Rapport de Veille Technologique - Application Desktop de Création et Résolution de Labyrinthes

KOUALIAGNIGNI NJUNDOM GLORIA STELLA

B1 CYBERSECURITE

YNOV CAMPUS

11/06/2025

Introduction

Ce rapport de veille technologique a été élaboré dans le but d'explorer en profondeur l'univers des applications dédiées à la création et à la résolution de labyrinthes. Nous y analyserons les approches algorithmiques les plus pertinentes et identifierons les meilleures pratiques, ainsi que des pistes d'amélioration concrètes pour notre projet d'application desktop développé avec Electron. L'objectif est de concevoir une solution complète et intuitive, permettant aux utilisateurs de générer, de visualiser, de modifier, de résoudre et de gérer leurs propres labyrinthes, le tout complété par une interface d'administration robuste.

1. Étude des applications similaires existantes

Pour mieux cerner le paysage actuel, nous avons examiné différents types de plateformes et d'applications qui proposent des fonctionnalités liées aux labyrinthes.

1.1. Les générateurs de labyrinthes en ligne

Le web regorge de sites offrant des générateurs de labyrinthes, souvent très simples d'utilisation. Ces outils sont majoritairement basés sur JavaScript, s'exécutant directement dans le navigateur du client.

- **Quelques exemples courants :**
    - **MazeGenerator.net :** Ce site propose la création de labyrinthes de diverses tailles et styles, avec une option d'impression pratique. Sa simplicité est un atout, mais ses fonctionnalités se limitent à la génération et à l'impression, sans réelle interactivité au-delà.
    - **Divers "Online Maze Generator" :** On trouve de nombreux équivalents offrant des interfaces conviviales pour une génération visuelle instantanée.

- **Leurs points forts :** Une grande accessibilité, pas besoin d'installer quoi que ce soit.
- **Leurs lacunes :** Les fonctionnalités sont souvent rudimentaires. Il n'y a pas de sauvegarde des données utilisateurs, ni de module de résolution automatique intégré, et encore moins d'interface d'administration. Ce ne sont pas des applications complètes et autonomes.

1.2. Les applications ludiques et jeux vidéo de labyrinthes

Le concept de labyrinthe est un grand classique du jeu vidéo, qu'il s'agisse de titres en 2D ou en 3D. Beaucoup de ces jeux intègrent des algorithmes de génération procédurale pour renouveler l'expérience.

- **Exemples emblématiques :**
    - **Pac-Man :** Bien que ses labyrinthes soient préconçus, il reste l'archétype du jeu de parcours en labyrinthe.
    - **Jeux d'aventure avec donjons générés :** Des titres comme *The Binding of Isaac* ou *Hades* illustrent parfaitement l'utilisation d'algorithmes complexes pour créer des niveaux labyrinthiques uniques à chaque session de jeu.
- **Leurs points forts :** Une immersion totale pour l'utilisateur, une grande richesse visuelle et ludique.
- **Leurs lacunes :** Leur vocation première est le divertissement. L'utilisateur n'a généralement pas la main sur la création et la gestion des labyrinthes. Les algorithmes sont "sous le capot" et rarement exposés pour une personnalisation facile.

1.3. Les bibliothèques et frameworks de génération/résolution

Pour les développeurs, il existe des bibliothèques open-source dans différents langages (comme Python ou JavaScript) qui fournissent les briques nécessaires pour générer et résoudre des labyrinthes. Elles sont la base technique pour intégrer ces fonctionnalités dans des applications plus vastes.

- **Exemples pertinents :**
    - **js-maze-generator (JavaScript) :** Une bibliothèque qui propose diverses implémentations d'algorithmes de génération.
    - **pathfinding.js (JavaScript) :** Une solution très complète pour la recherche de chemins sur des grilles, parfaitement adaptée à la résolution de labyrinthes, intégrant des algorithmes comme A*, Dijkstra, BFS, DFS, etc.

- **Leurs points forts :** Des algorithmes éprouvés et robustes, une grande flexibilité pour l'intégration, et une base solide pour démarrer un développement.
- **Leurs lacunes :** Elles nécessitent un travail conséquent pour construire une interface utilisateur conviviale et fonctionnelle autour d'elles.

