***UNIVERSITE SIDI MOHAMED BEN ABDELLAH***

***جامعة سيدي محمد بن عبد الله***

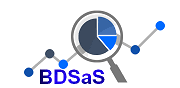
**كلية العلوم ظهر المهراز**

**- فاس-**

***Faculté des Sciences Dhar El Mahraz – Fès***

Master BDSAS

***Année universitaire 2023-2024***

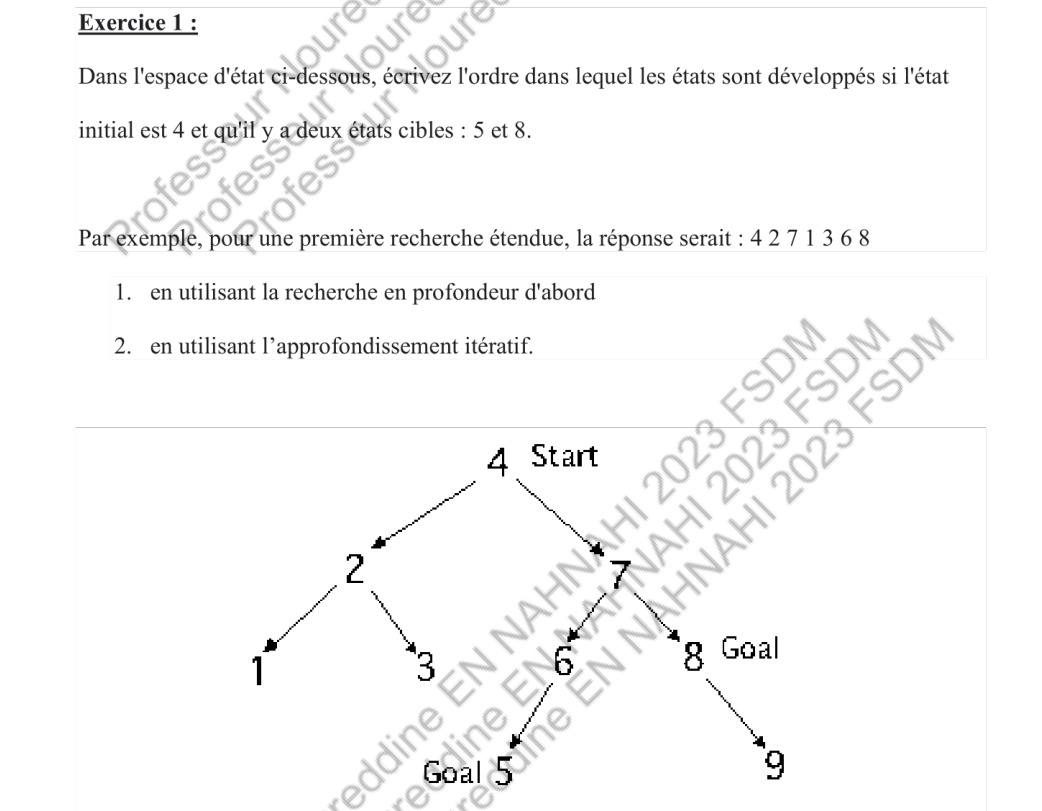
****

**Nom:ELHAGOUCHI**

**Prenom:HALIMA**

**Master BDSAS**

**Exercice 1**



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

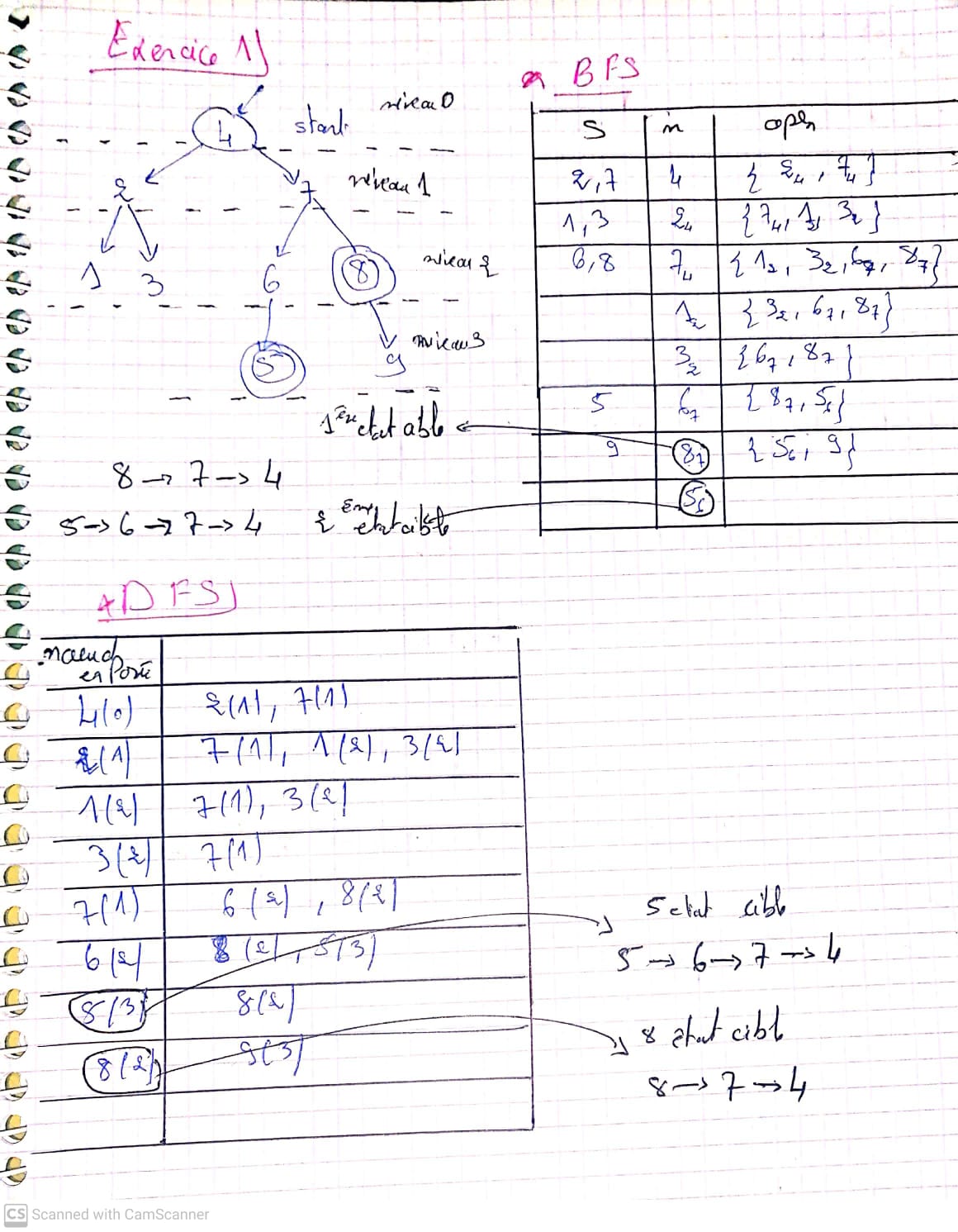
8

Goal 8 reached!

