

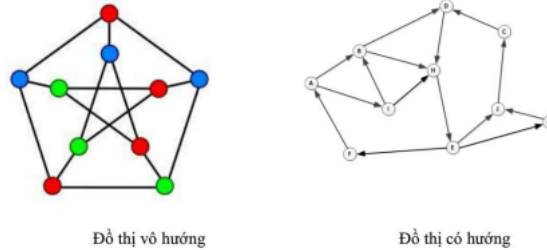
1 Đồ thị

1.1 Một số khái niệm cơ bản về đồ thị

1.1.1 Khái niệm đồ thị

Đồ thị là một cấu trúc rời rạc gồm các đỉnh và các cạnh nối với các đỉnh đó. Được mô tả hình thức: $G = (V, E)$. Trong đó, V là tập các **đỉnh**, E gọi là tập các **cạnh**. Có thể coi E là tập các cặp (u, v) với u và v là hai đỉnh của V .

Một số hình ảnh của đồ thị:



1.1.2 Các khái niệm cơ bản

Có thể phân loại đồ thị theo đặc tính và số lượng tập cạnh E . Cho đồ thị $G = (V, E)$, ta có một số khái niệm sau:

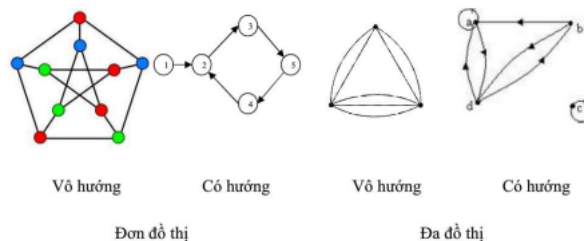
Đơn đồ thị G được gọi là đơn đồ thị nếu giữa hai đỉnh u, v của V có nhiều nhất là một cạnh nối trong E nối từ u tới v .

Đa đồ thị G được gọi là đa đồ thị nếu giữa hai đỉnh u, v của V có thể có nhiều hơn 1 cạnh trong E nối từ u tới v .

Đồ thị vô hướng G được gọi là đồ thị vô hướng nếu các cạnh trong E không định hướng, tức là cạnh nối hai đỉnh u, v bất kì cũng là cạnh nối hai đỉnh v, u . Hay nói cách khác, tập E gồm các cặp (u, v) không có tính thứ tự $(u, v) \equiv (v, u)$

Đồ thị có hướng G được gọi là đồ thị có hướng nếu các cạnh trong E là có định hướng, có thể có cạnh nối từ đỉnh u tới đỉnh v nhưng chưa chắc đã có cạnh nối từ đỉnh v tới đỉnh u . Hay nói cách khác, tập E gồm các cặp (u, v) có tính thứ tự: $(u, v) \neq (v, u)$

Ví dụ:



1.1.3 Cạnh liên thuộc, đỉnh kề, bậc

Đối với *đồ thị vô hướng* $G = (V, E)$, xét một cạnh $e \in E$, nếu $e = (u, v)$ thì ta nói hai đỉnh u và v là **kề nhau** (adjacent) và cạnh e này **liên thuộc** (incident) với đỉnh u và đỉnh v .

Với một đỉnh v trong đồ thị, ta định nghĩa **bậc** (degree) của v , kí hiệu $\deg(v)$ là số cạnh liên thuộc với v . Dễ thấy rằng trên đơn đồ thị thì số cạnh liên thuộc với v cũng là số đỉnh kề với v .

Đối với *đồ thị có hướng* $G = (V, E)$. Xét một cung $e \in E$, nếu $e = (u, v)$ thì ta nói **u nối tới v** và **v nối từ u** , cung e là **đi ra khỏi đỉnh u** và **đi vào đỉnh v** . Đỉnh u khi đó được gọi là đỉnh đầu, đỉnh v được gọi là đỉnh cuối của cung e .

Với mỗi đỉnh v trong đồ thị có hướng, ta định nghĩa: **Bán bậc ra** của v là số cung đi ra khỏi nó, **bán bậc vào** là số cung đi vào đỉnh đó.

Đường đi: Một đường đi độ dài k từ đỉnh u đến đỉnh v là dãy $(u = x_0, x_1, x_2, \dots, x_k = v)$ thỏa mãn $(x_i, x_{i+1}) \in E$ với $\forall i: (0 \leq i \leq k)$. Đỉnh u gọi là đỉnh xuất phát, v gọi là đỉnh kết thúc đường đi. Đường đi không có cạnh nào đi qua hơn 1 lần được gọi là **đường đi đơn**.