2. Décryptage des algorithmes de génération et de résolution

Chaque algorithme de labyrinthe possède sa propre "personnalité", influençant les performances, la complexité du labyrinthe et son aspect visuel.

2.1. Les algorithmes de Génération de Labyrinthes

a) L'algorithme de recherche en profondeur (DFS - Depth-First Search), ou "Recursive Backtracker"

- **Comment ça marche :** Il part d'un point choisi au hasard et creuse un chemin le plus loin possible. Quand il ne peut plus avancer, il fait marche arrière (backtrack) jusqu'à trouver une nouvelle voie.
- **Ses atouts :**
    - **Simplicité :** Il est relativement aisé à coder, que ce soit en utilisant la récursivité ou une structure de pile.
    - **Chemins sinueux :** Il a tendance à créer des chemins uniques et peu de culs-de-sac courts, ce qui peut rendre le labyrinthe plus stimulant à résoudre manuellement.
    - **Labyrinthes "parfaits" :** Il garantit un chemin unique entre n'importe quel point.
- **Ses points faibles :**
    - **Performances pour les très grands labyrinthes :** La récursivité profonde peut s'avérer gourmande en mémoire.
    - **Manque de diversité :** Les labyrinthes générés sont souvent des structures arborescentes, avec peu de boucles, ce qui peut simplifier la tâche pour certains usages.
    - **"Autoroutes" :** Parfois, il peut créer de très longues lignes droites, ce qui réduit la difficulté.

b) L'algorithme de Prim

- **Comment ça marche :** Inspiré de l'algorithme de Prim pour les arbres couvrants minimaux. Il débute avec une seule cellule et "étend" le labyrinthe en ajoutant des murs aléatoirement à partir des frontières des cellules déjà incluses.

- **Ses atouts :**
    - **Labyrinthes complexes et "branchus" :** Il génère des structures plus denses avec une multitude de culs-de-sac.
    - **Esthétique :** Les labyrinthes sont souvent visuellement plus équilibrés, avec une répartition plus homogène des chemins.
    - **Vitesse :** Efficace pour des grilles de grande taille.
- **Ses points faibles :**
    - **Légèrement plus complexe à mettre en œuvre que DFS.**
    - **Solutions "évidentes" :** Peut parfois générer des chemins plus directs, moins "serpentins" que DFS.

c) L'algorithme de Kruskal

- **Comment ça marche :** Il part du principe que toutes les cellules sont isolées. Il connecte ensuite des paires de cellules de manière aléatoire, à condition que cela ne crée pas de boucle. Il s'appuie sur une structure de données appelée "Union-Find".
- **Ses atouts :**
    - **Vitesse :** Très performant pour les grandes dimensions.
    - **Chemins variés :** La structure des chemins est moins prévisible, ce qui peut être un avantage.
- **Ses points faibles :**
    - **Complexité d'implémentation :** Nécessite une bonne maîtrise de l'algorithme Union-Find.
    - **"Îlots" :** Peut occasionnellement créer des chemins ou des zones qui semblent isolées du reste du labyrinthe.

Notre recommandation pour le projet : Pour la gestion de la difficulté (de 1 à 10), privilégier les algorithmes DFS ou Prim semble être un excellent point de départ. Ils offrent un bon compromis entre la simplicité de mise en œuvre et la richesse des labyrinthes générés. Pour moduler la difficulté, on pourrait par exemple ajuster la probabilité de "backtracking" pour DFS, ou contrôler le nombre de culs-de-sac avec Prim. À plus long terme, l'algorithme de Wilson pourrait être exploré pour des labyrinthes "parfaits" et "sans biais" (où aucun chemin n'est intrinsèquement favorisé).

