# BUỔI 3. TÌM ĐƯỜNG ĐI NGẮN NHẤT TRÊN ĐỒ THỊ

**Mục đích**

- *Thực hành cài đặt các thuật toán tìm đường đi nhất trên đồ thị*
- *Củng cố lý thuyết, rèn luyện kỹ năng lập trình*

**Nội dung**

- Biểu diễn đồ thị có trọng số
- Cài đặt thuật toán Moore - Dijkstra tìm đường đi ngắn nhất từ một đỉnh đến các đỉnh khác
- Cài đặt thuật toán Bellman – Ford
- Cài đặt thuật toán Floyd - Warshall

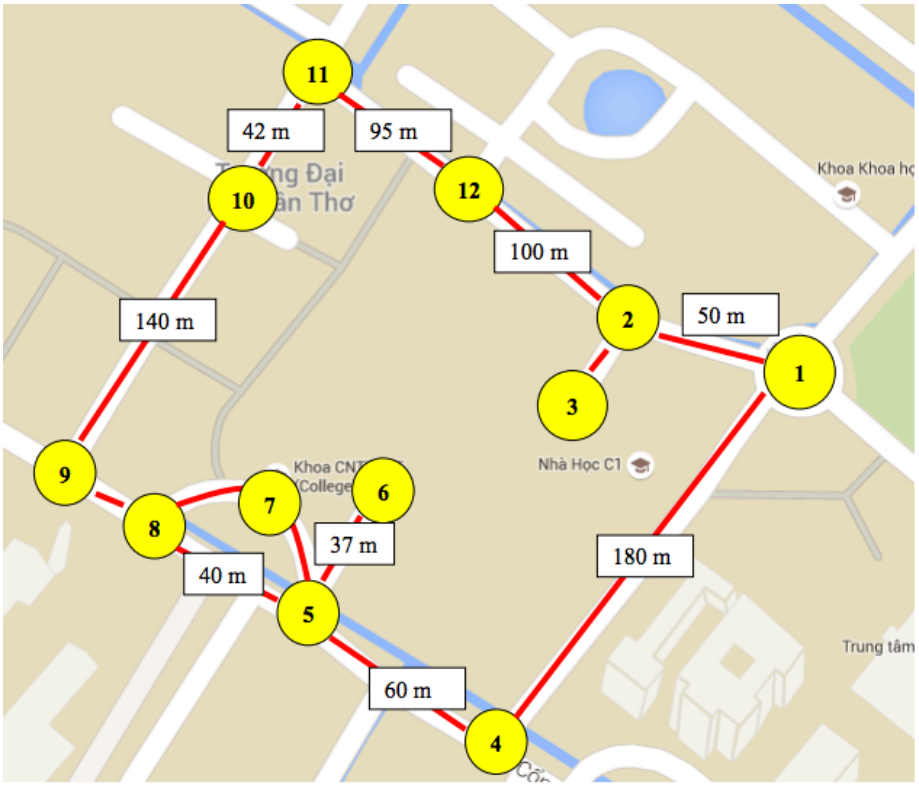
**Yêu cầu**

- Biểu diễn đồ thị
- Các phép toán cơ bản trên đồ thị

## 3.1 Đồ thị có trọng số

Xét bản đồ khu 2 ĐHCT như hình vẽ bên dưới. Giả sử ta muốn tìm đường đi ngắn nhất từ giao điểm 1 đến giao điểm 8. Ta mô hình hoá bản đồ về đồ thị với đỉnh là các giao điểm và cung là các con đường. Ta thu được đồ thị G = <V, E>.

Rõ ràng là để có thể biết được chiều dài của đường đi ta cần phải biết chiều của các con đường và phải biểu diễn chúng trong đồ thị. Vì thế, với mỗi cung của đồ thị (ứng với một con đường) ta gán cho nó 1 con số. Con số này được gọi là chiều dài của cung hay tổng quát hơn là *trọng số của cung* (edge weights). Một đồ thị có các cung được gán trọng số gọi là *đồ thị có trọng số* (weighted graph).



## 3.2 Biểu diễn đồ thị có trọng số

Xét một đồ thị G = <V, E>, với mỗi cung e = (u, v) của G được gán một trọng số w(e). Ta có thể mở rộng phương pháp **Ma trận kề (đỉnh – đỉnh)** thành ma trận trọn số để biểu diễn đồ thị có trọng số.

- Ma trận có số hàng và số cột bằng với số đỉnh
- Phần tử W[i][j] chứa trọng số (hay chiều dài) của cung (i, j)
- Nếu không có cung (i, j) phần tử W[i][j] chứa giá trị 0 hoặc một giá trị đặc biệt nào đó khác với trọng số của các cung, gọi giá trị này là **NO_EDGE**.

