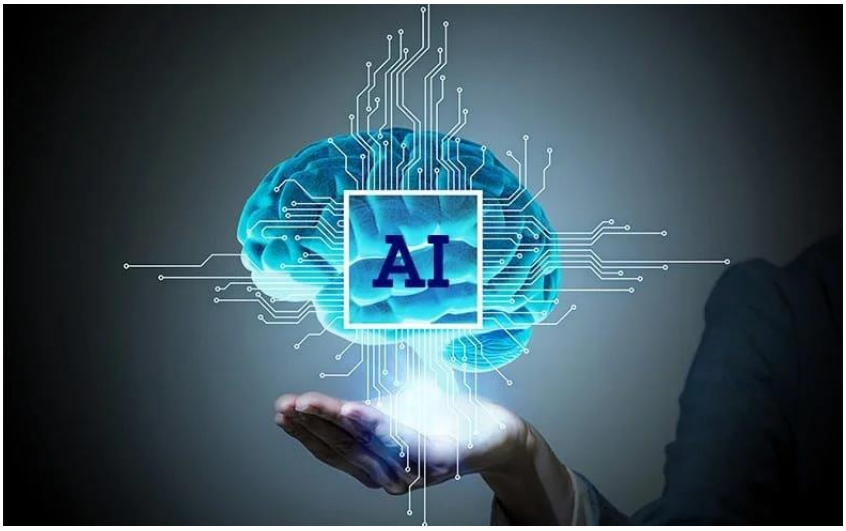
BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

KHOA TOÁN - TIN HỌC

BÀI TẬP THỰC HÀNH TUẦN 1

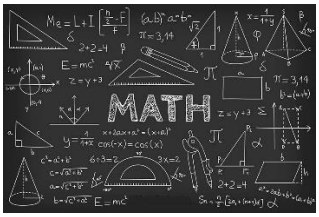


MÔN HỌC: **Nhập môn AI**

Sinh Viên: **Trần Công Hiếu - 21110294**

Lớp: **21TTH_KDL**

TP.HCM, ngày 30 tháng 10 năm 2023



TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN TP.HCM
KHOA TOÁN – TIN HỌC

BÀI BÁO CÁO BÀI TẬP THỰC HÀNH TUẦN 1

HK1 - NĂM HỌC: 2023-2024

MÔN: NHẬP MÔN TRÍ TUỆ NHÂN TẠO

SINH VIÊN: TRẦN CÔNG HIẾU

MSSV: 21110294

LỚP: 21TTH_KDL

NHẬN XÉT

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

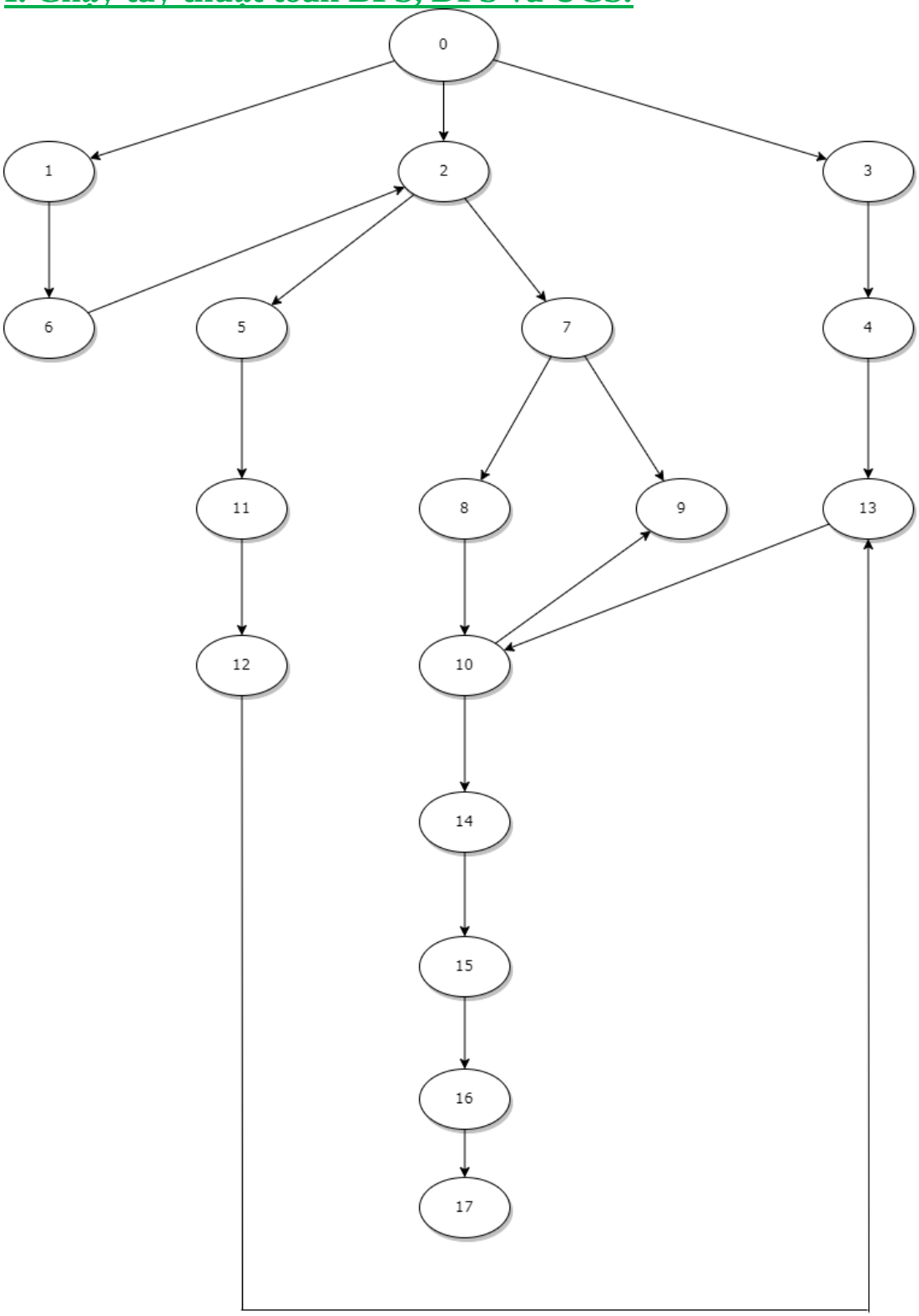
.....

