viettel

Theo cách của bạn

# ĐỒ THỊ VÀ THUẬT TOÁN TÌM ĐƯỜNG NGẮN NHẤT

Người trình bày: Nguyễn Đức Kiên

# NỘI DUNG

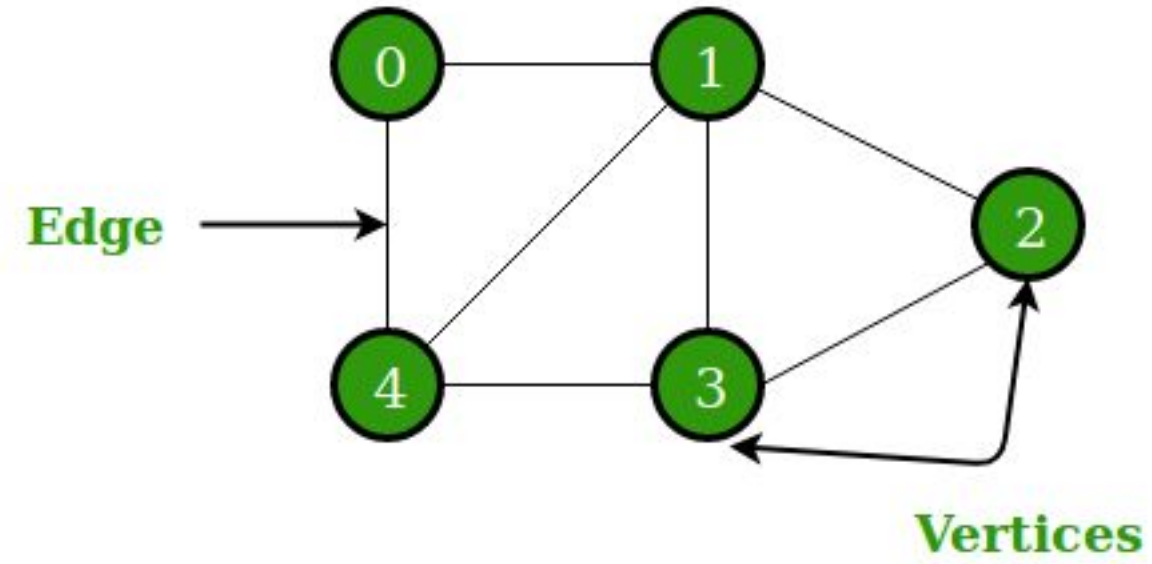
Đồ thị:
- Khái niệm
- Phân loại đồ thị
- Biểu diễn đồ thị bằng danh sách liên kết đôi

Bài toán tìm đường ngắn nhất trong đồ thị:
- Định nghĩa bài toán
- Thuật toán Dijkstra

Thực thi bài toán tìm đường ngắn nhất trên đồ thị với code C

# 01 Đồ thị

# 1. Khái niệm




Đồ thị G là cấu trúc rời rạc bao gồm 2 tập:

- Tập đỉnh V(G) là tập hữu hạn khác rỗng
- Tập cạnh E(G) là tập hữu hạn có thể là tập rỗng các cặp (u,v), u, v thuộc V

Ký hiệu G = (V, E).

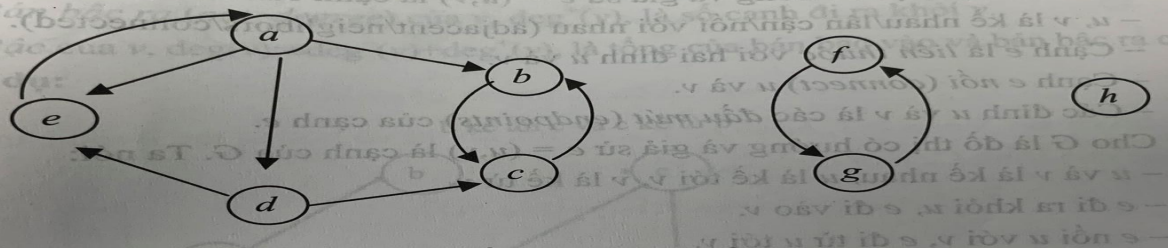
Phụ thuộc vào kiểu của cạnh nối và số lượng cạnh nối giữa 2 đỉnh mà ta phân biệt các loại đồ thị khác nhau.

# 2. Phân loại đồ thị

Đơn (đa) đồ thị vô hướng G = (V,E) là cặp gồm:

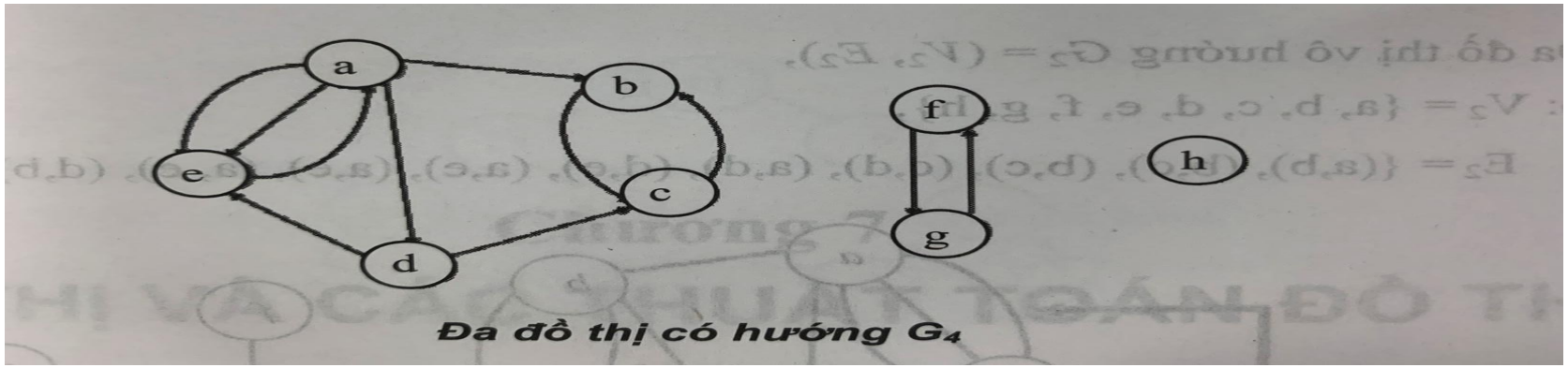
- Tập đỉnh V(G) là tập hữu hạn khác rỗng, các phần tử gọi là các đỉnh
- Tập cạnh E(G) là tập bộ không có thứ tự dạng (u,v) (u $\neq$ v). Các phần tử của E gọi là các cạnh

