Université de Tunis

Dauphine

Travaux Pratiques 1 : Intelligence artificielle et raisonnement

Binôme :
Aziz DHIF et Fatma CHAHED

1. Introduction

Ce travail explore la mise en œuvre et la comparaison de plusieurs algorithmes de recherche de chemin dans un environnement inconnu, modélisé sous forme de grille. Un robot, placé à une position de départ, doit atteindre un objectif en évitant des obstacles infranchissables (murs) et en tenant compte de zones de terrain difficile (eau), lesquelles engendrent un coût énergétique plus élevé.

Les déplacements du robot sont autorisés dans huit directions (horizontales, verticales et diagonales), chaque type de mouvement étant associé à un coût spécifique.

L'objectif principal est d'évaluer différentes stratégies de recherche — **BFS (Breadth-First Search)**, **DFS (Depth-First Search)**, **A*** avec une heuristique admissible, et **Forward Checking** avec gestion de contraintes — selon plusieurs critères : coût total du chemin, nombre de nœuds développés, temps d'exécution et optimalité des solutions.

Des visualisations graphiques sont également fournies afin d'illustrer et de mieux comprendre le comportement de chaque méthode.

Dans le cadre de cette étude, les algorithmes analysés peuvent être regroupés selon les catégories suivantes :

- **Recherche Non Informée :**
  - **DFS (Depth-First Search)**
  - **BFS (Breadth-First Search)**
- **Recherche Heuristique (Informée) :**
  - **A* (A-star)**
- **Recherche avec Élagage (Pruning) :**
  - **Forward Checking**

2. Modélisation du problème

2.1. Environnement : la grille de taille (12 $\times$11)

L'environnement est représenté sous forme de grille dans laquelle chaque cellule peut avoir un rôle spécifique :

- **Libre (blanche)** : déplacement normal.
- **Mur (gris)** : cellule infranchissable.
- **Eau (bleu)** : terrain difficile, déplacement coûteux.
- **Départ (rouge)** : position initiale de l'agent.
- **Objectif (vert)** : destination à atteindre.

2.2. Actions possibles A

Depuis chaque cellule, l'agent peut tenter les mouvements suivants :

- **Déplacements horizontaux ou verticaux (haut, bas, gauche, droite)** : coût=1.
- **Déplacements diagonaux** : coût = 1.41.
- **Déplacements dans l'eau (horizontal/vertical)** : coût = 3.
- **Déplacements dans l'eau (diagonale)** : coût = 4.24.
- **Murs** : mouvement non autorisé, mais le coût est payé et l'agent reste sur place.

$\Longrightarrow$ Le nombre d'actions possibles est égal à **8**.

2.3. Les coûts

La fonction de coût est définie comme suit :

- **Collision avec un mur** : 10.
- **Déplacement dans l'eau** :

$$\begin{cases} \text{Horizontal ou vertical} & : 3. \\ \text{Diagonal} & : 4.24. \end{cases}$$

- **Déplacement normal** :

$$\begin{cases} \text{Horizontal ou vertical} & : 1. \\ \text{Diagonal} & : 1.41. \end{cases}$$

3. Modélisation de la grille

Dans cette section, nous présentons la modélisation de la grille représentant le chemin dans un labyrinthe. La grille est codée sous forme d'une matrice de taille (12×11), où chaque cellule est encodée par une valeur entière correspondant à un type de case.

Les différentes valeurs utilisées dans la grille sont :

— **.** : une case vide, libre pour le déplacement.

— **|** : un mur, c'est-à-dire une case infranchissable.

— **w** : une case contenant de l'eau, franchissable mais pénalisante.

— **D** : la position de départ de l'agent.

— **A** : la position d'arrivée ou objectif à atteindre.

Cette représentation permet de simplifier l'implémentation des règles du jeu et la gestion des récompenses dans l'algorithme d'apprentissage par renforcement.

```
# Définition des constantes
DEPART = 'D'     # Position de départ (rouge)
ARRIVEE = 'A'      # Sortie (vert)
MUR = '|'      # Mur (gris)
EAU = 'W'     # Eau (bleu)
EMPTY = '.'      # Case vide

# Matrice adaptée avec vos indices
grid = [
    ['.', '.', '.', '.', '.', 'W', 'W', 'W', '.', 'W', 'W'],
    ['.', '.', '|', '|', '.', '|', 'W', 'W', '.', 'W', 'W'],
    ['.', '.', '.', '|', '.', '|', 'W', '|', '.', '.', '.'],
    ['.', '.', '.', '|', '.', '.', '.', '|', '.', '.', '.'],
    ['D', '.', '.', '|', '|', '.', 'W', '|', '.', '.', 'A'],
    ['.', '.', '.', '.', '|', '.', 'W', '|', '.', '.', '.'],
    ['.', '|', '.', '.', '|', '.', 'W', '|', '|', '|', '.'],
    ['.', '|', '|', '|', '|', '.', '.', '.', '|', '.', 'W'],
    ['.', '.', 'W', 'W', '.', '.', '|', '.', 'W', '.', '.'],
    ['.', '.', '.', 'W', '.', '.', '|', '.', 'W', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '|', '.', '.', '.', '.'],
    ['.', '.', '.', '.', '.', '.', '|', '.', '.', '.', '.']
]
```

4. Trouver les positions de départ et d'arrivée

La fonction **`find_positions`** permet de trouver les positions de départ et d'arrivée de l'agent dans une grille.

