UNIVERSITE SIDI MOHAMED BEN ABDELLAH
Faculté des Sciences Dhar El Mahraz – Fès
Master BDSAS
Année universitaire 2023-2024

جامعة سيدي محمد بن عبد الله
كلية العلوم ظهر المهراز
- فاس-

FACULTÉ DES SCIENCES DHAR EL MAHRAZ
UNIVERSITÉ SIDI MOHAMED BEN ABDELLAH

Nom:ELHAGOUCHI

Prenom:HALIMA

Master BDSAS

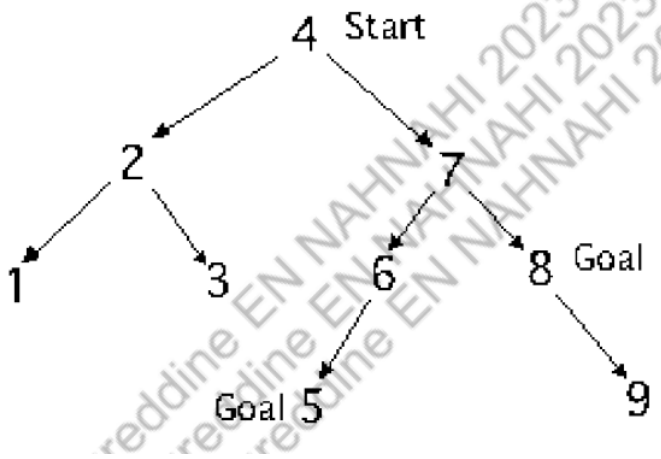
Exercice 1

Exercice 1 :

Dans l'espace d'état ci-dessous, écrivez l'ordre dans lequel les états sont développés si l'état initial est 4 et qu'il y a deux états cibles : 5 et 8.

Par exemple, pour une première recherche étendue, la réponse serait : 4 2 7 1 3 6 8

1. en utilisant la recherche en profondeur d'abord
2. en utilisant l'approfondissement itératif.



1. RESOLUTION AVEC PYTHON :

- **Dfs :**

```
def dfs_with_goals(graph, start, goals, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)  # You can do something with the node here

    if start in goals:
```

```
        print(f"Goal {start} reached!")
        goals.remove(start)  # Remove the goal from the list once reached
        if not goals:
            print("All goals reached!")
            return

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs_with_goals(graph, neighbor, goals, visited)

# Example usage with multiple goals
graph = {
    4: [2, 7],
    2: [1, 3],
    7: [6, 8],
    1: [],
    3: [],
    6: [5],
    8: [9],
    5: [],
    9: []
}

start_node = 4
```

goal_nodes = [8, 5]

dfs_with_goals(graph, start_node, goal_nodes)

L’AFFICHAGE:

```
4
2
1
3
7
6
5
Goal 5 reached!
8
Goal 8 reached!
All goals reached!
```

- **BFS:**

```
from collections import deque

def bfs_with_goals(graph, start, goals):
    visited = set()
    queue = deque([start])

    while queue:
        current_node = queue.popleft()
        print(current_node)  # Vous pouvez faire ce que vous voulez avec le nœud ici
```

```
        if current_node in goals:
            print(f"Goal {current_node} reached!")
            goals.remove(current_node)
            if not goals:
                print("All goals reached!")
                return

        visited.add(current_node)

        for neighbor in graph[current_node]:
            if neighbor not in visited and neighbor not in queue:
                queue.append(neighbor)

# Exemple d'utilisation avec deux objectifs spécifiés

graph = {
    4: [2, 7],
    2: [1, 3],
    7: [6, 8],
    1: [],
    3: [],
    6: [5],
    8: [9],
    5: [],
```

9: []