Đơn đồ thị có hướng $G_3 = (V_3, E_3)$,
trong đó: $V_3 = \{a, b, c, d, e, f, g, h\}$;
$E_3 = \{(a,b), (b,c), (c,b), (d,c), (a,d), (a,e), (d,e), (e,a), (f,g), (g,f)\}$




Đơn đồ thị có hướng $G_3$

Đa đồ thị có hướng $G_4 = (V_4, E_4)$,
trong đó: $V_4 = \{a, b, c, d, e, f, g, h\}$,
$E_4 = \{(a,b), (b,c), (c,b), (d,c), (a,d), (a,e), (a,e), (d,e), (e,a), (f,g), (g,f)\}$.

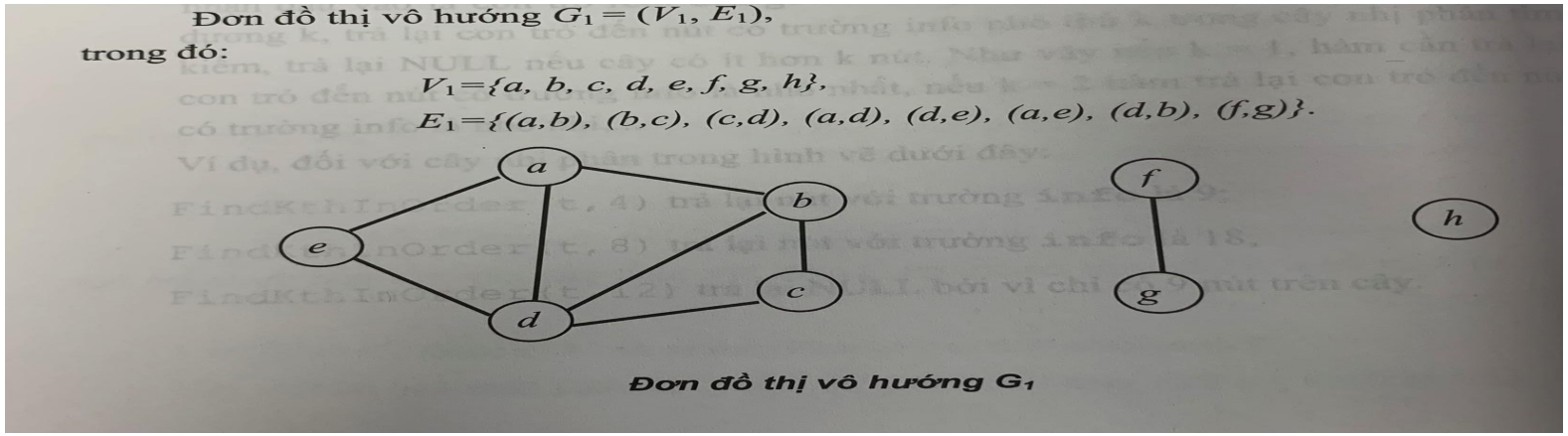


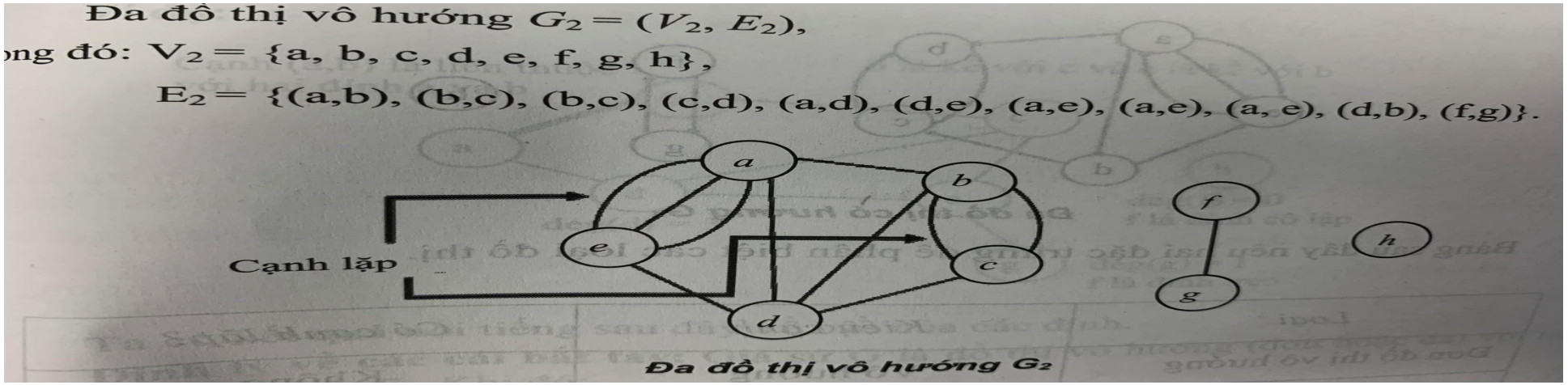

Đa đồ thị có hướng $G_4$

## 2. Phân loại đồ thị

Đơn (đa) đồ thị vô hướng G = (V,E) là cặp gồm:

