Estimation de la densité 2D par noyau dans l'espace urbain piéton

Sujet de projet d'analyse spatiale

Par

Mortada BOUZAFFOUR, Antoine HALTER-MINGAUD

ENSG Géomatique

2ème année du cycle ingénieur
Encadré par Paul Chapron

Janvier 2025

Pour avoir accès au code complet du projet, se rendre sur : Code GitHub

Résumé

L'estimation de densité par noyau (appelée KDE pour Kernel Density Estimation) est une méthode statistique qui sert à estimer la densité de probabilité d'une variable aléatoire en tout point d'un espace. À partir d'un échantillon de points d'intérêts localisés, la KDE permet de connaître la densité de présence de ces points d'intérêts partout ailleurs, ce qui «lisse» l'estimation, comme dans l'exemple ci-dessous :

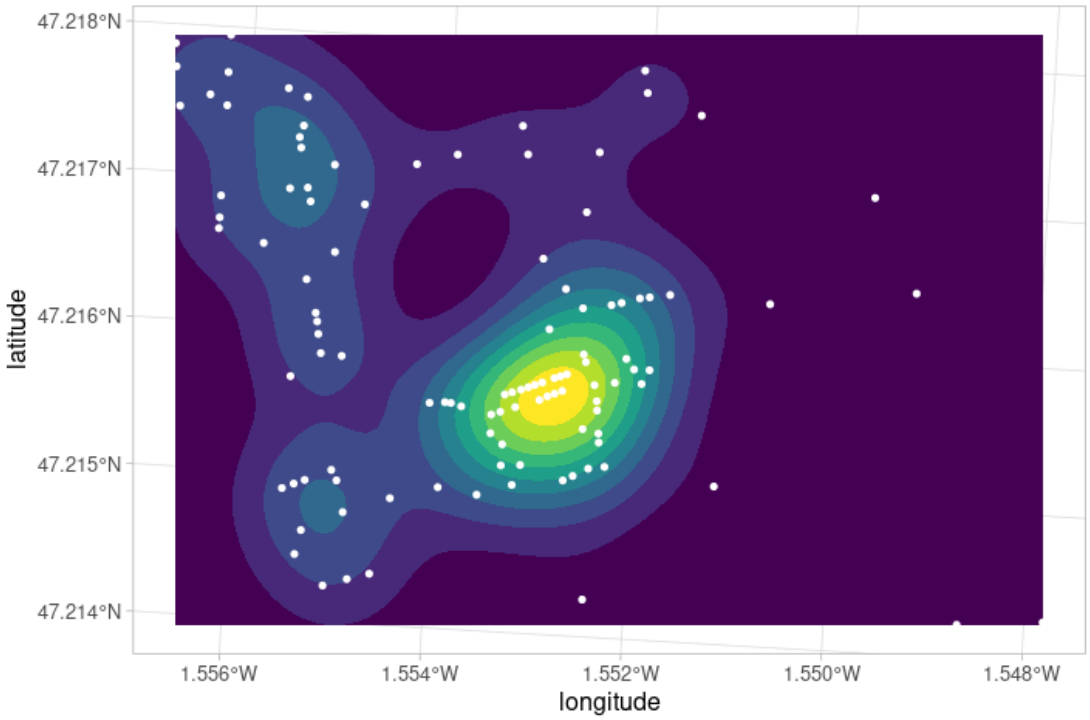


Figure 1: Estimation KDE d'un semis de points d'intérêts (en blanc) dans l'espace de leur emprise, discrétisée en 10 niveaux de densité. Plus la densité est importante, plus la teinte tire vers le jaune.

Problématique

La plupart des KDE sont réalisées dans un espace 2D continu, sans obstacles ni trous, et leur résultat est retourné sous la forme d'un maillage régulier, dont chaque maille contient un scalaire représentant la densité de la variable estimée : c'est un raster de densité. Le raster obtenu peut être vu comme un lissage de la probabilité de présence des points, de façon à ce que cette probabilité ne soit pas nulle hors des points d'intérêt eux-mêmes. Ce lissage utilise la distance entre les points d'intérêt et tout autre point de l'espace, de façon à ce que la probabilité de présence augmente lorsqu'on s'approche des points d'intérêt, et un paramètre appelé bande passante (ou bandwidth) qui contrôle l'intensité du lissage. Par défaut, la distance utilisée est euclidienne, ce qui se justifie lorsqu'on cherche à estimer une densité en espace ouvert, typiquement dans les espaces naturels, ou sans tenir compte des obstacles, par exemple lorsqu'on étudie la densité à des échelles plus larges, départementale par exemple. Mais lorsqu'on s'intéresse à la perception que peut avoir un humain de la proximité aux points d'intérêt, dans un espace urbain, typiquement un centre-ville, la distance euclidienne n'est plus satisfaisante : ces espaces sont 1contraints, moins ouverts, comportent des obstacles. Par ailleurs, à une échelle plus fine, comme celle d'un quartier, il peut être intéressant de restreindre ou d'adapter la densité obtenue par KDE à des distances telles que les perçoivent un piéton, en intégrant les détours, et les trajectoires particulières de la marche au centre ville. Il devient alors intéressant d'observer comment varie la densité estimée par KDE lorsqu'on utilise des distances plus réalistes en milieu urbain, comme celles obtenues par des algorithmes de pathfinding [1], ou des distances de plus court chemin sur le graphe «marchable» : trottoirs, passages piétons, places ouvertes, raccourcis, etc.