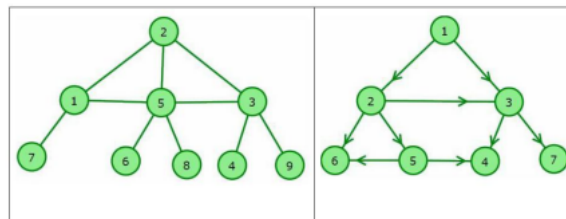
Chu trình: Đường đi có đỉnh xuất phát trùng với đỉnh kết thúc gọi là **chu trình**, tương tự ta có khái niệm **chu trình đơn**.

1.2 Bài toán tìm kiếm trên đồ thị

1.2.1 Phát biểu bài toán

Cho đồ thị $G = (V, E)$ và s và t là hai đỉnh của đồ thị. *Yêu cầu:* Hãy chỉ ra một đường đi từ s đến t (nếu có).

Ví dụ: Xét một đồ thị vô hướng và một đồ thị có hướng dưới đây:



Trên cả hai đồ thị, $(1, 2, 3, 4)$ là đường đi đơn độ dài 3 từ đỉnh 1 tới đỉnh 4. Bởi $(1, 2)$, $(2, 3)$, $(3, 4)$ đều là các cạnh (hay cung).

Làm sao để duyệt tất cả các đỉnh có thể đến được từ một đỉnh xuất phát nào đó? Vấn đề này đưa về một bài toán liệt kê mà yêu cầu của nó là không được bỏ sót hay lặp lại bất kì đỉnh nào. Vì vậy, cần phải xây dựng những thuật toán cho phép duyệt một cách hệ thống các đỉnh, những thuật toán như vậy gọi là những thuật toán tìm kiếm trên đồ thị. Trong lý thuyết đồ thị, người ta quan tâm đến hai thuật toán cơ bản nhất: thuật toán tìm kiếm theo chiều sâu và thuật toán tìm kiếm theo chiều rộng.

1.2.2 Thuật toán DFS

Tư tưởng của thuật toán được trình bày như sau: Bắt đầu từ s , mọi đỉnh u kề với s tất nhiên sẽ đến được từ s . Với mỗi đỉnh u đó, những đỉnh v kề với u cũng đến được từ s ... Ý tưởng đó gợi ý cho ta viết một thủ tục đệ quy $\text{DFS}(u)$ mô tả việc duyệt từ đỉnh u bằng cách thông báo thăm đỉnh u và tiếp tục quá trình duyệt $\text{DFS}(v)$ với v là một đỉnh chưa thăm kề với u . Để quá trình duyệt không lặp lại bất kì đỉnh nào, ta dùng kĩ thuật đánh dấu, khi thăm một đỉnh, ta sẽ đánh dấu đỉnh đó để các bước duyệt đệ quy tiếp theo không thăm lại đỉnh đó nữa.

Vấn đề còn lại: để in ra được đường đi từ đỉnh xuất phát s , trong quá trình duyệt $\text{DFS}(u)$, trước khi gọi đệ quy $\text{DFS}(v)$ với v là một đỉnh kề với u mà chưa đánh dấu, ta lưu lại **vết** đường đi từ u tới v bằng cách đặt $\text{trace}[v] = u$, tức là $\text{trace}[v]$ là đỉnh liền trước v trong đường đi từ s tới v . Khi quá trình tìm kiếm theo chiều sâu kết thúc, đường đi từ S tới F sẽ là:

$$t \leftarrow \text{trace}[t] \leftarrow \dots \leftarrow \text{trace}[u_1] \leftarrow \dots \leftarrow \text{trace}[s] \leftarrow s$$

Truy ngược đường đi này sẽ cho ta hành trình đi từ s đến t . Có thể mô tả thủ tục DFS dưới dạng giả mã như sau:

```

1 void DFS(vertex u)
2 {
3     < 1. Thông báo đã tới đỉnh u >;
4     < 2. Đánh dấu u là đã thăm >;
5     foreach (v kề với u)
6     if (v chưa được thăm)
7     {
8         trace[v] = u; //Truy vết được đi
9         DFS(v); //Gọi đệ quy bắt đầu với v
10    }
11 }
12
13 void DFSPath()
14 {

```

```

15 < Input: do thi G, s, t >;
16 < Khoi tao: Tat ca cac dinh deu chua bi danh dau >;
17
18 DFS(s);
19
20 if (v chua bi danh dau)
21     <Thong bao khong co duoc di tu s->t>;
22 else
23     <Dua vao trace tim duong di tu s->t>;
24 }

```

1.2.3 Thuật toán tìm kiếm theo chiều rộng BFS

Ý tưởng của phương pháp này là *lập lịch* duyệt các đỉnh. Khi thăm một đỉnh ta sẽ lên lịch thăm tất cả các đỉnh kề nó sao cho thứ tự duyệt là ưu tiên chiều rộng (đỉnh nào gần s hơn sẽ được duyệt trước).