5

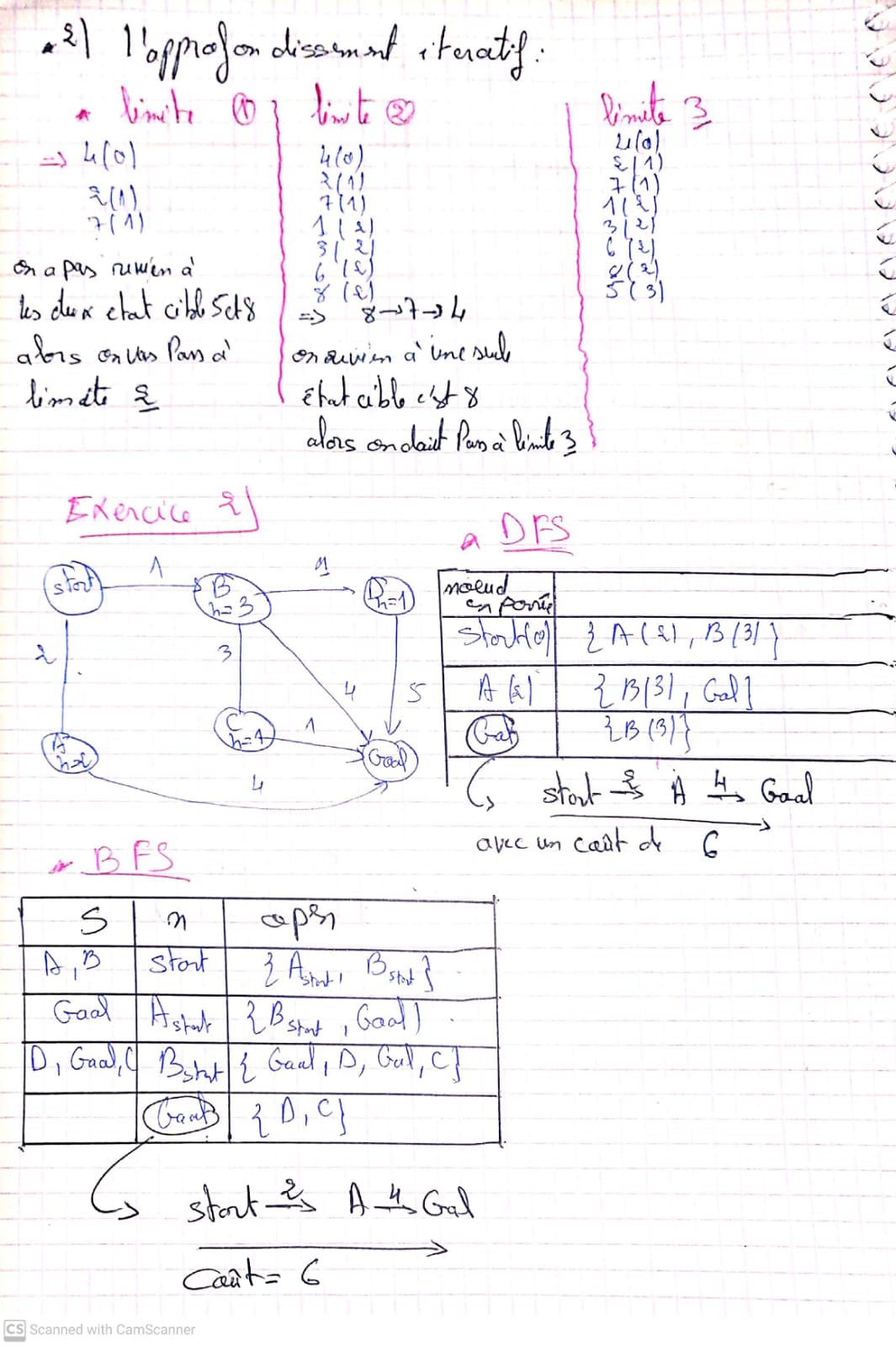
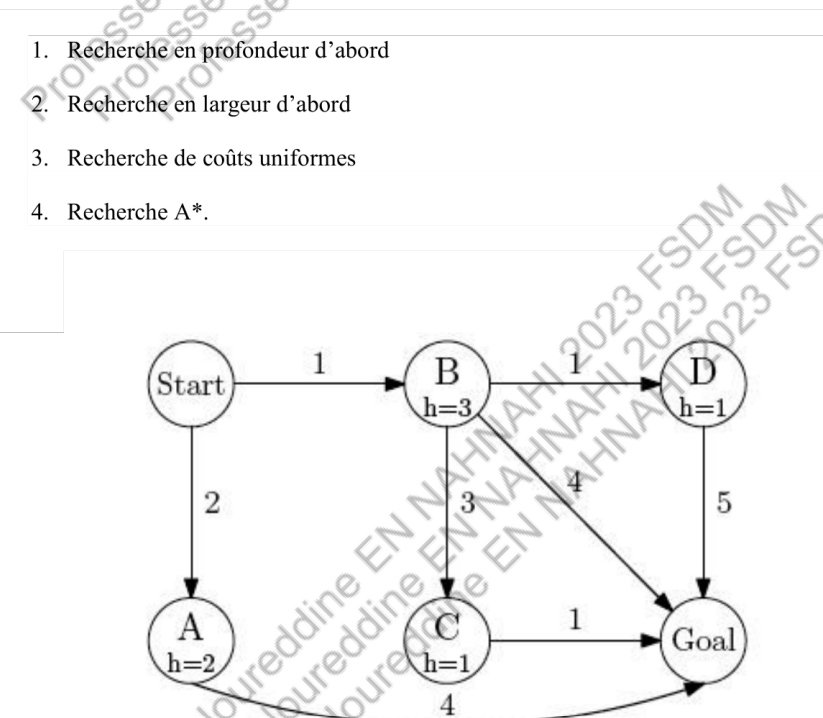
Goal 5 reached!

All goals reached!

****

1. **LA PROFONDEURE ITERATIF**
2. def depth\_limited\_dfs(graph, start, goals, depth\_limit):
3. stack = [(start, 0)] # Stack with the node and its depth
4. visited = set()
5. while stack:
6. current\_node, depth = stack.pop()
7. print(current\_node) # You can do whatever you want with the node here
8. if current\_node in goals:
9. print(f"Goal {current\_node} reached!")
10. goals.remove(current\_node)
11. if not goals:
12. print("All goals reached!")
13. return True
14. if depth < depth\_limit:
15. visited.add(current\_node)
16. # Add unvisited neighbors to the stack with increased depth
17. for neighbor in graph[current\_node]:
18. if neighbor not in visited:
19. stack.append((neighbor, depth + 1))
20. return False # No path found within depth limit
21. def iterative\_deepening\_dfs(graph, start, goals):
22. depth\_limit = 0
23. while True:
24. print(f"Searching at depth limit: {depth\_limit}")
25. if depth\_limited\_dfs(graph, start, goals.copy(), depth\_limit):
26. return True # Solution found
27. depth\_limit += 1
28. # Reset the set of visited nodes for the next iteration
29. visited.clear()
30. # Example usage with an undirected graph
31. graph = {
32. 4: [2, 7],
33. 2: [1, 3],
34. 7: [6, 8],
35. 1: [],
36. 3: [],
37. 6: [5],
38. 8: [9],
39. 5: [],
40. 9: []
41. }
42. start\_node = 4
43. goal\_nodes = [8,5]
44. iterative\_deepening\_dfs(graph, start\_node, goal\_nodes)
45. **L’AFFICHAGE:**
46. Searching at depth limit: 0
47. 4
48. Searching at depth limit: 1
49. 4
50. 7
51. 2
52. Searching at depth limit: 2
53. 4
54. 7
55. 8
56. Goal 8 reached!
57. 6
58. 2
59. 3
60. 1
61. Searching at depth limit: 3
62. 4
63. 7
64. 8
65. Goal 8 reached!
66. 9
67. 6
68. 5
69. Goal 5 reached!
70. All goals reached!

