École Polytechnique de l'Université de Tours
64, Avenue Jean
Portalis 37200
TOURS, FRANCE
(33)2-47-36-14-14
http://www.polytech.univ-tours.fr

Parcours des écoles d'ingénieur Polytech

Année 2023-2024

Projet Informatique : Labyrinthe

Étudiants

Pacôme Renimel–Lamiré
pacome.renimel–lamire@etu.univ-tours.fr
Esteban Laurent
esteban.laurent@etu.univ-tours.fr

Encadrant

Christophe Lenté
christophe.lente@univ-tours.fr

Avertissement

Ce document a été rédigé par Pacôme Renimel–Lamiré et Esteban Laurent susnommés les auteurs.

L'École Polytechnique de l'Université François Rabelais de Tours est représentée par Christophe Lenté susnommé le tuteur académique.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

Les auteurs reconnaissent assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non-respect des lois ou des droits d'auteur.

Les auteurs attestent que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

Les auteurs attestent ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

Les auteurs attestent que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

Les auteurs reconnaissent qu'ils ne peuvent diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable du tuteur académique et de l'entreprise.

Les auteurs autorisent l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source.

Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.

Table des matières

Introduction

Ce rapport a pour objectifs de décrire les différentes étapes de réalisation d'un programme informatique permettant de générer et résoudre automatiquement des labyrinthes, ainsi que de présenter les résultats obtenus.

L'objectif premier du projet consiste à créer un labyrinthe par une ou plusieurs méthodes différentes et à faire trouver à l'ordinateur un chemin menant vers la sortie. Pour ce faire, nous avons implémenté trois algorithmes différents : `depth-first search`, `recursive backtracking` et `A*`.

La suite du projet a été laissée relativement libre, nous invitant à faire preuve d'imagination et de créativité pour aller au-delà des objectifs initiaux. Nous avons donc décidé d'ajouter des fonctionnalités supplémentaires, notamment un mode de jeu permettant à l'utilisateur d'évoluer dans des labyrinthes générés aléatoirement tout en étant poursuivi par des ennemis et en collectant des objets.

Nous avons également essayé de rendre notre programme le plus interactif possible, en rajoutant une interface graphique agréable et complète permettant de générer et résoudre des labyrinthes en personnalisant différents paramètres.

Afin de décrire de façon détaillée le déroulement de ce projet, ce rapport est divisé en plusieurs chapitres.

Dans un premier temps, nous étudierons brièvement un état de l'art des labyrinthes et des algorithmes de génération et de résolution de labyrinthes.

Nous expliquerons ensuite la méthodologie que nous avons appliquée pour mener à bien ce projet, en décrivant les différentes étapes de développement du programme.

Nous poursuivrons en présentant l'implémentation du programme, en décrivant la structure du code et les différents obstacles rencontrés.

Nous présenterons ensuite les résultats obtenus, en détaillant les performances du programme et les statistiques obtenues.

Enfin, nous prendrons du recul sur le projet en discutant de potentielles améliorations pouvant être apportées au projet à l'avenir, ainsi que de notre ressenti par rapport à ce que cette expérience a pu nous apporter.

Il peut être utile pour le correcteur de pouvoir tester le programme ou en consulter le code source au besoin avant, durant ou après la lecture de ce rapport. Pour ce faire, nous avons ajouté en annexe une section dédiée à l'installation du programme et à la consultation du code source.

1 État de l'art et enjeux

1.1 Les labyrinthes : une fascination depuis l'Antiquité

Selon le dictionnaire Larousse, un labyrinthe peut être défini comme suit :

"*Dans l'Antiquité, [il s'agissait d'un] vaste édifice comprenant d'innombrables salles agencées de telle manière que l'on ne trouvait que difficilement l'issue.* [1]"

L'exemple le plus commun de labyrinthe associée à l'histoire humaine est probablement le labyrinthe de Crète, construit par Dédale pour emprisonner le Minotaure, un monstre mi-homme mi-taureau, au point que le mot dédale est rentré dans le langage courant pour désigner un lieu où l'on se perd facilement.

Toutefois, on retrouve des vestiges de tracés labyrinthiques depuis la fin de l'âge de bronze, la plus ancienne peuve de leur existence étant une tablette datant d'environ 1200 avant J.C. retrouvée à Pylos, en Grèce.[2]

Les historiens s'accordent sur le fait que l'idée du labyrinthe était principalement symbolique, leur tracé n'étant quasiment jamais retrouvé à une échelle humaine, mais plutôt représenté de façon décorative sur des pièces de monnaie, des poteries, des bâtiments, etc.