Une position est définie par des coordonnées binaires (x, y) et n'est retournée que si les deux points — départ et arrivée — existent dans la grille.

```python
# Trouver la position de d part et d'arrivee
def find_positions(grid):
    start, exit = None, None
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == DEPART:
                start = (i, j)
            elif grid[i][j] == ARRIVEE:
                exit = (i, j)
    return start, exit
```

Exemple : cost, new$_x$, $new_y = calculate_cost(grid, 10, 9, 10, 10) Coût : 1.0, Position : (10, 10)$

5. Validité de la position

```python
def is_valid(pos):
    r, c = pos
    return 0 <= r < rows and 0 <= c < cols
```

La fonction **is_valid** permet de vérifier si une position donnée dans la grille est valide. Pour ce faire, nous devons nous assurer que la position se situe bien dans les limites de la grille, définie par les variables `rows` (nombre de lignes) et `cols` (nombre de colonnes), soit :

$$0 \leq r < \texttt{rows} \quad \text{et} \quad 0 \leq c < \texttt{cols}$$

6. Définition de la fonction coût

```python
def calculate_cost(grid, x, y, dx, dy):
    """Calcule le co t de d placement de (x, y) vers (dx, dy) dans
        la grille."""

    # Verification que les positions sont valides
    if not is_valid((x, y)) or not is_valid((dx, dy)):
        return (float('inf'), x, y)  # Mouvement invalide (hors
            limites)

    # V rification que la destination n'est pas un mur
    if grid[dx][dy] == MUR:
        return (float('inf'), x, y)  # Mouvement bloqu

    # V rification que le mouvement est valide (1 case max en
        diagonale ou droite)
    delta_x = abs(dx - x)
```

```
    delta_y = abs(dy - y)
    if delta_x > 1 or delta_y > 1 or (delta_x == 0 and delta_y == 0):
        return (float('inf'), x, y)  # Mouvement invalide ou nul

    # Co t de base selon le type de mouvement
    is_diagonal = delta_x == 1 and delta_y == 1
    base_cost = 1.41 if is_diagonal else 1.0

    # Surco t pour les terrains sp ciaux
    if grid[dx][dy] == EAU:
        base_cost = 4.24 if is_diagonal else 3.0

    return (base_cost, dx, dy)
```

La fonction de calcul du coût, **calculate_cost**, détermine le coût de déplacement en fonction de la position actuelle et de la nouvelle position de l'agent. Les différents cas suivants sont pris en compte :

- **Mur** : la fonction retourne un coût de **10**, ce qui signifie que l'agent a heurté un mur sans pouvoir avancer.
- **Eau** : si l'agent se déplace en diagonale, le coût est de **4.24**, sinon, le coût est de **3**.
- **Autres cases** : pour les autres types de cases (par exemple, vides), le coût est de **1** pour un déplacement orthogonal, et de **1.41** pour un déplacement diagonal.

7. Détermination des voisins d'un neoud

```
# d terminer les voisins
def get_neighbors(grid, node):
    """Retourne les voisins accessibles avec leur co t"""
    x, y = node
    neighbors = []
    directions = [(-1,0),(1,0),(0,-1),(0,1), (-1,-1),(-1,1),(1,-1),(1,1)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]):
            cost = calculate_cost(grid, x, y, nx, ny)[0]  # On prend seulement le co t
            if cost != float('inf'):
                neighbors.append(((nx, ny), cost))
    return neighbors
```

Dans le cadre des algorithmes de recherche dans les graphes, on utilise la fonction **`get_neighbors`**, qui identifie tous les voisins accessibles d'un nœud donné dans une grille, en tenant compte de toutes les directions possibles (y compris les diagonales), et retourne une liste des positions voisines valides accompagnées de leur coût de déplacement, lequel est calculé à l'aide de la fonction `calculate_cost`. Cela permet à l'algorithme de navigation d'explorer les chemins possibles à partir du nœud courant.

8. Recherche Non Informée

Nous allons présenter deux algorithmes de recherche non informée, l'algorithme BFS et DFS.