Déroulement du projet

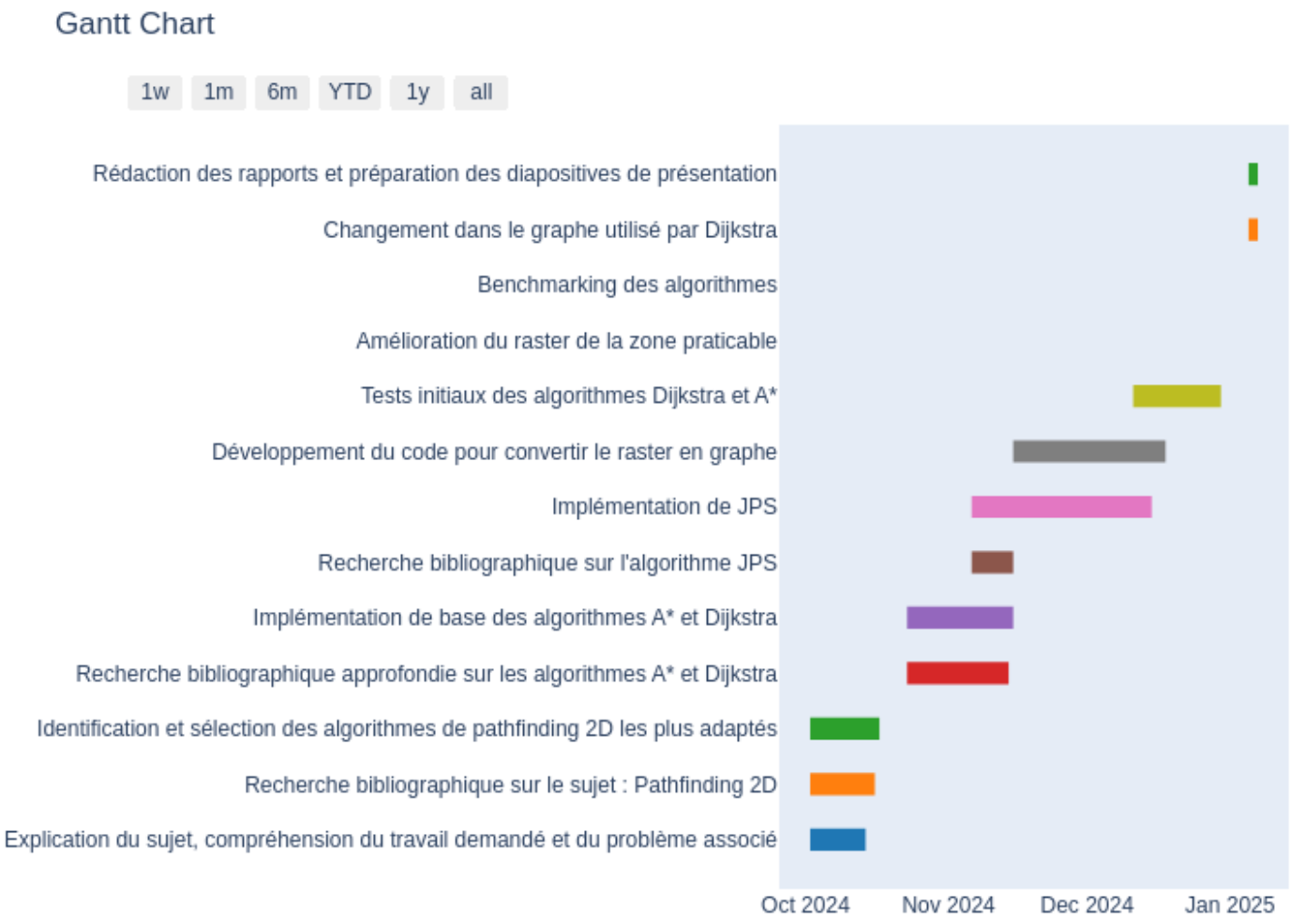


Figure 2: Diagramme de Gantt du projet

Remerciements

Nous tenons à remercier Monsieur Paul Chapron, chercheur au LASTIG, qui nous a encadré tout au long de ce projet et qui nous a fait partager ses connaissances académiques en informatique et en sciences géographiques. Qu'il soit aussi remercié pour sa disponibilité et pour les nombreux encouragements qu'il nous a prodigués.

Enfin nous remercions l'équipe de l'ENSG Géomatique d'avoir mis à disposition ce projet de recherche scientifique.

Author's declaration

Je soussigné, Mortada Bouzaffour, déclare sur l'honneur que ce rapport est le fruit d'un travail de groupe, et que nous n'avons ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui afin de la faire passer pour la nôtre. Toutes les sources d'information utilisées (supports papier, audiovisuels et numériques) ainsi que les citations d'auteurs ont été mentionnées conformément aux usages en vigueur. Nous sommes conscients que le fait de ne pas citer une source, ou de ne pas la citer de manière claire et complète, constitue un plagiat. Nous reconnaissons que le plagiat est considéré comme une faute grave au sein de l'Université et qu'il peut faire l'objet de sanctions sévères.

SIGNED: ..... DATE: .....

List of Tables

List of Figures

Table of Contents

Chapter 1

Estimation par Densité Noyau (KDE)

L'estimation par densité noyau, ou *Kernel Density Estimation* (KDE), est une méthode non paramétrique permettant d'estimer la densité de probabilité d'une variable aléatoire à partir d'un échantillon de données. Elle repose sur l'hypothèse que la densité peut être approximée par une somme pondérée de fonctions *noyau*, chacune centrée sur une donnée de l'échantillon.

1.1 Définition formelle

Soit $\{x_1, x_2, \ldots, x_n\}$ un échantillon de taille n extrait d'une variable aléatoire continue X avec une densité de probabilité inconnue $f(x)$. L'estimation par densité noyau de $f(x)$ est donnée par l'expression :

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right),$$

où :

- $K(\cdot)$ est la fonction noyau, qui est une fonction symétrique intégrant à 1, souvent choisie comme une densité de probabilité (exemple : gaussienne, uniforme, etc.),
- $h > 0$ est le paramètre de lissage, appelé *largeur de bande* (*bandwidth* en anglais), qui contrôle la régularisation de l'estimation.

Propriétés du noyau K

La fonction noyau $K(x)$ doit vérifier les propriétés suivantes :

1. Symétrie : $K(x) = K(-x)$,
2. Intégration à 1 : $\int_{-\infty}^{\infty} K(x)\,dx = 1$,
3. Habituellement, $K(x)$ est une fonction décroissante lorsque $|x|$ augmente.

Un exemple courant de noyau est le noyau gaussien, défini par :

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}.$$

1.2 Choix de la largeur de bande h

Le paramètre h joue un rôle crucial dans l'estimation. Un h trop petit conduit à une estimation trop bruitée (*overfitting*), tandis qu'un h trop grand lisse excessivement les données (*underfitting*). Il existe plusieurs méthodes pour choisir h, notamment :

- Les règles empiriques basées sur l'écart-type ou l'écart interquartile des données,
- Les techniques d'optimisation, telles que la minimisation de l'erreur quadratique intégrée (*Integrated Mean Square Error, IMSE*).

1.3 Applications

Dans notre sujet, la KDE est utilisée en analyse spatiale pour :

- La visualisation de la distribution d'un ensemble de données
- La détection d'anomalies
- Identification des zones d'activité piétonne

Chapter 2

Etat de l'art sur la recherche de chemin (pathfinding) 2D

Le pathfinding en 2D (ou recherche de chemin en 2D) est un problème d'intelligence artificielle consistant à déterminer le chemin le plus court ou le plus efficace entre deux points dans un environnement bidimensionnel, tel qu'un jeu, une simulation 2D, ou un système d'information géographique. Un tel algorithme de recherche de chemin vise à trouver un itinéraire qui évite les obstacles, minimise la distance ou le temps, et satisfait des contraintes spécifiques.

2.1 Quelques principes et approches clés sur le pathfinding 2D

- **Algorithmes de base** : Les algorithmes de base pour le pathfinding 2D incluent Dijkstra, A* (A-star), et Floyd-Warshall.
- **Heuristiques** : en informatique et en mathématique, une heuristique est une méthode ou une stratégie de calcul approximative permettant de résoudre un problème de manière rapide et efficace, sans garantir une solution optimale. L'objectif principal est de réduire le temps de calcul et la complexité d'un problème en donnant une solution acceptable.

  Dans le contexte du pathfinding 2D, certaines heuristiques guident les algorithmes pour trouver un chemin optimal.

- **Graphe et réseau** : Un problème de pathfinding 2D est souvent représenté à l'aide d'un graphe, où les nœuds représentent les positions dans l'environnement et les arêtes représentent les mouvements possibles.
- **Librairies et outils** : Il existe plusieurs librairies pour le pathfinding 2D, notamment *A* Pathfinding Project* pour Unity et *Pathfinding* pour Python.

2.2 Exemples d'application

- **Jeux vidéo** : les algorithmes de pathfinding 2D sont utilisés pour déplacer les personnages non jouables, les ennemis et les objets.
- **Simulateurs de robotique** : pour faire se déplacer les robots tout en évitant les obstacles.
- **Systèmes d'information géographique et cartes interactives** : le pathfinding est souvent utilisé pour la planification et le calcul des itinéraires.

2.3 Défis et limitations

- **Complexité de l'environnement** : Les environnements avec des obstacles irréguliers rendent le calcul du chemin difficile, et l'algorithme doit souvent recalculer ou ajuster le chemin en temps réel.
- **Coût de calcul élevé** : Dans des environnements étendus, les algorithmes de pathfinding sont plus exigeants en ressources, augmentant le temps de calcul.
- **Limites des algorithmes heuristiques** : Les heuristiques peuvent ignorer des chemins valides ou se concentrer trop sur des solutions locales.

Chapter 3

Sélection de trois algorithmes efficaces

3.1 Liste d'algorithmes de pathfinding

- *A* (A-star Algorithm)*
- *Dijkstra's Algorithm*
- *Bellman-Ford Algorithm*
- *Floyd-Warshall Algorithm*
- *Greedy Best-First Search*
- *Breadth-First Search (BFS)*
- *Depth-First Search (DFS)*
- *Johnson's Algorithm*
- *Bi-directional A-Star*
- *Jump Point Search (JPS)*

Cette liste inclut les algorithmes les plus répondus du pathfinding sur des grilles 2D, nous allons ensuite mettre en question l'efficacité de 3 algorithmes: les plus adaptés pour notre problèmatique.