De plus, le tracé le plus commun d'un labyrinthe, le labyrinthe unicursal, est un tracé continu sans embranchements, ce qui le rend impossible à résoudre de façon classique. Malgré le fait qu'il ne représente aucun intérêt récréatif, c'est celui qu'on retrouve le plus souvent dans les représentations de cette époque, ce qui montre bien la prévalence du symbolisme sur la fonctionnalité.

Figure 1.1: Pièce d'argent représentant un labyrinthe unicursal, 400 avant J.C. - AlMare - CC-BY-SA-3

Ainsi, les labyrinthes que nous connaissons n'ont réellement fait leur apparition qu'au XVI^{e} siècle[3], en tant que jeu de réflexion et de divertissement.

1.2 Labyrinthes et informatique

Aujourd'hui, on retrouve de nombreuses installations permettant d'évoluer dans des labyrinthes, souvent végétaux (Haies, Maïs, etc.). Le labyrinthe est devenu un symbole de réflexion, de concentration et de patience, et est souvent utilisé dans des jeux vidéos, des films, des livres, etc.

Cette notion a notamment pris une dimension toute particulière dans les domaines des mathématiques et plus tard, de l'informatique. En effet, un labyrinthe peut être vu comme un graphe, où chaque case est un nœud et chaque passage entre deux cases est une arête.

Ainsi, utiliser un labyrinthe est un excellent moyen d'en apprendre plus sur les algorithmes de recherche, qui sont indispensables à d'innombrables technologies que nous utilisons aujourd'hui : les jeux vidéos, les applications de navigation, les moteurs de recherche, les intelligences artificielles, etc.

De nombreux problèmes mathématiques bien connus sont d'ailleurs basés sur la théorie des graphes et donc sur les labyrinthes, comme le problème du cavalier (un cavalier doit visiter une unique fois chaque case d'un échiquier) ou celui des sept ponts de Königsberg, tous deux résolus par Euler dès le milieu du XVIII^{e} siècle[4][5].

De la même façon que les labyrinthes ont été utilisés afin d'étudier la mémoire chez les rats, des compétitions de résolution de labyrinthes sont organisées chaque année, notamment par la société américaine *Micromouse*, qui propose aux participants de créer un robot capable de résoudre un labyrinthe en un temps record, sans aucune connaissance préalable du tracé du labyrinthe.[6]

1.3 Algorithmes de génération et de résolution de labyrinthes

1.3.1 Génération de labyrinthes

Un peu de théorie

Afin de générer un labyrinthe, il peut être très utile d'en apporter une définition mathématique plus rigoureuse.

Si on choisit l'approche de la théorie des graphes, alors un labyrinthe peut être décrit comme un graphe grille : un graphe non orienté où chaque nœud est relié à ses quatre voisins (haut, bas, gauche, droite) s'ils existent.

Chaque nœud représente un chemin du labyrinthe, et chaque arête représente le passage d'une case à une autre. Certaines propriétés intéressantes sont déjà présentes : on ne peut par exemple pas se déplacer d'une case à une autre si elles ne sont pas adjacentes, c'est à dire reliées par une arête.

Pour ajouter le concept de mur au labyrinthe, il suffit de retirer la connexion entre deux nœuds pour ajouter un "obstacle virtuel" entre les cases, empêchant ainsi le passage.

À partir de cette définition, on peut considérer que n'importe quel graphe ainsi dérivé d'un graphe grille, avec un nombre arbitraire d'arêtes manquantes, est un labyrinthe. Toutefois, afin de rendre les choses plus intéressantes, il est bon de rajouter quelques contraintes.

Il peut par exemple être intéressant de nécessiter l'existence d'au moins un chemin entre n'importe quelle case de départ, et n'importe quelle case d'arrivée. Cette contrainte est appelée la connexité du graphe, et est souvent utilisée pour garantir qu'un labyrinthe est "solvable".

On peut également pousser le vice et exiger que le chemin qui relie deux cases du labyrinthe soit unique. Si on considère notre graphe grille, alors un tel labyrinthe serait représenté par un arbre, c'est à dire un graphe convexe et acyclique (il est impossible de partir d'un sommet et d'y revenir sans rebrousser chemin au moins une fois). Plus encore, il s'agit d'un arbre couvrant, puisqu'il couvre l'ensemble des sommets du graphe de départ.