**EXERCICE2**



* **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

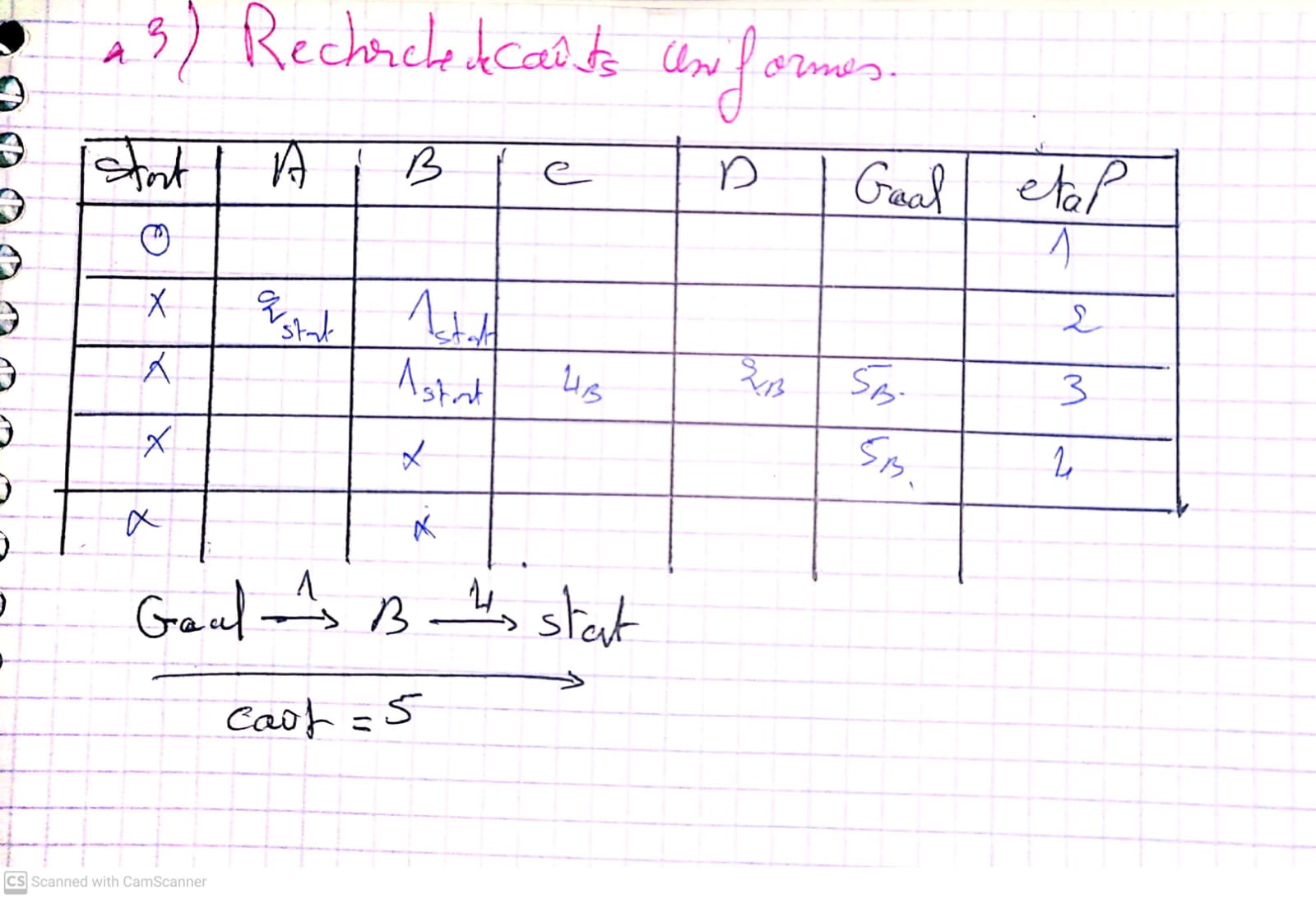
A

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