Tp.HCM, Ngày……Tháng……Năm 2021

Giảng viên Bộ môn

Mục lục

I. Chạy tay thuật toán BFS, DFS và UCS.



Từ file Input.txt ta có được hình phác họa sơ đồ trên.

1. Thuật toán BFS.

1. L = [0]

2. Node = 0, L = [1, 2, 3], father[1, 2, 3] = 0

3. Node = 1, L = [2, 3, 6], father[6] = 1

4. Node = 2, L = [3, 6, 5, 7], father[5, 7] = 2

5. Node = 3, L = [6, 5, 7, 4], father [4] = 3

6. Node = 6, L = [5, 7, 4], father [] = 6

7. Node = 5, L = [7, 4, 11], father[11] = 5

8. Node = 7, L = [4, 11, 8, 9], father[8, 9] = 7

9. Node = 4, L = [11, 8, 9, 13], father[13] = 4

10. Node = 11, L = [8, 9, 13, 12], father[12] = 11

11. Node = 8, L = [9, 13, 12, 10], father[10] = 8

12. Node = 9, L = [13, 12, 10], father[] = 9

13. Node = 13, L = [12, 10], father[] = 13

14. Node = 12, L = [10], father[] = 12

15. Node = 10, L = [14], father[14] = 10

16. Node = 14, L = [15], father[15] = 14

17. Node = 15, L = [16], father[16] = 15

16. Node = 16, L = [17], father[17] = 16

17. Node = 17

⇒ Đường đi từ node start đến node end trong bài là:

$$0 \rightarrow 2 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 14 \rightarrow 15 \rightarrow 16 \rightarrow 17$$

2. Thuật toán DFS.

- Xét chiều xử lý node từ trái sang phải.

1. L = 0

2. Node = 0, L = [1, 2, 3], father[1, 2, 3] = 0

3. Node = 1, L = [6, 2, 3], father[6] = 1

4. Node = 6, L = [2, 3], father[] = 6

5. Node = 2, L = [5, 7, 3], father[5, 7] = 2

6. Node = 5, L = [11, 7, 3], father[11, 7] = 5

7. Node = 11, L = [12, 7, 3], father[12] = 11

8. Node = 12, L = [13, 7, 3], father[13] = 12

9. Node = 13, L = [10, 7, 3], father[10] = 13

10. Node = 10, L = [9, 14, 7, 3], father[9, 14] = 10

11. Node = 9, L = [14, 7, 3], father[] = 9

12. Node = 14, L = [15, 7, 3], father[15] = 14

13. Node = 15, L = [16, 7, 3], father[16] = 15

14. Node = 16, L = [17, 7, 3], father[17] = 16

15. Node = 17

⇒ Đường đi từ node start đến node end trong bài là:

0 → 2 → 5→ 11 → 12→ 13 → 10 → 14 → 15 → 16 → 17

- Xét chiều xử lý node từ phải sang trái.