Chú ý:

- Phương pháp này chỉ sử dụng được với đơn đồ thị có hướng và vô hướng
- Không sử dụng được với đồ thị có đa cung (vì chỉ có 1 ô W[i][j] duy nhất nên chỉ chứa được 1 giá trị)

Khai báo cấu trúc dữ liệu đồ thị:

```
#define MAXN    100
#define NO_EDGE 0
//hoặc 1 giá trị đặc biệt nào đó

typedef struct {
      int n;
      int W[MAXN][MAXN];
} Graph;
```

Khởi tạo đồ thị ta cho tất cả phần tử của L đều bằng NO_EDGE: không có cung nào.

```
//Khởi tạo đồ thị n đỉnh, 0 cung
void init_graph(Graph *pG, int n) {
      pG->n = n;
      pG->m = 0;
      int u, v;
      for (u = 1; u <= n; u++)
            for (v = 1; v <= n; v++)
                  pG->W[u][v] = NO_EDGE;
}
```

Thêm cung (u, v) có trọng số (chiều dài) w vào đồ thị có thể thực hiện dễ dàng bằng phép gán:

```
      ...
      pG->W[u][v] = w;
      ...
```

Hãy sử dụng lệnh này trong lúc đọc đồ thị từ tập tin hoặc trong hàm `add_edge()`.

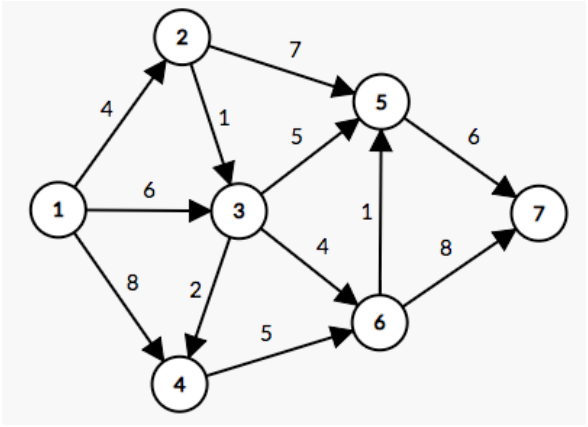
## 3.3 Lưu trữ đồ thị có trọng số trong tập tin

Tương tự như đồ thị không có trọng số, ta có thể lưu trữ đồ thị trong tập tin bằng cách thêm trọng số của cung kế bên cạnh cung. Định dạng của tập tin như sau:

```
n m
u1  v1  w1
u2  v2  w2
...
um  vm  w
```

Trong đó, n là số đỉnh, m là số cung. m dòng kế tiếp lưu thông tin của m cung, mỗi cung lưu đỉnh đầu, đỉnh cuối và trọng số của cung.

Ví dụ:




```
7 12
1 2 4
1 3 6
1 4 8
2 3 1
2 5 7
3 4 2
3 5 5
3 6 4
4 6 5
5 7 6
6 5 1
6 7 8
```

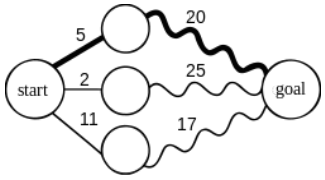
## 3.4 Bài toán tìm đường đi ngắn nhất

Cho đồ thị G = <V, E>, tìm đường đi ngắn nhất từ đỉnh u đến đỉnh v. Đường đi ngắn nhất từ u đến v chỉ tồn tại nếu như trên đường đi không chứa chu trình âm.

Bài toán tìm đường đi ngắn nhất là bài toán có *cấu trúc con tối ưu* (optimal substructure).

Nếu đường đi: u → $x_1$ → $x_2$ → … → $x_i$ → … → $x_k$ → v là đường đi ngắn nhất từ u đến v thì

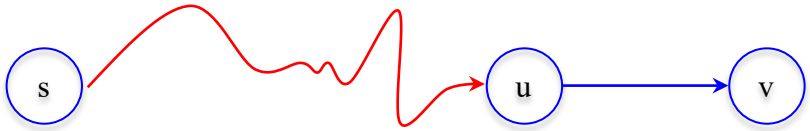
- Đoạn đường đi từ u đến $x_i$ cũng là đường đi ngắn nhất từ u đến $x_i$, và
- Đoạn đường đi từ $x_i$ đến v cũng là đường đi ngắn nhất từ $x_i$ đến v.




Tính chất này cho phép sử dụng kỹ thuật Quy hoạch động để giải bài toán. Thuật toán Moore-Dijkstra là một thuật toán Quy hoạch động.

**Biểu diễn (và lưu trữ) đường đi ngắn nhất:**

Xét một đường đi ngắn nhất từ s đến v như bên dưới.




Do đường đi ngắn nhất có tính chất cấu trúc con tối ưu nên đoạn đường đi từ s đến u (trên đường đi này) cũng là đường đi ngắn nhất từ s đến u. vì thế, đường đi ngắn nhất từ s đến và chính là đường đi có được bằng cách ghép *đường đi ngắn nhất từ s đến u* và *cung (u,v)*.

Nói cách khác để có được đường đi ngắn nhất từ s đến v, thì đối với v ta chỉ cần biết *u là đỉnh trước của nó*. Đến lượt u, nó chỉ cần biết được đỉnh trước của nó. Và cứ như thế, mỗi đỉnh chỉ cần biết *đỉnh trước* của nó trên đường đi ngắn nhất.

Nếu gọi p[u] là đỉnh trước của đỉnh u trong đường đi ngắn nhất, thì ta có được tất cả các đường đi ngắn nhất từ s đến các đỉnh còn lại trên đồ thị.

Để lưu trữ n đường đi ngắn nhất ta chỉ cần lưu các p[u] (một mảng p có n phần tử).

## 3.5 Thuật toán Moore – Dijkstra

Thuật toán Moore – Dijkstra cho phép tìm các đường đi ngắn nhất từ s đến các đỉnh còn lại trên một đồ thị vô hướng (hoặc có hướng) có trọng số.