3.2 Critères de sélection

Le choix de trois algorithmes est basé sur plusieurs critères. Le tableau ci-dessous compare certains de ces algorithmes sur ces critères.

Critères	A*	Dijkstra	JPS
Adaptation aux grilles 2D	Très bien adapté	Bien adapté	Très bien adapté
Coût et Performance (temps)	Optimal avec heuristique	Optimal mais plus lent	Optimal et très rapide
Utilisation d'heuristique	Oui	Non	Oui (optimisé)
Gestion des obstacles	Efficace	Efficace mais exhaustive	Optimisé pour obstacles
Capacité à trouver le chemin optimal	Toujours si heuristique correcte	Toujours	Toujours

Table 3.1: Tableau comparatif des algorithmes de recherche de chemin

3.3 Les 3 algorithmes choisis :

- *Dijkstra :* Principe : Explore tous les chemins en calculant le coût le plus bas depuis un point de départ jusqu'à chaque autre nœud.
- *A* / A* Bidirectionnel :* Principe : utilise une fonction heuristique pour explorer les chemins les plus prometteurs, en utilisant le coût réel et une estimation de la distance restante.
- *Jump Point Search (JPS):* Principe : JPS est une optimisation de A* pour les grilles 2D, en sautant les nœuds inutiles sur les lignes droites et se concentrant uniquement sur les points de décision critiques.

Chapter 4

Implémentation des trois algorithmes

4.1 Algorithme de Dijkstra [3]

Cet algorithme permet de trouver le chemin optimal dentre deux points sur un graphe pondéré, non orienté ou orienté, avec des coûts positifs. Il fonctionne en explorant tous les nœuds d'un graphe en calculant le coût le plus bas à chaque itération. Cet algorithme a été introduit par Edsger W. Dijkstra en 1956.

4.2 Concepts Clés de l'Algorithme de Dijkstra

L'algorithme de Dijkstra repose sur plusieurs concepts clés :

- **Distance minimale** : Distance minimale entre un nœud de départ et chaque nœud du graphe.
- **Poids des arêtes** : Les arêtes du graphe sont pondérées, c'est-à-dire que chaque arête a un poids ou un coût associé.
- **Liste des nœuds non traités** : Les nœuds sont initialement marqués comme non traités. Au fur et à mesure que l'algorithme avance, les nœuds sont marqués comme traités une fois leur distance minimale trouvée.
- **Liste des nœuds traités** : Cette liste stocke les nœuds pour lesquels le plus court chemin depuis le nœud de départ est déterminé.

- **Table des distances** : Une structure qui garde une trace de la distance minimale connue pour chaque nœud depuis le nœud de départ.

4.3 Fonctionnement de l'Algorithme de Dijkstra

L'algorithme de Dijkstra fonctionne en étapes successives pour parcourir le graphe à partir du nœud de départ et trouver les distances minimales pour atteindre chaque nœud. Voici le déroulement en étapes :

Étape 1 : Initialisation

- Attribuer à chaque nœud une distance infinie (∞), sauf pour le nœud de départ qui reçoit une distance de 0.
- Placer tous les nœuds dans une liste de nœuds non traités.

Étape 2 : Sélection du Nœud Courant

- Parmi les nœuds non traités, sélectionner le nœud avec la distance minimale comme nœud courant.

Étape 3 : Mise à Jour des Distances

- Pour chaque voisin du nœud courant, calculer la distance potentielle en passant par ce nœud courant.
- Si la distance trouvée est inférieure à la distance déjà enregistrée pour ce voisin, mettre à jour la distance de ce voisin.

Étape 4 : Marquer le Nœud comme Traité

- Une fois les distances des voisins du nœud courant mises à jour, marquer le nœud courant comme traité.

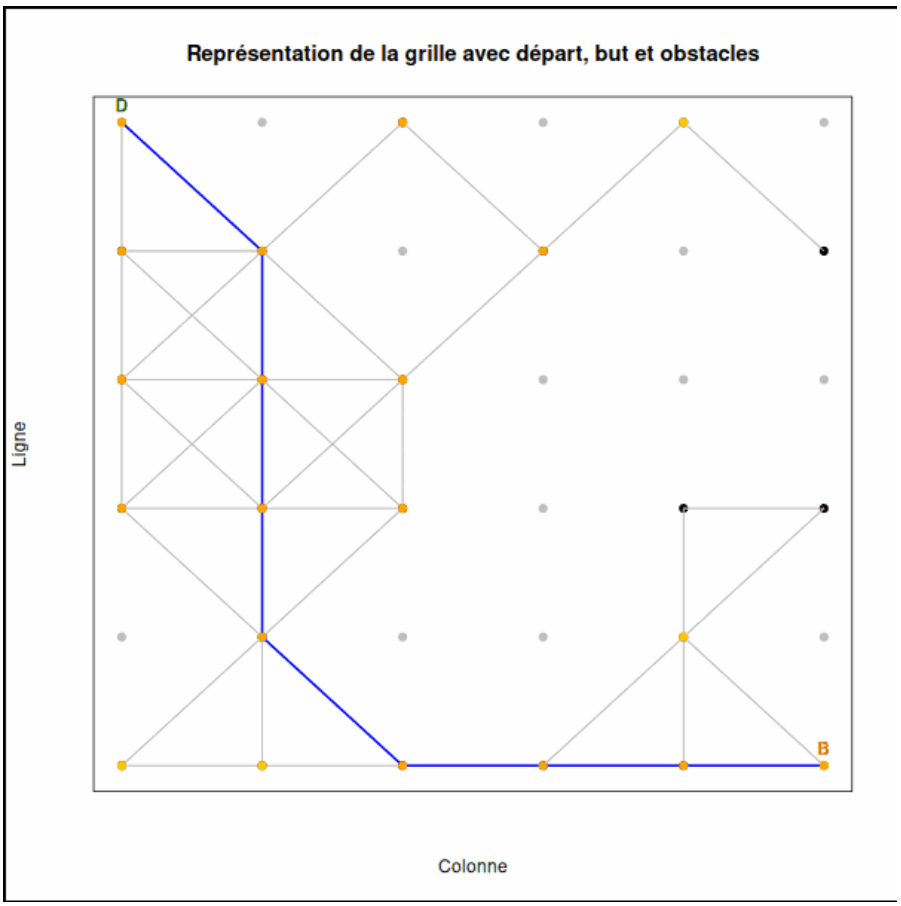


Figure 4.1: Capture d'écran de l'animation de l'exécution de Djikstra

Étape 5 : Répéter jusqu'à ce que tous les Nœuds soient Traités

- Répéter les étapes 2 à 4 jusqu'à ce que tous les nœuds soient traités ou jusqu'à ce que la distance minimale atteigne le nœud de destination, si seul le chemin vers un nœud spécifique est recherché.

4.4 Pseudo-code de l'Algorithme de Dijkstra

```
1 def Dijkstra(graph, source):
2     # Create a distance table with infinite values for each node
3     distance = {node: float('inf') for node in graph.nodes}
4     distance[source] = 0
5     unvisited_nodes = set(graph.nodes)
```

```
    predecessor_table = {}

    while unvisited_nodes:
        # Select the node with the smallest distance in unvisited_nodes
        current_node = min(unvisited_nodes, key=lambda node: distance[node])
        unvisited_nodes.remove(current_node)

        for neighbor in graph.neighbors(current_node):
            new_distance = distance[current_node] + weight(current_node,
            ↪  neighbor)
            if new_distance < distance[neighbor]:
                distance[neighbor] = new_distance
                predecessor_table[neighbor] = current_node

    return distance, predecessor_table

def find_path(predecessor_table, destination):
    path = []
    node = destination
    while node is not None:
        path.insert(0, node)  # Add node to the start of the path
        node = predecessor_table.get(node)  # Move to the predecessor

    return path
```

Code en R : Pour consulter le code , fichier : `Code/DIJKSTRA.R`.

4.5 Algorithme A*

L'algorithme A* [4] combine la recherche en largeur et la recherche en coût uniforme. Il utilise une fonction heuristique pour estimer le coût restant jusqu'à l'objectif.