8.1. BFS

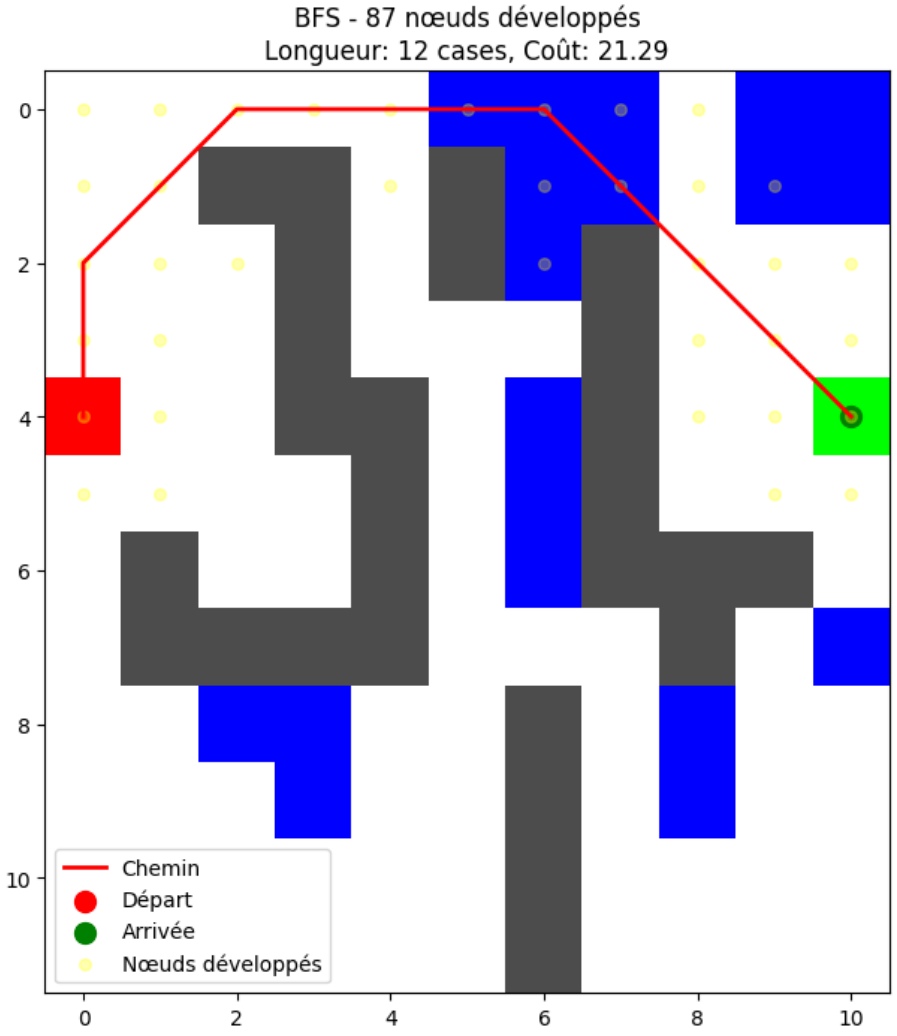
```
    # 1. BFS avec co t
from collections import deque
def bfs(grid, start, goal):
    queue = deque([(start, [start], 0)])
    visited = set([start])
    developed_nodes = 0

    while queue:
        current, path, cost = queue.popleft()
        developed_nodes += 1

        if current == goal:
            return path, cost, developed_nodes

        for neighbor, move_cost in get_neighbors(grid, current):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor], cost +
                    move_cost))

    return None, float('inf'), developed_nodes
```

Ce code implémente l'algorithme de **recherche en largeur (BFS)** adapté à une grille pondérée, c'est-à-dire en tenant compte du **coût de déplacement** entre les nœuds.

— Il initialise une **file FIFO (`deque`)** contenant le nœud de départ, le chemin parcouru jusqu'ici et le coût associé (initialement nul).

— À chaque itération, le nœud en tête de **file** est extrait, et s'il correspond au nœud objectif (`goal`), la fonction retourne le chemin, le coût total et le nombre de nœuds développés.

— Sinon, les **voisins accessibles** (obtenus via la fonction `get_neighbors`) sont explorés, et ceux qui n'ont pas encore été visités sont ajoutés à la file avec le **chemin mis à jour** et le **coût cumulé**.
— Un ensemble `visited` permet d'éviter les répétitions et les boucles infinies.
— Un compteur `developed_nodes` permet de suivre le **nombre total de nœuds explorés**.
— Si aucun chemin n'est trouvé, la fonction retourne `None`, un coût infini et le nombre de nœuds développés.
— Cette version de BFS permet donc une **exploration des chemins avec prise en compte des coûts**, bien qu'elle ne garantisse pas nécessairement le chemin au coût minimal si les coûts ne sont pas uniformes.




```
[(4, 0), (3, 0), (2, 0), (1, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (1, 7), (2, 8), (3, 9), (4, 10)]
Coût total avec BFS: 21.29
```

FIGURE 1 – Le chemin du BFS

8.2. DFS

```python
# 2. DFS avec co t
def dfs(grid, start, goal):
    stack = [(start, [start], 0)]
    visited = set([start])
    developed_nodes = 0

    while stack:
        current, path, cost = stack.pop()
        developed_nodes += 1

        if current == goal:
            return path, cost, developed_nodes

        for neighbor, move_cost in get_neighbors(grid, current):
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append((neighbor, path + [neighbor], cost +
                    move_cost))

    return None, float('inf'), developed_nodes
```

Cette fonction implémente l'algorithme de **recherche en profondeur (DFS)** avec prise en compte du **coût de déplacement** dans une grille.

- Elle utilise une **pile (`stack`)** pour gérer les nœuds à explorer, ce qui permet une exploration en profondeur du graphe, c'est-à-dire qu'elle suit un chemin aussi loin que possible avant de revenir en arrière.
- La **pile** contient des tuples composés du nœud courant, du chemin parcouru, et du coût cumulé. Elle est initialisée avec le nœud de départ.
- À chaque itération, le dernier élément ajouté à la pile est retiré (logique **LIFO**), et s'il correspond au but, la fonction retourne le chemin trouvé, le coût total et le nombre de nœuds développés.
- Les **voisins accessibles** sont récupérés via la fonction `get_neighbors`, et seuls ceux qui n'ont pas encore été visités sont ajoutés à la pile, avec le chemin et le coût mis à jour.
- Un ensemble `visited` permet d'éviter les redondances et les cycles.
- Un compteur `developed_nodes` garde la trace du nombre total de nœuds explorés.