**Ý tưởng:**

- Khởi tạo đường đi ngắn nhất trực tiếp từ s đến các đỉnh còn lại.
- Sau đó lần lượt cập nhật lại đường đi nếu đường đi mới tốt hơn đường đi cũ.

**Các biến hỗ trợ:**

- π[u]: chiều dài đường đi ngắn nhất từ s đến u (tính đến thời điểm đang xét).
- p[u]: đỉnh trước đỉnh u trên đường đi ngắn nhất từ s đến u.
- mark[u]: cho biết đỉnh u đã được đánh dấu chưa.
- Đồ thị được biểu diễn bằng phương pháp ma trận trọng số với W[u][v] chứa chiều dài (trọng số) của cung (u, v).

**Thuật toán**

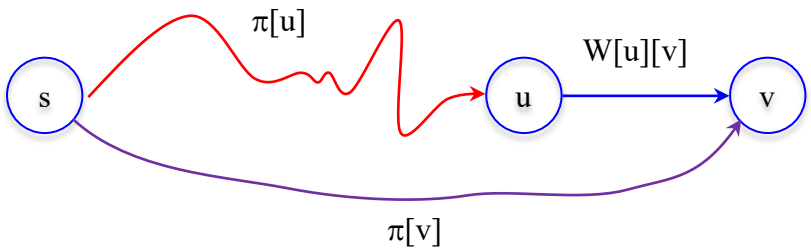
Khởi tạo:

- π[u] = ∞ với mọi u ≠ s
- π[s] = 0;
- mark[u] = 0 với mọi u

Lặp (n – 1 lần)

- Chọn đỉnh u chưa đánh dấu (mark[u] == 0) có chiều dài của đường đi từ s đến nó (π[u]) nhỏ nhất.
- Đánh đấu đã xét u bằng cách đặt mark[u] = 1
- Xem xét cập nhật π[v] và p[v] các đỉnh kề của u chưa được đánh dấu (mark[v] == 0)
    - Đỉnh v sẽ được cập nhật nếu đường đi mới (thông qua u) tốt hơn đường đi cũ:

```
if (π[u] + W[u][v] < π[v]) {
        π[v] = π [i] + W[u][v]      //cập nhật chiều dài đường đi ngắn nhất
        p[v] = u                    //cập nhật đường đi ngắn nhất
}
```



**Cài đặt thuật toán:**

```
//Định nghĩa giá trị vô cùng
#define oo 999999

int mark[MAXN];
int pi[MAXN];
int p[MAXN];

void MooreDijkstra(Graph *pG, int s) {
	int u, v, it;
	for (u = 1; u <= pG->n; u++) {
		pi[u] = oo;
		mark[u] = 0;
	}

	pi[s] = 0; //chiều dài đường đi ngắn nhất từ s đến chính nó bằng 0
	p[s] = -1; //trước đỉnh s không có đỉnh nào cả


	//Lặp n-1 lần
	for (it = 1; it < pG->n; it++) {

		//1. Tìm u có mark[u] == 0 và có pi[u] nhỏ nhất
		int j, min_pi = oo;
		for (j = 1; j <= pG->n; j++)
			if (mark[j] == 0 && pi[j] < min_pi) {
				min_pi = pi[j];
				u = j;
			}

		//2. Đánh dấu u đã xét
		mark[u] = 1;


		//3. Cập nhật pi và p của các đỉnh kề của v (nếu thoả)
		for (v = 1; v <= pG->n; v++)
			if (pG->W[u][v] != NO_EDGE && mark[v] == 0)
				if (pi[u] + pG->W[u][v] < pi[v]) {
					pi[v] = pi[u] + pG->W[u][v];
					p[v] = u;
				}
	}
}
```

Sau khi gọi hàm `MooreDijkstra()` kết quả sẽ được lưu trong 2 mảng `pi[]` và `p[]`. Ta có thể in kết quả này ra màn hình để kiểm tra:

```
for (u = 1; u <= G.n; u++)
	printf("pi[%d] = %d, p[%d] = %d\n", u, pi[u], u, p[u]);
```

Dựa vào kết quả này, ta có thể trả lời các câu hỏi thuộc 2 dạng sau:

Hỏi: "Chiều dài đường đi ngắn nhất từ s đến u là bao nhiêu?"

Trả lời: pi[u]

Hỏi: "Đường đi ngắn nhất từ s đến u là đường nào?"

Trả lời: ta có thể sử dụng đoạn chương trình bên dưới để in đường đi từ s đến u: ta lần ngược từ u về p[u] rồi p[p[u]] … cho đến -1.

```
//Tìm đường đi ngắn nhất
int path[MAXN]; //Lưu các đỉnh trên đường đi
int k = 0;      //số đỉnh của đường đi
int current = u;


//Lần ngược theo p để lấy đường đi
while (current != -1) {
      path[k] = current; k++;
      current = p[current];
}

//In ra màn hình theo chiều ngược lại
int u;
for (u = k-1; u >= 0; u--)
      printf("%d ", path[u]);
```

**Bài tập 1a.** Viết chương trình đọc một **đơn đồ thị có hướng, có trọng số không âm** từ bàn phím. Cài đặt thuật toán Moore – Dijkstra để tìm (các) đường đi ngắn nhất từ đỉnh 1 đến các đỉnh còn lại. In các thông tin pi[u] và p[u] của các đỉnh ra màn hình theo mẫu:

```
pi[1] = 0, p[1] = -1
pi[2] = 2, p[2] = 1
...
```

**Bài tập 1b.** Tương tự bài tập 1a nhưng với đồ thị **vô hướng**.

*Chú ý:* đối với đồ thị vô hướng khi thêm cung (u, v) vào đồ thị ta thêm luôn cung (v, u) vào đồ thị.

