Rapport :

Exploration de Graphe et Implémentation en Python

Réalisé par :

_ Souci Nour el houda 202031050569

_ Bouyemmout Meriem 202031029144

Introduction

L'objectif de ce projet est d'implémenter et d'explorer un graphe d'espace d'états représentant un problème de recherche avec six sommets (`S`, `A`, `B`, `C`, `D`, `G`). Nous avons utilisé plusieurs algorithmes d'exploration tels que :

1. **Recherche en largeur d'abord (BFS)**.
2. **Recherche à coût uniforme (UCS)**.
3. **Recherche en profondeur d'abord (DFS)**.
4. **Recherche A*** avec une heuristique cohérente.

Le graphe est également visualisé avec les chemins trouvés par chaque algorithme.

Étape 1 : Représentation du Graphe

Le graphe a été représenté sous forme de dictionnaire d'adjacence en Python. Cette méthode est efficace pour stocker des graphes dirigés pondérés. Voici la structure utilisée :

```
graph = {
    'S': {'A': 1, 'G': 12},
    'A': {'B': 3, 'C': 1},
    'B': {'D': 3},
    'C': {'D': 1, 'G': 2},
    'D': {'G': 3},
    'G': {}
}
```

Chaque clé est un sommet, et sa valeur est un dictionnaire contenant les voisins accessibles avec les coûts des arcs.

Étape 2 : Génération de la Matrice d'Adjacence

Une matrice d'adjacence permet de visualiser les connexions et coûts entre les sommets du graphe. Voici les étapes :

1. Liste de tous les sommets : `['S', 'A', 'B', 'C', 'D', 'G']`.
2. Matrice initialisée avec des coûts infinis (∞) pour représenter l'absence d'arc.
3. Mise à jour des coûts pour chaque paire de sommets connectés.

Code :

```
import numpy as np

nodes = ['S', 'A', 'B', 'C', 'D', 'G']
n = len(nodes)
```

```
adj_matrix = np.full((n, n), float('inf'))

for i, node in enumerate(nodes):
    for neighbor, cost in graph.get(node, {}).items():
        j = nodes.index(neighbor)
        adj_matrix[i][j] = cost

print("Matrice d'adjacence:")
print(adj_matrix)
```

Étape 3 : Algorithmes d'Exploration

a. Recherche en largeur d'abord (BFS)

L'algorithme BFS explore tous les voisins d'un sommet avant de passer au niveau suivant. L'implémentation utilise une file (queue) et un ensemble pour éviter de revisiter les nœuds.

Code :

```
from collections import deque

def bfs(graph, start, goal):
    queue = deque([[start]])
    visited = set()

    while queue:
        path = queue.popleft()
        node = path[-1]

        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for neighbor in sorted(graph[node]):
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)

    return None

print("Chemin BFS:", bfs(graph, 'S', 'G'))
```

---

b. Recherche à coût uniforme (UCS)

UCS est similaire à BFS, mais il explore les nœuds avec le coût cumulatif le plus faible. Une file de priorité (`heapq`) est utilisée pour gérer les nœuds.

Code :

```
import heapq

def ucs(graph, start, goal):
    queue = [(0, start, [])]
    visited = set()
```

```
    while queue:
        (cost, node, path) = heapq.heappop(queue)

        if node in visited:
            continue
        visited.add(node)

        path = path + [node]

        if node == goal:
            return path, cost

        for neighbor, edge_cost in sorted(graph[node].items()):
            if neighbor not in visited:
                heapq.heappush(queue, (cost + edge_cost, neighbor, path))

    return None

ucs_path, ucs_cost = ucs(graph, 'S', 'G')
print("Chemin UCS:", ucs_path, "Coût:", ucs_cost)
```

---

c. Recherche en profondeur d'abord (DFS)

L'algorithme DFS explore autant que possible le long d'un chemin avant de revenir en arrière. Une pile (stack) est utilisée pour implémenter DFS.