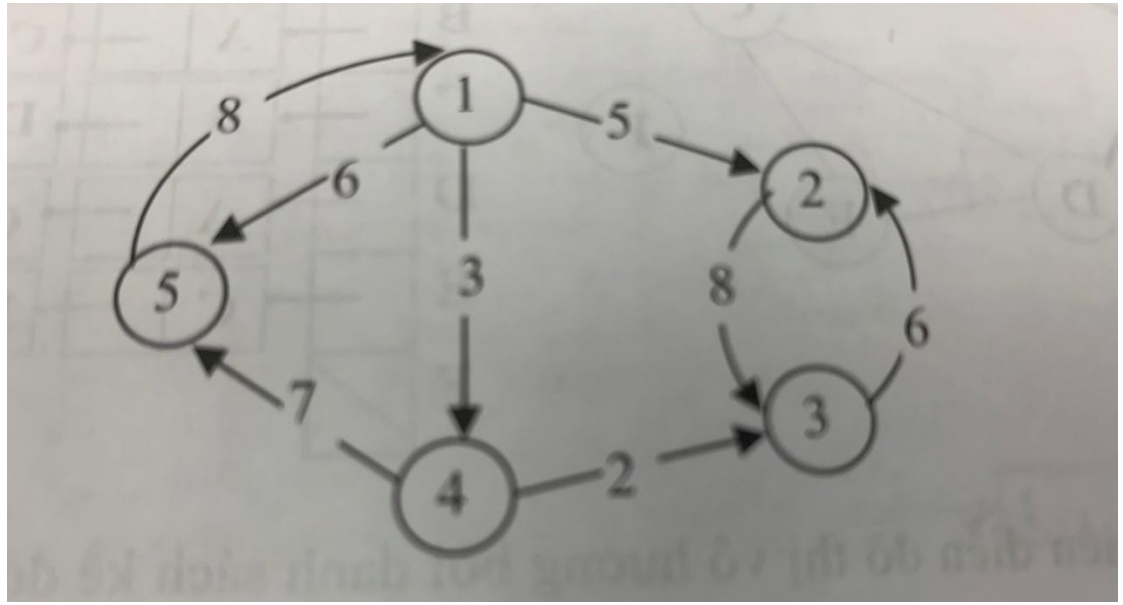
- Tập đỉnh V(G) là tập hữu hạn khác rỗng, các phần tử gọi là các đỉnh
- Tập cạnh E(G) là tập bộ không có thứ tự dạng (u,v) (u $\neq$ v). Các phần tử của E gọi là các cạnh



## 3. Biểu diễn đồ thị bằng danh sách liên kết đôi

Với mỗi cạnh e = (u,v) có trọng số không âm trên cạnh, ta cất giữ:
begin[e] = u; end[e] = v; length = trọng số cạnh




| e | begin[e] | end[e] | length |
|---|---|---|---|
| 1 | 1 | 5 | 6 |
| 2 | 5 | 1 | 8 |
| 3 | 4 | 5 | 7 |
| 4 | 1 | 4 | 3 |
| 5 | 1 | 2 | 5 |
| 6 | 4 | 3 | 2 |
| 7 | 2 | 3 | 8 |
| 8 | 3 | 2 | 6 |

# 02 Bài toán tìm đường ngắn nhất

## 1. Định nghĩa bài toán

Cho đồ thị có hướng G = (V,E) với trọng số không âm trên cạnh c(e). Giả sử s, t là 2 đỉnh và P(s,t) là đường đi từ s đến t trên đồ thị:

P(s,t): s = v0, v1,…, v(k-1), vk = t.

Ta gọi độ dài của đường đi P(s,t) là tổng trọng số trên các cung của nó, tức là nếu kí hiệu ρ(P(s,t)) thì ρ(P(s,t)) = Σc(vi,v(i+1)), i từ 0 đến k – 1.

Ta gọi đường đi ngắn nhất từ s đến t là đường đi có độ dài nhỏ nhất trong số tất cả các đường đi từ s đến t trên đồ thị.

Bài toán xét đến có 3 dạng:

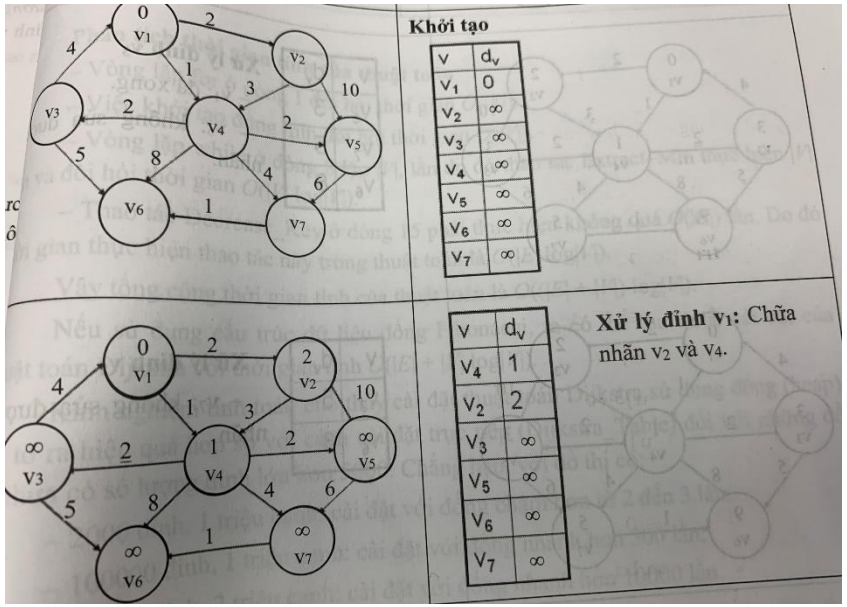
- Tìm đường ngắn nhất giữa hai đỉnh cho trước
- Tìm đường ngắn nhất từ một đỉnh nguồn s đến tất cả các đỉnh còn lại
- Tìm đường ngắn nhất giữa hai đỉnh bất kì