**Bài tập 2a.** Viết chương trình đọc một đồ thị **có hướng, có trọng số không âm** từ bàn phím, in ra màn hình chiều dài đường đi ngắn nhất từ đỉnh 1 đến đỉnh n. Nếu không có đường đi từ 1 đến n, in ra -1.

**Bài tập 2b.** Tương tự bài tập 2a nhưng với đồ thị **vô hướng**.

**Bài tập 3a.** Viết chương trình đọc một đồ thị **có hướng** từ bàn phím, tìm đường đi ngắn nhất từ đỉnh s đến đỉnh t (s và t được đọc từ bàn phím). In đường đi ra màn hình theo mẫu:

```
s -> u1 -> u2 -> … -> t
```

Nếu có nhiều đường đi ngắn nhất có chiều dài bằng nhau, in đường đi nào cũng được.

**Bài tập 3b.** Tương tự bài tập 3a nhưng với đồ thị vô hướng.

**Bài tập 4 – Mê cung số (number maze)**. Cho một mê cung số được biểu diễn bằng một mảng 2 chiều chứa các con số từ 0 đến 9 (xem hình bên dưới). Một con robot được đặt tại góc trên bên trái của mê cung và muốn đi đến góc dưới bên phải của mê cung. Con robot có thể đi lên, xuống, qua trái và qua phải 1 ô. Chi phí để đi đến một ô bằng với con số bên trong ô đó.

| | | | | |
|---|---|---|---|---|
| 0 | 3 | 1 | 2 | 9 |
| 7 | 3 | 4 | 9 | 9 |
| 1 | 7 | 5 | 5 | 3 |
| 2 | 3 | 4 | 2 | 5 |

Hãy tìm cách giúp con robot đi đến ô ở góc dưới phải sao cho tổng chi phí thấp nhất. Đường đi có chi phí thấp nhất cho ví dụ này bằng 24.

Dữ liệu đầu vào được cho trong một tập có định dạng như sau:

- Dòng đầu chứa 2 số nguyên M N (M: số hàng, N: sốc ột)
- m dòng tiếp theo mô tả các số trong mê cung

Ví dụ trên sẽ được lưu như sau:

```
4 5
0 3 1 2 9
7 3 4 9 9
1 7 5 5 3
2 3 4 2 5
```

Dữ liệu đầu ra: in chi phí thấp nhất để con robot đi từ góc trên bên trái về góc dưới bên phải. Ví dụ trên, cần in ra màn hình:

```
24
```

**Gợi ý giải:**

- Mô hình hoá bài toán về đồ thị có hướng
  - Đỉnh ⇔ ô
  - Cung ⇔ hai ô cạnh nhau
  - Trọng số cung (u, v) ⇔ giá trị của ô tương ứng với đỉnh v.

**Cách 1 - Đánh số ô**

Giả sử các ô trong mê cung được đánh chỉ số từ (0, 0): góc trên trái đến (M-1, N-1): góc dưới phải.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | 6 | 7 | 8 | 9 | 10 |
| 2 | 11 | 12 | 13 | 14 | 15 |
| 3 | 16 | 17 | 18 | 19 | 20 |

- Ô (i, j) sẽ tương ứng vớI đỉnh (i*N + j) + 1
- Đỉnh u sẽ tương ứng với ô ở hàng (u-1)/N và cột (u-1)%N

*Liệt kê các đỉnh kề của 1 đỉnh:*

1 ô có thể kề với 4 ô xung quanh nó nên 1 đỉnh sẽ có nhiều nhất 4 đỉnh kề tương ứng. Ta sẽ dùng công thức biến đổi ô -> đỉnh và đỉnh -> ô bên trên để tìm đỉnh kề của 1 đỉnh.

Giả sử ta muốn tìm đỉnh kề của đỉnh u, ta sẽ làm như sau:

- Đổi u thành hàng i = (u – 1)/N và cột j = (u – 1)%N
- Tìm 4 ô xung quanh ô (i, j) là (i – 1, j), (i + 1, j), (i, j – 1) và (i, j + 1)
- Đổi 4 ô thành 4 đỉnh (nếu ô vẫn nằm trong phạm vi (0, 0) và (M-1, N-1)).