Dans ce contexte, un labyrinthe qui vérifie ces deux conditions est appelé un labyrinthe parfait. Ainsi, la plupart des algorithmes de génération ont été conçus pour produire des labyrinthes parfaits.

Toutefois, il est très souvent plus intéressant de travailler avec des labyrinthes non-parfaits, et où plusieurs chemins sont possibles. On peut ainsi tester les algorithmes de résolution sur un critère supplémentaire : la longueur du chemin trouvé, que l'on précisera dans la suite de cette partie.

Parallèlement, un algorithme de génération de labyrinthe peut être comparé sur la base de deux critères : le temps de génération, ainsi que la "difficulté" du labyrinthe généré, ce qui est assez peu objectif.

L'algorithme "Depth-first search"

Dans le cadre de ce projet, nous avons choisi de générer des labyrinthes en utilisant l'algorithme de *depth-first search*, également appelé *recursive backtracking*, un algorithme de type "diviser pour régner" qui consiste à diviser le problème en sous-problèmes plus petits, les résoudre, puis les combiner pour obtenir la solution du problème initial.

Malgré son nom, son implémentation ne nécéssite pas d'utiliser une récursion au sens strict, et est souvent plus performante de façon itérative, en utilisant une **pile** pour stocker les états précédents. L'utilisation d'une pile, en plus d'être plus sûre (en évitant d'éventuells erreurs liées à un manque de mémoire dans la pile d'appel des fonctions), permet de faciliter la visualisation de l'algorithme, en rendant accessibles toutes les données qui y sont liées à tout moment.

L'algorithme fonctionne de la façon suivante :

1. On considère un labyrinthe où chaque case est entourée par des murs. On définit une pile vide.
2. Choisir un point de départ dans le labyrinthe.
3. Marquer ce point comme visité et l'ajouter à la pile.
4. Tant qu'il existe des voisins non visités :
    a) Retirer la case en haut de la pile et la définir comme la "case courante"
    b) Si la case courante possède au moins un voisin qui n'a pas été visité :
        i. Ajouter la case courante à la pile
        ii. Choisir un de ses voisins non visités (aléatoirement)

iii. Retirer le mur entre la case courante et le voisin choisi

iv. Marquer le voisin comme visité et l'ajouter à la pile

Cet algorithme garantit la génération d'un labyrinthe parfait. De plus, il est relativement simple à implémenter et produit des labyrinthes avec une structure intéressante.

Après considération des alternatives existantes, nous avons jugé que l'intérêt d'implémenter d'autres algorithmes de génération de labyrinthes n'était pas suffisant pour justifier le temps et les ressources nécessaires à leur implémentation. En effet, les alternatives existantes telles que les algorithmes de Kruskal ou de Prim ne présentent pas d'avantages significatifs ou de résultats significativement différents par rapport à l'algorithme de *depth-first search*.

La principale différence entre les labyrinthes générés par ces algorithmes réside dans sa structure : le *depth-first search* génère des labyrinthes avec de longs couloirs, tandis que les algorithmes de Kruskal et de Prim génèrent des labyrinthes avec des couloirs plus courts, rendant une grande partie du labyrinthe fondamentalement inutile du point de vue de la résolution.

Afin d'ajouter de la diversité à nos labyrinthes, et de rendre le processus de résolution beaucoup plus intéressant, nous avons décider d'implémenter ce que nous avons appelé un "facteur de bouclage" : il s'agit d'un réel compris entre 0 et 1, et qui permet de transformer un labyrinthe parfait en labyrinthe comportant de nombreux chemins différents entre deux cases.

Une fois un labyrinthe généré, on sélectionne aléatoirement un certain nombre de murs en fonction du facteur de bouclage (un facteur de 0.5 signifiant que la moitié des murs du labyrinthe seront retirés), et on les retire du labyrinthe. On obtient ainsi un labyrinthe non-parfait, mais toujours solvable.

1.3.2 Résolution de labyrinthes

Un peu de théorie

L'élucidation d'un labyrinthe consiste principalement en la recherche d'un chemin dans un graphe, tout en tenant compte des propriétés spécifiques de ce dernier, comme abordé précédemment.