Ví dụ:

Bắt đầu ta thăm đỉnh s. Quá trình thăm s sẽ lên lịch duyệt những đỉnh (u_1, u_2, \dots, u_p) kề với s (những đỉnh gần s nhất). Tiếp theo sẽ thăm đỉnh u_1 , khi thăm u_1 sẽ lại lên lịch duyệt những đỉnh (v_1, v_2, \dots, v_q) kề với u_1 . Nhưng rõ ràng các đỉnh này xa s hơn những đỉnh u nên chúng chỉ được duyệt đến khi tất cả những đỉnh u đã duyệt xong, tức là thứ tự duyệt đỉnh sau khi đã thăm u_1 sẽ là: $(u_2, u_3, \dots, u_p, v_1, v_2, \dots, v_q)$. Do việc lập lịch như mô tả ở trên nên cần phải xếp hàng cho các đỉnh đã lên lịch theo đúng thứ tự. Khi thêm đỉnh nào đó, ta sẽ thêm vào cuối hàng. Chính vì nguyên tắc đó nên danh sách chứa những đỉnh đang chờ sẽ được tổ chức dưới dạng hàng đợi - *queue*.

Ta sẽ xây dựng giải thuật như sau:

```

1 void BFS(vertex v)
2 {
3     //Khoi tao
4     free[s] = true;
5     free[v] = false; //v in V, v != s
6     queue.push(v); //queue chỉ chứa v
7
8     while (!queue.empty())
9     {
10        u = queue.pop();
11        free[u] = false;
12        foreach(v ke voi u)
13            if (free[v] == true)
14            {
15                free[v] = false;
16                trace[v] = u;
17                queue.push(v);
18            }
19        //Cac dinh ke u da duoc ghe tham
20    }
21
22 void BFSPath()
23 {
24     < Input: do thi G, s, t >;
25     < Khoi tao: Tat ca cac dinh deu chua bi danh dau >;
26
27     BFS(s);
28
29     if (v chua bi danh dau)
30         <Thong bao khong co duoc di tu s->t>;
31     else
32         <Dua vao trace tim duong di tu s->t>;
33 }

```

1.2.4 Độ phức tạp của thuật toán DFS và BFS

Quá trình tìm kiếm trên đồ thị bắt đầu từ một đỉnh có thể thăm các đỉnh còn lại, khi đó cách biểu diễn đồ thị có ảnh hưởng lớn tới chi phí về thời gian thực hiện giải thuật.

Trong trường hợp ta biểu diễn đồ thị bằng danh sách kề, cả hai thuật toán BFS và DFS đều có độ phức tạp tính toán là $O(n + m) = O(\max(n, m))$. Đây là cách cài đặt tốt nhất. Nếu ta biểu diễn đồ thị bằng ma trận kề thì độ phức tạp tính toán trong trường hợp này là $O(n + n^2) = O(n^2)$. Nếu ta biểu diễn đồ thị bằng danh sách cạnh, thao tác duyệt những đỉnh kề với đỉnh u sẽ dẫn tới phải duyệt qua toàn bộ danh sách cạnh, đây là cài đặt tồi nhất, nó có độ phức tạp tính toán là $O(n.m)$.

1.3 Bài toán tìm đường đi ngắn nhất trên đồ thị có trọng số

1.3.1 Phát biểu bài toán

Bài toán được phát biểu dưới dạng tổng quát như sau: Cho đồ thị $G = (V, E)$ là đồ thị không có chu trình âm. Hãy tìm một đường đi ngắn nhất (tổng trọng số qua các đỉnh trên đường đi là bé nhất) từ đỉnh xuất phát $s \in V$ đến đỉnh đích $t \in V$. Nếu như đồ thị có chu trình âm thì khoảng cách giữa một số cặp đỉnh nào đó có thể không xác định vì bằng cách đi vòng theo chu trình này một số lần đủ lớn, ta có thể chỉ ra đường đi giữa hai đỉnh nào đó trong chu trình này nhỏ hơn bất kỳ một số cho trước nào. Trong trường hợp như vậy có thể đặt vấn đề tìm đường đi cơ bản (đường đi không có đỉnh lặp lại) ngắn nhất. Vấn đề đó là một bài toán hết sức phức tạp mà ta không bàn tới ở đây.

Dưới đây trình bày hai thuật toán để giải bài toán này là thuật toán Bellman-Ford và thuật toán Dijkstra.

1.3.2 Giới thiệu thuật toán Bellman-Ford

Thuật toán Bellman-Ford có thể phát biểu rất đơn giản:

Với đỉnh xuất phát S , gọi $d(v)$ là khoảng cách từ S tới v . Ban đầu $d(S)$ được gán bằng 0, còn các $d(v)$ với $v \neq S$ được khởi gán bằng $+\infty$

Sau đó, ta tối ưu hóa dần các $d(v)$ như sau:

Xét mọi cặp đỉnh u, v của đồ thị, nếu có một cặp đỉnh u, v mà $d(v) > d(u) + c(u, v)$ thì ta đặt lại $d(v) = d(u) + c(u, v)$. Tức là độ dài đường đi từ S tới v lại lớn hơn tổng độ dài đường đi từ S tới u cộng với chi phí đi từ u tới v thì ta sẽ hủy bỏ đường đi từ S tới v đang có và coi đường đi từ S tới v chính là đường đi từ S tới u , sau đó đi tiếp từ u tới v .