⇒ Contrairement au BFS, cette version n'explore pas tous les voisins au même niveau mais s'enfonce dans un chemin avant d'en tester un autre, ce qui peut être **moins optimal** pour trouver le chemin au coût minimal.

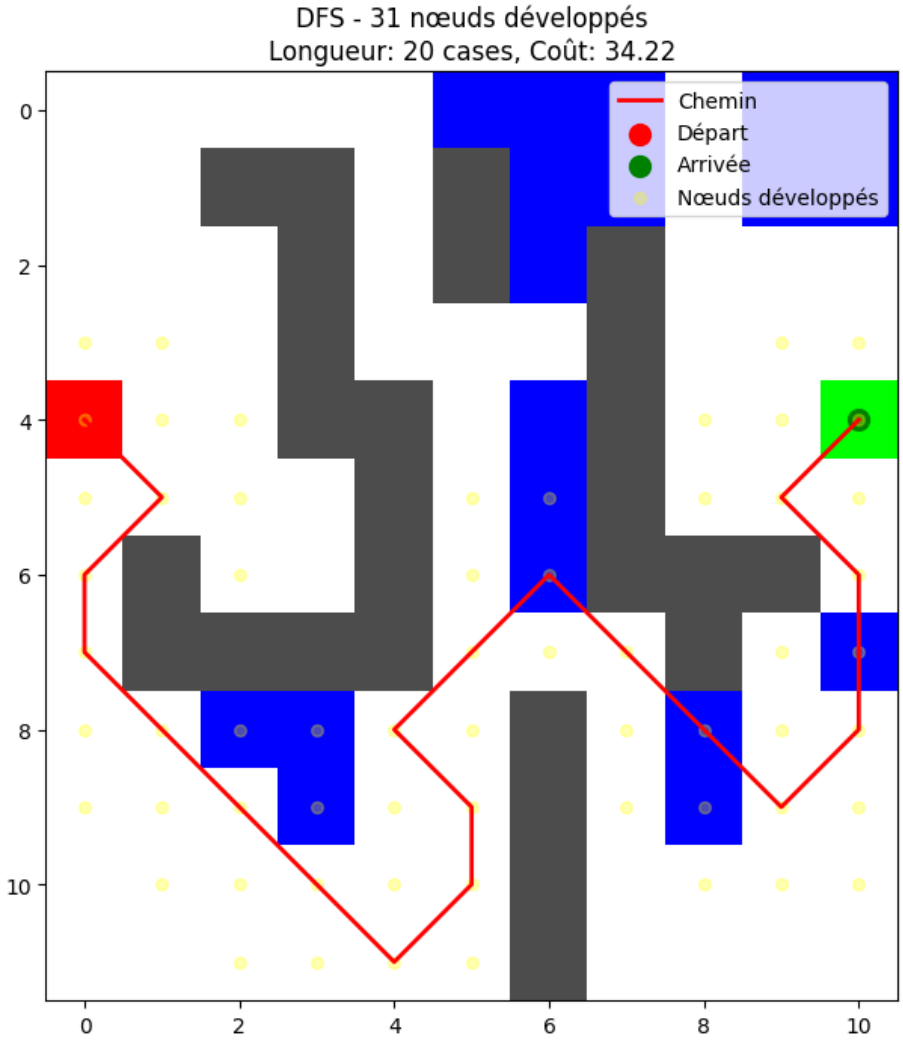


FIGURE 2 – le chemin du dfs : [(4, 0), (5, 1), (6, 0), (7, 0), (8, 1), (9, 2), (10, 3), (11, 4), (10, 5), (9, 5), (8, 4), (7, 5), (6, 6), (7, 7), (8, 8), (9, 9), (8, 10), (7, 10), (6, 10), (5, 9), (4, 10)]

9. Recherche Heuristique (L'algorithme A*)

Ce code implémente l'algorithme **A*** en utilisant deux fonctions principale :

9.1. Fonction `heuristic(a, b)`

```python
def heuristic(a, b):
    """Distance de Manhattan"""
    """(admissible pour les mouvements orthogonaux et diagonaux)"""
    dx = abs(a[0] - b[0])
    dy = abs(a[1] - b[1])
    return max(dx, dy) + 0.41 * min(dx, dy)  # Approximation pour
        diagonales
```

— Calcule la **distance heuristique** entre deux points `a` et `b` sur une grille.
— Utilise une **distance de type Chebyshev améliorée**, adaptée aux déplacements orthogonaux (haut, bas, gauche, droite) et diagonaux.
— La formule `max(dx, dy) + 0.41 * min(dx, dy)` est une approximation qui donne un coût légèrement plus élevé aux mouvements diagonaux. C'est une heuristique admissible et consistante, donc A* garantit d'explorer efficacement et de trouver le chemin optimal.