Ainsi, il est tout à fait possible d'utiliser n'importe quel algorithme de recherche de chemin dans un graphe pour résoudre n'importe quel labyrinthe, comme par exemple l'algorithme de Dijkstra.

Toutefois, le contexte du labyrinthe permet également de rajouter certaines contraintes optionnelles à la résolution, comme par exemple le fait de se mettre à la place d'un joueur ne pouvant pas voir l'ensemble du labyrinthe à la fois. De plus, certains algorithmes ne fonctionnent que sur des arbres couvrants, c'est à dire des labyrinthes parfaits.

On peut citer par exemple deux algorithmes triviaux de résolution de labyrinthes : le mur suiveur, qui consiste à suivre un mur à sa droite ou à sa gauche jusqu'à trouver la sortie, et celui de la "souris aléatoire", qui consiste à se déplacer aléatoirement dans le labyrinthe jusqu'à trouver la sortie.

Ce dernier algorithme est d'ailleurs plus intéressant qu'il en a l'air : il est évidemment très peu performant, mais c'est un exemple d'*algorithme de Las Vegas*, c'est à dire un

algorithme aléatoire qui est garanti de trouver une solution correcte au problème qui lui est présenté.

De son côté l'algorithme de mur suiveur fonctionne uniquement sur des labyrinthes parfaits, et peut rentrer dans des boucles infinies si ce n'est pas le cas.

Comme nos labyrinthes ne sont pas parfaits, et que l'objectif reste d'en trouver une solution dans un temps raisonnable, nous avons choisi d'implémenter deux algorithmes beaucoup plus performants que ceux-ci : `recursive backtracking` et `A*`.

L'algorithme "Recursive backtracking"

L'algorithme `recursive backtracking` est un algorithme qui consiste à identifier les impasses du labyrinthes, et de les "bannir" afin de ne plus les emprunter. Cet algorithme est bien le même que celui qui permet de générer des labyrinthes, mais en sens inverse. Il porte donc le même nom, mais afin d'éviter toute confusion, nous avons fait le choix de l'appeler exclusivement "*recursive backtracking*", tandis que l'algorithme de génération est appelé "*depth-first search*".

Bien que sur le papier, il soit similaire à celui que nous avons déjà étudié, il est en réalité légèrement différent dans son implémentation. En plus de la pile et de la liste de cases déjà visitées, on va également utiliser une liste de cases bannies, qui sont des cases qui ne mènent à aucune solution.

L'algorithme fonctionne de la façon suivante :

1. On considère un labyrinthe quelconque, avec une case de départ et une case d'arrivée. On initialise une pile contenant la case de départ, une liste de cases déjà visitées vide, et une liste de cases bannies vide.
2. Tant que la pile ne contient pas la case d'arrivée :
   a) On choisit une case voisine non visitée, non bannie, et non séparée par un mur de la case en haut de la pile.
   b) Si aucune case n'est disponible, on retire la case en haut de la pile et on l'ajoute à la liste de cases bannies.
   c) Sinon, on ajoute la case choisie à la pile et on la marque comme visitée.
3. Lorsque la pile contient la case d'arrivée, on a trouvé une solution : la pile contient le chemin entre la case de départ et la case d'arrivée.

Cet algorithme est raisonnablement performant, mais n'est pas nécessairement garanti de trouver le chemin le plus court entre deux cases. Il est toutefois garanti de trouver une solution si elle existe.

L'algorithme "A*"

L'algorithme `A*` est un algorithme de recherche de chemin dans un graphe qui utilise une heuristique pour guider la recherche. Il est souvent utilisé dans des contextes où la recherche de chemin est difficile, comme par exemple dans les jeux vidéos, les applications de navigation, etc.

Une heuristique est une fonction qui permet de "guider" l'algorithme dans ses choix en faisant des hypothèses jugées "raisonnables". Par exemple, dans l'algorithme

précédent, lorsque plusieurs cases sont valides, on en choisit une au hasard. Ici, on fera appel à une fonction qui permettra de choisir la case la plus ”prometteuse” parmi les cases valides.

Cette fonction peut grandement varier en fonction du contexte, par exemple en tenant compte du poids d'une connexion dans un graphe (la vitesse de circulation sur une route par exemple). Ici, on utilisera la distance de Manhattan, c'est à dire la somme des distances horizontales et verticales entre deux cases.