Sử dụng khung chương trình bên dưới để cặp nhật pi và p của các đỉnh kề của đỉnh u trong giải thuật Moore – Dijkstra.

```
//4 ô xung quanh của 1 ô: trên, dưới, trái, phải
int di[] = {-1, 1,  0, 0};
int dj[] = { 0, 0, -1, 1};


//Đổi đỉnh u thành ô (i, j)
int i = (u – 1)/N;
int j = (u – 1)%N;


//Duyệt qua 4 ô kề với ô (i, j)
for (k = 0; k < 4; k++) {
      ii = i + di[k];
      jj = j + dj[k];

      //Kiểm tra ô (ii, jj) có nằm trong mê cung không
      if (ii >= 0 && ii < M && jj >= 0 && jj < N) {
            v = ii*N + jj; //đổi ô (ii,jj) thành đỉnh v

            //v là đỉnh kề của đỉnh u
            ...
      }
}
```

**Cách 2 – sử dụng chỉ số hàng và côt của ô làm chỉ số đỉnh**

- Không cần đổi ô thành đỉnh
- Sử dụng mảng pi[i][j] (2 chiều) thay vì pi[u] (1 chiều)

**Bài tập 5 - Ô kiều**

*"Ngưu Lang là vị thần chăn trâu của Ngọc Hoàng Thượng đế, vì say mê một tiên nữ phụ trách việc dệt vải tên là Chức Nữ nên bỏ bê việc chăn trâu, để trâu đi nghênh ngang vào điện Ngọc Hư. Chức Nữ cũng vì mê tiếng tiêu của Ngưu Lang nên trễ nải việc dệt vải. Ngọc Hoàng giận dữ, bắt cả hai phải ở cách xa nhau, người đầu sông Ngân, kẻ cuối sông.*

*Sau đó, Ngọc Hoàng thương tình nên ra ơn cho hai người mỗi năm được gặp nhau vào ngày bảy tháng bảy âm lịch. Khi tiễn biệt nhau, Ngưu Lang và Chức Nữ khóc sướt mướt. Nước mắt của họ rơi xuống trần hóa thành cơn mưa và được người dưới trần gian đặt tên là mưa ngâu."* (Theo wikipedia.com)

Để gặp được nhau vào ngày 7/7, Ngưu Lang và Chức Nữ phải nhờ đến bầy quạ đen bắt cầu, gọi là Ô kiều, cho mình đi qua để gặp nhau.

Sông Ngân Hà có n ngôi sao, giả sử được đánh số từ 1 đến n. Ngưu Lang ở tại ngôi sao Ngưu (Altair), được đánh số 1, còn Chức Nữ ở tại ngôi sao Chức Nữ (Vega) được đánh số n. Để bắt được một cây cầu từ ngôi sao này sang ngôi sao kia cần một số lượng quạ nào đó. Một khi con quạ ở cây cầu nào thì phải ở đó cho đến khi Ngưu Lang và Chức Nữ gặp được nhau.

Quạ thì càng ngày càng hiếm, nên Ngưu Lang và Chức Nữ phải tính toán sao cho số lượng quạ ít nhất có thể.

Hãy giúp Ngưu Lang và Chức Nữ viết chương trình tính xem cần phải nhờ đến ít nhất bao nhiêu con quạ để bắt cầu cho họ gặp nhau.

**Đầu vào (Input)**
Dữ liệu đầu vào được nhập từ bàn phím với định dạng:

- Dòng đầu tiên chứa 2 số nguyên n và m, tương ứng là số ngôi sao và số cặp sao có thể bắt cầu.
- m dòng tiếp theo mỗi dòng chứa 3 số nguyên u v q nói rằng để bắt 1 cây cầu bắt qua hai ngôi sao u và v cần phải tốn q con quạ.

**Đầu ra (Output)**

- In ra màn hình số lượng quạ cần thiết.

**Gợi ý:**
Mô hình hoá

- Ngôi sao ⇔ đỉnh
- Cây cầu ⇔ cung
- Số quạ cần thiết cho mỗi cây cầu ⇔ trọng số/chiều dài cung

Bài toán

- Tìm đường đi ngắn nhất từ 1 đến n

Số quạ cần thiết = chiều dài đường đi ngắn nhất từ 1 đến n.

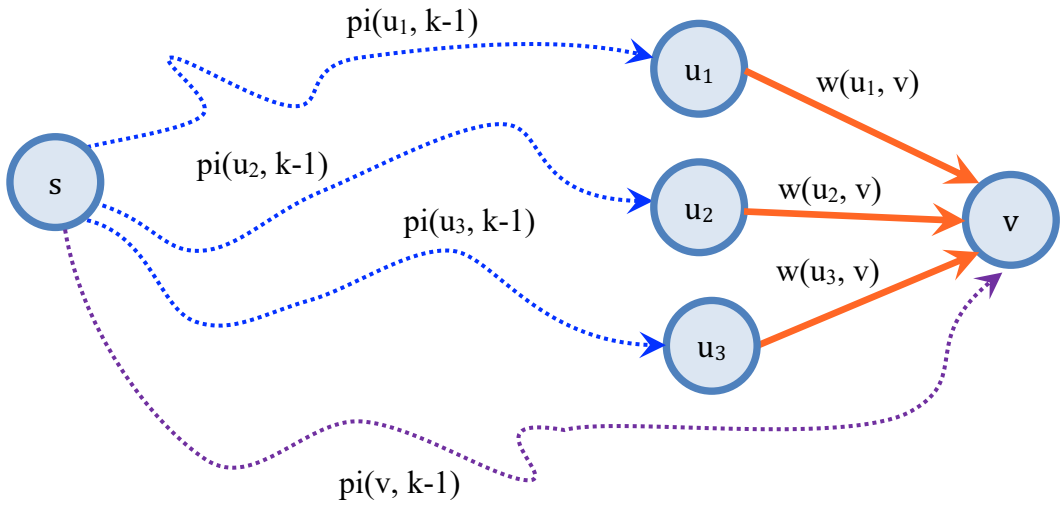
## 3.6 Thuật toán Bellman – Ford

Tương tự thuật toán Moore – Dijkstra, thuật toán Bellman – Ford cũng cho phép tìm đường đi ngắn nhất từ 1 đỉnh s đến các đỉnh khác nhưng làm việc được đối với đồ thị có trọng số âm.