}

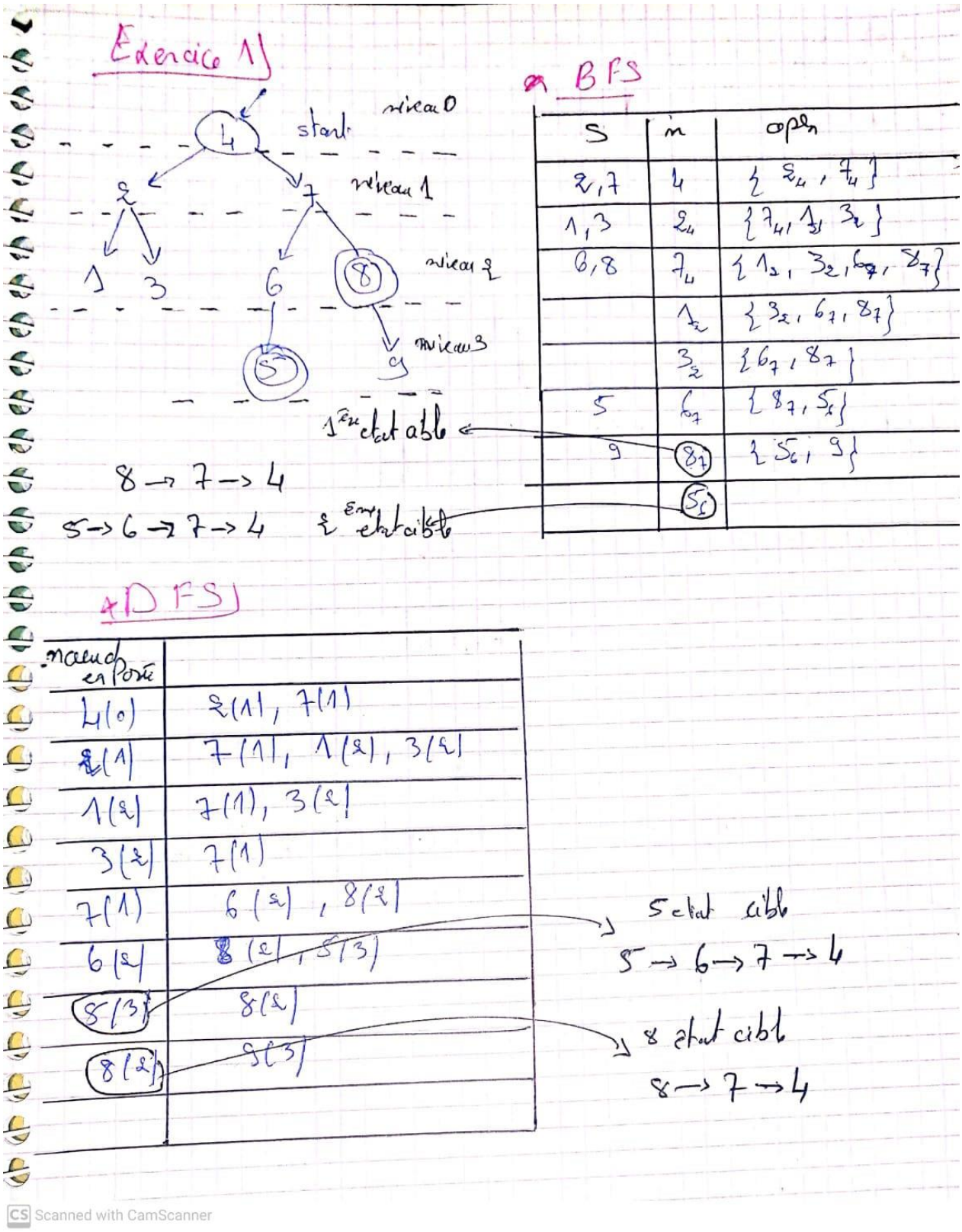
start_node = 4

goal_nodes = [8,5]

bfs_with_goals(graph, start_node, goal_nodes)

L’AFICHAGE:

```
4
2
7
1
3
6
8
Goal 8 reached!
5
Goal 5 reached!
All goals reached!
```



S	n	open
2,7	4	{2₄, 7₄}
1,3	2₄	{7₄, 1₂, 3₂}
6,8	7₄	{1₂, 3₂, 6₇, 8₇}
	1₂	{3₂, 6₇, 8₇}
	3₂	{6₇, 8₇}
5	6₇	{8₇, 5₆}
9	8₇	{5₆, 9}
	5₆	

noeud en Pose	
4(0)	2(1), 7(1)
2(1)	7(1), 1(2), 3(2)
1(2)	7(1), 3(2)
3(2)	7(1)
7(1)	6(2), 8(2)
6(2)	8(2), 5(3)
5(3)	8(2)
8(2)	5(3)

