**Structure Générale**

Le code implémente plusieurs algorithmes de recherche de chemin sur un graphe représenté par un fichier de carte. Les algorithmes comprennent :

1. **Dijkstra :** Garantit le chemin le plus court.
2. **A* :** Une recherche informée qui utilise une heuristique pour guider la recherche, souvent plus rapide que Dijkstra.
3. **WA* (A* Pondéré) :** Une variante de A* qui permet des solutions sous-optimales, mais peut être plus rapide.
4. **BFS (Recherche en largeur) :** Trouve le chemin le plus court en termes de nombre d'arêtes (sans tenir compte du poids des arêtes).
5. **Recherche gloutonne du meilleur d'abord :** Utilise une heuristique pour explorer de manière gourmande le chemin le plus prometteur, mais n'est pas garanti d'être optimal.

**Structures de Données**

- **Node (Nœud) :**
    - *position::Tuple{Int64, Int64} :* Représente les coordonnées (x, y) du nœud dans la grille.
    - *neighbors::Dict{Tuple{Int64, Int64}, Float64} :* Un dictionnaire où les clés sont les positions des nœuds voisins, et les valeurs sont les poids (coûts) des arêtes pour atteindre ces voisins.
- **Graph (Graphe) :**
    - *nodes::Dict{Tuple{Int64, Int64}, Node} :* Un dictionnaire où les clés sont les positions des nœuds (x, y), et les valeurs sont les objets Node eux-mêmes. Cela permet un accès rapide à n'importe quel nœud dans le graphe.
- **priority_queue (File de Priorité) :**
    - *elements::Vector{Tuple{Tuple{Int64, Int64}, Float64}}* : Un vecteur de tuples. Chaque tuple contient une clé (un tuple représentant des coordonnées) et une priorité (un nombre Float64). Cela représente la file de priorité.

**Fonctions Clés Expliquées**

1. **Structure priority_queue et Fonctions Associées**
    - **Objectif :** Implémente une file de priorité, cruciale pour Dijkstra, A* et la recherche gloutonne du meilleur d'abord. Elle permet de récupérer efficacement le nœud avec la plus petite priorité.

***priority_queue()*** **(Constructeur) :**

Initialise une file de priorité vide. Le champ elements est un vecteur qui stockera les nœuds et leurs priorités.

***add_to_queue!(pq::priority_queue, key::Tuple{Int64, Int64}, new_priority::Float64) :***

Ajoute un nouvel élément (key et new_priority) à la file de priorité, puis appelle rebalancing_up! pour maintenir la propriété de tas (heap).

***is_empty(pq::priority_queue) :***

Vérifie si la file de priorité est vide.

***remove_to_queue!(pq::priority_queue) :***

Supprime l'élément avec la priorité minimale (la racine du tas). Elle échange la racine avec le dernier élément, supprime le dernier élément, puis appelle rebalancing_down! pour restaurer la propriété de tas.

***update_priority!(pq::priority_queue, key::Tuple{Int64, Int64}, new_priority::Float64) :***

Met à jour la priorité d'un élément dans la file. Si la nouvelle priorité est inférieure, elle déplace l'élément vers le haut dans le tas ; si elle est supérieure, elle le déplace vers le bas. Si l'élément n'existe pas, elle l'ajoute à la file.

***rebalancing_up!(pq::priority_queue, index::Int) :***

Restaure la propriété de tas en déplaçant un élément vers le haut de l'arbre si sa priorité est inférieure à celle de son parent.

***rebalancing_down!(pq::priority_queue, index::Int) :***

Restaure la propriété de tas en déplaçant un élément vers le bas de l'arbre si sa priorité est supérieure à celle d'un de ses enfants.

***change!(vec::Vector, i::Int, j::Int) :***

Échange deux éléments dans le vecteur.

- **Algorithme :** La file de priorité est implémentée sous forme de tas binaire. Cela garantit que l'élément avec la plus petite priorité peut être récupéré en O(1) temps, et les opérations d'insertion/suppression prennent O(log n) temps, où n est le nombre d'éléments dans la file.

- **Justification :** Une file de priorité est essentielle pour sélectionner efficacement le prochain nœud à explorer dans Dijkstra, A* et la recherche gloutonne du meilleur d'abord. Elle garantit que l'algorithme développe toujours le nœud avec le coût de chemin le plus prometteur.

2. ***movement_cost(value::Char)***

- **Objectif :** Détermine le coût de déplacement vers une cellule spécifique sur la carte en fonction de sa représentation de caractère.

- **Algorithme :** Une simple table de consultation. @ représente un mur (coût infini), W, S et T représentent différents types de terrain avec des coûts associés, et tout autre caractère a un coût par défaut de 1.0.

- **Justification :** Cette fonction permet à la carte de représenter différents types de terrain avec des coûts de déplacement variables, ce qui rend le problème de recherche de chemin plus réaliste.

3. ***read_graph_from_file(filename::String)***

- **Objectif :** Lit une carte à partir d'un fichier et construit un objet Graph représentant la carte.
- **Algorithme :**

  Lit le fichier de carte, en ignorant les 4 premières lignes d'en-tête.