L'algorithme A* est considéré comme l'un des meilleurs algorithmes de *pathfinding*. Il est généralement utilisé pour trouver le plus court chemin dans un graphe en combinant deux concepts principaux de *pathfinding* :

La recherche en largeur (BFS) : La recherche en largeur d'abord (BFS - Breadth-First Search) consiste à explorer un graphe en largeur, c'est-à-dire qu'il parcourt d'abord tous les voisins immédiats du nœud de départ avant de s'aventurer plus loin. Pour ce faire, BFS utilise une file (queue) pour garder la trace des nœuds à visiter dans l'ordre où ils ont été découverts.

La recherche en coût uniforme (Dijkstra) : *But :* Trouver le chemin le moins coûteux (ou le plus court) entre un point de départ et tous les autres points dans un graphe pondéré. Dijkstra explore les chemins en minimisant le coût cumulé depuis le point de départ vers les autres nœuds. Le terme "uniforme" signifie ici que l'algorithme traite chaque nœud en fonction du coût réel accumulé, et non en fonction d'une estimation (comme c'est le cas pour A*, qui utilise une heuristique).

4.6 Concepts clés dans A*

- **Graphe** : Une grille qui peut représenter, par exemple, une maquette de jeu 2D, une carte, ou un système de navigation GPS, où chaque nœud représente un emplacement.
- **Nœuds** : Un nœud est une position dans l'espace que l'algorithme va explorer. Chaque nœud possède un coût pour être atteint.
- **Heuristique** : Une fonction qui estime le coût restant pour atteindre l'objectif à partir d'un nœud donné. Elle permet à l'algorithme de choisir le chemin qui semble le plus prometteur.
- $g(n)$: Le coût exact du chemin allant du point de départ jusqu'au nœud actuel n.
- $h(n)$: L'estimation (heuristique) du coût restant pour aller du nœud actuel n à l'objectif. Elle ne doit jamais surestimer le coût réel pour garantir que A* trouve le chemin optimal.
- $f(n)$: La somme des deux coûts, $f(n) = g(n) + h(n)$. C'est cette valeur qui est utilisée pour déterminer quel nœud explorer ensuite.

4.7 Principe de fonctionnement

Initialisation :

- Placez le point de départ dans une file appelée *open set* (ensemble ouvert), qui contient les nœuds à explorer.
- Définissez $g(\text{start}) = 0$ (le coût de départ est 0) et calculez $f(\text{start}) = g(\text{start}) + h(\text{start})$.

Boucle principale : Tant que l'open set n'est pas vide :

- Sélectionnez le nœud n ayant la plus petite valeur dans l'open set (le nœud le plus prometteur).
- Si ce nœud est le nœud objectif, l'algorithme est terminé : vous avez trouvé le chemin optimal.
- Sinon, déplacez le nœud n de l'open set vers un ensemble appelé *closed set* (ensemble fermé), ce qui signifie qu'il a été exploré.
- Explorez tous les voisins du nœud n :
  - Pour chaque voisin v, calculez le coût pour aller au voisin en passant par le nœud n.
  - Si ce coût est inférieur au coût précédent calculé pour v, mettez à jour le chemin et ajoutez le voisin v à l'open set si ce n'est pas déjà fait.

Reconstruction du chemin : Une fois l'objectif atteint, l'algorithme remonte le chemin en utilisant les nœuds parents de chaque nœud pour obtenir le chemin complet du départ à l'objectif.

4.8 Les heuristiques les plus utilisées dans A*

La qualité de l'heuristique [5] $h(n)$ a un impact considérable sur l'efficacité de l'algorithme A*. Les heuristiques les plus couramment utilisées sont les suivantes :

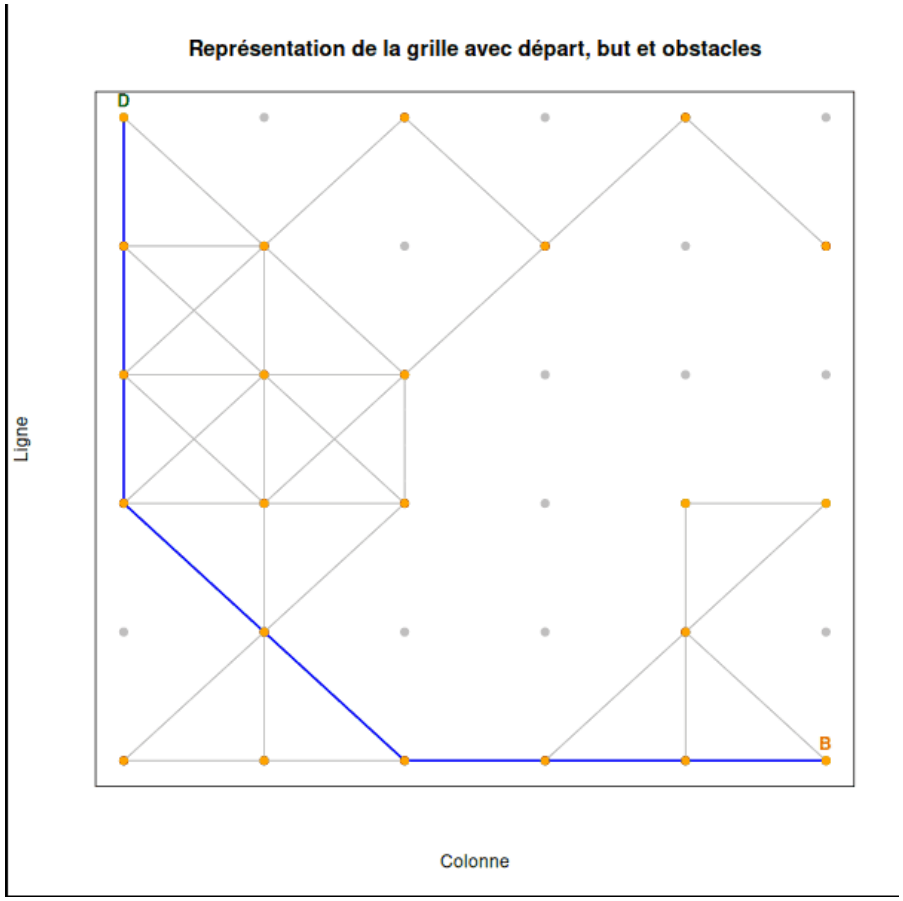


Figure 4.2: Capture d'écran de l'animation de l'exécution de A*

- **Distance de Manhattan** : Utilisée dans les grilles où le mouvement est uniquement permis horizontalement ou verticalement (comme dans les jeux ou cartes 2D).
- **Distance Euclidienne** : Utilisée lorsque le mouvement est permis dans toutes les directions (2D ou 3D).
- **Distance diagonale** : Utilisée dans des grilles où le déplacement en diagonale est permis.

4.9 Pseudo-code de l'Algorithme A*

```
def A_star(graph, start, goal):
    # Initialization
    open_list = priority_queue()  # Nodes to explore
```

```
    closed_set = set()           # Explored nodes

    g_score = {start: 0}
    f_score = {start: g_score[start] + heuristic(start, goal)}

    open_list.add(start, f_score[start])

    while not open_list.is_empty():
        # Select the node with the lowest f_score
        current_node = open_list.extract_min()

        # If the goal is reached, reconstruct the path
        if current_node == goal:
            return reconstruct_path(predecessor, current_node)

        # Otherwise, move current_node from open_list to closed_set
        closed_set.add(current_node)

        for neighbor in graph.neighbors(current_node):
            if neighbor in closed_set:
                continue

            # Calculate the cost to reach neighbor via current_node
            temp_g_score = g_score[current_node] + cost(current_node,
            ↪  neighbor)

            if neighbor not in open_list:
                open_list.add(neighbor, temp_g_score + heuristic(neighbor,
                ↪  goal))
            elif temp_g_score >= g_score.get(neighbor, float('inf')):
                continue  # This path is not better

            # Update the optimal path and g_score
            predecessor[neighbor] = current_node
            g_score[neighbor] = temp_g_score
            f_score[neighbor] = g_score[neighbor] + heuristic(neighbor,
            ↪  goal)
```

```
38
39              # Update the priority of neighbor in open_list
40              open_list.update(neighbor, f_score[neighbor])
41
42      return "Failure"  # No path found
43
44  def reconstruct_path(predecessor, node):
45      path = []
46      while node is not None:
47          path.append(node)
48          node = predecessor.get(node)
49      return path[::-1]
```

Code en R : Pour consulter le code , fichier : `Code/A_star.R`.