2. LA PROFONDEURE ITERATIF

```
def depth_limited_dfs(graph, start, goals, depth_limit):
    stack = [(start, 0)]  # Stack with the node and its depth
    visited = set()

    while stack:
        current_node, depth = stack.pop()
        print(current_node)  # You can do whatever you want with the node here

        if current_node in goals:
            print(f"Goal {current_node} reached!")
            goals.remove(current_node)
            if not goals:
                print("All goals reached!")
                return True

        if depth < depth_limit:
            visited.add(current_node)

            # Add unvisited neighbors to the stack with increased depth
            for neighbor in graph[current_node]:
                if neighbor not in visited:
                    stack.append((neighbor, depth + 1))

    return False  # No path found within depth limit

def iterative_deepening_dfs(graph, start, goals):
    depth_limit = 0

    while True:
        print(f"Searching at depth limit: {depth_limit}")
        if depth_limited_dfs(graph, start, goals.copy(), depth_limit):
            return True  # Solution found
        depth_limit += 1

```

```
        # Reset the set of visited nodes for the next iteration
        visited.clear()

# Example usage with an undirected graph
graph = {
    4: [2, 7],
    2: [1, 3],
    7: [6, 8],
    1: [],
    3: [],
    6: [5],
    8: [9],
    5: [],
    9: []
}

start_node = 4
goal_nodes = [8,5]
iterative_deepening_dfs(graph, start_node, goal_nodes)
```

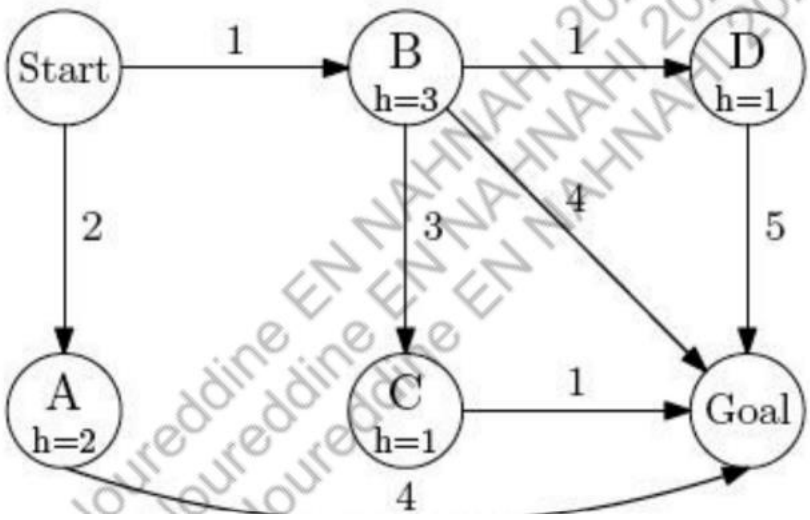
L'AFFICHAGE:

```
Searching at depth limit: 0
4
Searching at depth limit: 1
4
7
2
Searching at depth limit: 2
4
7
8
Goal 8 reached!
6
2
3
1
Searching at depth limit: 3
4
7
8
Goal 8 reached!
9
6
```

```
5
Goal 5 reached!
All goals reached!
```

EXERCICE2

1. Recherche en profondeur d'abord
2. Recherche en largeur d'abord
3. Recherche de coûts uniformes
4. Recherche A*.



* 2) l'approfondissement itératif :

* limite ① :

⇒ 4(0)
3(1)
7(1)

on a pas rencontré les deux états cibles 5 et 8 alors on va passer à limite 2

* limite ② :

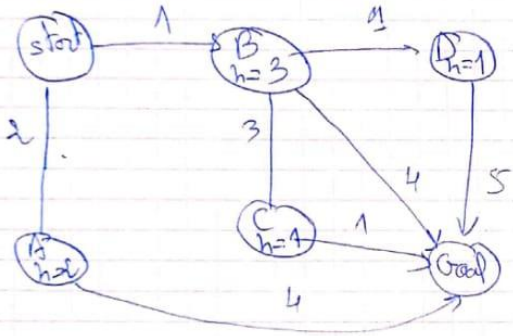
4(0)
3(1)
7(1)
1(2)
3(2)
6(2)
8(2)
⇒ 8 → 7 → 4

on a rencontré une seule état cible c'est 8 alors on doit passer à limite 3

* limite 3 :