Comme l'algorithme est basé sur des hypothèses, il n'est également pas garanti de trouver une solution optimale. Toutefois, il trouve généralement des solutions qui s'en approchent en un temps très court. De plus, il est garanti de trouver une solution si elle existe.

Ses nombreux avantages et sa grande adaptabilité en font un algorithme très populaire : il est notamment utilisé par défaut dans de nombreux moteurs de jeux vidéos, et est souvent utilisé dans des compétitions de résolution de labyrinthes.

D'autres algorithmes

Il existe de très nombreux autres algorithmes de résolution de labyrinthe, ayant tous leurs avantages et inconvénients. On peut citer par exemple l'algorithme de Dijkstra, l'algorithme de Bellman-Ford, l'algorithme de Floyd-Warshall, etc.

Bien que leur implémentation ne soit pas nécéssairement plus complexe que celle des algorithmes que nous avons choisi d'implémenter, nous avons préféré passer plus de temps à créer des visualisations et des fonctionnalités supplémentaires pour le programme plutôt que d'implémenter de nouveaux algorithmes.

L'ajout de ces algorithmes pourrait être une piste d'améliorations pour le futur, comme il le sera précisé dans le cinquième et dernier chapitre de ce rapport.

2 Méthodologie

Au sein de ce chapitre, nous allons décrire les différentes étapes de développement du programme, en décrivant les outils utilisés ainsi que les différentes étapes de conception.

Il est bon de rappeler que des instructions détaillées sur l'installation du programme et la consultation du code source sont disponibles en annexe 1. Il est également possible de consulter directement le dépôt Git du projet à l'adresse suivante : `https://github.com/HerbeMalveillante/ProjetPeiP24`

2.1 Versionnage

Garder une trace de l'évolution du code source est une étape cruciale dans tout projet informatique, surtoût lorsqu'il est réalisé en équipe.

Conserver le projet stocké en "local" sur une machine est la meilleure façon de s'exposer à des pertes de données, des erreurs de manipulation, etc. De plus, il est souvent nécessaire de travailler en même temps sur un même fichier, ce qui peut être une source de conflits évidente.

Imaginons que deux développeurs modifient le même fichier en même temps, et que l'un des deux enregistre ses modifications avant l'autre. Lorsque le second enregistre ses modifications, il écrase celles du premier, et les modifications de ce dernier sont perdues. C'est ce qu'on appelle un conflit.

De plus, si un changement majeur dans le code source provoque un bug et qu'il est difficile de faire machine arrière, il est souvent nécessaire de revenir à une version antérieure du code source, ce qui peut faire perdre beaucoup de progression si les sauvegardes n'ont pas été régulières.

Pour éviter tous ces problèmes, il est nécessaire d'utiliser un logiciel de gestion de version, qui permet de conserver un historique de toutes les modifications apportées au code source, de revenir à une version antérieure, de créer des branches pour travailler sur des fonctionnalités différentes, etc.

L'outil que nous avons choisi d'utiliser est Git[7], associé à la plateforme en ligne GitHub[8].

2.1.1 Git

Git est un logiciel libre de gestion de version créé par Linus Torvalds (le créateur du noyau Linux) en 2005. Il est aujourd'hui le logiciel de ce type le plus utilisé au monde, et est considéré comme un standard de facto dans le monde du développement logiciel de part sa nature libre et gratuite, et sa grande robustesse.

Git peut fonctionner de façon décentralisée, c'est à dire que chaque développeur peut avoir une copie du code source sur sa machine, et travailler dessus de façon indépendante. Il est également possible de travailler en équipe, en partageant le code

source sur un serveur distant, et en synchronisant les modifications apportées par chaque développeur.

Lorsqu'on a terminé une fonctionnalité, on peut "commit" nos modifications, c'est à dire les enregistrer dans l'historique du projet. On peut ensuite "push" ces modifications sur le serveur distant, où elles seront visibles par les autres membres de l'équipe qui pourront les "pull" pour les télécharger sur leur machine.

Si des conflits surviennent, Git est capable de les gérer de façon très efficace, en permettant de les résoudre manuellement, ou en utilisant des outils de résolution de conflits.

Cet outil est également très utile pour travailler collaborativement grâce à son système de branches, qui permet de travailler sur des fonctionnalités différentes en parallèle, et de fusionner ces branches une fois les fonctionnalités terminées afin d'éviter de mettre en ligne des fonctionnalités non abouties sur la branche principale.