**Ý tưởng:** áp dụng *kỹ thuật quy hoạch động* tìm đường đi ngắn nhất từ đỉnh `s` đến đỉnh `v` có số cung nhiều nhất là `k`.

Gọi `pi(v, k)` là chiều dài đường đi ngắn nhất từ `s` đến `v` có số cung không vượt quá `k`, ta có:

- `pi(v, 0)` = chiều dài đường đi trực tiếp (số cung = 0) từ `s` đến `v`.
- `pi(v, k)` = `min{pi(v, k-1), pi(u, k-1) + w(u, v)}` với mọi cung `(u, v)`.




Chiều dài đường đi ngắn nhất từ `s` đến `v` là: `pi(v, n-1)`. Đường đi ngắn nhất sẽ có tối đa n - 1 cung.

**Biến hỗ trợ**

- `pi[u]`: chiều dài đường đi ngắn nhất từ s đến u
- `p[u]`: đỉnh trước đỉnh u trên đường đi ngắn nhất

**Thuật toán**

- Khởi tạo: giống Moore – Dijkstra
  - `pi[u]=oo` với mọi `u != s`
  - `pi[s] = 0`
  - `p[s] = -1`
- Lặp n - 1 lần
  - Duyệt qua tất cả các cung (u, v) và cập nhật `pi[v]` và `p[v]` nếu thoả điều kiện

```
if (pi[u] + W[u][v] < pi[v]) {
      pi[v] = pi[u] + W[u][v];
      p[v] = u;
}
```

Về cơ bản, giải thuật Bellman – Ford có cùng nguyên lý như giải thuật Moore – Dijkstra. Điểm khác biệt là ở mỗi lần lặp: giải thuật Dijkstra chọn ra đỉnh có pi[u] bé nhất và cập nhật các đỉnh kề của u; trong khi đó giải thuật Bellman – Ford duyệt qua kết tất cả các cung (u, v) và cập nhật các đỉnh v (gần như cập nhật tất cả các đỉnh của đồ thị).

Vì thế để thuận tiện cho giải thuật Bellman – Ford ta phải biểu diễn đồ thị sao cho duyệt qua các cung của nó dễ dàng. Cách đơn giản nhất là lưu **danh sách các cung** của đồ thị.

**Biểu diễn đồ thị**

```
//Biểu diễn đồ thị có trọng số bằng phương pháp danh sách cung
#define M 1000

typedef struct {
	int u, v; // đỉnh đầu v, đỉnh cuối v
	int w;    // trọng số w
} Edge;

typedef struct {
	int n, m;          //n: đỉnh, m: cung
	Edge edges[MAX_M]; //các cung của đồ thị
} Graph;
```

Khởi tạo đồ thị

```
void init_graph(Graph* G, int n) {
	G->n = n;
	G->m = 0;
}
```

Thêm 1 cung vào đồ thị

```
void add_edge(Graph *pG, int u, int v, int w) {
	pG->edges[pG->m].u = u;
	pG->edges[pG->m].v = v;
	pG->edges[pG->m].w = w;
	pG->m++;
}
```

**Thuật toán Bellman – Ford:**

```
#define oo 999999

int pi[MAXN];
int p[MAXN];
void BellmanFord(Graph *pG, int s) {
	int u, v, w, it, k;
	for (u = 1; u <= pG->n; u++)
		pi[u] = oo;
	pi[s] = 0;
	p[s] = -1; //trước đỉnh s không có đỉnh nào cả

	// lặp n-1 lần
	for (it = 1; it < pG->n; it++) {

		// Duyệt qua các cung và cập nhật (nếu thoả)
		for (k = 0; k < pG->m; k++) {
			u = pG->edges[k].u;
			v = pG->edges[k].v;
			w = pG->edges[k].w;
			if (pi[u] == oo)  //Chưa có đường đi đến u, bỏ qua cung u,v
				continue;

			if (pi[u] + w < pi[v]) {
				pi[v] = pi[u] + w;
				p[v] = u;
			}
		}
	}

	//Làm thêm 1 lần nữa để kiểm tra chu trình âm (nếu cần thiết)
}
```

Kết quả thuật toán Bellman – Ford: cách sử dụng kết quả giống thuật toán Moore – Dijkstra.

***Chú ý:***

- Bản cài đặt này đã được đơn giản hoá (và tối ưu) so với phần ý tưởng, chỉ sử dụng mảng 1 chiều thay vì 2 chiều để lưu `pi[u]`. Đúng ra phải lưu là `pi[u]`. Tuy nhiên kết quả vẫn không đổi và tiết kiệm được bộ nhớ.
- Việc kiểm tra `pi[u] == oo` để rất quan trọng để tránh tình trạng **vô cùng + số âm**.