4(0)
3(1)
7(1)
1(2)
3(2)
6(2)
8(2)
5(3)

Exercice 2)



* DFS

noeud en cours	
start(0)	{A(2), B(3)}
A(2)	{B(3), Goal}
Goal	{B(3)}

start → A → Goal (2, 4)

avec un coût de 6

* BFS

S	n	opérateurs
A, B	start	{A_start, B_start}
Goal	A_start	{B_start, Goal}
D, Goal, C	B_start	{Goal, D, Goal, C}
	Goal	{D, C}

start → A → Goal (2, 4)

Coût = 6

- **DFS:**

```
def dfs_with_goal(graph, start, goal, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)  # Vous pouvez faire ce que vous voulez avec le nœud ici

    if start == goal:
        print("Goal reached!")
        return

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs_with_goal(graph, neighbor, goal, visited)
            # Ajoutez une condition pour arrêter la recherche si le goal est atteint
            if goal in visited:
                return

# Exemple d'utilisation avec un objectif spécifié
graph = {'start': ['A', 'B'], 'A': ['goal'], 'B': ['C','goal','D'],
         'D': ['goal'], 'C': ['goal']}

start_node = 'start'
goal_node = 'goal'
dfs_with_goal(graph, start_node, goal_node)
```

L'AFFICHAGE:

```
start
A
```

```
goal
Goal reached!
```

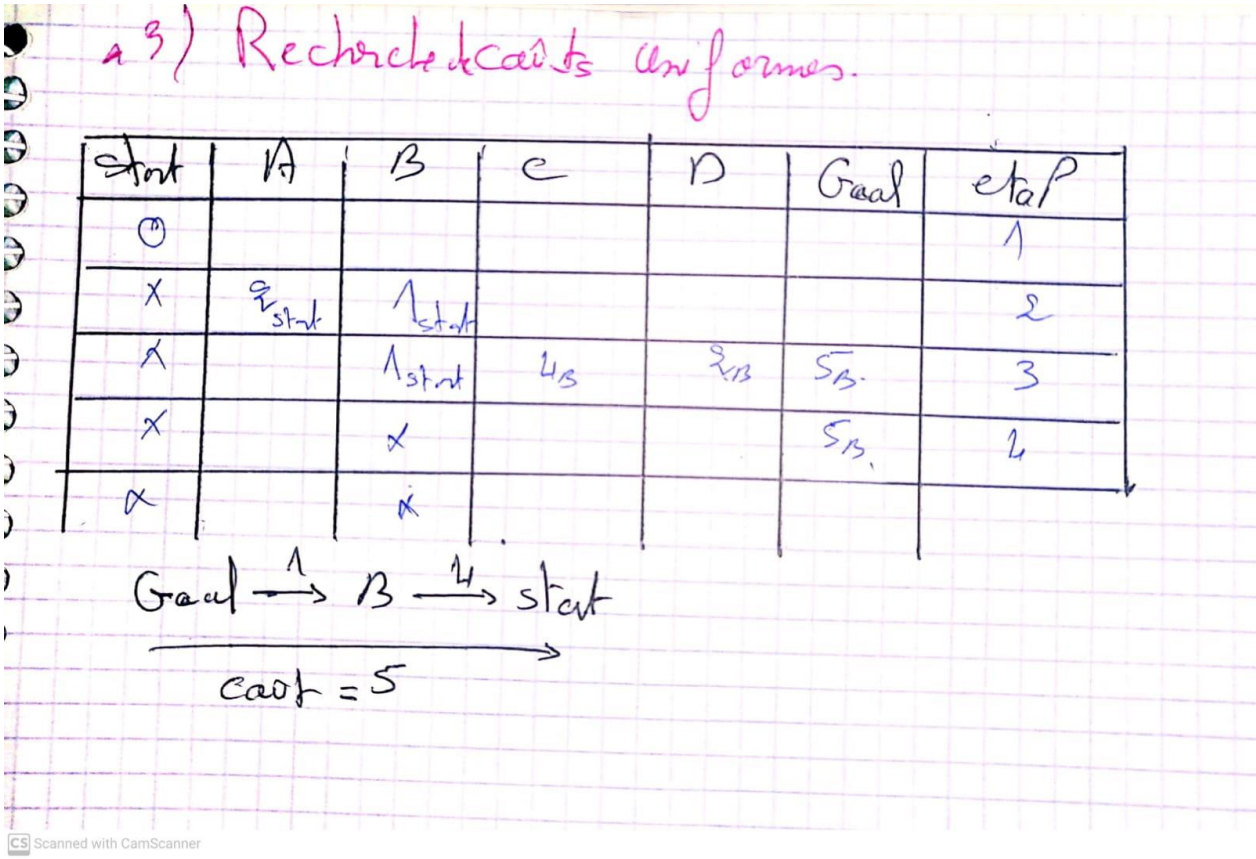
- **BFS:**

```
from collections import deque

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

    while queue:
        current_node = queue.popleft()
        print(current_node)  # Vous pouvez faire ce que vous voulez avec le nœud ici

        if current_node == goal:
            print("Goal reached!")
            return

        visited.add(current_node)

        for neighbor in graph[current_node]:
            if neighbor not in visited and neighbor not in queue:
                queue.append(neighbor)

# Exemple d'utilisation avec un objectif spécifié
graph = {'start': ['A', 'B'], 'A': ['goal'], 'B': ['C','goal','D'],
        'D': ['goal'], 'C': ['goal']}

start_node = 'start'
goal_node = 'goal'
bfs(graph, start_node, goal_node)
```