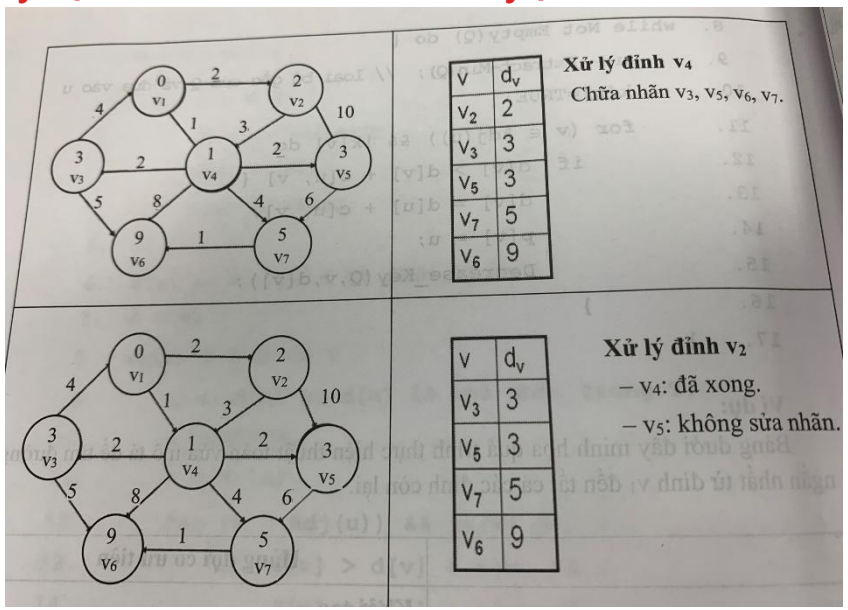
Bài toán xét đến với thuật toán Dijkstra là bài toán thứ 2!

## 2. Thuật toán Dijkstra

Ý tưởng của thuật toán:

- Bước 1: Từ đỉnh gốc, khởi tạo khoảng cách tới chính nó là 0, với các đỉnh khác là vô cùng lớn.
- Bước 2: Chọn đỉnh a có khoảng cách nhỏ nhất trong danh sách và ghi nhận. Lần duyệt tới sẽ không xét đến đỉnh này.
- Bước 3: Lần lượt xét các đỉnh kề b của đỉnh a vừa chọn. Nếu khoảng cách từ gốc tới đỉnh b mà lớn hơn tổng khoảng cách từ gốc tới đỉnh a và khoảng cách từ a đến b thì cập nhật khoảng cách từ gốc tới đỉnh b, đồng thời cập nhật thông tin đỉnh a là đỉnh mà đường đi ngắn nhất từ gốc đến b đi qua.
- Bước 4: Sau khi xét tất cả đỉnh kề b của a thì quay lại bước 2 đến khi duyệt hết các đỉnh.