2.2. Les algorithmes de Résolution de Labyrinthes

Résoudre un labyrinthe, c'est en fait trouver le chemin le plus efficace dans un graphe.

a) La Recherche en largeur (BFS - Breadth-First Search)

- **Comment ça marche :** Cet algorithme explore tous les voisins d'un point donné avant de passer aux voisins du niveau suivant. Il est garanti de trouver le chemin le plus court en termes de nombre de pas.
- **Ses atouts :**
    - **Optimalité :** Il découvre toujours le chemin le plus direct.
    - **Simplicité :** Son implémentation est facile, basée sur une structure de file (queue).
    - **Visualisation claire :** L'exploration "couche par couche" est intuitive à suivre.
- **Ses points faibles :**
    - **Consommation mémoire :** Peut stocker un grand nombre de nœuds dans la file, surtout pour des labyrinthes complexes avec beaucoup de chemins possibles.

b) La Recherche en profondeur (DFS - Depth-First Search)

- **Comment ça marche :** Il explore un chemin aussi loin que possible, puis revient en arrière si nécessaire.
- **Ses atouts :**
    - **Moins gourmand en mémoire que BFS :** Utilise une pile (stack) et se concentre sur une seule branche à la fois.
    - **Simplicité d'implémentation :** Peut être codé récursivement ou de manière itérative.
- **Ses points faibles :**
    - **Non-optimalité :** Il ne garantit pas de trouver le chemin le plus court. Pour la visualisation, cela peut être moins satisfaisant si le chemin n'est pas le plus direct.

c) L'algorithme A* (A-star)

- **Comment ça marche :** C'est un algorithme de recherche de chemin intelligent qui utilise une "heuristique" pour estimer la distance restante vers la destination. Il combine la garantie d'optimalité de Dijkstra avec la rapidité de la recherche gloutonne (grâce à l'heuristique).
- **Ses atouts :**

    - **Optimalité garantie :** Il trouve toujours le chemin le plus court (à condition que l'heuristique soit bien choisie).
    - **Efficacité :** Bien plus rapide que BFS/DFS pour les grands labyrinthes, car il se dirige intelligemment vers la cible, réduisant l'espace de recherche.
    - **Idéal pour une visualisation optimisée du chemin.**
- **Ses points faibles :**
    - **Complexité d'implémentation :** Nécessite une structure de file de priorité et une fonction heuristique.
    - **Dépendance à l'heuristique :** Une heuristique mal conçue peut le rendre aussi lent que BFS.

Recommandation pour le projet : Pour l'affichage visuel du chemin de résolution, l'algorithme A* est sans conteste le plus pertinent. Il assure le chemin le plus court et offre d'excellentes performances. Si la complexité d'A* est un obstacle initial, BFS reste une alternative solide, garantissant également l'optimalité et une visualisation claire.

3. Pistes d'améliorations et fonctionnalités additionnelles

Au-delà des exigences de base, nous avons identifié plusieurs pistes pour enrichir l'expérience utilisateur et la robustesse de l'application :

- **Visualisation animée de la génération :** Imaginez pouvoir suivre le processus de création du labyrinthe étape par étape ! Cela ajouterait une dimension éducative et captivante. Il suffirait d'intégrer des pauses ou des contrôles de vitesse dans l'algorithme.
- **Un choix d'algorithmes de génération :** Proposer à l'utilisateur de choisir parmi différents algorithmes (DFS, Prim, Kruskal) lui permettrait de comprendre l'impact de chacun sur la structure et la difficulté du labyrinthe. Une belle opportunité de découverte !
- **Un éditeur de labyrinthes plus poussé :** Au-delà des fonctions CRUD classiques, pourquoi ne pas permettre à l'utilisateur de dessiner ou d'effacer des murs directement sur la grille ? Un "mode dessin" intuitif ouvrirait de nouvelles possibilités créatives.
- **Personnalisation visuelle étendue :** En plus des thèmes existants (comme votre "Alien Cosmique" déjà si prometteur !), offrir des options de personnalisation des couleurs, des textures de murs, et même de l'apparence du chemin de résolution.
- **Partage des créations :** Permettre aux utilisateurs de partager leurs labyrinthes avec d'autres, par exemple via un code unique ou en les rendant accessibles

publiquement via l'interface d'administration. Une fonctionnalité d'export/import au format JSON serait parfaite pour cela.

- **Statistiques détaillées de résolution :** Après avoir résolu un labyrinthe, afficher des données comme le temps écoulé, le nombre de pas effectués, ou comparer la performance de l'utilisateur avec le chemin optimal trouvé automatiquement.
- **Historique des tentatives :** Conserver un historique des tentatives de résolution de chaque utilisateur pour un labyrinthe donné.
- **Une interface utilisateur hyper-réactive :** Il est crucial que l'application reste fluide, même lors de la génération ou de la résolution de labyrinthes très complexes. L'utilisation de "web workers" dans Electron est la solution idéale pour décharger les calculs lourds en arrière-plan et garantir une expérience sans accroc.