1. L = 0

2. Node = 0, L = [3, 2, 1], father[1, 2, 3] = 0

3. Node = 3, L = [4, 2, 1], father[4] = 3

4. Node = 4, L = [13, 2, 1], father[13] = 4

5. Node = 13, L = [10, 2, 1], father[10] = 13

6. Node = 10, L = [14, 9, 2, 1], father[14, 9] = 10

7. Node = 14, L = [15, 9, 2, 1], father[15] = 14

8. Node = 15, L = [16, 9, 2, 1], father[16] = 15

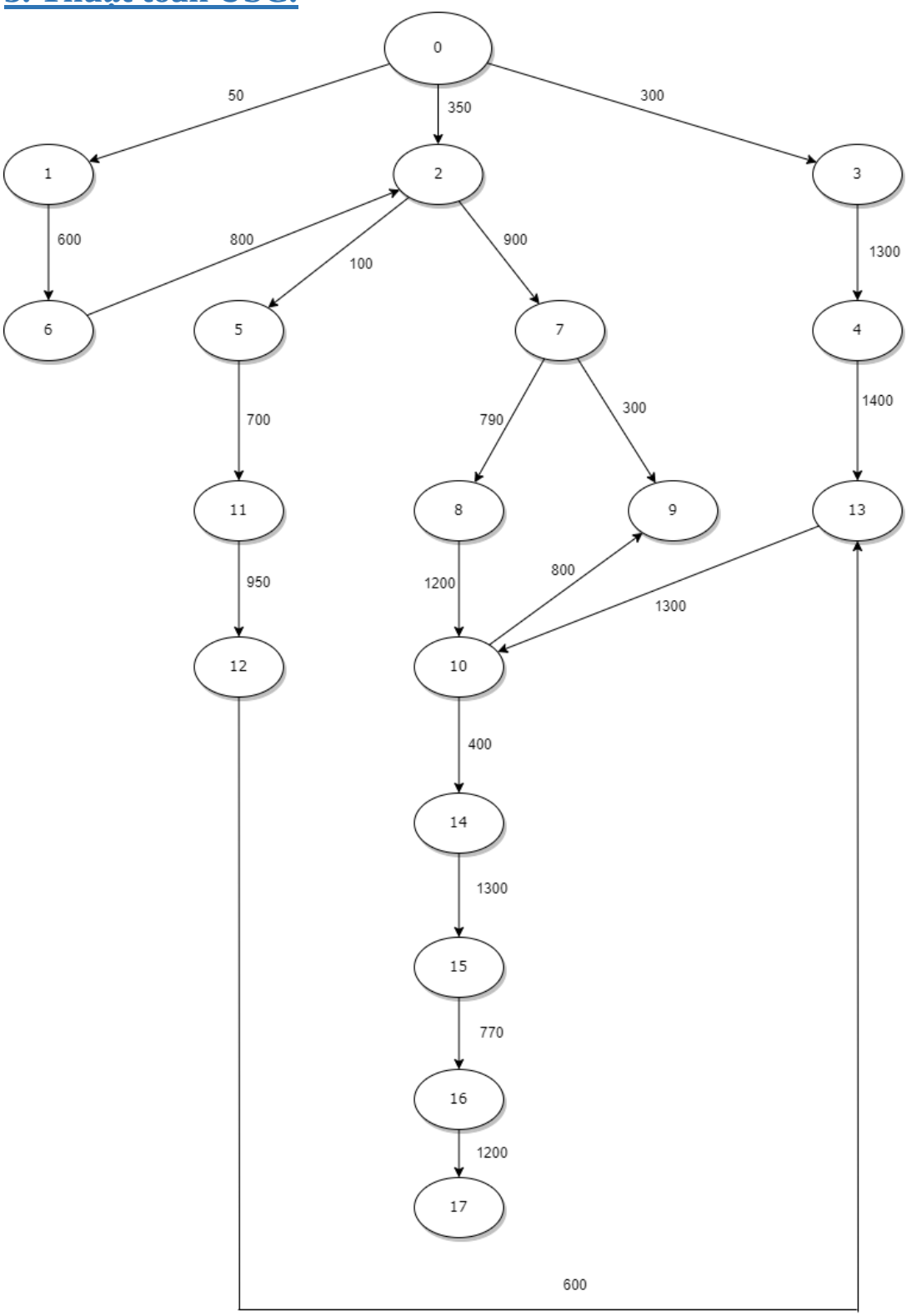
7. Node = 16, L = [17, 9, 2, 1], father[17] = 16

9. Node = 17

⇒ Đường đi từ node start đến node end trong bài là:

0 → 3 → 4→ 13 → 10 → 14 → 15 → 16 → 17

3. Thuật toán USC.



Từ file Input.txt ta có được hình phác họa sơ đồ trên.

Quy ước (a, b) là (tên_node, chi_phí)

1. PQ = (0, 0)

2. PQ = (1, 50), (3, 300), (2, 350)

3. PQ = (3, 300), (2, 350), (6, 650)

4. PQ = (2, 350), (4, 1600)

5. PQ = (5, 450), (7, 1250), (4, 1600)

6. PQ = (11, 1150), (7, 1250), (4, 1600)

7. PQ = (7, 1250), (4, 1600), (12, 2100)

8. PQ = (9, 1550), (4, 1600), (8, 2040), (12, 2100)

9. PQ = (4, 1600), (8, 2040), (12, 2100)

10. PQ = (8, 2040), (12, 2100), (13, 3000)

11. PQ = (12, 2100), (13, 3000), (10, 3240)

12. PQ = (13, 2700), (10, 3240)

13. PQ = (10, 3240)

14. PQ = (14, 3640)

15. PQ = (15, 4940)

16. PQ = (16, 5710)

17. PQ = (17, 6910)

⇒ Đường đi từ node start 0 đến node đích end 17 là:

0 → 2 → 7 → 8 → 10 → 14 → 15 → 16 → 17 với chi phí là 6910.

II. Kiểm tra tính đúng đắn BFS, DFS và UCS đã cho.

1. Kiểm tra BFS.

- Thuật toán BFS đã cho trong file không sai. Cân nhắc bỏ dòng "visited.append(start)" bởi khi vào vòng lặp while, sau khi qua dòng if đầu tiên thì list visited đã append vào rồi, nhằm tránh sự trùng lặp.

2. Kiểm tra DFS.

- Tương tự thuật toán BFS, thuật toán DFS đã cho trong file không sai. Cân nhắc bỏ dòng "visited.append(start)" bởi khi vào vòng lặp while, sau khi qua dòng if đầu tiên thì list visited đã append vào rồi, nhằm tránh sự trùng lặp.

3. Kiểm tra UCS.

- Thuật toán UCS đã cho trong file sai. Cũng cân nhắc bỏ dòng "visited.append(start)" như cài đặt của hai thuật toán trên. Hơn nữa sai về mặt logic trong vòng lặp while nên phải chỉnh sửa lại.

III. Giải thích thuật toán BFS, DFS và UCS.

1. Giải thích cài đặt đọc đầu vào, đầu ra.

```
def read_txt(file):
  size = int(file.readline())
  start, goal = [int (num) for num in file.readline().split(' ')]
  matrix = [[int (num) for num in line.split(' ')] for line in file]
  return size, start, goal, matrix
```

"size = int(file.readline())": dòng này đọc dòng đầu tiên của tệp văn bản (file) và chuyển nó thành một số nguyên. Dòng đầu tiên chứa số node có trong đồ thị ta đang xét.