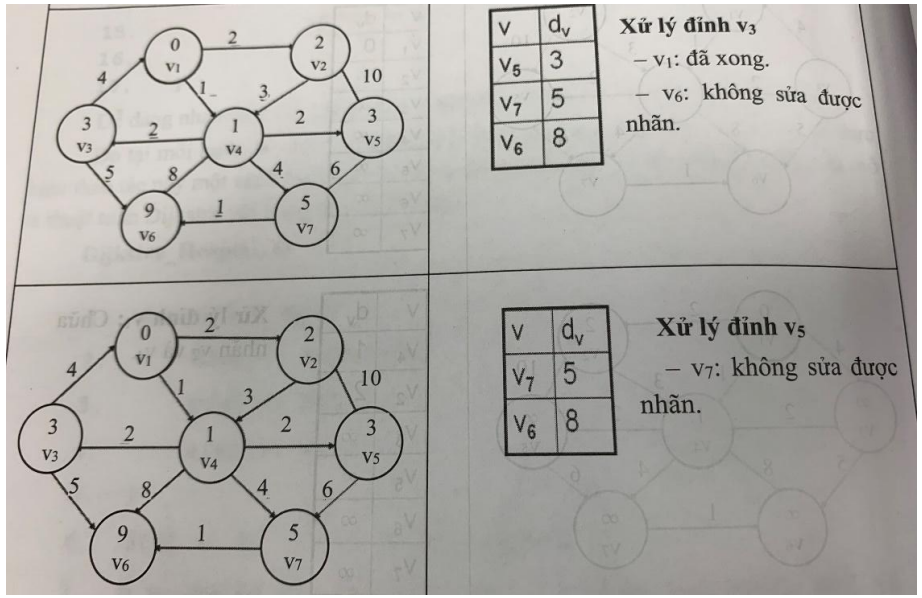


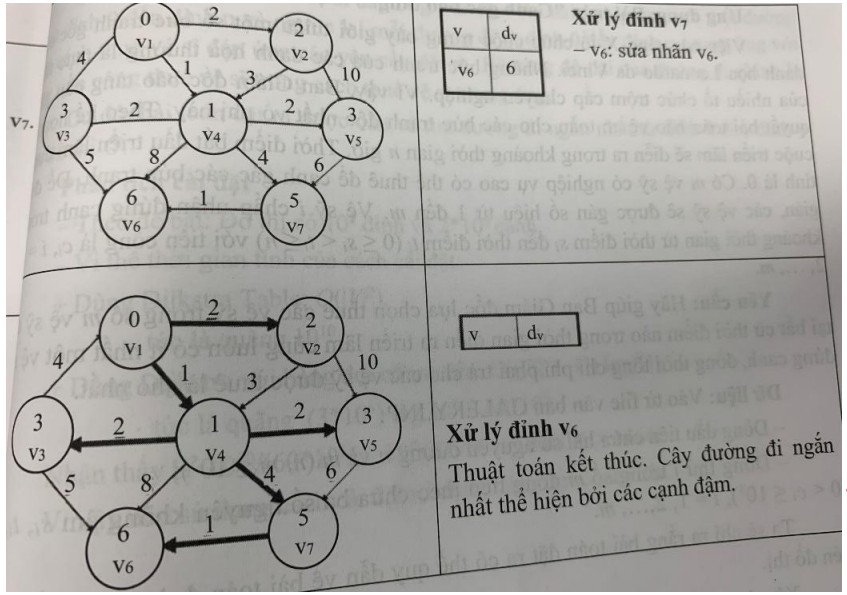



# 2. Thuật toán Dijkstra

Ý tưởng của thuật toán:

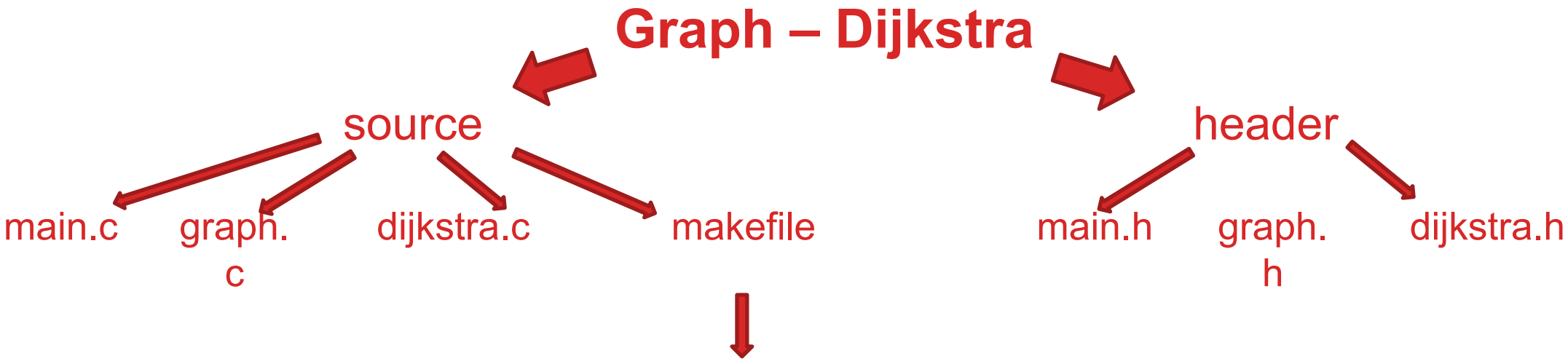
- Bước 1: Từ đỉnh gốc, khởi tạo khoảng cách tới chính nó là 0, với các đỉnh khác là vô cùng lớn.
- Bước 2: Chọn đỉnh a có khoảng cách nhỏ nhất trong danh sách và ghi nhận. Lần duyệt tới sẽ không xét đến đỉnh này.
- Bước 3: Lần lượt xét các đỉnh kề b của đỉnh a vừa chọn. Nếu khoảng cách từ gốc tới đỉnh b mà lớn hơn tổng khoảng cách từ gốc tới đỉnh a và khoảng cách từ a đến b thì cập nhật khoảng cách từ gốc tới đỉnh b, đồng thời cập nhật thông tin đỉnh a là đỉnh mà đường đi ngắn nhất từ gốc đến b đi qua.
- Bước 4: Sau khi xét tất cả đỉnh kề b của a thì quay lại bước 2 đến khi duyệt hết các đỉnh.



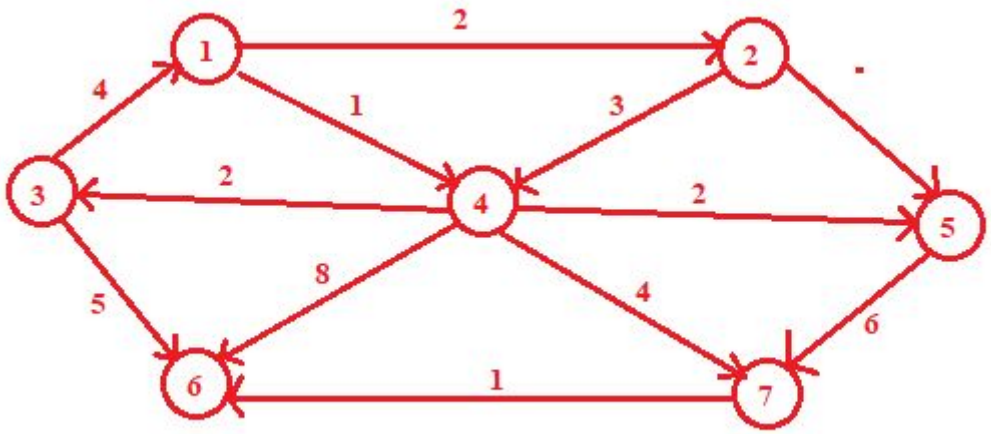
# 03 Thực thi bài toán tìm đường ngắn nhất với code C