L'AFFICHAGE:

```
start
A
B
goal
Goal reached!
```

- **RECHERCHE DE COUT UNIFORME**



'''

import heapq: Importe le module heapq, qui fournit une implémentation de l'algorithme de tas (heap). Il est utilisé ici pour créer une file de priorité

pour une manipulation efficace des nœuds avec le coût cumulatif le plus bas.

'''

```
import heapq

def uniform_cost_search(graph, start, goal):
    # Priority queue pour stocker les nœuds à explorer
```

'''

priority_queue = [(0, start)]: Initialise une file de priorité avec un tuple contenant le

coût cumulatif (initialisé à 0) et le nœud de départ. La file de

priorité est un tas min, assurant que le nœud avec le coût cumulatif le plus bas est en tête.

'''

priority_queue = [(0, start)] # (coût cumulatif, nœud)

'''

costs = {start: 0}: Initialise un dictionnaire (costs) pour suivre les coûts

cumulatifs associés à chaque nœud. Le coût pour le nœud de départ est initialement fixé à 0.

'''

Dictionnaire pour suivre les coûts cumulatifs actuels

costs = {start: 0}

'''

while priority_queue::

Démarre une boucle qui continue tant qu'il y a des nœuds dans la file de priorité à traiter.

'''

while priority_queue:

'''

current_cost, current_node = heapq.heappop(priority_queue): Dépile le nœud avec le

coût cumulatif le plus bas de la file de priorité. Ce nœud devient le nœud actuel à traiter.

'''

```
    current_cost, current_node = heapq.heappop(priority_queue)


    if current_node == goal:
        print(f"Goal {goal} reached with cost {current_cost}")
        return
```

'''

for neighbor, edge_cost in graph[current_node].items()::

Itère à travers les voisins du nœud actuel et leurs coûts d'arête associés.

new_cost = current_cost + edge_cost: Calcule le nouveau coût cumulatif pour atteindre le nœud voisin.

'''

```
    for neighbor, edge_cost in graph[current_node].items():
        new_cost = current_cost + edge_cost
```

'''

Vérifie si le voisin n'a pas été visité ou si le nouveau coût est inférieur au coût précédemment enregistré pour le voisin. Si c'est vrai, met à jour

le coût et réinsère le voisin dans la file de priorité avec le nouveau coût cumulatif.

'''

```
        if neighbor not in costs or new_cost < costs[neighbor]:
            costs[neighbor] = new_cost
            heapq.heappush(priority_queue, (new_cost, neighbor))


# Exemple d'utilisation avec un graphe pondéré
graph = {'start': {'A':2, 'B':1}, 'A': {'goal':4}, 'B': {'C':3,'goal':4,'D':1},
      'D': {'goal':5}, 'C': {'goal':1}}


start_node = 'start'
goal_node = 'goal'
uniform_cost_search(graph, start_node, goal_node)
```

L'AFFICHAGE:

```
Goal goal reached with cost 5
```