4.10 Algorithme JPS (Jump Point Search)

L'algorithme Jump Point Search (JPS) [6] est une version améliorée de l'algorithme A*, combinant des règles d'élagage simples et des mécanismes de saut récursifs pour identifier et éliminer de nombreuses symétries de chemins dans une grille connectée en 8 directions. Le JPS est principalement conçu pour des grilles bidimensionnelles à coût uniforme.

4.11 Règles d'élagage (pruning rules)

Les règles d'élagage [2] permettent de décider si un nœud voisin doit être conservé ou élagué pour l'étape suivante. Soient x le nœud actuel, n le nœud suivant, et p le nœud précédent. Voici les deux principales règles d'élagage :

- **Règle 1 : Élagage des Chemins Plus Longs**
  - **But :** Ne pas conserver les chemins qui ne sont pas les plus courts.
  - **Principe :** Si un chemin alternatif plus court est trouvé entre p et n (par un autre nœud y), le chemin passant par x n'est plus nécessaire, et le chemin (p, x, n) est élagué.

- **Règle 2 : Élagage des Chemins Sans Mouvement Diagonal Optimal**
  - **But :** Minimiser les déplacements en ligne droite lorsque des diagonales peuvent être utilisées.
  - **Principe :** Si un autre chemin permet d'atteindre n avec un déplacement en diagonale plus tôt, au lieu de parcours horizontaux ou verticaux, ce chemin est préféré. Par exemple, si un chemin $C2 = (p, y, n)$ est équivalent à $C1 = (p, x, n)$ mais utilise une diagonale plus tôt, alors $C1$ sera élagué.

Ces règles d'élagage assurent une exploration plus directe des directions diagonales, qui sont souvent plus avantageuses dans les grilles.

4.12 Règles de Saut (Jumping Rules)

L'algorithme JPS applique une procédure récursive de "saut" à chaque voisin forcé ou naturel du nœud actuel x. L'objectif est de remplacer chaque voisin n par un successeur alternatif n', situé plus loin. Au lieu de se déplacer de nœud en nœud, l'algorithme JPS effectue des sauts suivant une direction donnée jusqu'à atteindre l'une des situations suivantes :

- Un point de saut, où une décision de changement de direction est requise.
- Un obstacle.
- Le nœud de destination.

4.13 Concepts clés dans JPS

- **Points de saut (Jump Points) :** Un point de saut est un nœud important où une décision de direction doit être prise. JPS identifie ces points pour éviter les nœuds intermédiaires inutiles.
- **Élagage des voisins (Neighbour Pruning) :** JPS utilise des règles d'élagage pour ignorer les nœuds voisins qui ne contribuent pas aux chemins les plus courts vers la destination.

- **Successeurs naturels :** Un nœud n est un successeur naturel de x si la direction de déplacement de $p(x)$ à x, puis de x à n, reste la même et qu'il n'y a pas d'obstacle nécessitant un changement de direction.
- **Successeurs forcés :** Un nœud n est un successeur forcé de x si ce nœud n'est pas un successeur naturel de x et si une contrainte, comme un obstacle, empêche l'algorithme d'ignorer ce nœud.

4.14 Principe de fonctionnement

Dans A*, chaque nœud est examiné individuellement, alors que JPS identifie et explore uniquement les points de saut (jump points). Le fonctionnement de l'algorithme peut être résumé en 5 étapes clés :

1. **Identification des points de saut :** Un nœud y est considéré comme un point de saut s'il est atteint à partir d'un nœud x suivant une direction donnée, et si l'une des conditions suivantes est remplie :
   - y est le nœud de destination.
   - y possède un voisin forcé dans une direction perpendiculaire à d.
   - Pour un mouvement diagonal, y possède un successeur dans une direction perpendiculaire.

   Mathématiquement, $y = x + k \cdot \vec{d}$, où k est un entier et y est un point de saut.
2. **Élagage des voisins :** À chaque nœud x, l'algorithme applique les règles d'élagage pour réduire le nombre de voisins à considérer. Seuls les successeurs naturels et forcés sont conservés. Pour chaque voisin .
   pour un voisin n ∈ neighbors(x), une comparaison de longueur est effectuée pour vérifier si un chemin alternatif sans x est plus court :
   - Pour un déplacement horizontal ou vertical :
     Nous vérifions si n peut être atteint directement sans passer par x. mathématiquement la condition s'écrit:

     $$\text{len}(\langle p(x), n\rangle \setminus x) \leq \text{len}(\langle p(x), x, n\rangle)$$

     Si cette condition est vraie, cela signifie qu'il est plus court de se rendre directement de p(x) à n sans passer par x, et donc x est inutile il sera élagué.

- Pour un déplacement diagonal : Ici la condition devient plus stricte :

$$\text{len}(\langle p(x), n\rangle \setminus x) < \text{len}(\langle p(x), x, n\rangle)$$

3. **Application du mécanisme de saut :** Une fois les points de saut identifiés, des sauts sont effectués dans la direction optimale, remplaçant ainsi les déplacements successifs par des mouvements directs. Le saut est une procédure récursive qui continue dans la direction $\vec{d}$ jusqu'à rencontrer un point de saut ou un obstacle. Pour chaque saut, la fonction `jump(x, d)` cherche un successeur y de x en avançant dans la direction $\vec{d}$.

   - Si y est un point de saut, il est renvoyé comme successeur et la récursion s'arrête.
   - Sinon, la fonction continue en appelant `jump(y, d)` de manière récursive.

4. **Calcul des coûts :** Comme pour A*, les coûts de chaque mouvement sont calculés, mais la recherche est plus rapide grâce aux sauts et à l'élagage. Le JPS conserve l'optimalité de l'algorithme A* en assurant que chaque segment entre deux points de saut est le plus court possible. Le coût entre deux points de saut x et y est calculé selon :

$$g(y) = g(x) + \text{dist}(x, y)$$

   où g est la fonction coût jusqu'à présent et $\text{dist}(x, y)$ est la distance de déplacement entre x et y.

5. **Vérification de l'optimalité**

4.15 Pseudo-code de l'Algorithme JPS :

Fonctions utilisées :