Code :

```
def dfs(graph, start, goal):
    stack = [(start, [start])]
    visited = set()

    while stack:
        (node, path) = stack.pop()

        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for neighbor in sorted(graph[node], reverse=True):
                stack.append((neighbor, path + [neighbor]))

    return None

print("Chemin DFS:", dfs(graph, 'S', 'G'))
```

---

d. Recherche A*

L'algorithme A* utilise une heuristique pour estimer le coût restant vers le but. Cette implémentation utilise une file de priorité pour prioriser les nœuds.

Heuristique utilisée :

```
heuristic = {
    'S': 7, 'A': 6, 'B': 2, 'C': 1, 'D': 1, 'G': 0
}
```

Code :

```
def a_star(graph, start, goal, heuristic):
    queue = [(heuristic[start], 0, start, [])]
    visited = set()

    while queue:
        (f_cost, cost, node, path) = heapq.heappop(queue)

        if node in visited:
            continue
        visited.add(node)

        path = path + [node]

        if node == goal:
            return path, cost

        for neighbor, edge_cost in sorted(graph[node].items()):
            if neighbor not in visited:
                g_cost = cost + edge_cost
                f_cost = g_cost + heuristic[neighbor]
                heapq.heappush(queue, (f_cost, g_cost, neighbor, path))

    return None

a_star_path, a_star_cost = a_star(graph, 'S', 'G', heuristic)
print("Chemin A*:", a_star_path, "Coût:", a_star_cost)
```

Étape 4 : Visualisation avec `networkx`

Le module `networkx` a été utilisé pour visualiser le graphe et les chemins. Chaque méthode d'exploration est représentée par une couleur distincte.

Code :

```
import networkx as nx
import matplotlib.pyplot as plt

# Création du graphe
G = nx.DiGraph()
for node, neighbors in graph.items():
    for neighbor, cost in neighbors.items():
        G.add_edge(node, neighbor, weight=cost)

# Positionnement des nœuds
pos = nx.spring_layout(G)

# Couleurs des chemins
path_colors = {
    'BFS': ('blue', bfs(graph, 'S', 'G')),
    'UCS': ('green', ucs_path),
    'DFS': ('orange', dfs(graph, 'S', 'G')),
```

```
    'A*': ('red', a_star_path)
}

# Dessiner le graphe de base
nx.draw(G, pos, with_labels=True, node_color="lightblue", node_size=2000,
font_size=10, font_weight="bold")
nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v): G[u][v]['weight']
for u, v in G.edges})

# Dessiner les chemins de chaque méthode en couleurs différentes
for method, (color, path) in path_colors.items():
    if path:
        edges_in_path = [(path[i], path[i + 1]) for i in range(len(path) -
1)]
        nx.draw_networkx_edges(G, pos, edgelist=edges_in_path,
edge_color=color, width=2, label=method)

plt.legend(handles=[plt.Line2D([0], [0], color=color, lw=4) for color, _ in
path_colors.values()],
           labels=path_colors.keys(), loc='upper left')
plt.title("Différents chemins d'exploration dans le graphe")
plt.show()
```

Conclusion

Ce projet a permis d'explorer différents algorithmes et de comprendre leurs comportements dans un problème de recherche. La visualisation avec `networkx` facilite l'interprétation des résultats. Chaque méthode a ses avantages selon les cas d'usage :

- **BFS** : Optimal pour les chemins courts en nombre de nœuds.
- **UCS** : Optimal pour les chemins à coût minimum.
- **DFS** : Rapide mais peut ne pas être optimal.
- **A*** : Combinaison puissante avec une bonne heuristique.

Bibliothèques utilisées

1. `numpy` : Manipulation de la matrice d'adjacence.
2. `collections` : Structures comme `deque` et `heapq`.
3. `networkx` : Création et visualisation du graphe.
4. `matplotlib` : Tracé du graphe.