**Phát hiện chu trình âm:**

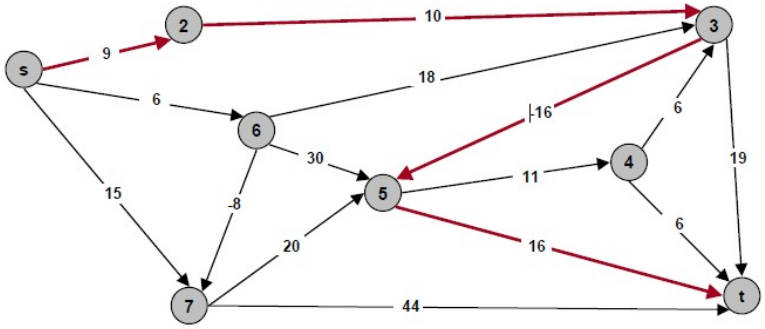
Thuật toán Bellman – Ford hơn thuật toán More – Dijkstra ở chỗ có thể chạy được với đồ thị có trọng số âm. Sau khi chạy xong thuật toán, ta có thể phát hiện được chu trình âm bằng cách duyệt qua các cung một lần nữa nếu tiếp tục cải thiện được `pi[v]` thì có nghĩa là đồ thị *có chu trình âm*.

Thêm đoạn code này vào cuối bản cài đặt ở trên:

```
	// Kiểm tra chu trình âm bằng cách Duyệt qua các cung một lần nữa
	int negative_cycle = 0;
	for (k = 0; k < G->m; k++) {
		int u = G->edges[k].u;
		int v = G->edges[k].v;
		int w = G->edges[k].w;
		if (pi[u] + w < pi[v]) { //Nếu có đường đi mới đến v tốt hơn
			negative_cycle = 1;//Có chu trình âm
			break;
		}
	}
```

**Bài tập 6a.** Viết chương trình đọc vào một đơn đồ thị có hướng có trọng số (có thể âm) từ bàn phím, tìm đường đi ngắn nhất từ đỉnh s đến đỉnh t (s và t cũng được đọc từ bàn phím). In ra màn hình chiều dài đường đi ngắn nhất từ s đến t.

Hãy kiểm thử với đồ thị bên dưới (s: đỉnh 1, t: đỉnh 8):




**Bài tập 6b.** Viết chương trình đọc vào một đơn đồ thị có hướng có trọng số (có thể âm), tìm đường đi ngắn nhất từ đỉnh s đến đỉnh t (s và t được đọc từ bàn phím). In ra màn hình đường đi ngắn nhất từ s đến t theo mẫu:

```
s -> u1 -> u2 -> … -> t
```

**Bài tập 7a.** Áp dụng thuật toán Bellman – Ford kiểm tra xem một đồ thị (có hướng, có trọng số) có chứa chu trình âm *trong quá trình tìm đường đi ngắn nhất từ s đến các đỉnh khác không*. In kết quả YES (nếu có chu trình âm) hoặc NO (trường hợp ngược lại).

**Bài tập 7b (nâng cao).** Tương tự bài tập 7, nhưng thay vì in ra YES/NO hãy in ra các đỉnh trong chu trình âm này.

**Bài tập 8 (ứng dụng).**

## 3.7 Thuật toán Floyd – Warshall

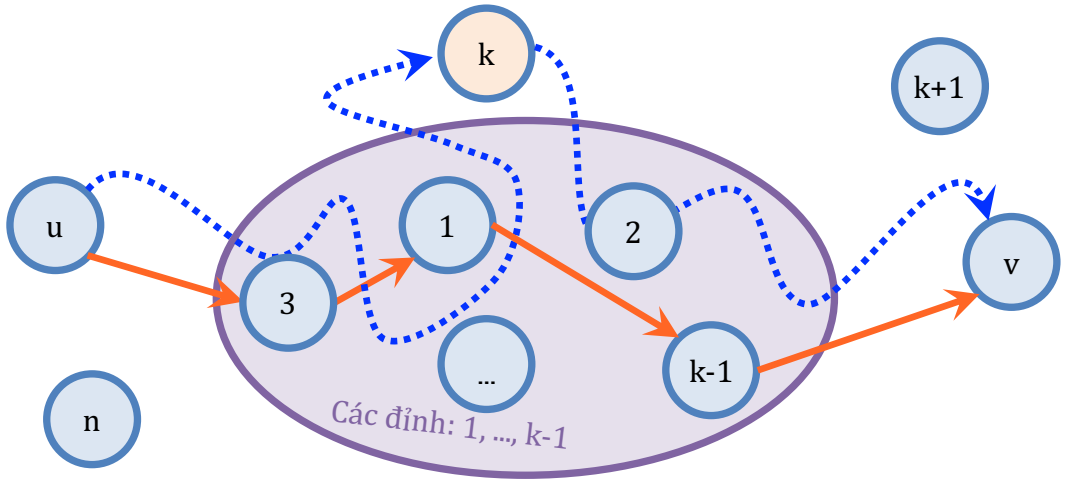
Tìm đường đi ngắn nhất giữa *tất cả các cặp đỉnh* (all pair shortest path).

**Ý tưởng:** áp dụng *kỹ thuật quy hoạch động* tìm đường đi ngắn nhất giữa 2 đỉnh `u`, `v` thông qua các *đỉnh trung gian* `{1, 2, …, k}`.

Gọi `pi(u, v, k)` là chiều dài đường đi ngắn nhất từ u đến v chỉ đi qua các đỉnh `{1, 2, …, k}`, ta có:

- `pi(u, v, 0)` = trọng số cung (u,v): đi trực tiếp từ u đến v
- `pi(u, v, k) = min{pi(u, v, k-1), pi(u, k, k-1) + pi(k, v, k-1)}`

Chiều dài đường đi ngắn nhất từ `u` đến `v` là: `pi(u, v, n)`.