Chú ý rằng ta đặt $c[u, v] = +\infty$ nếu (u, v) không là cung. Thuật toán sẽ kết thúc khi không thể tối ưu thêm bất kỳ một nhãn $d[v]$ nào nữa.

Tính dừng của thuật toán

Tại bước lặp 0 Bước khởi tạo $d(S) = 0; d(v) = +\infty$ với $v \neq S$ thì dãy $d(v)$ chính là độ dài đường đi ngắn nhất từ S tới v qua không quá 0 cạnh.

Tại bước lặp thứ i $d(v)$ bằng độ dài ngắn nhất từ S tới v qua không quá i cạnh, do tính chất đường đi từ S tới v qua không quá $i + 1$ cạnh sẽ phải thành lập bằng cách lấy một đường đi từ S tới một đỉnh u nào đó qua không quá i cạnh rồi đi tiếp tới v bằng cung (u, v) nên độ dài đường đi ngắn nhất từ S tới v qua không quá $i + 1$ cạnh sẽ được tính bằng giá trị nhỏ nhất trong các giá trị (Nguyên lý tối ưu Bellman):

- Độ dài đường đi ngắn nhất từ S tới v qua không quá i cạnh
- Độ dài đường đi ngắn nhất từ S tới u qua không quá i cạnh cộng với trọng số cạnh (u, v) ($\forall u$)

nên sau bước lặp tối ưu các $d(v)$ bằng công thức $d(v)_{bci+1} = \min(d(v)_{bci}, d(u)_{bci} + c(u, v))$ thì các $d(v)$ sẽ bằng độ dài đường đi ngắn nhất từ S tới v qua không quá $i + 1$ cạnh.

Sau bước lặp tối ưu hóa thứ $n-1$ Ta có $d(v)$ là độ dài đường đi ngắn nhất từ S tới v qua không quá $n-1$ cạnh. Vì đồ thị không có chu trình âm nên sẽ có một đường đi ngắn nhất từ S tới v là đường đi cơ bản (qua không quá $n-1$ cạnh). Tức là $d(v)$ sẽ là độ dài đường đi ngắn nhất từ S tới v .

Vậy thì số bước lặp tối ưu hóa sẽ không quá $n-1$ bước.

Nếu mỗi bước ta mô tả dưới dạng:

```

for (u=0; u<n; u++)
    for (v=0; v<n; v++)
        if (d[v] > d[u] + c[u][v])
            d[v] = d[u] + c[u][v];

```

Do sự tối ưu bắc cầu (dùng $d(u)$ tối ưu $d(v)$ rồi lại có thể dùng $d(v)$ tối ưu $d(w)$ nữa ...) nên chỉ làm tốc độ tối ưu nhân $d(v)$ tăng lên chứ không thể giảm đi được.

1.3.3 Giới thiệu thuật toán Dijkstra

Thuật toán Dijkstra (E.Dijkstra - 1959) có thể mô tả như sau:

Bước 1: Khởi tạo Với đỉnh $v \in V$, gọi nhãn $d[v]$ là độ dài đường đi ngắn nhất từ s tới v . Ta sẽ tính các $d[v]$. Ban đầu $d[v]$ được khởi gán bằng $c[s][v]$. Nhãn của mỗi đỉnh có hai trạng thái tự do hay cố định, nhãn tự do nghĩa là có thể tối ưu hơn được nữa, và nhãn cố định tức là $d[v]$ đã là độ dài đường đi ngắn nhất từ s tới v nên không thể tối ưu thêm. Để làm điều này ta có thể sử dụng kĩ thuật đánh dấu $free[v] = true$ hay $false$ tùy theo $d[v]$ tự do hay cố định. Ban đầu các nhãn đều tự do.

Bước 2: Lặp

- Cố định nhãn: Chọn trong các đỉnh có nhãn tự do, lấy ra đỉnh u là đỉnh có $d[u]$ nhỏ nhất và cố định nhãn đỉnh u .
- Sửa nhãn: Dùng đỉnh u xét tất cả những đỉnh v và sửa lại các $d[v]$ theo công thức: $d[v] = \min(d[v], d[u] + c[u][v])$

Bước lặp sẽ kết thúc khi mà đỉnh đích t được cố định nhãn, hoặc tại thao tác cố định nhãn, tất cả các đỉnh tự do đều có nhãn là $+\infty$.