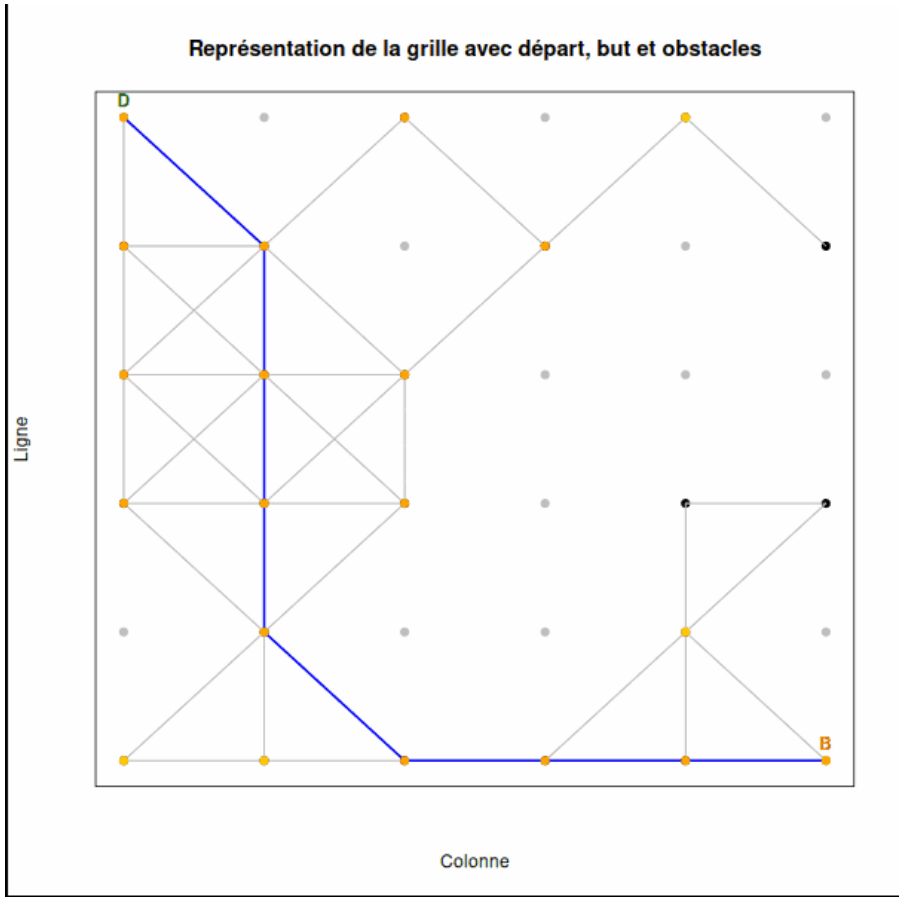
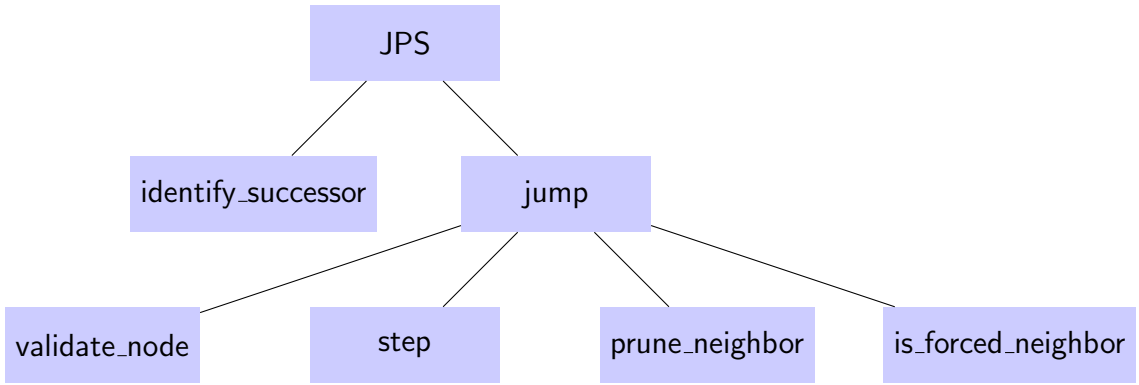


Figure 4.3: Capture d'écran de l'animation de l'exécution de JPS

Figure 4.4: Schéma des fonctions utilisées pour l'algorithme JPS




```
# Main functions:
def JPS(start, goal, grid):
    open_list = [start]
```

```
    closed_list = []
    while open_list:
        current_node = get_node_with_lowest_cost(open_list)
        if current_node == goal:
            return reconstruct_path(current_node)
        closed_list.append(current_node)
        for direction in possible_directions:
            successor = jump(current_node, direction, grid, goal)
            if successor and successor not in closed_list:
                open_list.append(successor)

# Auxiliary functions:
# 1. Jump
def jump(current_node, direction, grid, goal):
    next_node = move_forward(current_node, direction)
    if is_obstacle(next_node, grid) or out_of_bounds(next_node, grid):
        return None
    if next_node == goal or has_forced_neighbors(next_node, grid):
        return next_node
    # Filter neighbors of next_node
    neighbors = prune_neighbors(next_node, find_neighbors(next_node, grid))
    for neighbor in neighbors:
        if is_forced(neighbor):
            return next_node
    if is_diagonal(direction):
        for orthogonal_direction in orthogonal_directions(direction):
            if jump(next_node, orthogonal_direction, grid):
                return next_node
    return jump(next_node, direction, grid)

# 2. Reconstruct Path
def reconstruct_path(current_node, parent_list):
    path = []
    while current_node is not None:
        path.append(current_node)
        current_node = parent_list[current_node]
    return path[::-1]
```

```
41
42  # 3. Move Forward
43  def move_forward(current_node, direction):
44      next_node = current_node + direction
45      if out_of_bounds(next_node, grid) or is_obstacle(next_node, grid):
46          return None
47      return next_node
48
49  # 4. Prune Neighbors
50  def prune_neighbors(current_node, neighbors):
51      pruned_neighbors = []
52      for neighbor in neighbors:
53          if is_forced(neighbor) or is_natural(neighbor):  # Apply pruning
            ↪  rules
54              pruned_neighbors.append(neighbor)
55      return pruned_neighbors
56
57  # 5. Find Neighbors
58  def find_neighbors(current_node, grid):
59      neighbors = []
60      for direction in {up, down, left, right, diagonals}:
61          neighbor = current_node + direction
62          if in_bounds(neighbor, grid):
63              neighbors.append(neighbor)
64      return neighbors
```

Chapter 5

Analyse des algorithmes de recherche de chemin sur des rasters

5.1 Introduction

Dans cette étude, nous avons comparé les performances des algorithmes de recherche de chemin Dijkstra et A* appliqués sur des rasters représentant des zones marchables. Les rasters initiaux et agrégés ont été utilisés pour évaluer les performances en termes de temps d'exécution et d'efficacité.

5.2 Description des données et prétraitement

5.2.1 Raster Initial

Le raster d'origine contient une dimension qui dépend de sa réolution, pour une résolution de 0.5 2251 $\times$ 1870 , soit un total de 4209370 pixels. Parmi ces pixels, Il ya ceux qui représentant les zones marchables (ils ont une valeur de 31 dans le raster utilisé), et autres les zones non marchables. La recherche d'un compromis entre la résolution retenue pour le raster qui sera utilisé et la précision des résultats qui seront obtenues est une question cruciale dont on detaillera après.

5.2.2 Raster agrégé

Afin d'optimiser le calcul, le raster initial a été agrégé par un facteur de k=2, k=3, k=4 ... lors des tests, réduisant ainsi chaque groupe de (k) pixels à un seul pixel. Le raster agrégé

présente une dimension plus petite que le raster d'origine, donc il permet d'optimier le temps de calcul des chemins par les différentes algorithmes.

5.3 Méthodologie

5.3.1 Structure et extraction des données

Les pixels ayant une valeur de 31 ont été identifiés, et leurs indices ont été convertis en coordonnées matricielles. Une liste d'adjacence a ensuite été construite en identifiant les voisins de chaque pixel en utilisant une connectivité à 8 directions.

5.3.2 Construction du graphe

Deux approches ont été utilisées pour construire le graphe des pixels marchables. Nous avons utilisé la librairie iGraph [7] de R pour l'analyse et la construction de graphes :

1. **À partir d'une liste d'arêtes :** Chaque paire de pixels voisins a été transformée en une liste d'arêtes unique.
2. **À partir de la liste d'adjacence :** Une fonction a été utilisée pour convertir directement la liste d'adjacence en graphe.

5.3.3 Algorithmes implémentés

5.3.3.1 Algorithme de Dijkstra

L'algorithme de Dijkstra a été appliqué pour trouver le chemin le plus court entre deux pixels marchables. L'implémentation a utilisé une structure de graphe non orienté.

5.3.3.2 Algorithme A*

L'algorithme A* a été implémenté en utilisant une heuristique basée sur la distance euclidienne entre le pixel courant et le pixel cible. Cet algorithme a permis de réduire les temps de calcul grâce à une exploration plus ciblée.

5.4 Résultats

5.4.1 Performance sur le Raster Initial

- **Dijkstra :** Temps d'exécution moyen de 14 secondes.
- **A* :** Temps d'exécution moyen de 2,8 secondes.
- **JPS :** Temps d'exécution moyen de 2,8 secondes.

