'Algorithme Visualizer est une application puissante qui fait appel à des algorithmes de pointe pour comprendre, analyser rapidement n'importe quel algorithme tels que les algorithmes de tri, les algorithmes de plus courts chemins ou encore, des algorithmes sur les arbres binaires.

PRÉSENTATION

Ce site présente notre projet Algorithme Visualizer 2024. Les élements fonctionnels du programme sont: les différents algorithmes proposés tels que les arbres binaires, graph et tri, puis notre interface graphique.

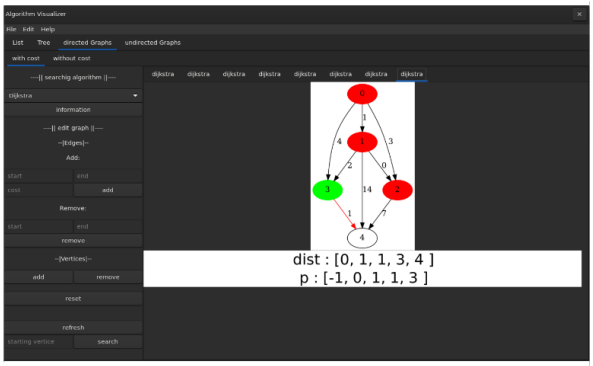
UTILITAIRES

Nous disposons également de petits utilitaires créés pour simplifier divers opérations répétitives (formatage du code, compilation de l'ensemble du projet, ...).

PLANNING

Le projet s'est déroulé en respectant les délais demandés et les choses se présentent de façon à ce que le projet soit terminé pour l'échéance requise. Chacun a travaillé dur pour proposer un travail de qualité.

Les aspects techniques



Projet information

Catégorie: Techniques

Date du projet: 03 Juin 2024

Résumé

Algorithme Dijkstra's
Algorithme A* Search
ABR (Arbres Binaire de Recherche)
BFS.DFS (graph)
Algorithme Floyd-Warshall
Algorithme de Ford-Bellman
Algorithme glouton de recherche Best-First
Algorithme Spanning Trees minimum
Insertion sort
Merge sort
Counting sort
Interface graphique

Algorithme Dijkstra's

L'algorithme de Dijkstra est un algorithme de recherche de chemin le plus court dans un graphe, c'est-à-dire un réseau de nœuds connectés par des arêtes ayant des poids ou des coûts associés. L'algorithme détermine le chemin le plus court entre un nœud de départ et tous les autres nœuds du graphe.

ALL TECHNIQUES UTILITAIRES PROBLÈMES HISTORIQUE

Le deuxième site web que nous avons développé est un site de présentation du projet. Il s'agit d'un site épuré, mettant en avant les éléments les plus importants pour comprendre comment utiliser notre application, ainsi que les personnes qui ont contribué à sa création.

7 Conclusion

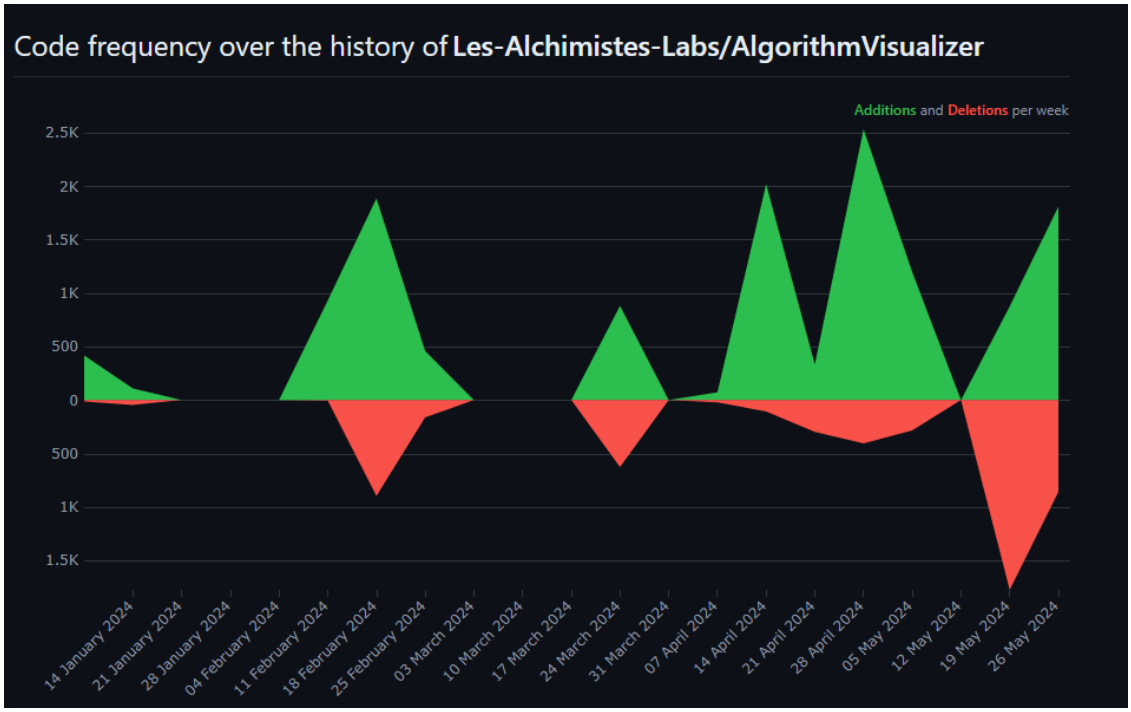
Ainsi, nous sommes satisfaits du résultat final du projet et nous estimons que tout s'est déroulé conformément à nos attentes. Les fonctionnalités principales du logiciel ont été implémentées avec succès. Les difficultés rencontrées ont été surmontées grâce à notre travail en équipe et notre capacité d'adaptation.

Dans le cadre de ce projet, nous avons eu l'opportunité de travailler en groupe, ce qui a été une expérience enrichissante. La mise en commun des compétences de chacun et le travail d'équipe nous ont permis de respecter le planning et d'atteindre les objectifs que nous nous étions fixés. Nous sommes fiers d'avoir réalisé ce projet et nous espérons que notre application vous aura intéressée autant qu'elle nous a passionée.

En conclusion, nous avons appris de nombreuses choses qui ont alimenté notre bibliothèque à savoir notre cerveau durant cette période en Rust et nous sommes impatients de relever les futurs défis qui seront présents lors de l'année suivante : les piscines !

8 Annexes

8.1 Historique de développement



8.2 Lien

Afin d'accéder à notre projet, voici le lien de notre github :

https ://github.com/Les-Alchimistes-Labs/AlgorithmVisualizer