Có thể đặt câu hỏi, ở thao tác 1, tại sao đỉnh u như vậy lại được cố định nhãn, giả sử $d[u]$ còn có thể tối ưu thêm được nữa thì tất phải có một đỉnh t mang nhãn tự do sao cho $d[u] > d[t] + c[t][u]$. Do trọng số $c[t][u]$ không âm nên $d[u] > d[t]$, trái với cách chọn $d[u]$ là nhỏ nhất. Tất nhiên trong lần lặp đầu tiên thì S là đỉnh được cố định nhãn do $d[s] = 0$.

Bước 3 Kết hợp với việc lưu vết đường đi trên từng bước sửa nhãn, thông báo đường đi ngắn nhất tìm được hoặc cho biết không tồn tại đường đi ($d[t] = +\infty$).

Có thể mô tả ngắn gọn thuật toán bằng mã giả như sau:

```

void Dijkstra()
{
    /Khoi tao
    foreach(v thuoc V)
    {
        d[v] = c[s][w];
        p[v] = s;
    }
    d[s] = 0;
    S = {s}; //tap cac nhan co dinh co dinh
    T = V \ S; //T la tap cac dinh co nhan tam thoi

    //Buoc lap
    while (u!=t && !T.empty())
    {
        <Tim dinh u thuoc T thoa man d[u] = min(d[z] | z thuoc T)>
        T = T \ {u};
        S = S U {u}; //Co dinh nhan cua dinh u

        //Gan lai nhan cho cac dinh trong T
        foreach (v thuoc T)
        {
            if (d[v] > d[u] + c[u][v])
            {
                d[v] = d[u] + c[u][v];
                trace[v] = u;
            }
        }
    }
}

```

}
}

1.3.4 Độ phức tạp

Nếu đồ thị có nhiều đỉnh, ít cạnh, ta có thể sử dụng danh sách kề kèm trọng số để biểu diễn đồ thị, tuy nhiên tốc độ của thuật toán Dijkstra vẫn khá chậm trong trường hợp xấu nhất, nó cần n lần cố định nhân và mỗi lần cố định nhân sẽ mất một đoạn chương trình độ phức tạp $O(n)$. Vậy độ phức tạp của thuật toán Dijkstra là $O(n^2)$. Để tăng tốc độ, người ta sử dụng cấu trúc dữ liệu Heap để lưu các đỉnh chưa cố định nhân. Heap ở đây là một cây nhị phân hoàn chỉnh thỏa mãn: Nếu u là đỉnh lưu nút cha và v là đỉnh lưu ở nút con thì $d[u] \leq d[v]$ (Đỉnh r lưu ở gốc Heap là đỉnh có $d[r]$ nhỏ nhất).

1.4 Các thuật toán tìm kiếm trên cây khung

1.4.1 Bài toán cây khung

Khái niệm cây khung: Cho đồ thị $G = (V, E)$ vô hướng, liên thông và $T = (V, E')$ là một đồ thị con của G với $E' \subseteq E$. Khi đó, T được gọi là cây khung (cây khung bao trùm) nếu T liên thông và không có chu trình đơn.

Cho $G = (V, E, w)$ là đồ thị vô hướng liên thông có trọng số, với một cây khung T của G , ta gọi trọng số của cây T là tổng trọng số các cạnh trong T .

Yêu cầu: Trong số các cây khung của G , chỉ ra cây khung nào có trọng số nhỏ nhất.

Cây khung như vậy được gọi là cây khung nhỏ nhất của đồ thị, và bài toán xây dựng cây khung nhỏ nhất. Dưới đây ta xét một trong hai thuật toán thông dụng để giải bài toán cây khung nhỏ nhất của đơn đồ thị vô hướng có trọng số.

1.4.2 Giới thiệu thuật toán Prim

Một trong hai thuật toán quan trọng để giải bài toán tìm cây khung nhỏ nhất là thuật toán Prim. Thuật toán đó có thể phát biểu hình thức như sau: Đơn đồ thị vô hướng $G = (V, E, w)$. Xét cây T trong G và một đỉnh v , gọi khoảng cách từ v tới T là trọng số nhỏ nhất trong số các cạnh nối v với một đỉnh nào đó trong T :

$$d[v] = \min\{w[u, v] | u \in T\}$$

Ban đầu khởi tạo cây T chỉ gồm có mỗi đỉnh 1. Sau đó cứ chọn trong số các đỉnh ngoài T ra một đỉnh gần T nhất, kết nạp đỉnh đó vào T đồng thời kết nạp luôn cả cạnh tạo ra khoảng cách gần nhất đó, cứ làm như vậy cho tới khi:

- Hoặc đã kết nạp được tất cả n đỉnh thì ta có T là cây khung nhỏ nhất
- Hoặc chưa kết nạp được hết n đỉnh nhưng mọi đỉnh ngoài T đều có khoảng cách tới T là $+\infty$. Khi đó đồ thị đã cho không liên thông, ta thông báo việc tìm cây thất bại.