4. Les outils et ressources clés pour le projet

Pour mener à bien ce projet, nous nous appuierons sur les ressources suivantes :

4.1. Forums et Communautés d'entraide

- **Stack Overflow :** Une mine d'or pour trouver des solutions à des problèmes de programmation spécifiques, notamment avec les technologies clés (Electron, Node.js, SQLite, bcryptjs, JWT) et les algorithmes.
- **GitHub :** Indispensable pour trouver des exemples de projets Electron, des implémentations d'algorithmes de labyrinthes (en JavaScript, Python), et pour explorer le code de bibliothèques utiles. C'est aussi la plateforme privilégiée pour le travail collaboratif en groupe via Git.
- **Discord / Reddit (communautés de développeurs) :** Idéals pour des questions plus générales ou pour obtenir des retours sur des choix d'architecture ou d'algorithmes.

4.2. Documentations et tutoriels de référence

- **Documentation officielle Electron :** La référence incontournable pour maîtriser le fonctionnement d'Electron, ses API (processus principal, processus de rendu) et les bonnes pratiques de développement desktop.
- **MDN Web Docs (Mozilla Developer Network) :** La bible pour tout ce qui concerne HTML, CSS, JavaScript. Essentiel pour la création de l'interface.
- **Documentation Node.js :** Pour toutes les questions liées à l'environnement Node.js (modules, gestion des fichiers, etc.).

- **Documentation SQLite (sqlite.org) :** Pour comprendre les requêtes SQL et les spécificités de cette base de données légère.
- **Documentations de bcrypt.js et jsonwebtoken :** Pour une implémentation rigoureuse et sécurisée de l'authentification.
- **Sites spécialisés en algorithmique (ex: GeeksforGeeks, ou les ouvrages comme "Introduction to Algorithms" de Cormen et al.) :** Pour approfondir les rouages des algorithmes de labyrinthes.

4.3. Outils et Bibliothèques techniques spécifiques

- **Electron :** Le cœur de notre application desktop.
- **SQLite (via des modules Node.js comme sqlite3 ou better-sqlite3) :** Notre choix pour la base de données embarquée. better-sqlite3 est souvent préféré pour ses performances et son API synchrone simplifiée.
- **bcryptjs :** Pour hacher les mots de passe de manière sécurisée.
- **jsonwebtoken (JWT) :** Pour la gestion des sessions utilisateur après l'authentification.
- **Bibliothèques de génération/résolution de labyrinthes (JavaScript) :**
    - pathfinding.js : Très utile pour la résolution.
    - Pour la génération, une implémentation maison des algorithmes peut être plus formatrice et flexible, mais les bibliothèques existantes peuvent servir d'excellentes références.
- **Canvas API (HTML5) :** L'outil parfait pour dessiner dynamiquement les labyrinthes et visualiser la résolution, offrant une grande flexibilité.

4.4. Outils d'aide à la conception et au développement

- **Figma / Adobe XD :** Des logiciels indispensables pour la conception de l'interface utilisateur (UI/UX) et la création des maquettes
- **ESLint / Prettier :** Pour assurer une qualité de code irréprochable et une lisibilité constante en JavaScript, HTML et CSS.
- **Git / GitHub Desktop (ou ligne de commande) :** Absolument essentiel pour la collaboration en équipe, le contrôle de version et le maintien d'un dépôt propre et structuré.

Conclusion

Ce rapport de veille souligne l'importance d'une sélection judicieuse des algorithmes pour la génération et la résolution des labyrinthes, afin de trouver le juste équilibre entre complexité, performance et une expérience utilisateur agréable. L'intégration des technologies imposées (Electron, SQLite, bcryptjs, JWT) fournira une base solide et sécurisée à notre application. Les propositions d'améliorations visent à enrichir l'application en profondeur, la rendant encore plus attrayante et fonctionnelle. Enfin, l'utilisation éclairée des sources et des outils identifiés sera la clé du succès pour garantir la qualité et la pertinence du produit final.