Propriétés de l'heuristique : admissibilité et consistance

Pour que l'algorithme A* garantisse un chemin optimal, l'heuristique utilisée doit satisfaire deux propriétés essentielles :

— **Admissibilité** : l'heuristique ne doit jamais surestimer le vrai coût pour atteindre l'objectif, autrement dit, elle doit toujours être inférieure ou égale au coût réel.
— **Consistance (ou monotonicité)** : pour tout nœud n et tout successeur n' de n, l'heuristique h doit respecter la condition suivante :

$$h(n) \leq c(n, n') + h(n')$$

où $c(n, n')$ est le coût réel pour aller de n à n'. Cette propriété garantit que l'estimation devient plus précise à mesure que l'on s'approche de l'objectif.

La distance euclidienne, bien qu'intuitivement correcte dans un espace continu, peut violer la consistance dans les grilles si les coûts de déplacement ne sont pas strictement proportionnels à la distance géométrique. Cela peut conduire à des incohérences dans la sélection des nœuds à explorer, compromettant l'optimalité du chemin trouvé.

9.2. Fonction `a_star(grid, start, goal)`

```
def a_star(grid, start, goal):
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    g_score = {start: 0}
    f_score = {start: heuristic(start, goal)}
    developed_nodes = 0

    while open_set:
        current = heapq.heappop(open_set)[1]
        developed_nodes += 1

        if current == goal:
            path = [current]
```

```
15              while current in came_from:
16                  current = came_from[current]
17                  path.append(current)
18              path.reverse()
19              return path, g_score[goal], developed_nodes
20
21          for neighbor, move_cost in get_neighbors(grid, current):
22              tentative_g = g_score[current] + move_cost
23              if neighbor not in g_score or tentative_g < g_score[
                    neighbor]:
24                  came_from[neighbor] = current
25                  g_score[neighbor] = tentative_g
26                  f_score[neighbor] = tentative_g + heuristic(neighbor
                        , goal)
27                  heapq.heappush(open_set, (f_score[neighbor],
                        neighbor))
28      return None, float('inf'), developed_nodes
```

L'algorithme **A*** est un algorithme de recherche de chemin qui utilise une combinaison d'une recherche en largeur et d'une heuristique pour guider l'exploration vers la solution la plus prometteuse.

- **Initialisation** :
  - Une **file ouverte** est utilisée pour stocker les nœuds à explorer, triés par un score de priorité basé sur la fonction $f(n) = g(n) + h(n)$.
  - Le score $g(n)$ représente le coût du chemin depuis le début jusqu'au nœud n.
  - Le score $h(n)$ est l'heuristique qui estime le coût restant du nœud n à l'objectif.
- À chaque itération :
  - Le nœud avec le score $f(n)$ le plus bas est extrait de la file ouverte.
  - Si le nœud extrait est l'objectif, l'algorithme reconstruit le chemin en suivant les nœuds parents jusqu'à la position de départ.
  - Pour chaque voisin du nœud courant, on calcule un nouveau score g et un score f.
  - Si le voisin n'a pas encore été exploré ou si un chemin plus court est trouvé, le voisin est ajouté à la file ouverte.
- L'algorithme continue jusqu'à ce que l'objectif soit atteint ou que la file ouverte soit vide (absence de solution).

Avantages :

- Trouve toujours le chemin optimal si l'heuristique est admissible (ne surestime jamais le coût réel).
- Utilise l'heuristique pour guider efficacement la recherche.

Limites :

- La performance dépend de la qualité de l'heuristique.

— Si l'heuristique est mal choisie ou inexistante, il se comporte comme une recherche en largeur et peut être coûteux en termes de temps et de mémoire.