"start, goal = [int(num) for num in file.readline().split(' ')]": dòng này đọc dòng thứ hai của tệp văn bản, chia dòng thành các từ riêng lẻ bằng dấu cách thông qua split(' ') và sau đó chuyển các từ thành số nguyên. Dòng này chứa hai số đại diện cho node ban đầu và node đích mà ta muốn xét.

"matrix = [[int(num) for num in line.split(' ')] for line in file]": tương tự như ở câu lệnh trên, dòng này đọc các dòng tiếp theo trong tệp văn bản và tạo một ma trận bằng cách chuyển đổi từng dòng thành một danh sách các số nguyên. Mỗi dòng trong tệp thường tương ứng với một hàng trong ma trận, và các số trong mỗi dòng được chia tách bằng dấu cách. Kết quả là một ma trận 2D, và ma trận này được gán cho biến matrix.

Cuối cùng, hàm trả về bốn giá trị: size, start, goal, và matrix. Khi ta gọi hàm read_txt và truyền cho nó một tệp văn bản, sẽ nhận về được bốn giá trị này để sử dụng trong chương trình của mình.

```
def convert_graph(a):
  adjList = defaultdict(list)
  for i in range (len(a)):
    for j in range(len(a[i])):
      if a[i][j] == 1:
        adjList[i].append(j)
  return adjList
```

Tạo hàm conver_graph() để chuyển ma trận kề thành danh sách kề với tham số đầu vào là một ma trận kề a.

“adjList = defaultdict(list)”: adjList là một biến được sử dụng để lưu trữ danh sách kề (adjacency list), defaultdict(list) là một cấu trúc dữ liệu trong Python từ thư viện collections, và nó tạo một từ điển mà giá trị mặc định của mỗi khóa là một danh sách rỗng. Trong trường hợp này, từ điển này sẽ chứa danh sách các đỉnh kề cho mỗi đỉnh trong đồ thị.

Sau đó, hai vòng lặp lồng nhau được sử dụng để duyệt qua từng phần tử của ma trận a. Trong đó giá trị a[i][j] là 1 nếu có một cạnh nối từ đỉnh i đến đỉnh j, và là 0 nếu không có cạnh nối.

“if a[i][j] == 1:”: dòng này kiểm tra xem giá trị tại vị trí a[i][j] có phải là 1 hay không. Nếu đúng, điều này có nghĩa rằng có một cạnh nối từ đỉnh i đến đỉnh j.

“adjList[i].append(j)”: nếu a[i][j] là 1, thì đoạn code này thêm đỉnh j vào danh sách kề của đỉnh i. Nó sẽ xây dựng danh sách kề dựa trên ma trận kề và lưu trữ các đỉnh kề trong biến adjList.

Kết quả của hàm convert_graph sẽ là một danh sách kề (adjacency list) được lưu trữ trong biến adjList, thể hiện cách các đỉnh trong đồ thị nối với nhau thông qua các cạnh.

```
def convert_graph_weight(a):
  adjList = defaultdict(list)
  for i in range(len(a)):
    for j in range(len(a[i])):
      if a[i][j] != 0:
        adjList[i].append((j, a[i][j]))
  return adjList
```

Tạo hàm convert_graph_weight tương tự như hàm convert_graph trong việc chuyển đổi một ma trận kề a thành một danh sách kề, nhưng với sự khác biệt quan trọng là nó xử lý trọng số của các cạnh.

“adjList = defaultdict(list)”: đây là bước khởi tạo, giống như trong hàm convert_graph, adjList là một từ điển, và mỗi khóa của từ điển tương ứng với một đỉnh trong đồ thị. Giá trị của mỗi khóa là một danh sách, và danh sách này sẽ lưu trữ các đỉnh kề của đỉnh tương ứng.

Hai vòng lặp lồng nhau sẽ duyệt qua từng phần tử của ma trận a. Giống như hàm convert_graph, ma trận a là ma trận kề, trong đó giá trị a[i][j] thể hiện có một cạnh nối từ đỉnh i đến đỉnh j. Tuy nhiên, ở đây, không phải kiểm tra giá trị a[i][j] có bằng 1 mà thay vào đó là kiểm tra giá trị a[i][j] khác 0.

“if a[i][j] != 0:”: dòng này kiểm tra xem giá trị a[i][j] có khác 0 hay không. Nếu khác 0, điều này có nghĩa rằng có một cạnh từ đỉnh i đến đỉnh j, và giá trị của a[i][j] thể hiện trọng số của cạnh đó.

“adjList[i].append((j, a[i][j]))”: nếu a[i][j] khác 0, thì đoạn mã này thêm một cạnh có đỉnh đích j và trọng số a[i][j] vào danh sách kề của đỉnh i. Cụ thể, một cặp (j, a[i][j]) được thêm vào danh sách kề của đỉnh i, thể hiện cạnh từ i đến j có trọng số là a[i][j].

Kết quả của hàm convert_graph_weight sẽ là một danh sách kề (adjacency list) lưu trữ trong biến adjList, thể hiện cách các đỉnh trong đồ thị nối với nhau qua các cạnh với trọng số tương ứng.

2. Giải thích cài đặt thuật toán BFS.

1. Khởi tạo hàm.

```
def BFS(graph, start, end):
  visited = []
  frontier = Queue()
```

- Định nghĩa hàm BFS với tham số đầu vào là graph, start (điểm đầu, xuất phát), end (điểm kết thúc, muốn đến).

- Tạo list để cho việc lưu lại các node đã đi qua và queue để lưu lại các cạnh của node đang xét.