1.4.3 Giới thiệu thuật toán Kruskal

Thuật toán Kruskal dựa trên mô hình xây dựng cây khung bằng thuật toán hợp nhất, chỉ có điều thuật toán không phải xét các cạnh với thứ tự tùy ý mà xét các cạnh theo thứ tự đã sắp xếp. Với đồ thị vô hướng $G = (V, E)$ có n đỉnh. Khởi tạo cây T ban đầu không có cạnh nào. Xét tất cả các cạnh của đồ thị từ cạnh có trọng số nhỏ đến cạnh có trọng số lớn, nếu việc thêm cạnh đó vào T không tạo thành chu trình đơn trong T thì kết nạp thêm cạnh đó vào T , cứ làm như vậy cho tới khi:

- Hoặc đã kết nạp được $n-1$ cạnh vào trong T thì ta được T là cây khung nhỏ nhất
- Hoặc chưa kết nạp đủ $n-1$ cạnh nhưng thêm bất cứ một cạnh bất kì trong số các cạnh còn lại thì sẽ tạo thành chu trình đơn. Trong trường hợp này đồ thị G là không liên thông, việc tìm cây khung thất bại.

Như vậy có hai vấn đề quan trọng khi cài đặt thuật toán Kruskal:

Thứ nhất, làm thế nào để xét được các cạnh từ trọng số nhỏ tới cạnh có trọng số lớn. Ta có thể thực hiện bằng cách sắp xếp danh sách cạnh theo thứ tự không giảm của trọng số, sau đó duyệt từ đầu tới cuối danh sách cạnh, nên sử dụng các thuật toán sắp xếp hiệu quả để đạt được tốc độ nhanh trong trường hợp số cạnh lớn. Trong trường hợp tổng quát, thuật toán HeapSort là hiệu quả nhất bởi nó cho phép chọn lần lượt các cạnh từ cạnh có trọng số nhỏ nhất tới cạnh có trọng số lớn nhất ra khỏi Heap và có thể xử lý (bỏ qua hay thêm vào cây) luôn.

Thứ hai, làm thế nào để kiểm tra xem việc thêm một cạnh có tạo thành chu trình đơn trong T hay không. Để ý rằng các cạnh trong T ở các bước sẽ tạo thành một rừng (đồ thị không có chu trình đơn). Muốn thêm một cạnh (u, v) vào T mà không tạo thành chu trình đơn thì (u, v) phải nối hai cây khác nhau của rừng T , bởi nếu u, v thuộc cùng một cây thì sẽ tạo thành chu trình đơn trong cây đó. Ban đầu, ta khởi tạo rừng T gồm n cây, mỗi cây chỉ gồm đúng một đỉnh, sau đó mỗi khi set đến cạnh mới hai cây khác nhau của rừng T thì ta kết nạp cạnh đó vào T , đồng thời hợp nhất hai cạnh đó tạo thành một cây.

1.4.4 Độ phức tạp

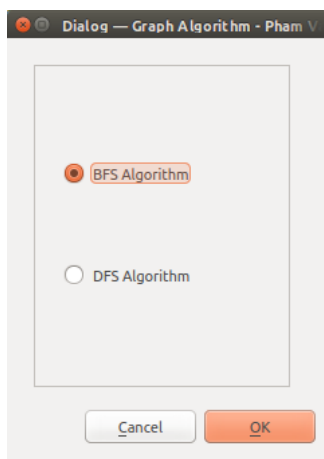
Xét về độ phức tạp tính toán, thuật toán Prim có độ phức tạp là $O(n^2)$. Tương tự thuật toán Dijkstra, nếu kết hợp thuật toán Prim với cấu trúc Heap sẽ được một thuật toán với độ phức tạp $O((m+n)\log n)$.

Đối với thuật toán Kruskal, ta có thể chứng minh rằng thao tác GetRoot có độ phức tạp là $O(\log_2 n)$ còn thao tác Union là $O(1)$. Giả sử ta đã có danh sách cạnh đã sắp xếp rồi thì xét vòng lặp dựng cây khung, nó duyệt qua danh sách cạnh và mỗi cạnh nó gọi 2 lần thao tác GetRoot, như vậy thì độ phức tạp là $O(m \log_2 n)$, nếu đồ thị có cây khung thì $m \geq n - 1$ nên ta thấy chi phí thời gian chủ yếu nằm ở thao tác sắp xếp danh sách cạnh bởi độ phức tạp của HeapSort là $O(m \log_2 m)$. Vậy độ phức tạp của thuật toán là $O(m \log_2 m)$ trong trường hợp xấu nhất. Tuy nhiên, phải lưu ý rằng để xây dựng cây khung thì ít khi thuật toán phải duyệt toàn bộ danh sách cạnh mà chỉ một phần của danh sách cạnh mà thôi.