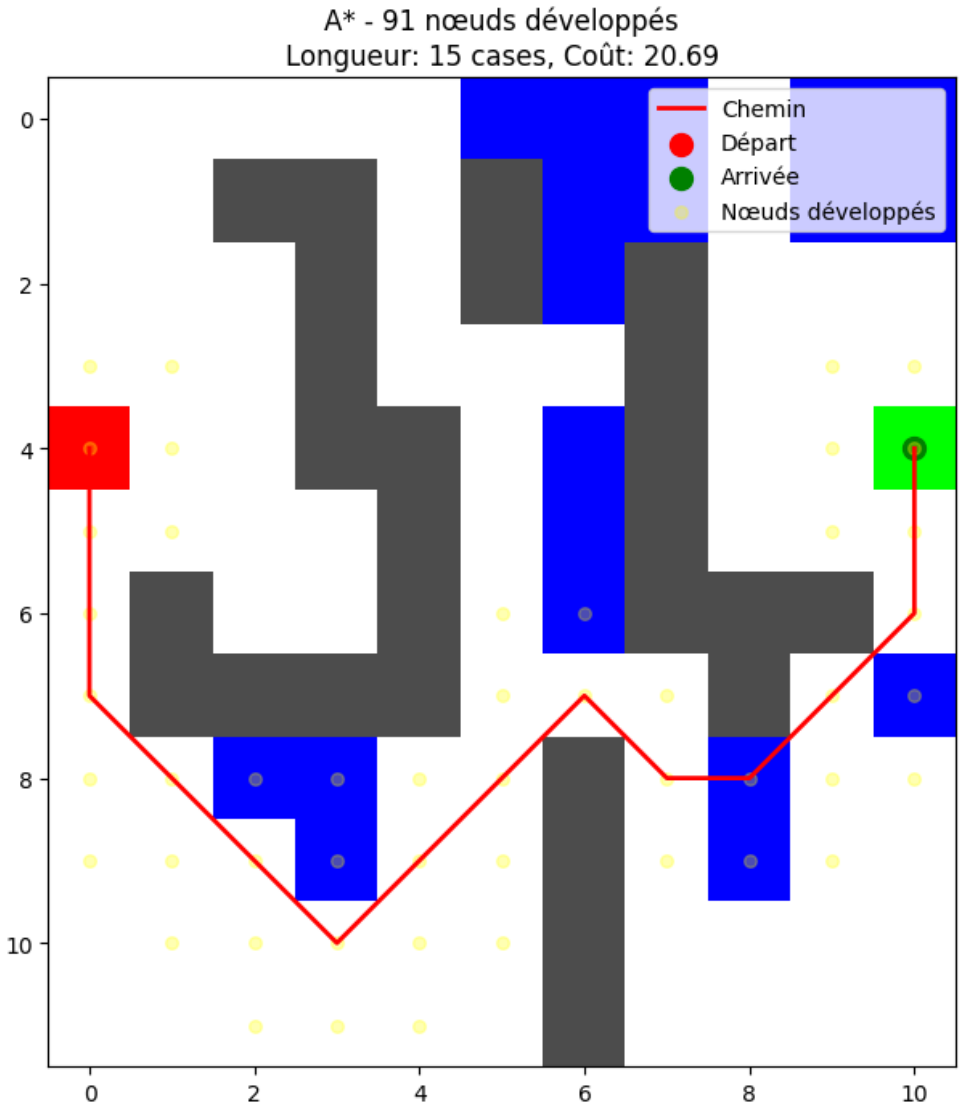


FIGURE 3 – le chemin trouvé : [(4, 0), (5, 0), (6, 0), (7, 0), (8, 1), (9, 2), (10, 3), (9, 4), (8, 5), (7, 6), (8, 7), (8, 8), (7, 9), (6, 10), (5, 10), (4, 10)]

10. Pruning/Élagage (L'algoritheme Forward checking)

```python
# 4. Forward Checking
def forward_checking(grid, start, goal):
    stack = [(start, [start], set([start]), 0)]  # (pos, path, domain, cost)
    developed_nodes = 0

    while stack:
        current, path, domain, cost = stack.pop()
        developed_nodes += 1

        if current == goal:
            return path, cost, developed_nodes

        neighbors = []
        for neighbor, move_cost in get_neighbors(grid, current):
            if neighbor not in domain:
                new_domain = domain.copy()
                new_domain.add(neighbor)
                neighbors.append((neighbor, move_cost, new_domain))

        # Tri par heuristique (optionnel)
        neighbors.sort(key=lambda x: heuristic(x[0], goal))

        for neighbor, move_cost, new_domain in neighbors:
            stack.append((neighbor, path + [neighbor], new_domain,
                cost + move_cost))

    return None, float('inf'), developed_nodes
```

L'algorithme **Forward Checking** est une variante améliorée de la recherche en profondeur (DFS) qui tente de réduire l'espace de recherche en anticipant les chemins invalides.

- Il utilise une **pile** pour explorer les chemins de manière récursive (approche en profondeur).
- Chaque état dans la pile contient :
  - la position actuelle,
  - le chemin parcouru,
  - le **domaine** (ensemble des nœuds déjà visités),
  - et le **coût** accumulé.
- À chaque itération :
  - les voisins du nœud courant sont générés,
  - seuls les voisins **non présents dans le domaine** sont considérés,
  - ces voisins peuvent être **triés selon une heuristique** (ex : distance à la cible),

— chaque voisin est ajouté à la pile avec le nouveau domaine, le chemin mis à jour, et le nouveau coût.

— Le processus s'arrête dès que la position objectif est atteinte.

Avantages :

— Évite de revisiter les nœuds déjà explorés.

— Réduit l'exploration inutile grâce à une anticipation locale.

Limites :

— Peut développer un très grand nombre de nœuds si les contraintes sont peu restrictives.

— Moins performant que des méthodes comme A* pour trouver des chemins optimaux.