2. Khởi gán cho node start.

```
#thêm node start vào frontier
frontier.put(start)

#start không có node cha
parent = dict()
parent[start] = None

path_found = False
```

- Thêm node start vào hàng đợi thông qua phương thức put() của Queue.

- Tạo thư viện parent và tạo None cho start trong “từ điển” parent.

- Khởi gán False cho path_found.

3. Tạo vòng lặp while.

```
while True:
  if frontier.empty():
    raise Exception("No way Exception")
  current_node = frontier.get()
  visited.append(current_node)
```

“ while True: ”: Đây là một vòng lặp vô hạn, nghĩa là nó sẽ lặp mã bên trong vô tận cho đến khi có điều kiện nào đó để thoát khỏi vòng lặp.

Kiểm tra hàng đợi frontier có rỗng hay không thông qua empty(). Nếu frontier rỗng, điều này có nghĩa là không còn node nào để xem xét và không có đường đi nào từ nút bắt đầu đến nút kết thúc. Trong trường hợp này, một ngoại lệ (Exception) sẽ được ném ra với thông báo "No way Exception" để báo lỗi và thoát khỏi vòng lặp vô hạn.

Nếu không rỗng sẽ đi xuống câu lệnh dưới “current_node = frontier.get()”, gọi node đầu tiên trong hàng đợi và gán cho biến current_node. Và đánh dấu đã “thăm” node đó bằng câu lệnh “visited.append(current_node)”.

```
# Kiểm tra current_node có là end hay không
if current_node == end:
  path_found = True
  break

for node in graph[current_node]:
  if node not in visited:
    frontier.put(node)
    parent[node] = current_node
    visited.append(node)
```

Kiểm tra current_node có là node end hay không, nếu là node end thì gán path_found giá trị True rồi thoát khỏi vòng lặp while bằng break. Nếu không là node end thì xuống vòng lặp for, duyệt từng node trong graph của node current_node, nếu node nào chưa được "thăm" thì ta put node đó vào hàng đợi frontier, lưu "node cha" và đánh dấu đã thuộc list visited của node đang được duyệt.

```
# Xây dựng đường đi
path = []
if path_found:
  path.append(end)
  while parent[end] is not None:
    path.append(parent[end])
    end = parent[end]
  path.reverse()
return path
```

"path = []": tạo một danh sách rỗng có tên path để lưu trữ đường đi từ nút đích (end) đến nút xuất phát (start). Ở bước tạo path, ta sẽ đi từ node end về lại node start bằng việc truy ngược "node cha" đã lưu trước đó thông qua parent. Câu lệnh if kiểm tra giá trị path_found là True hay False; nếu False, trả về path "rỗng" như vừa mới khởi tạo ở trên; nếu True thì đưa phần từ cuối vào path "path.append(end)". Vòng lặp while sẽ kiểm tra parent[end] có là not None hay không, tức parent[end] có khác giá trị

parent của start, chính là None hay không. Nếu không là None thì ta thêm node cha của node end đang xét đó vào list path rồi cập nhật lại end, lúc này thì end chính là node cha của node end ban đầu. Cứ lặp lại cho đến khi node cha của node end mang giá trị None, tức node end này cũng chính là node start, thì dừng lại. Đảo ngược path bằng hàm reverse(). Sau cùng là return về path.

3. Giải thích cài đặt thuật toán DFS.

1. Khởi tạo hàm.

```
def DFS(graph, start, end):
  visited = []
  frontier = []
```

Định nghĩa hàm DFS với tham số đầu vào là graph, start (điểm đầu, xuất phát), end (điểm kết thúc, muốn đến).

Tạo list để cho việc lưu lại các node đã đi qua và list frontier để lưu lại các cạnh của node đang xét. Lý thuyết thường thấy thì dùng stack cho frontier, ở đây list cũng có cấu trúc tương tự.

2. Khởi gán cho node start.

```
#thêm node start vào frontier
frontier.put(start)

#start không có node cha
parent = dict()
parent[start] = None

path_found = False
```

- Thêm node start vào list frontier thông qua phương thức put() của Queue.

- Tạo thư viện parent và tạo None cho start trong “từ điển” parent.

- Khởi gán False cho path_found.

3. Tạo vòng lặp while.

```
while True:
  if frontier == []:
    raise Exception("No way Exception")
  current_node = frontier.pop()
  visited.append(current_node)
```

“ while True: ”: Đây là một vòng lặp vô hạn, nghĩa là nó sẽ lặp mã bên trong vô tận cho đến khi có điều kiện nào đó để thoát khỏi vòng lặp.

Kiểm tra list frontier có rỗng hay không. Nếu frontier rỗng, điều này có nghĩa là không còn node nào để xem xét và không có đường đi nào từ nút bắt đầu đến nút kết thúc. Trong trường hợp này, một ngoại lệ (Exception) sẽ được ném ra với thông báo "No way Exception" để báo lỗi và thoát khỏi vòng lặp vô hạn.