5.4.2 Performance sur le Raster Agrégé

- **Dijkstra :** Temps d'exécution moyen de 0,7 seconde.
- **A* :** Temps d'exécution moyen de 0,46 seconde.
- **JPS :** Temps d'exécution moyen de 2,8 secondes.

5.5 Conclusion

L'algorithme A* s'est avéré plus performant que Dijkstra en termes de temps d'exécution, grâce à son heuristique. L'agrégation du raster a également permis de réduire considérablement les temps de calcul, au prix d'une perte de précision due à la diminution de la résolution. Ces résultats soulignent l'importance de choisir un algorithme et une granularité adaptés à la nature des données et aux objectifs de l'application.

Chapter 6

Analyse et Évaluation des Algorithmes de Recherche de Chemin dans une Zone Marchable

6.1 Introduction

Dans ce chapitre, nous analysons trois algorithmes fondamentaux de recherche de chemin : *Dijkstra*, *A** et *Jump Point Search* (*JPS*). Nous testons ces algorithmes sur une grille 2D modélisant une zone marchable du centre-ville de Nantes. Cette analyse inclut la génération dynamique d'obstacles et de points de test, ainsi que la sauvegarde des résultats dans des fichiers pour une étude ultérieure. Cette grille appelée *Zone 1* ne représente pas les véritables zones marchables mais elle fait la même taille que le véritable raster des zones marchables.

6.2 Étape 1 : Génération de la grille et des points de test

Pour simuler une zone marchable, une grille de 270×270 est générée. Chaque nœud de la grille représente une cellule. Les obstacles sont disposés aléatoirement et enregistrés dans un fichier `.rds`.

6.2.1 Définition des obstacles

Un total de 10 000 obstacles est généré de manière aléatoire en excluant certains points pour garantir qu'un chemin reste possible entre deux points sélectionnés. Ces obstacles sont sauvegardés pour être utilisés dans toutes les étapes suivantes.

```
num_obstacles <- 10000
obstacles <- sample(1:total_nodes, num_obstacles, replace = FALSE)
saveRDS(obstacles, file = "obstacles.rds")
```

6.2.2 Couples de points de test

Pour tester les algorithmes, 25 couples de points de départ et d'arrivée sont créés, en s'assurant que ces points ne se situent pas dans des obstacles. Ces données sont également enregistrées pour garantir leur réutilisation.

```
generate_accessible_point <- function(excluded_points) { ... }
test_pairs <- vector("list", num_pairs)
for (i in 1:num_pairs) {
  ...
}
saveRDS(test_pairs, "test_pairs.rds")
```

6.3 Étape 2 : Construction du graphe et heuristique

6.3.1 Liste d'Adjacence

La grille est convertie en une liste d'adjacence. Chaque nœud (non obstacle) est relié à ses voisins adjacents ou diagonaux, avec un coût de déplacement calculé en fonction de la distance euclidienne.

```
adjacency_list <- vector("list", total_nodes)
for (node in 1:total_nodes) {
  if (node %in% obstacles) {
    adjacency_list[[node]] <- NULL
    next
```

```
  }
  ...
}
```

6.3.2 Heuristique euclidienne

L'heuristique utilisée dans l'algorithme *A** est basée sur la distance euclidienne entre le nœud courant et le nœud d'arrivée.

```
euclidean_distance <- function(row1, col1, row2, col2) { ... }
```

6.4 Étape 3 : Implémentation et évaluation des algorithmes

6.4.1 Algorithme de Dijkstra

L'algorithme de Dijkstra est exécuté sur le graphe pour trouver le chemin le plus court entre un point de départ et un point d'arrivée. Le coût total du chemin est calculé.

```
resultat_djikstra <- shortest_paths(g, from = start_node, to = goal_node)
cout_total <- calculate_path_cost(resultat_djikstra$vpath[[1]], grid_cols)
```

6.4.2 Algorithme A*

L'algorithme *A**, utilisant l'heuristique euclidienne, améliore les performances en explorant préférentiellement les nœuds proches du but.

```
result <- a_star(adjacency_list, euclidean_heuristic, start_node, goal_node, grid_cols)
```

6.4.3 Algorithme Jump Point Search (JPS)

Le *Jump Point Search* optimise davantage la recherche de chemin en sautant les nœuds redondants. Il identifie les nœuds pertinents, ou points de saut , réduisant significativement les calculs.

```
result <- a_star_jps(adjacency_list, euclidean_heuristic, start_node, goal_node, grid_c
```

6.4.4 Évaluation des algorithmes et analyse

Les résultats des tests de performance montrent des variations marquées entre les algorithmes, en particulier en ce qui concerne le temps d'exécution. Le tableau suivant résume les temps d'exécution observés pour les deux rasters utilisés.

Algorithme	**Temps (Zone 1 Test)**	**Temps (Zone 2)**
Dijkstra	0.1 s	0.7 s
A*	2.3 s	0.46 s
JPS	5.7 s	0.4 s

Table 6.1: Temps d'exécution des algorithmes sur les différents rasters

Les résultats qualitatifs révèlent des différences notables entre les algorithmes. A* et JPS sont en effet bien plus performants grâce à l'utilisation de l'heuristique et grâce au pruning. Ces deux optimisations permettent ainsi de visiter beaucoup moins de noeuds que Dijkstra et donc d'améliorer le temps de calcul.

Pour le pruning et le saut, il est nécessaire de calculer les distances du noeud d'où l'on vient vers le noeud où on veut aller, ce qui implique de calculer jusqu'à huit distances supplémentaires.

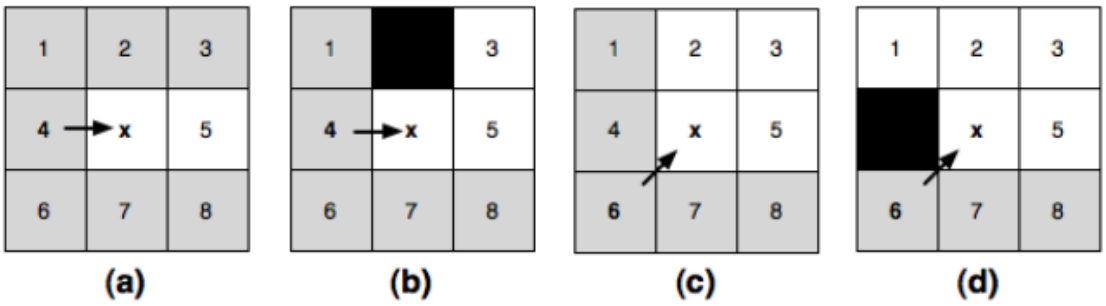


Figure 6.1: Schéma du processus de l'élagage avec JPS [2]

Donc cela rajoute beaucoup d'exécutions supplémentaires. Notre conjecture était que le saut, même avec des sous exécutions de A* supplémentaires, permettrait au final sur l'ensemble de la grille, d'avoir moins de noeuds à visiter.

Les résultats permettent de confirmer cette conjecture.