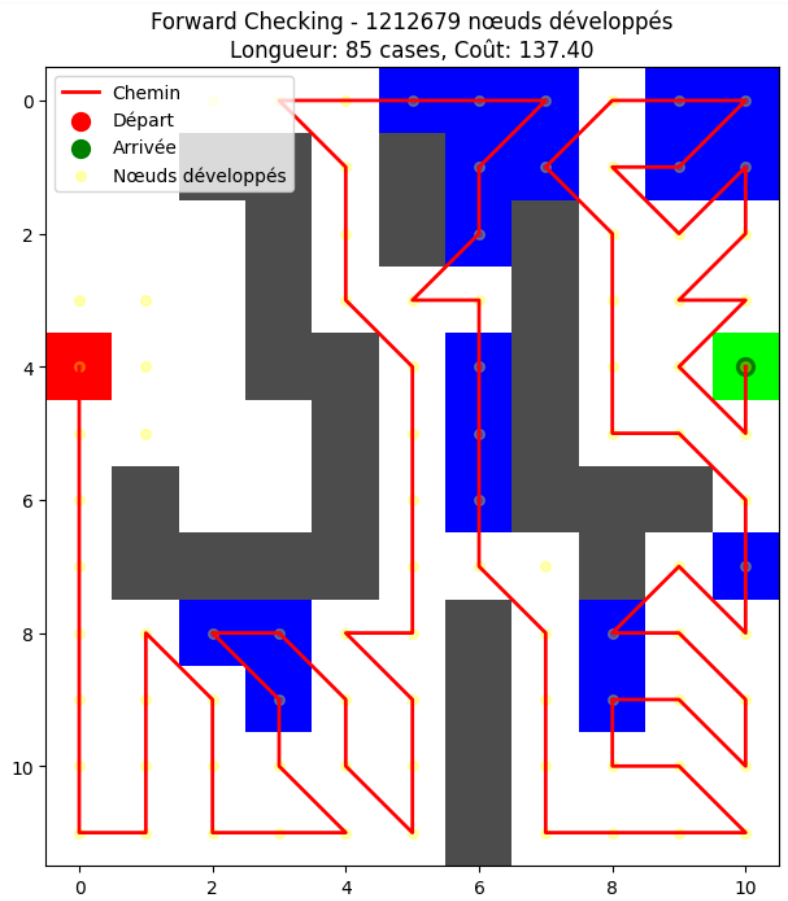


FIGURE 4 – le chemin trouvé :[(4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (10, 0), (11, 0), (11, 1), (10, 1), (9, 1), (8, 1), (9, 2), (10, 2), (11, 2), (11, 3), (11, 4), (10, 3), (9, 3), (8, 2), (8, 3), (9, 4), (10, 4), (11, 5), (10, 5), (9, 5), (8, 4), (8, 5), (7, 5), (6, 5), (5, 5), (4, 5), (3, 4), (2, 4), (1, 4), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (1, 6), (2, 6), (3, 5), (3, 6), (4, 6), (5, 6), (6, 6), (7, 6), (8, 7), (9, 7), (10, 7), (11, 7), (11, 8), (11, 9), (11, 10), (10, 9), (10, 8), (9, 8), (9, 9), (10, 10), (9, 10), (8, 9), (8, 8), (7, 9), (8, 10), (7, 10), (6, 10), (5, 9), (5, 8), (4, 8), (3, 8), (2, 8), (1, 7), (0, 8), (0, 9), (0, 10), (1, 9), (1, 8), (2, 9), (1, 10), (2, 10), (3, 9), (3, 10), (4, 9), (5, 10), (4, 10)]

11. Analyse des performances des algorithmes

11.1. Résumé des résultats obtenus

Le tableau ci-dessous récapitule les coûts, le nombre de nœuds développés, le temps d'exécution, ainsi que l'information sur l'optimalité du chemin trouvé.

	Algorithme	Coût total	Nœuds développés	Temps (ms)	Optimal
0	BFS	21.29	87	1.33	Non
1	DFS	34.22	31	0.48	Non
2	A*	20.69	91	1.72	Oui
3	Forward Checking	137.40	1212679	14092.30	Non

11.2. Visualisations et interprétations

Dans cette partie, nous proposons 5 graphiques pour comparer la performance des algorithmes DFS, BFS, A* et Forward Checking.

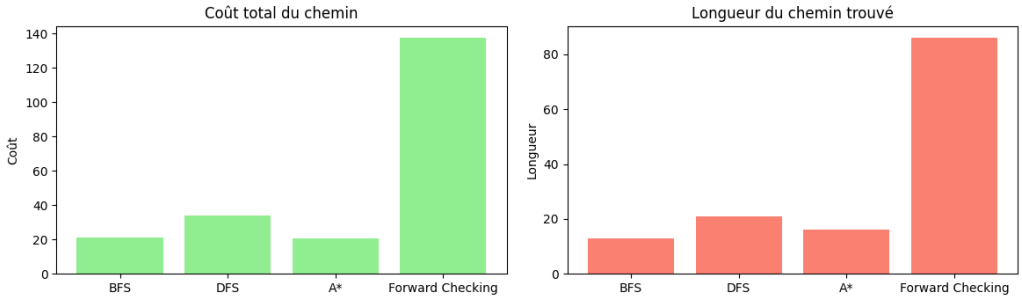


FIGURE 5 – Comparaison en terme du cout total et de la longueur du chemin trouvé

Graphique de gauche – Coût total du chemin

Le coût total du chemin est :

- Le plus bas pour BFS et A* (~21).
- Moyen pour DFS (~35).
- Très élevé pour Forward Checking (~138).

⇒ **Forward Checking** trouve un chemin très coûteux, ce qui montre une inefficacité dans l'optimisation du trajet.

Graphique de droite – Longueur du chemin trouvé

La longueur du chemin :

- Est la plus courte pour BFS et A* (autour de 15–16).
- Légèrement plus longue pour DFS (~21).
- Extrêmement longue pour Forward Checking (~87).