Nếu không rỗng sẽ đi xuống câu lệnh dưới “current_node = frontier.pop()”, gọi node được đưa vào cuối cùng của list và gán cho biến current_node. Và đánh dấu đã “thăm” node đó bằng câu lệnh “visited.append(current_node)”.

```
# Kiểm tra current_node có là end hay không
if current_node == end:
  path_found = True
  break

for node in graph[current_node]:
  if node not in visited:
    frontier.append(node)
    parent[node] = current_node
    visited.append(node)
```

Kiểm tra current_node có là node end hay không, nếu là node end thì gán path_found giá trị True rồi thoát khỏi vòng lặp while bằng break. Nếu

không là node end thì xuống vòng lặp for, duyệt từng node trong graph của node current_node, nếu node nào chưa được "thăm" thì ta put node đó vào list frontier, lưu "node cha" và đánh dấu đã thuộc list visited của node đang được duyệt.

```
# Xây dựng đường đi
path = []
if path_found:
  path.append(end)
  while parent[end] is not None:
    path.append(parent[end])
    end = parent[end]
  path.reverse()
return path
```

"path = []": tạo một danh sách rỗng có tên path để lưu trữ đường đi từ nút đích (end) đến nút xuất phát (start). Ở bước tạo path, ta sẽ đi từ node end về lại node start bằng việc truy ngược "node cha" đã lưu trước đó thông qua parent. Câu lệnh if kiểm tra giá trị path_found là True hay False; nếu False, trả về path "rỗng" như vừa mới khởi tạo ở trên; nếu True thì đưa phần từ cuối vào path "path.append(end)". Vòng lặp while sẽ kiểm tra parent[end] có là not None hay không, tức parent[end] có khác giá trị parent của start, chính là None hay không. Nếu không là None thì ta thêm node cha của node end đang xét đó vào list path rồi cập nhật lại end, lúc này thì end chính là node cha của node end ban đầu. Cứ lặp lại cho đến khi node cha của node end mang giá trị None, tức node end này cũng chính là node start, thì dừng lại. Đảo ngược path bằng hàm reverse(). Sau cùng là return về path.

4. Giải thích cài đặt thuật toán UCS.

1. Khởi tạo hàm.

```
def UCS(graph, start, end):
  visited = []
  frontier = PriorityQueue()
```

Định nghĩa hàm UCS với tham số đầu vào là graph, start (điểm đầu, xuất phát), end (điểm kết thúc, muốn đến).

Tạo list visited để cho việc lưu lại các node đã đi qua và hàng đợi ưu tiên frontier để lưu lại các cạnh hay các node gắn với trọng số để xét độ ưu tiên nào thấp hơn và được kiểm tra trước.

2. Khởi gán cho node start.

```
frontier.put((0, start))

parent = dict()
parent[start] = None

path_found = False
```

- Thêm node start vào với trọng số (chi phí) là 0.

- Tạo thư viện parent và tạo None cho start trong “từ điển” parent.

- Khởi gán False cho path_found.

3. Tạo vòng lặp while.

```
while True:
  if frontier.empty():
    raise Exception("No way Exception")
  current_w, current_node = frontier.get()
  visited.append(current_node)
```

“ while True: ”: Đây là một vòng lặp vô hạn, nghĩa là nó sẽ lặp mã bên trong vô tận cho đến khi có điều kiện nào đó để thoát khỏi vòng lặp.

Kiểm tra list frontier có rỗng hay không. Nếu frontier rỗng, điều này có nghĩa là không còn node nào để xem xét và không có đường đi nào từ nút bắt đầu đến nút kết thúc. Trong trường hợp này, một ngoại lệ (Exception) sẽ được ném ra với thông báo "No way Exception" để báo lỗi và thoát khỏi vòng lặp vô hạn.

Nếu không rỗng sẽ đi xuống câu lệnh dưới “current_w, current_node = frontier.get()”, gọi node dựa trên sự ưu tiên trọng số và lần lượt gán cho biến current_w, current_node. Và đánh dấu đã “thăm” node đó bằng câu lệnh “visited.append(current_node)”.

```
if current_node == end:
  path_found = True
  break
```

Kiểm tra current_node có là node end hay không, nếu là node end thì gán path_found giá trị True rồi thoát khỏi vòng lặp while bằng break.

```
for neighbor, cost in graph[current_node]:
  if neighbor not in visited:
    if neighbor not in [node for _, node in frontier.queue]:
      frontier.put(((current_w + cost), neighbor))
      parent[neighbor] = current_node
    else:
      for _, node in frontier.queue:
        if node == neighbor and current_w + cost < _:
          frontier.queue.remove((_, node))
          frontier.put(((current_w + cost), neighbor))
          parent[neighbor] = current_node
```

Nếu không là node end thì xuống vòng lặp for, duyệt từng node hàng xóm và cost trong graph của node current_node. Hiểu cost ở đây là chi phí đi từ node current đến node neighbor, tức là chi phí của cạnh giữa 2 node đang xét.

“if neighbor not in [node for _, node in frontier.queue]:” dòng này kiểm tra xem node hàng xóm có trong frontier không. Nếu không có, nó thêm hàng xóm vào frontier với trọng số cập nhật (là tổng trọng số hiện tại và chi phí đến hàng xóm đó) và đánh dấu current_node làm nút cha của hàng xóm.

Nếu hàng xóm đã có trong frontier, dòng này thực hiện việc so sánh giữa chi phí hiện tại để đến hàng xóm với giá trị chi phí đã lưu trữ. Nếu chi phí hiện tại nhỏ hơn, nghĩa là có một đường đi ngắn hơn đến hàng xóm, thì cập nhật thông tin về hàng xóm trong frontier với trọng số mới và đánh dấu current_node làm nút cha của hàng xóm. Trong vòng lặp for có dùng “_” được sử dụng khi ta không muốn gán một tên biến cụ thể cho giá trị mà ta không có ý định sử dụng sau này, hoặc khi ta chỉ muốn lặp qua các phần tử mà không cần quan tâm đến giá trị của chúng.