Git est disponible sous forme de ligne de commande, mais il existe également de nombreuses interfaces graphiques qui permettent de l'utiliser de façon plus intuitive.

2.1.2 GitHub

GitHub est une plateforme en ligne qui fait office de serveur distant pour les projets Git. Elle permet d'héberger des projets gratuitement afin de les rendre accessibles à tous facilement et est ainsi le berceau de nombreux projets open-source tels que le noyau Linux lui-même[9], le langage de programmation Python[10], etc.

En plus de cette fonctionnalité, GitHub propose de nombreux outils supplémentaires pour faciliter la coopération et permettre à des collaborateurs externes de facilement travailler sur un projet, en créant sa propre "version alternative" appelée "fork", en proposant des modifications au projet original appelées "pull requests", en faisant des rapports de bugs, etc.

Parfois, certaines versions alternatives peuvent devenir aussi populaires que le projet original, comme c'est le cas de la version communautaire de Pygame, qui est utilisée pour ce projet.

Tout au long du projet, nous avons utilisé Git et GitHub pour conserver un historique de nos modifications, ce qui s'est avéré très utile. Nous avons également pu utiliser GitHub pour en mettre à disposition le code source.

2.2 Programme principal

2.2.1 Langage de programmation, librairies et outils.

Le langage de programmation que nous avons utilisé pour ce projet est Python, associé à la version communautaire de la librairie Pygame "Pygame-CE"[11].

Le choix de cette librairie au lieu d'autres librairies graphiques a été basé sur sa simplicité d'utilisation, sa grande versatilité, ainsi que nos connaissances préalables avec cette librairie.

La version communautaire a été préférée pour ses nombreuses améliorations par rapport à la version officielle, notamment en termes de performances et de compatibilité avec les dernières versions de Python. En effet, la version originale de Pygame est de moins en moins maintenue. Les programmes écrits avec la version officielle

fonctionnent correctement avec la version communautaire, mais l'inverse n'est pas garanti.

Des outils de formattage externe tels que `black`[12] ont également été utilisés pour garantir une cohérence dans le code source, et une conformité avec les standards `PEP 8`[13] qui définissent les conventions en matière de style de code Python.

Les librairies `rich` et `graphviz` ont également été utilisées pendant le développement, la première pour améliorer la lisibilité des messages affichés dans la console (notamment lors de l'affichage de longs tableaux et listes), et la seconde lors de nos recherches sur les structures de données à utiliser pour stocker les labyrinthes. Ces librairies n'ont pas été utilisées dans la version finale du programme.

2.2.2 Conception

La première étape de la conception du programme a été de définir les différentes fonctionnalités que nous souhaitions implémenter, ainsi que de réaliser de nombreuses recherches sur les algorithmes de génération et de résolution de labyrinthes.

Nous avons ensuite défini une "liste de tâches" à réaliser, en les classant par ordre de priorité, avant de commencer à travailler sur des versions primitives du programme. Ces versions primitives nous ont permis de tester les algorithmes de génération et de résolution de labyrinthes, ainsi que de nous familiariser avec la librairie Pygame-CE et les diverses optimisations qu'il convient d'apporter pour obtenir les meilleures performances. Cette liste de tâches nous a également permis d'éclaircir les objectifs de la partie "libre" du projet, sur laquelle nous n'étions pas guidés par le sujet.

Un point majeur de cette étape fut de définir la structure de donnée à utiliser pour stocker les labyrinthes. En effet, il est tout à fait possible d'utiliser la programmation orientée objet pour définir un labyrinthe sous la forme d'un graphe, d'une matrice, d'une liste de cases, etc. L'enjeu était de trouver une structure adaptée aux algorithmes, mais également facile à utiliser pour l'adapter à l'interface graphique.

Une fois que nous avons obtenu des résultats satisfaisants et que nous avons défini une direction à suivre, nous avons pu commencer à travailler sur la version finale du programme, en ajoutant des fonctionnalités au fur et à mesure et en les testant régulièrement.

2.3 Rédaction du rapport

La rédaction du rapport a été réalisée en utilisant le langage de composition de documents LaTeX(LaTeX), qui est un langage de balisage avancé permettant de créer des documents de grande qualité typographique adaptés à la publication scientifique.

Ce langage est très populaire dans le monde de la recherche, et est souvent utilisé pour rédiger des articles, des thèses, des rapports, etc. Il permet de gérer de façon très efficace les références bibliographiques, les tableaux, les figures, les équations, etc.