2 Mô phỏng thuật toán DFS và BFS

2.1 Khung chương trình

Khởi động chương trình sẽ có *dialog* cho phép người sử dụng lựa chọn thuật toán mô phỏng là DFS hay BFS:



Hình 1: Dialog lựa chọn thuật toán

Khung chương trình cho phép người dùng lựa chọn các thao tác:

Select Lựa chọn đỉnh để kéo thả vào các vị trí thích hợp. Đỉnh được lựa chọn sẽ có dấu màu đỏ.

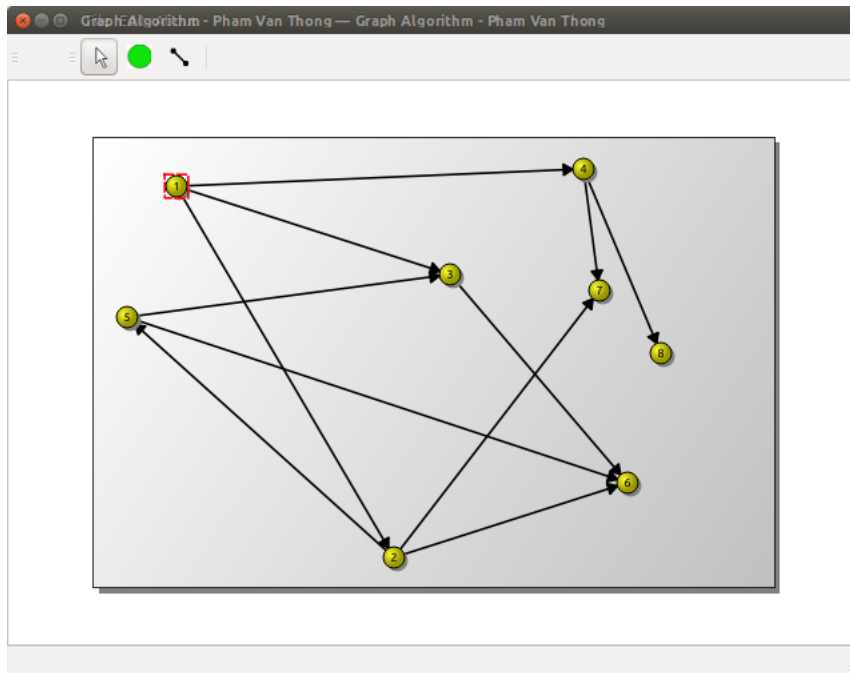
Draw Vertex Vẽ đỉnh trên màn hình. Nhân của đỉnh được đánh số, tự động tăng khi có một đỉnh được tạo ra. Đỉnh được tạo ra tại vị trí con trỏ chuột, đỉnh sau khi tạo ra có thể **select** để chuyển đi vị trí mới.

Draw Edge Vẽ cạnh giữa hai đỉnh. Cạnh tự thay đổi theo vị trí của đỉnh.

Play Animation Chạy chương trình. Với mỗi bước của thuật toán, trạng thái các đỉnh, cạnh trên màn hình sẽ thay đổi. Cụ thể, đối với đỉnh, màu vàng nghĩa là chưa được duyệt, màu xanh là đã được thăm nhưng chưa hoàn thành, và màu xám là đã hoàn thành quá trình thăm. Với cạnh, các cạnh trong cây sẽ được tô màu xanh, lần lượt theo từng bước của thuật toán.

2.2 Tổng quan về chương trình

Chương trình được lập trình trên C++, dựa trên bộ công cụ QT 5. Chương trình được chia thành các module nhỏ. Có 2 module chính:



Hình 2: Cửa sổ chính chương trình

Graphics Tool Thực hiện việc thiết kế giao diện dành cho quá trình mô phỏng, bao gồm việc nhập liệu cho người dùng như tạo đỉnh, cạnh, di chuyển đỉnh, và hiển thị các bước thay đổi của thuật toán. Bao gồm các class chính: *GraphScene*, *GraphWidget*, *MainWindow*, *Dialog*, *Vertex*, *Edge*

Algorithm Thực hiện cài đặt mô hình thuật toán, lên kịch bản cho quá trình mô phỏng. Bao gồm các class chính: *Algorithm*, *DFSAnimation*, *BFSAnimation*, *Vertex*, *Edge*

Giữa hai module luôn có mối quan hệ khăng khít với nhau, một module nhận thông tin về các đối tượng đỉnh và cạnh được do người dùng nhập từ bàn phím chuyển tới cho module kia. Module còn lại tiếp nhận thông tin đó, và thực thi thuật toán, sau đó lên kịch bản để mô phỏng rồi chuyển tới cho module kia để trình diễn.