Tóm lại, đoạn mã đang thực hiện việc thăm các node hàng xóm của current_node trong đồ thị, và đối chiếu chi phí của cạnh nối tới mỗi node hàng xóm đó. Nếu node hàng xóm chưa được thăm và nó không nằm trong hàng đợi frontier, thì chi phí của cạnh này sẽ được cộng vào current_w và node hàng xóm sẽ được thêm vào hàng đợi frontier với chi phí tính toán này. Nếu node hàng xóm đã có mặt trong hàng đợi frontier, thì chương trình sẽ kiểm tra xem chi phí mới tính toán có thấp hơn chi phí hiện tại của node trong hàng đợi hay không. Nếu có, thì nó sẽ cập nhật chi phí của node và cập nhật parent của node hàng xóm đó.

Sau khi vòng lặp kết thúc (hoặc nếu nó kết thúc vì tìm thấy đường đi), thuật toán sẽ xây dựng đường đi từ parent để trả về đường đi ngắn nhất và chi phí của nó.

```
path = []
if path_found:
  while parent[end] is not None:
    path.append(parent(end))
    end = parent[end]
  path.reverse()
return current_w, path
```

“path = []”: tạo một danh sách rỗng có tên path để lưu trữ đường đi từ nút đích (end) đến nút xuất phát (start). Ở bước tạo path, ta sẽ đi từ node end về lại node start bằng việc truy ngược “node cha” đã lưu trước đó thông qua parent. Câu lệnh if kiểm tra giá trị path_found là True hay False; nếu False, trả về path “rỗng” như vừa mới khởi tạo ở trên; nếu True thì đưa phần từ cuối vào path “path.append(end)”. Vòng lặp while sẽ kiểm tra parent[end] có là not None hay không, tức parent[end] có khác giá trị parent của start, chính là None hay không. Nếu không là None thì ta thêm node cha của node end đang xét đó vào list path rồi cập nhật lại end, lúc này thì end chính là node cha của node end ban đầu. Cứ lặp lại cho đến khi node cha của node end mang giá trị None, tức node end này cũng chính là node start, thì dừng lại. Đảo ngược path bằng hàm reverse(). Sau cùng là return về chi phí current_w và path.

IV. Kết quả chạy đoạn mã.

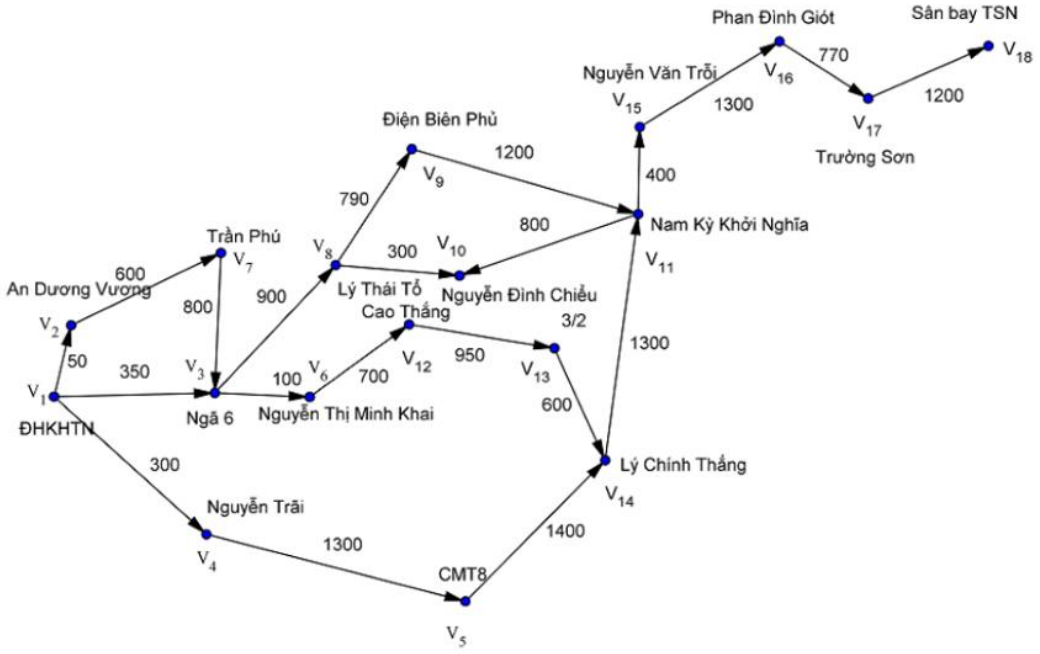
```
C:\Users\PC-LENOVO\AppData\Local\Programs\Python\Python310\python.exe
Kết quả sử dụng thuật toán BFS:
 [0, 2, 7, 8, 10, 14, 15, 16, 17]
Kết quả sử dụng thuật toán DFS:
 [0, 3, 4, 13, 10, 14, 15, 16, 17]
Kết quả sử dụng thuật toán UCS:
 [0, 2, 7, 8, 10, 14, 15, 16] với tổng chi phí là 6910

Process finished with exit code 0
```

V. Nhận xét.

Các kết quả từ chạy tay và code nhìn chung đều giống nhau. Chỉ lưu ý ở thuật toán DFS, vì việc lựa chọn ưu tiên đưa node nào ra trước thì sẽ ảnh hưởng tới việc đường đi từ node start đến node end của bài toán như trong bài tập trên, việc đi theo node ưu tiên từ trái qua phải và ngược lại thì cho ra kết quả khác nhau.