# 1. Cấu trúc project C




```
1  shortest_dis: main.o graph.o dijkstra.o
2      gcc -o shortest_dis main.o graph.o dijkstra.o
3  main.o: main.c
4      gcc -c main.c -I ../header/
5  graph.o: graph.c
6      gcc -c graph.c -I ../header/
7  dijkstra.o: dijkstra.c
8      gcc -c dijkstra.c -I ../header/
9  clean:
10     rm shortest_dis *.o
```

# 2. Xây dựng đồ thị




```c
/* data structures for storing graph data */
typedef struct edge_list {
        int begin;
        int end;
        float length;
} edge_list;

typedef struct g_node {
        edge_list *data;
        struct g_node *pre;
        struct g_node *next;
} g_node;

typedef struct graph {
        g_node *first;
        g_node *last;
        int count;
} graph;
```

```c
/* create a graph holds primitive data */
graph *graph_data = create_graph();

insert_data_graph(graph_data, 1, 2, 2.0);
insert_data_graph(graph_data, 1, 4, 1.0);
insert_data_graph(graph_data, 2, 4, 3.0);
insert_data_graph(graph_data, 2, 5, 10.0);
insert_data_graph(graph_data, 3, 1, 4.0);
insert_data_graph(graph_data, 3, 6, 5.0);
insert_data_graph(graph_data, 4, 3, 2.0);
insert_data_graph(graph_data, 4, 5, 2.0);
insert_data_graph(graph_data, 4, 6, 8.0);
insert_data_graph(graph_data, 4, 7, 4.0);
insert_data_graph(graph_data, 5, 7, 6.0);
insert_data_graph(graph_data, 7, 6, 1.0);
```

```c
int insert_node_graph(graph *p_graph, g_node *p_node)
{
        if (NULL == p_graph || NULL == p_node)
        {
                return -1;
        }
        if (0 == p_graph->count)
        {
                p_graph->first = p_node;
                p_graph->last = p_node;
                p_graph->count++;
                return 0;
        }
        else
        {
                p_graph->last->next = p_node;
                p_node->pre = p_graph->last;
                p_graph->last = p_node;
                p_graph->count++;
                return 0;
        }
}
int insert_data_graph(graph *p_graph, int begin, int end, float length)
{
        edge_list *p_data = create_data(begin, end, length);
        g_node *p_node = create_g_node(p_data);
        return insert_node_graph(p_graph, p_node);
}
```

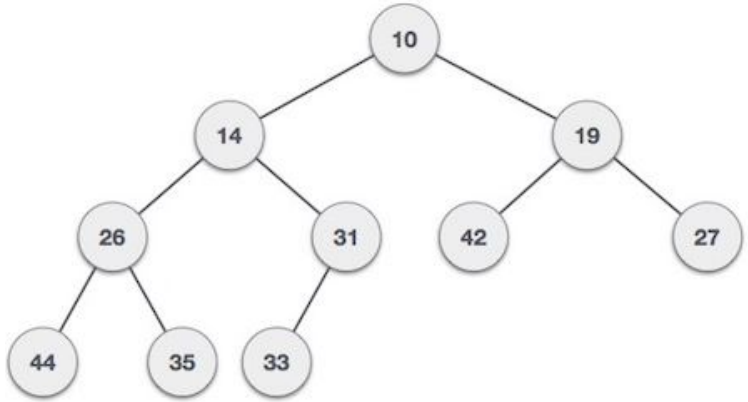
# 2. Xây dựng đồ thị

## Lấy dữ liệu là trọng số cạnh từ 2 đỉnh cho trước

```
/* function finds node having begin-end point, return length (begin, end) if node finded, else retun INFINITY */
float find_node(graph *p_graph, int begin, int end)
{
        if (NULL == p_graph || 0 > begin || 0 > end)
        {
                return -INFINITY;
        }
        g_node *tmp_node = p_graph->first;
        while (NULL != tmp_node)
        {
                if ((begin == tmp_node->data->begin) && (end == tmp_node->data->end))
                {
                        return tmp_node->data->length;
                }
                else
                {
                        tmp_node = tmp_node->next;
                }
        }
        return INFINITY;
}
```

# 2. Hàng đợi có ưu tiên

Hàng đợi có ưu tiên (priority queue) được xây dựng từ vun đống dữ liệu key từ 1 danh sách liên kết đôi




min(max) heap là một trường hợp của cây nhị phân cân bằng (chiều cao của cây con trái và phải của 1 nút bất kì không chênh lệch quá 1) nhưng không có tính chất của cây nhị phân tìm kiếm (miễn key của nút cha luôn nhỏ(lớn) hơn key của nút con.

Tính chất:

- Nút gốc có chỉ số là 1
- Nút con trái của nút chỉ số i có chỉ số 2*i
- Nút con phải của nút chỉ số i có chỉ số 2*i+1
- Nút cha của nút con chỉ số i có chỉ số i/2

```
/* data structures for dijkstra algorithm */
typedef struct dijk_vertex {
        int vtx; /* vertex v */
        bool k_sv; /* = true if we find shortest distance from s to v */
        float d_sv; /* moment shortest distance from s to v */
        int p_v; /* vertex is begin of edge having end point v */
} dijk_vertex;

typedef struct q_node {
        dijk_vertex *vertex;
        struct q_node *pre;
        struct q_node *next;
} q_node;

typedef struct pri_queue {
        q_node *first;
        q_node *last;
        int count;
} pri_queue;
```