**Biến hỗ trợ**

- `pi[u][v]`: chiều dài đường đi ngắn nhất từ u đến v
- `next[u][v]`: đỉnh kế tiếp đỉnh u trên đường đi ngắn nhất từ u đến v

**Thuật toán**

*Khởi tạo*

- `pi[u][v] = ∞` (vô cùng), với mọi u, v
- `pi[u][u] = 0`, với mọi u
- `pi[u][v] = W[u][v]`, với mọi cung (u, v) của đồ thị
- `next[u][v] = -1`, với mọi cặp u, v.
- `next[u][v] = v` với mọi cung (u, v) của đồ thị

*Lặp k = 1 đến n*

- Với mọi cặp đỉnh (u, v), cập nhật lại pi[u][v] nếu thoả điều kiện

```
if (pi[u][k] + pi[k][v] < pi[u][v]) {
      pi[u][v] = pi[u][k] + pi[k][v];
      next[u][v] = next[u][k];
}
```

*Dựng lại đường đi từ u đến v*

```
Đưa u vào path;
while (u != v) {
      u = next[u][v];
      Thêm u vào path. Ta cũng có thể in u ra màn hình thay vì lưu vào path
}
```

**Cài đặt:**

```
// Thuật toạn Floyd – Warshall tìm đường đi ngắn nhất giữa mọi cặp đỉnh
#define oo 999999

int pi[MAXN][MAXN];
int next[MAXN][MAXN];

void FloydWarshall(Graph *pG) {
	int u, v, k;
	for (u = 1; u <= pG->n; u++)
		for (v = 1; v <= pG->n; v++) {
			pi[u][v] = oo;
			next[u][v] = -1;
		}
	for (u = 1; u <= pG->n; u++)
		pi[u][u] = 0;

	for (u = 1; u <= pG->n; u++)
		for (v = 1; v <= pG->n; v++)
			if (pG->W[u][v] != NO_EDGE) {
				pi[u][v] = pG->W[u][v]; //đi trực tiếp từ u -> v
				next[u][v] = v;
			}

	for (k = 1; k <= pG->n; k++)
		for (u = 1; u <= pG->n; u++)
			for (v = 1; v <= pG->n; v++)
				if (pi[u][k] + pi[k][v] < pi[u][v]) {
					pi[u][k] = pi[u][k] + pi[k][v];
					next[u][v] = next[u][k];
				}

	//Kiểm tra chu trình âm (nếu cần thiết)
}
```

***Chú ý:***

- Bản cài đặt này đã được đơn giản hoá (và tối ưu) so với phần ý tưởng, chỉ sử dụng mảng 2 chiều thay vì 3 chiều để lưu `pi[u][v]`. Đúng ra phải lưu là `p[u][v][k]`. Tuy nhiên kết quả vẫn không đổi và tiết kiệm được bộ nhớ.
- Có 1 lỗi trong bản cài đặt trên (phần tô vàng) hãy tìm và sửa nó.
- Hãy điều chỉnh bản cài đặt để xử lý được trường hợp vô cùng + số âm, ví dụ: oo + (-3) = ?

**Bài tập 9a.** Viết chương trình đọc đồ thị có trọng số từ bàn phím. Áp dụng thuật toán Floyd – Warshall tìm đường đi ngắn nhất giữa các cặp đỉnh. In chiều dài ngắn nhất giữa các cặp đỉnh ra màn hình theo mẫu:

```
u -> v: chiều dài
...
```

**Bài tập 9b.** Viết chương trình đọc đồ thị có trọng số từ bàn phím. In đường đi ngắn nhất giữa các cặp đỉnh ra màn hình theo mẫu:

```
u -> v: u -> x1 -> x2 -> … -> v
```

**Phát hiện chu trình âm**

Sau khi chạy thuật toán Floyd – Warshall nếu trên đường chéo của ma trận chiều dài đường đi có phần tử `pi[u][u] < 0` (với u bất kỳ) thì đồ thị đã cho chứa ít nhất 1 chu trình âm.

Lý do là khi khởi tạo, ta đã gán `pi[u][u] = 0`. Giá trị `pi[u][u]` chỉ được cập nhật khi đường đi từ `u -> … -> u` có chiều dài âm. Tức là có chu trình âm!

Thêm đoạn chương trình này vào cuối thuật toán Floyd – Warshall để kiểm tra chu trình âm.

```
//Kiểm tra chu trình âm
int negative_cycle = 0;
for (u = 1; u <= pG->n; u++)
      if (pi[u][u] < 0) {
            //Đồ thị có chứa chu trình âm
            negative_cycle = 1;
            break;
}
```

**Bài tập 10a.** Viết chương trình đọc đồ thị có hướng, có trọng số từ bàn phím. Áp dụng thuật toán Floyd – Warshall kiểm tra đồ thị có chứa chu trình âm nào không. Nếu có, in ra "Negative cycle detected", nếu không in ra "No negative cycle".

**Bài tập 10b*.** Tương tự bài 10b nhưng bổ sung thêm yêu cầu: nếu đồ thị có chứa chu trình âm thì in ra các đỉnh trong chu trình âm bất kỳ.