  Stocke la carte dans un tableau 2D (grid).

  Parcourt chaque cellule de la grille, en créant un objet Node pour chaque cellule.

  Pour chaque nœud, elle vérifie les quatre voisins possibles (haut, bas, gauche, droite).

Si un voisin est dans les limites de la grille et n'est pas un mur (le coût n'est pas infini), elle ajoute le voisin au dictionnaire neighbors du nœud avec le coût approprié.

Ajoute le nœud à l'objet Graph.

- **Justification :** Cette fonction encapsule le processus de lecture de la carte à partir d'un fichier et de conversion en une représentation de graphe que les algorithmes de recherche de chemin peuvent utiliser.

### 4. *dijkstra(graph::Graph, source::Tuple{Int64, Int64}, target::Tuple{Int64, Int64})*

- **Objectif :** Implémente l'algorithme de Dijkstra pour trouver le chemin le plus court d'un nœud source à un nœud cible dans un graphe.
- **Algorithme :**

Initialise distance (la distance la plus courte de la source à chaque nœud) à l'infini pour tous les nœuds, sauf pour le nœud source, qui est défini sur 0.

Initialise previous (le nœud précédent dans le chemin le plus court depuis la source) à nothing pour tous les nœuds.

Crée une file de priorité Q et ajoute le nœud source avec sa distance (0).

Tant que la file de priorité n'est pas vide :

- Supprime le nœud u avec la plus petite distance de la file de priorité.
- Si u a déjà été visité, passez à l'itération suivante.
- Marque u comme visité.
- Pour chaque voisin v de u :
  - Calcule la distance alternative alt de la source à v en passant par u.
  - Si alt est inférieure à la distance actuelle à v, met à jour la distance à v et définit le nœud précédent de v sur u.
  - Ajoute v à la file de priorité avec la distance mise à jour.

Renvoie le dictionnaire previous (utilisé pour reconstruire le chemin), la distance au nœud cible et le nombre d'états évalués.

- **Justification :** L'algorithme de Dijkstra garantit de trouver le chemin le plus court dans un graphe avec des poids d'arête non négatifs. La file de priorité garantit que l'algorithme explore toujours le nœud avec la plus petite distance connue de la source.

5. ***Breadth_First_Search(graph::Graph, source::Tuple{Int64, Int64}, target::Tuple{Int64, Int64})***

   - **Objectif :** Implémente la recherche en largeur (BFS) pour trouver le chemin le plus court d'un nœud source à un nœud cible dans un graphe. BFS trouve le chemin le plus court en termes de *nombre d'arêtes* (sauts), sans tenir compte du poids des arêtes.
   -
   - **Algorithme :**

     Initialise shortest_path (la distance la plus courte de la source à chaque nœud) à l'infini pour tous les nœuds, sauf pour le nœud source, qui est défini sur 0.

     Initialise previous (le nœud précédent dans le chemin le plus court depuis la source) à nothing pour tous les nœuds.

     Initialise mark à false pour tous les nœuds, indiquant qu'aucun nœud n'a été visité.

     Crée une file d'attente tail et ajoute le nœud source.

     Marque le nœud source comme visité.

     Tant que la file d'attente n'est pas vide :

     - Supprime le premier nœud current de la file d'attente.
     - Si current est le nœud cible, l'algorithme est terminé.
     - Pour chaque voisin de current :
       - Si le voisin n'a pas été visité :
         - Définit le chemin le plus court vers le voisin sur le chemin le plus court vers current plus 1.
         - Définit le nœud précédent du voisin sur current.
         - Marque le voisin comme visité.
         - Ajoute le voisin à la file d'attente.

Renvoie le dictionnaire shortest_path (utilisé pour reconstruire le chemin), le dictionnaire previous et le nombre d'états évalués.

- **Justification :** BFS explore tous les voisins à la profondeur actuelle avant de passer aux nœuds au niveau de profondeur suivant. En d'autres termes, il développe le nœud non développé le moins profond.

6. ***heuristic_cost(source::Tuple{Int, Int}, target::Tuple{Int, Int})***

- **Objectif :** Calcule la distance de Manhattan entre deux nœuds, utilisée comme heuristique dans A* et la recherche gloutonne du meilleur d'abord.
- **Algorithme :** Calcule la somme des différences absolues des coordonnées x et y.
- **Justification :** La distance de Manhattan est une heuristique admissible (ne surestime jamais le coût réel) pour la recherche de chemin basée sur une grille où les mouvements diagonaux ne sont pas autorisés. Cela aide A* à trouver le chemin optimal plus efficacement.

7. ***a_star(graph::Graph, source::Tuple{Int64, Int64}, target::Tuple{Int64, Int64})***

- **Objectif :** Implémente l'algorithme de recherche A* pour trouver le chemin le plus court d'un nœud source à un nœud cible dans un graphe, en utilisant l'heuristique de la distance de Manhattan.
- **Algorithme :**

Initialise g (coût du départ au nœud actuel) à l'infini pour tous les nœuds, sauf pour le nœud source, qui est défini sur 0.