linked-list (hoặc mảng): 10, 14, 19, 26, 31, 42, 27, 44, 35, 33

## 2. Hàng đợi có ưu tiên

Lấy các đỉnh của đồ thị và khởi tạo các giá trị ban đầu cho dữ liệu của hàng đợi có ưu tiên

```
int find_vertex(pri_queue *queue, int ver)
{
        if (NULL == queue)
        {
                return 0;
        }
        q_node *tmp_node = queue->first;
        while(NULL != tmp_node)
        {
                if (ver == tmp_node->vertex->vtx)
                {
                        return 1;
                }
                tmp_node = tmp_node->next;
        }
        return 0;
}
```

```
if (0 == queue->count)
{
        insert_data_queue(queue, tmp_g_node->data->begin, false, INFINITY, -1);
        tmp_g_node = tmp_g_node->next;
}
else
{
        j = find_vertex(queue, tmp_g_node->data->begin);
        if (0 == j)
        {
                insert_data_queue(queue, tmp_g_node->data->begin, false, INFINITY, -1);
                tmp_g_node = tmp_g_node->next;
        }
        else
        {
                tmp_g_node = tmp_g_node->next;
        }
}
```

# 2. Hàng đợi có ưu tiên

## Xây dựng hàng đợi có ưu tiên từ danh sách liên kết đôi đã khởi tạo

```
/* funtion heapifies graph from position i, n is total number of element in graph */
int min_heapify(pri_queue *queue, int i, int n)
{
	if (NULL == queue || i > n)
	{
		return 0;
	}
	int smallest = i;
	q_node *i_node = browse_queue_node(queue, i);
	q_node *smallest_node = browse_queue_node(queue, smallest);

	int left = 2*i;
	if (left > n)
	{
		return -1;
	}
	q_node *left_child = browse_queue_node(queue, left);
	if (smallest_node->vertex->d_sv > left_child->vertex->d_sv)
	{
		smallest = left;
		smallest_node = browse_queue_node(queue, smallest);
	}
```

```
	int right = 2*i+1;
	if (right > n)
	{
		swap(smallest_node, i_node);
		return 0;
	}
	q_node *right_child = browse_queue_node(queue, right);
	if (smallest_node->vertex->d_sv > right_child->vertex->d_sv)
	{
		smallest = right;
		smallest_node = browse_queue_node(queue, smallest);
	}

	if (smallest != i)
	{
		swap(smallest_node, i_node);
	}
	else
	{
		++smallest;
	}
	min_heapify(queue, smallest, n);
	return 0;
}
```

```
/* function swaps two data pointers of 2 nodes */
int swap(q_node *node1, q_node *node2)
{
	if (NULL == node1 || NULL == node2)
	{
		return -1;
	}
	dijk_vertex *tmp_ver;
	tmp_ver = node1->vertex;
	node1->vertex = node2->vertex;
	node2->vertex = tmp_ver;
	return 0;
}
```

```
/* function browses to a node at pos in pri_queue */
q_node *browse_queue_node(pri_queue *queue, int pos)
{
	if (NULL == queue || 1 > pos || queue->count < pos)
	{
		return NULL;
	}
	int i;
	q_node *tmp_node = queue->first;
	for (i = 1; i < pos; ++i)
	{
		tmp_node = tmp_node->next;
	}
	return tmp_node;
}
```

```
build_min_heap(pri_queue *queue)

    if (NULL == queue)
    {
        return -1;
    }
    int n = queue->count;
    int i;
```

```
for (i = n/2; i > 0; --i)
{
    min_heapify(queue, i, n);
}
```

# 3. Thực thi thuật toán Dijkstra

## Tách phần tử có key nhỏ nhất từ hàng đợi ưu tiên

```
/* function removes root of min heap, return this root */
q_node *extract_min(pri_queue *queue)
{
        if (NULL == queue || 0 >= queue->count)
        {
                return NULL;
        }
        if (1 == queue->count)
        {
                q_node *tmp_node = create_q_node(queue->last->vertex);
                free(queue->last);
                queue->first = NULL;
                queue->last = NULL;
                queue->count--;
                return tmp_node;
        }
        else
        {
                swap(queue->first, queue->last);
                q_node *last_node = queue->last;
                q_node *tmp_node = create_q_node(last_node->vertex);
                queue->last = last_node->pre;
                free(last_node);
                queue->count--;
                build_min_heap(queue);
                return tmp_node;
        }
}
```

## Thay đổi key của một node trong hàng đợi

```
/* funtion changees d_sv into decrease_d in a node having begin data begin_ver */
int change_key(pri_queue *queue, int begin_ver, float decrease_d)
{
        if (NULL == queue || 0 > begin_ver || 0 > decrease_d)
        {
                return -1;
        }
        int i;
        q_node *tmp_node = queue->first;
        for (i = 0; i < queue->count; ++i)
        {
                if (begin_ver == tmp_node->vertex->vtx)
                {
                        tmp_node->vertex->d_sv = decrease_d;
                        tmp_node = tmp_node->next;
                }
                else
                {
                        tmp_node = tmp_node->next;
                }
        }
        build_min_heap(queue);
        return 0;
}
```

# 3. Thực thi thuật toán Dijkstra

Ý tưởng của thuật toán (bước 1 khởi tạo):