Ainsi, nous avons pu nous concentrer sur le fond du rapport, LaTeX se chargeant automatiquement de la forme. Le code source du rapport est également disponible sur le dépôt Git du projet.

3 Implémentation

Dans ce chapitre, nous allons décrire la structure du programme, ainsi que les différentes fonctionnalités que nous avons implémentées et les obstacles que nous avons rencontrés.

3.1 Structure du programme

Le programme que nous avons réalisé est divisé en plusieurs fichiers Python, chacun regroupant un certain nombre de fonctionnalités spécifiques.

Nous avons utilisé un paradigme de programmation orientée objet tout au long du projet, afin de créer des classes qui représentent les différents éléments du programme, comme les labyrinthes, les menus, les éléments graphiques, les entités évoluant au sein du jeu, etc.

Afin d'éviter de créer des conflits d'importation de type "importation circulaire" (deux fichiers s'importent mutuellement), la division du programme peut avoir lieu de façon assez naturelle, en isolant les classes utilisées à plusieurs endroits dans un fichier à part.

Cette façon de procéder encourage la réutilisation du code source, ce qui a beaucoup été fait dans ce projet, notamment pour les éléments graphiques et les labyrinthes en eux-même.

De nombreuses classes héritent de classes fournies par `pygame-ce`, ce qui permet de faciliter l'implémentation de certaines fonctionnalités, comme la gestion des événements, l'affichage des éléments graphiques, etc.

Ainsi, de façon générale, chaque classe qui représente un élément visible à l'écran est responsable de son propre affichage, de sa mise à jour, et de sa gestion des événements, le tout étant piloté par une classe principale `Menu` qui gère l'affichage des différents menus du programme.

Nous avons également fait le choix de stocker les constantes régulièrement utilisées dans un fichier à part, afin d'y avoir accès facilement et de pouvoir les modifier rapidement si besoin. Ce fichier peut également faire office de "configuration" du programme, en regroupant toutes les options modifiables par l'utilisateur.

3.2 Classes, fonctions et méthodes principales

3.2.1 La classe `Labyrinthe`

La classe `Labyrinthe` est la classe principale du programme, qui représente un labyrinthe. Elle contient toute la logique nécéssaire à la génération la résolution et l'affichage d'un labyrinthe à l'écran.

Elle hérite de la classe `pygame.sprite.Sprite`, ce qui lui fournit par défaut des attributs de taille, de position, une image, une forme, ainsi que des méthodes pour l'affichage et la mise à jour de l'objet.

Lors de l'initialisation, cette classe prend en paramètre la taille du labyrinthe, le facteur de bouclage, et les algorithmes de génération et de résolution à utiliser. Elle crée ensuite un labyrinthe vide et initialise les algorithmes.

Plutôt que de stocker les informations du labyrinthe sous forme de matrice ou de graphe, nous avons choisi de simplement stocker tous les murs qui le composent dans une liste, sous la forme d'un `tuple` des deux cases qu'il sépare.

Les cases sont quand à elles identifiées par un identifiant entier unique. Si on imagine le labyrinthe comme une grille, alors on peut numéroter chaque case de 0 à $n^2 - 1$, où n est la taille d'un côté du labyrinthe.

Ainsi, nous avons facilement pu implémenter de nombreuses méthodes utilitaires permettant d'ajouter ou de retirer des murs, d'obtenir une liste des cases adjacentes à une case donnée, de vérifier si un mur existe entre deux cases, etc. Ces méthodes sont utilisées par les algorithmes qui manipulent le labyrinthe.

4 Résultats

Présenter les résultats obtenus

Ajouter des statistiques comme le temps de génération d'un labyrinthe, le nombre de nœuds explorés par A*, etc.

On peut aussi ajouter des statistiques sur les performances du programme, comme le temps d'exécution, la consommation de mémoire, etc, ainsi que sur le nombre de lignes, etc.

5 Discussion et perspectives

Interpréter les résultats obtenus et les comparer à ce qui était attendu
- Analyser les avantages et inconvénients de chaque algorithme
- Présenter ce que ce projet nous a permis d'apprendre
- Proposer de potentielles améliorations qui pourraient être apportées à l'avenir

6 Sandbox pour tester des trucs

Conclusion

Résumer les points clés du projet

Réiterer sur le but de ce projet et ce que la recherche dans ce domaine permet d'apporter à la science

Donner son impression générale par rapport à ce que nous a apporté le projet

Bibliographie

[1] Larousse : définition d'un labyrinthe. https://www.larousse.fr/dictionnaires/francais/labyrinthe/45804. Consulté le 08-04-2024.