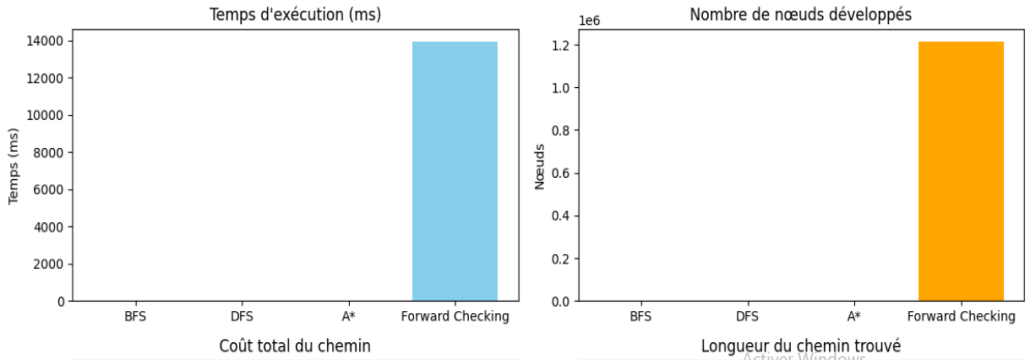


FIGURE 6 – Comparaison entre les algorithmes en termes de temps d'exécution et nombre de neuds développés

⇒ **Cela confirme** que Forward Checking trouve un chemin bien plus long et donc probablement non optimal, malgré sa stratégie de « prévoyance ».

Graphique de gauche – Temps d'exécution (ms)

Forward Checking est nettement plus lent que les autres algorithmes : il atteint environ 14 000 ms (~14 secondes).

BFS, DFS, et A* semblent avoir un temps négligeable en comparaison (quasiment invisible ici).

⇒ Cela suggère que Forward Checking, bien qu'il essaie d'éviter les mauvais chemins, effectue beaucoup plus de calculs, ce qui ralentit fortement l'algorithme.

Graphique de droite – Nombre de nœuds développés

Forward Checking développe plus de 1,2 million de nœuds, ce qui est très élevé.

Les autres algorithmes (BFS, DFS, A*) développent beaucoup moins de nœuds (leurs barres sont invisibles ici car négligeables en comparaison).

⇒ Cela indique que Forward Checking explore beaucoup trop de combinaisons ou que sa stratégie d'évitement des obstacles n'est pas efficace dans ce cas.

12. Interprétation de la consommation mémoire

Consommation mémoire par algorithme :

— BFS (Breadth-First Search) est de loin le plus gourmand en mémoire (~1,85 KB).

— A* et DFS utilisent une quantité modérée (~0,95 KB et ~0,91 KB respectivement).

— Forward Checking consomme le moins de mémoire (~0,82 KB).

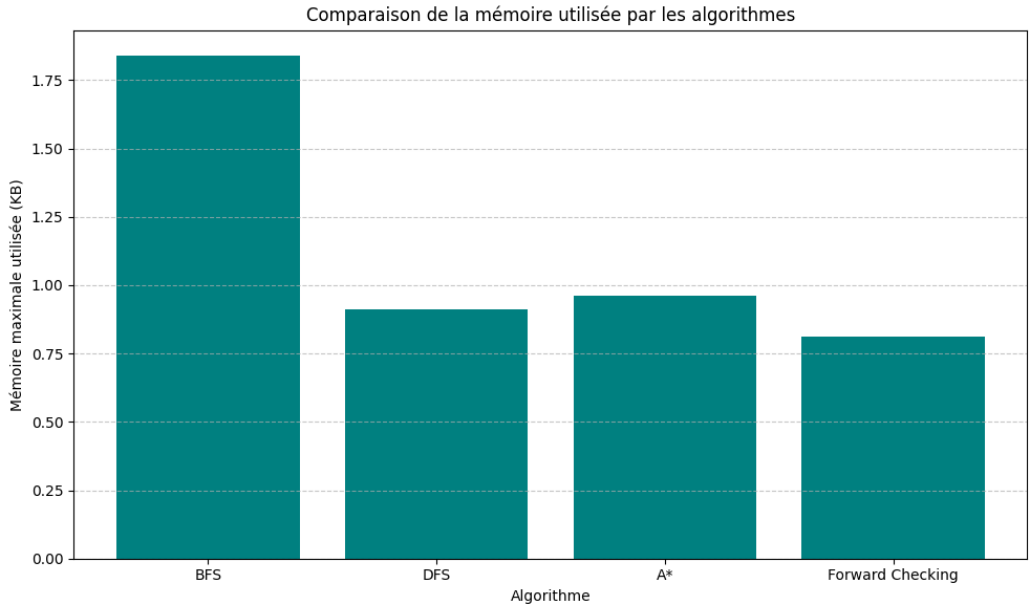


FIGURE 7 – Comparaison de la consommation mémoire entre les algorithmes

Analyse :

— BFS explore tous les voisins à chaque niveau, ce qui nécessite de stocker beaucoup de nœuds en mémoire, expliquant sa consommation élevée.

— DFS suit un chemin en profondeur et ne garde en mémoire que les nœuds de la branche actuelle, ce qui réduit son empreinte mémoire.

— A* garde les scores et une file de priorité, donc une mémoire modérée, mais plus optimisée que BFS.

— Forward Checking, bien qu'inefficace en coût et longueur de chemin (selon les graphiques précédents), gère sa mémoire efficacement, probablement car il évite l'exploration inutile de certaines branches.