- Bước 2: Chọn đỉnh a có khoảng cách nhỏ nhất trong danh sách và ghi nhận. Lần duyệt tới sẽ không xét đến đỉnh này.
- Bước 3: Lần lượt xét các đỉnh kề b của đỉnh a vừa chọn. Nếu khoảng cách từ gốc tới đỉnh b mà lớn hơn tổng khoảng cách từ gốc tới đỉnh a và khoảng cách từ a đến b thì cập nhật khoảng cách từ gốc tới đỉnh b, đồng thời cập nhật thông tin đỉnh a là đỉnh mà đường đi ngắn nhất từ gốc đến b đi qua.
- Bước 4: Sau khi xét tất cả đỉnh kề b của a thì quay lại bước 2 đến khi duyệt hết các đỉnh.

```
/* funtion processes Dijikstra Algorimth */
int dijkstra_heap(graph *p_graph, pri_queue *dijk_queue, pri_queue *store_queue, int origin_ver)
{
        if (NULL == p_graph || NULL == dijk_queue || NULL == store_queue || 0 >= origin_ver)
        {
                return -1;
        }
        q_node *tmp_node = dijk_queue->first;
        while (NULL != tmp_node)
        {
                if (origin_ver == tmp_node->vertex->vtx)
                {
                        tmp_node->vertex->d_sv = 0.0;
                        tmp_node->vertex->p_v = origin_ver;
                        break;
                }
                else
                {
                        tmp_node = tmp_node->next;
                }
        }

        build_min_heap(dijk_queue);
```

# 3. Thực thi thuật toán Dijkstra

Ý tưởng của thuật toán (bước 1 khởi tạo):

- Bước 2: Chọn đỉnh a có khoảng cách nhỏ nhất trong danh sách và ghi nhận. Lần duyệt tới sẽ không xét đến đỉnh này.
- Bước 3: Lần lượt xét các đỉnh kề b của đỉnh a vừa chọn. Nếu khoảng cách từ gốc tới đỉnh b mà lớn hơn tổng khoảng cách từ gốc tới đỉnh a và khoảng cách từ a đến b thì cập nhật khoảng cách từ gốc tới đỉnh b, đồng thời cập nhật thông tin đỉnh a là đỉnh mà đường đi ngắn nhất từ gốc đến b đi qua.
- Bước 4: Sau khi xét tất cả đỉnh kề b của a thì quay lại bước 2 đến khi duyệt hết các đỉnh.

```
while (0 != dijk_queue->count)
{
        q_node *extract_node = extract_min(dijk_queue);
        extract_node->vertex->k_sv = true;
        insert_node_queue(store_queue, extract_node);

        if (0 == dijk_queue)
        {
                break;
        }
```

```
int i;
q_node *tmp_node = dijk_queue->first;
for (i = 0; i < dijk_queue->count; ++i)
{
        if (false == tmp_node->vertex->k_sv)
        {
                if (tmp_node->vertex->d_sv > (extract_node->vertex->d_sv + find_node(p_graph, extract_node->vertex->vtx, tmp_node->vertex->vtx)))
                {
                        tmp_node->vertex->d_sv = extract_node->vertex->d_sv + find_node(p_graph, extract_node->vertex->vtx, tmp_node->vertex->vtx);
                        tmp_node->vertex->p_v = extract_node->vertex->vtx;
                        change_key(dijk_queue, tmp_node->vertex->vtx, tmp_node->vertex->d_sv);
                }
        }
        tmp_node = tmp_node->next;
}
```

# 3. Thực thi thuật toán Dijkstra

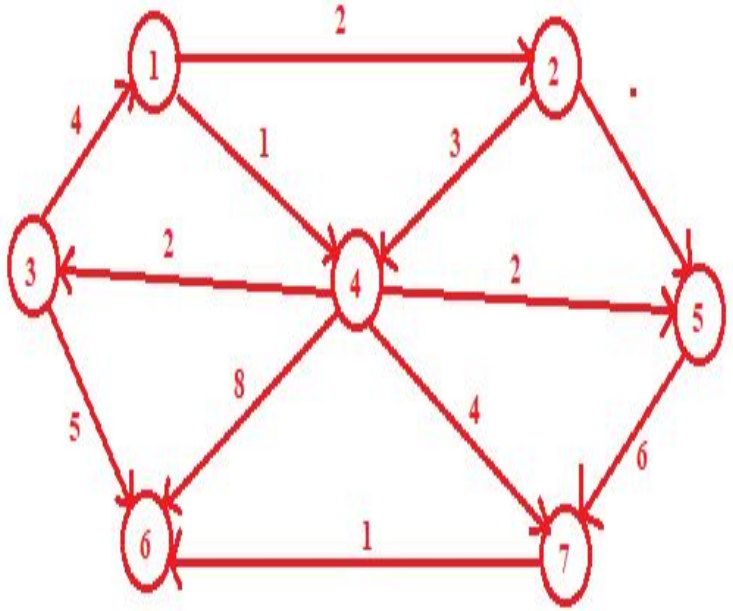
## Kết quả chạy project:

```
kiennd@cntd101:/data/workspace/kiennd/graph_dijkstra/source$ make
gcc -c main.c -I ../header/
gcc -c graph.c -I ../header/
gcc -c dijkstra.c -I ../header/
gcc -o shortest_dis main.o graph.o dijkstra.o
kiennd@cntd101:/data/workspace/kiennd/graph_dijkstra/source$ ./shortest_dis
Data of graph given in edge linked-list:
(1,2) = 2.00    (1,4) = 1.00    (2,4) = 3.00    (2,5) = 10.00   (3,1) = 4.00   (
3,6) = 5.00     (4,3) = 2.00    (4,5) = 2.00    (4,6) = 8.00    (4,7) = 4.00   (
5,7) = 6.00     (7,6) = 1.00



Shortest distance from 1 to others implemented by Dijkstra algorithm:
(1,1) = 0.00, pre-ver of 1 is 1;
(1,4) = 1.00, pre-ver of 4 is 1;
(1,2) = 2.00, pre-ver of 2 is 1;
(1,5) = 3.00, pre-ver of 5 is 4;
(1,3) = 3.00, pre-ver of 3 is 4;
(1,7) = 5.00, pre-ver of 7 is 4;
(1,6) = 6.00, pre-ver of 6 is 7;

kiennd@cntd101:/data/workspace/kiennd/graph_dijkstra/source$
```




viettel

Theo cách của bạn