[2] Herman Kern. *Through the Labyrinth*. Prestel, 2000. catalog item 103–104.

[3] David McCullough. *The Unending Mystery : A Journey Through Labyrinths and Mazes*. Pantheon, 2004.

[4] Gerald Alexanderson. Euler and königsberg's bridges: a historical view. *Bull. Amer. Math. Soc.*, 43:567, 2006.

[5] Euler. *Solution d'une question curieuse qui ne paraît soumise à aucune analyse*. Mémoires de l'Académie Royale des Sciences et Belles Lettres, 1759.

[6] Veritasium. The fastest maze-solving competition on earth (youtube). https://youtu.be/ZMQbHMgK2rw, 2023. Consulté le 08-04-2024.

[7] Site web du projet git. https://git-scm.com/, 2024. Consulté le 08-04-2024.

[8] Site web de github. https://github.com/, 2024. Consulté le 08-04-2024.

[9] Page github de linux. https://github.com/torvalds/linux, 2024. Consulté le 08-04-2024.

[10] Page github de python. https://github.com/python/cpython, 2024. Consulté le 08-04-2024.

[11] Page github de pygame. https://github.com/pygame-community/pygame-ce, 2024. Consulté le 08-04-2024.

[12] Documentation du projet black. https://black.readthedocs.io/en/stable/index.html, 2024. Consulté le 08-04-2024.

[13] Alyssa Coghlan Guido van Rossum, Barry Warsaw. Pep 8. https://peps.python.org/pep-0008/, 2013. Consulté le 08-04-2024.

Compte rendus hebdomadaires

Annexes

Annexe 1 : Code source et installation

Prérequis

Afin de pouvoir installer et exécuter le programme, il est nécessaire de disposer des éléments suivants :

1. Python 3.10 ou supérieur. Les versions antérieures n'ont pas été testées et ne sont pas garantie de fonctionner.
2. Des notions de base avec l'interface en ligne de commande de votre système d'exploitation.

Installation

Afin d'obtenir le code source du programme et l'installer sur votre machine, veuillez suivre les instructions suivantes (également décrites dans le fichier `README.md` du dépôt Git) :

1. Télécharger le code source du programme en suivant l'une des trois méthodes suivantes :
   - Cloner le dépôt Git à l'aide de la commande suivante :

     ```
     git clone https://github.com/HerbeMalveillante/ProjetPeiP24.git
     ```

   - Télécharger le code source sous forme d'archive ZIP en cliquant sur le bouton "Code" du dépôt Git, puis sur "Download ZIP".
   - Obtenir le code source à partir du fichier ZIP fourni avec ce rapport.
2. Ouvrir un terminal et se placer dans le dossier contenant le code source du programme.
3. Vérifier la version de Python utilisée dans le PATH à l'aide de la commande suivante :

   ```
   python --version
   ```

   Il est bon de noter que la commande permettant d'invoquer Python peut dépendre de votre système d'exploitation et/ou de votre installation de Python. Les commandes les plus courantes sont `python`, `python3` et `py`.

4. (optionnel) Créer un environnement virtuel Python à l'aide de la commande suivante :

```
python3 -m venv .envi

# activer l'environnement virtuel sur Unix
source .envi/bin/activate

# activer l'environnement virtuel sur Windows
.envi\Scripts\activate
```

5. Installer les dépendances du programme à l'aide de la commande suivante :

```
python3 -m pip install --upgrade pip
python3 -m pip install -r requirements.txt
```

   Veuillez noter que si une erreur liée à Pygame survient, il convient de vérifier que la version installée est bien la version "communauté" de Pygame (`pygame-ce`), et non la version officielle (`pygame`).

6. Lancer le programme à l'aide de la commande suivante :

```
python3 main.py
```

Projet Informatique : Labyrinthe

Résumé

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Mots-clés

Labyrinthe, ajouter, d'autres, mots, clés

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

Keywords

Labyrinth, add, other, keywords

Encadrant académique	Étudiants
Christophe Lenté	Pacôme Renimel–Lamiré Esteban Laurent