6.5 Résultats et visualisation

Les chemins trouvés par chaque algorithme sont visualisés sur la grille. Les obstacles, les points de départ et d'arrivée, ainsi que le chemin trouvé sont affichés pour une analyse visuelle.

```
plot_grid(grid_rows, grid_cols, adjacency_list, start_node, goal_node, result$path, obs
```

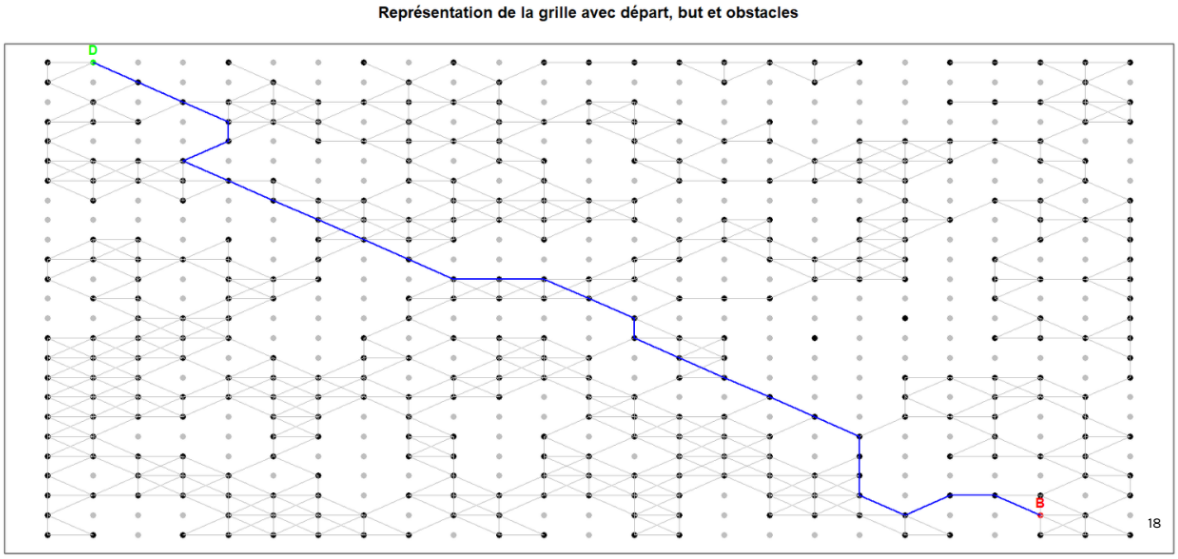


Figure 6.2: Résultat de Dijkstra sur la *Zone 1*

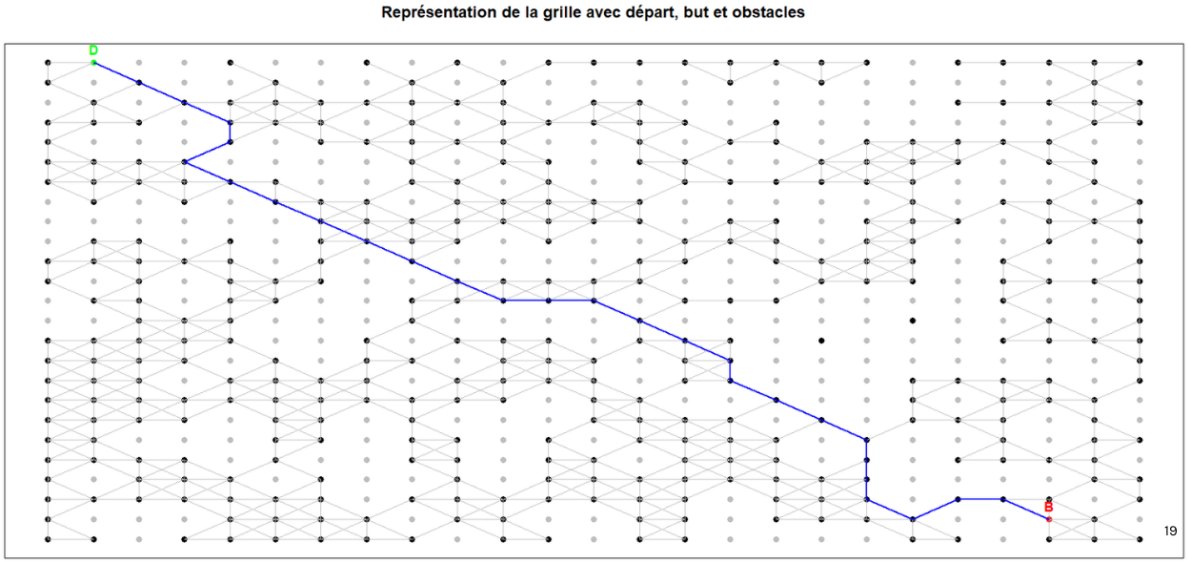


Figure 6.3: Résultat de A* sur la *Zone 1*

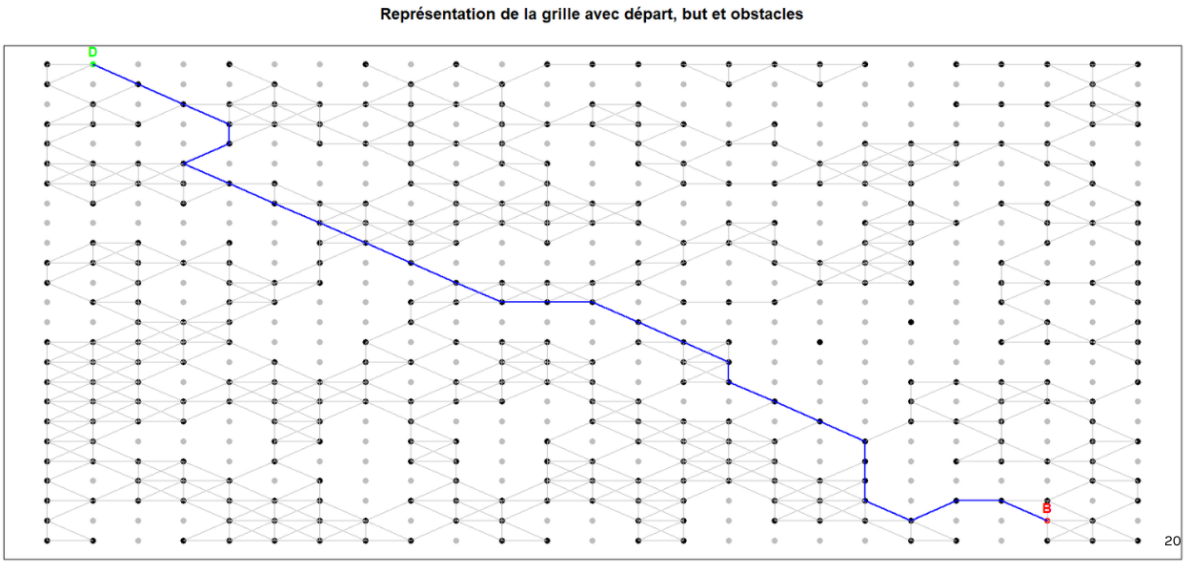


Figure 6.4: Résultat de JPS sur la *Zone 1*

6.6 Conclusion

Ce chapitre a présenté une méthodologie pour tester trois algorithmes de recherche de chemin dans une zone marchable du centre-ville de Nantes. Les performances de chaque algorithme, en termes de coût et de temps d'exécution, sont comparées pour évaluer leur efficacité respective.

Chapter 7

Conclusion générale

Cette étude démontre l'importance d'adapter les algorithmes de recherche de Cette étude met en évidence l'importance d'adapter les algorithmes de recherche de chemin aux contraintes spécifiques des environnements urbains. Au delà du type d'algorithme utilisé, ainsi que la méthodologie adoptée pour appliquer ces algorithmes, influencent significativement le temps d'exécution. L'algorithme JPS s'est révélé être le plus performant en offrant un excellent compromis entre déterminisme et rapidité d'exécution.

Appendix A

Appendix A

Begins an appendix

Bibliography

[1] Dorian Essamir.
Le pathfinding a* et l'optimisation pour des grilles 2d, mémoire de master.
https://www.guillaumelevieux.com/siteperso/contents/recherchem1/essamir/pathfinding.pdf.
Disponible en ligne.

[2] Daniel Harabor and Alban Grastien.
Online graph pruning for pathfinding on grid maps.
volume 2, 01 2011.

[3] ETUDESTECH.com.
Guide complet de l'algorithme de dijkstra.
https://etudestech.com/decryptage/algorithme-dijkstra-calculer-chemin-plus-court-som
Disponible en ligne.

[4] Amit Patel.
Introduction to a*.
https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.
Disponible en ligne.

[5] Blai Bonet and Héctor Geffner.
Planning as heuristic search.
Artificial Intelligence, 129(1):5–33, 2001.

[6] Nathan Witmer.
A visual explanation of jump point search.
https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search/, 2013.

Disponible en ligne.

[7] Laurent Tichit Alberto Valdeolivas Urbelz.
igraph.
https://www.dil.univ-mrs.fr/~tichit/bin/tp1/igraph_tutorial.html.
Disponible en ligne.