VI. Bài tập.

Cho đồ thị như hình vẽ bên dưới Tìm đường đi ngắn nhất từ trường Đại học Khoa học



Tự nhiên (V_1) tới sân bay Tân Sơn Nhất (V_{18}) dùng các thuật toán sau:

1. Thuật toán BFS:

1. L = [V_1]

2. Node = V_1, L = [V_2, V_3, V_4], father[V_2, V_3, V_4] = V_1

3. Node = V_2, L = [V_3, V_4, V_7], father[V_7] = V_2

4. Node = V_3, L = [V_4, V_7, V_6, V_8], father[V_6, V_8] = V_3

5. Node = V_4, L = [V_7, V_6, V_8, V_5], father [V_5] = V_4

6. Node = V_7, L = [V_6, V_8, V_5], father [] = V_7

7. Node = V_6, L = [V_8, V_5, V_{12}], father[V_{12}] = V_6

8. Node = V_8, L = [V_5, V_{12}, V_9, V_{10}], father[V_9, V_{10}] = V_8

9. Node = V_5, L = [V_{12}, V_9, V_{10}, V_{14}], father[V_{14}] = V_5

10. Node = V_{12}, L = [V_9, V_{10}, V_{14}, V_{13}], father[V_{13}] = V_{12}

11. Node = V_9, L = [V_{10}, V_{14} ,V_{13}, V_{11}], father[V_{11}] = V_9

12. Node = V_{10}, L = [V_{14} ,V_{13}, V_{11}], father[] = V_{10}

13. Node = V_{14}, L = [V_{13}, V_{11}], father[] = V_{14}

14. Node = V_{13}, L = [V_{11}], father[] = V_{13}

15. Node = V_{11}, L = [V_{15}], father[V_{15}] = V_{11}

16. Node = V_{15}, L = [V_{16}], father[V_{16}] = V_{15}

15. Node = V_{16}, L = [V_{17}], father[V_{17}] = V_{16}

15. Node = V_{17}, L = [V_{18}], father[V_{18}] = V_{17}

16. Node = V_{18}

⇒ Đường đi từ trường Đại học Khoa học Tự nhiên (V_1) tới sân bay Tân Sơn Nhất (V_{18}): $V_1 \rightarrow V_3 \rightarrow V_8 \rightarrow V_9 \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$

2. Thuật toán DFS.

1. L = [V_1]

2. Node = V_1, L = [V_2, V_3, V_4], father[V_2, V_3, V_4] = V_1

3. Node = V_2, L = [V_7, V_3, V_4], father[V_7] = V_2

4. Node = V_7, L = [V_3, V_4], father[V_3] = V_7

5. Node = V_3, L = [V_8, V_6, V_4], father[V_8, V_6] = V_3

6. Node = V_8, L = [V_9, V_{10}, V_6, V_4], father[V_9, V_{10}] = V_8

7. Node = V_9, L = [V_{11}, V_{10}, V_6, V_4], father[V_{11}] = V_9

8. Node = V_{11}, L = [V_{15}, V_{10}, V_6, V_4], father[V_{15}] = V_{11}

9. Node = V_{15}, L = [V_{16}, V_{10}, V_6, V_4], father[V_{16}] = V_{15}

10. Node = V_{16}, L = [V_{17}, V_{10}, V_6, V_4], father[V_{17}] = V_{16}

11. Node = V_{17}, L = [V_{18}, V_{10}, V_6, V_4], father[V_{18}] = V_{17}

12. Node = V_{18}

⇒ Đường đi từ trường Đại học Khoa học Tự nhiên (V_1) tới sân bay Tân Sơn Nhất (V_{18}): $V_1 \rightarrow V_2 \rightarrow V_7 \rightarrow V_3 \rightarrow V_8 \rightarrow V_9 \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$

3. Thuật toán USC.

1. PQ = (V_1, 0)

2. PQ = (V_2, 50), (V_4, 300), (V_3, 350)

3. PQ = (V_4, 300), (V_3, 350), (V_7, 650)

4. PQ = (V_3, 350), (V_5, 1600)

5. PQ = (V_6, 450), (V_8, 1250), (V_5, 1600)

6. PQ = (V_{12}, 1150), (V_8, 1250), (V_5, 1600)

7. PQ = (V_8, 1250), (V_5, 1600), (V_{13}, 2100)

8. PQ = (V_{10}, 1550), (V_5, 1600), (V_9, 2040), (V_{13}, 2100)

9. PQ = (V_5, 1600), (V_9, 2040), (V_{13}, 2100)

10. PQ = (V_9, 2040), (V_{13}, 2100), (V_{14}, 3000)

11. PQ = (V_{13}, 2100), (V_{14}, 3000), (V_{11}, 3240)

12. PQ = (V_{14}, 2700), (V_{11}, 3240)

13. PQ = (V_{11}, 3240)

14. PQ = (V_{15}, 3640)

15. PQ = (V_{16}, 4940)

16. PQ = (V_{17}, 5710)

17. PQ = (V_{18}, 6910)

⇒ Đường đi từ trường Đại học Khoa học Tự nhiên (V_1) tới sân bay Tân Sơn Nhất (V_{18}) là: $V_1 \rightarrow V_3 \rightarrow V_8 \rightarrow V_9 \rightarrow V_{11} \rightarrow V_{15} \rightarrow V_{16} \rightarrow V_{17} \rightarrow V_{18}$. Với chi phí là 6910.

4. Nhận xét.

Nếu để ý thì hình ảnh phác họa đồ thị từ file InputUCS.txt chính là đồ thị của bài tập này. Ứng với node thứ $i, i \in [0, 17]$ trong ảnh phác họa chính là node V_{i+1}.

Do đó, khi không quan tâm trọng số, thì kết quả chạy code của BFS, DFS cho bài trên cũng là của bài tập này. Và khi quan tâm trọng số, thì kết quả chạy code của UCS cho bài trên chính là của bài tập này. Tức kết quả chạy tay và chạy code là giống nhau.