Initialise f (coût estimé du départ à la cible en passant par le nœud actuel) à l'infini pour tous les nœuds.

Calcule f pour le nœud source comme le coût heuristique de la source à la cible.

Crée une file de priorité Q et ajoute le nœud source avec sa valeur f.

Tant que la file de priorité n'est pas vide :

- Supprime le nœud current avec la plus petite valeur f de la file de priorité.
- Si current a déjà été visité, passez à l'itération suivante.
- Marque current comme visité.
- Si current est le nœud cible, l'algorithme est terminé.
- Pour chaque voisin de current :

- Calcule la valeur g provisoire pour le voisin.
- Si la valeur g provisoire est inférieure à la valeur g actuelle pour le voisin :
  - Met à jour la valeur g pour le voisin.
  - Met à jour la valeur f pour le voisin.
  - Définit le nœud précédent du voisin sur current.
  - Ajoute le voisin à la file de priorité avec sa valeur f.

Renvoie le dictionnaire g (utilisé pour reconstruire le chemin), le dictionnaire previous et le nombre d'états évalués.

- **Justification :** A* est un algorithme de recherche informé qui utilise une heuristique pour guider la recherche, ce qui se traduit souvent par des performances plus rapides que l'algorithme de Dijkstra, en particulier pour les grands graphes. L'heuristique aide A* à prioriser les nœuds qui sont susceptibles de se trouver sur le chemin le plus court vers la cible.

8. ***bounded_suboptimal_search(graph::Graph, source::Tuple{Int64, Int64}, target::Tuple{Int64, Int64}, w::Float64)***

- **Objectif :** Implémente l'algorithme de recherche A* pondéré (WA*), une variante de A* qui permet des solutions sous-optimales, mais peut être plus rapide.
- **Algorithme :**

L'algorithme est presque identique à A*, mais il multiplie l'heuristique par un poids w.

Si w est 1, WA* est équivalent à A*.

Si w est supérieur à 1, WA* devient plus gourmand, ce qui peut lui permettre de trouver une solution sous-optimale plus rapidement.

- **Justification :** WA* permet un compromis entre l'optimalité de la solution et le temps de recherche. En augmentant le poids de l'heuristique, l'algorithme explore moins de nœuds, mais peut trouver un chemin plus long que le chemin optimal.

9. ***greedy_best_first_search(graph::Graph, source::Tuple{Int64, Int64}, target::Tuple{Int64, Int64})***

- **Objectif :** Implémente l'algorithme de recherche gloutonne du meilleur d'abord pour trouver un chemin d'un nœud source à un nœud cible dans un

graphe, en utilisant l'heuristique de la distance de Manhattan.

**Algorithme :**

Initialise previous à nothing pour tous les nœuds.

Crée une file de priorité Q et ajoute le nœud source avec son coût heuristique à la cible.

Tant que la file de priorité n'est pas vide :

- Supprime le nœud current avec le plus petit coût heuristique de la file de priorité.
- Si current a déjà été visité, passez à l'itération suivante.
- Marque current comme visité.
- Si current est le nœud cible, l'algorithme est terminé.
- Pour chaque voisin de current :
  - Si le voisin n'a pas été visité :
    - Définit le nœud précédent du voisin sur current.
    - Ajoute le voisin à la file de priorité avec son coût heuristique à la cible.

Renvoie le dictionnaire previous (utilisé pour reconstruire le chemin) et le nombre d'états évalués.

- **Justification :** La recherche gloutonne du meilleur d'abord est un algorithme simple et rapide qui explore de manière gourmande le chemin le plus prometteur en fonction de l'heuristique. Cependant, elle ne garantit pas de trouver le chemin le plus court.

### *10. print_shortest_path(source, target)*

- **Objectif :** Reconstruit et imprime le chemin le plus court de la source à la cible, en utilisant le dictionnaire previous renvoyé par les algorithmes de recherche.
- **Algorithme :**

Commence à partir du nœud cible et suit itérativement les liens previous jusqu'à ce qu'il atteigne le nœud source.

Inverse le chemin pour obtenir l'ordre correct de la source à la cible.

Imprime le chemin.

- **Justification :** Cette fonction permet de visualiser la solution trouvée par les algorithmes de recherche de chemin.

11. **Fonctions principales :** ***algo_dijkstra, algo_bfs, algo_a_star, algo_bss, algo_greedy_bfs***

    - **Objectif :** Ces fonctions orchestrent l'exécution de chaque algorithme de recherche de chemin. Elles :

      Lis le graphe à partir du fichier en utilisant read_graph_from_file.

      Enregistre l'heure de début.

      Appelle l'algorithme de recherche de chemin correspondant.

      Enregistre l'heure de fin.

      Imprime les résultats, y compris le temps d'exécution, la distance, le nombre d'états évalués et le chemin le plus court.

    - **Justification :** Ces fonctions fournissent un moyen clair et organisé d'exécuter chaque algorithme et d'évaluer ses performances.