2.3 Xây dựng các class chính

2.3.1 Class Vertex

Class *Vertex* mô phỏng đối tượng đỉnh, kế thừa từ lớp *QGraphicsObject*. Được cài đặt như sau:

```

1  #ifndef VERTEX_H
2  #define VERTEX_H
3
4  #include <QGraphicsItem>
5  #include <QGraphicsObject>
6  #include <QList>
7
8  class Edge;
9  class GraphWidget;
10 class QGraphicsSceneMouseEvent;
11
12
13 class Vertex : public QGraphicsObject
14 {
15     Q_OBJECT
16     Q_PROPERTY(VertexColor color READ color WRITE setColor)
17     Q_ENUMS(VertexColor)
18 public:
19     Vertex(qreal x, qreal y);
20
21     void addInEdge(Edge *edge);

```



```

22     void addOutEdge(Edge *edge);
23
24     bool removeInEdge(Edge *edge);
25     bool removeOutEdge(Edge *edge);
26
27
28     QList<Edge *> inEdges() const {return inEdgeList;}
29     QList<Edge *> outEdges() const {return outEdgeList;}
30
31     enum { Type = UserType + 1 };
32     int type() const Q_DECL_OVERRIDE { return Type; }
33
34     QRectF boundingRect() const Q_DECL_OVERRIDE;
35     QPainterPath shape() const Q_DECL_OVERRIDE;
36     void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget) Q_DECL_OVERRIDE;
37
38     enum VertexColor{Init, Visited, Source, Destination, Discovered};
39     static QColor vertexColor[5][2];
40
41
42     static int interconnect;
43
44     VertexColor color() const {return m_color;}
45     void setColor(VertexColor);
46
47     int getId() const {return id;}
48
49     void setAssociate(int m) {m_associate = m;}
50     int associate() const {return m_associate;}
51
52 protected:
53     QVariant itemChange(GraphicsItemChange change, const QVariant &value) Q_DECL_OVERRIDE;
54
55     void mousePressEvent(QGraphicsSceneMouseEvent *event) Q_DECL_OVERRIDE;
56     void mouseReleaseEvent(QGraphicsSceneMouseEvent *event) Q_DECL_OVERRIDE;
57     void mouseMoveEvent(QGraphicsSceneMouseEvent *event) Q_DECL_OVERRIDE;
58
59 private:
60     QList<Edge *> inEdgeList;
61     QList<Edge *> outEdgeList;
62     QPointF newPos;
63     static int vertexID;//new
64
65     int m_associate;
66     int id;//new
67     VertexColor m_color;
68 };
69
70 #endif // VERTEX_H

```

Các thuộc tính và phương thức chính là:

Hàm tạo Tạo đỉnh ở tọa độ là tham số được truyền vào. Xác định một số cờ hiệu và trạng thái mặc định của đỉnh.

Các hàm thêm, xóa cung Các hàm addInEdge(), addOutEdge(), removeInEdge(), removeOutEdge() thực hiện chức năng thêm, xóa các cung vào, ra tại đỉnh.

Hàm inEdges(), outEdges() Trả về danh sách cạnh tới đỉnh và ra khỏi đỉnh.

Hàm boundingRect() Định nghĩa lại phương thức ảo từ lớp QGraphicsObject, xác định phạm vi của đối tượng trong QGraphicsScene. Ví dụ nó giúp phát hiện được người dùng có đang trỏ chuột vào đối tượng không.

Hàm `paint()` Định nghĩa lại phương thức ảo từ lớp `QGraphicsObject`, xác định cách mà đỉnh đó được vẽ ra trên `QGraphicsScene`.

Thuộc tính lớp `interconnect` Xác định số thành phần liên thông của đồ thị.

Thuộc tính `m_associate` Xác định đỉnh thuộc thành phần liên thông nào, giữ luôn vai trò đánh dấu trong các thuật toán BFS và DFS.

Các hàm `mouseEvent()` Vẽ lại đối tượng khi có thay đổi trên đối tượng.

Các hàm và thuộc tính khác

2.3.2 Class `Edge`

Class `Edge` mô phỏng đối tượng cạnh, được kế thừa từ lớp `QGraphicsObject`. Cài đặt như sau:

```
1  #ifndef EDGE_H
2  #define EDGE_H
3
4  #include <QGraphicsObject>
5  #include <QColor>
6
7  class Vertex;
8
9  class Edge : public QGraphicsObject
10 {
11     Q_OBJECT
12     Q_PROPERTY(EdgeState state READ state WRITE setState)
13     Q_ENUMS(EdgeState)
14 public:
15     Edge(Vertex *sourceVertex, Vertex *destVertex);
16
17     Vertex *sourceVertex() const;
18     Vertex *destVertex() const;
19
20     void adjust();
21
22     enum { Type = UserType + 2 };
23     int type() const Q_DECL_OVERRIDE { return Type; }
24
25     enum EdgeState {Init, Relaxed, InPath, Error};
26     static QColor EdgeColor[4];
27     EdgeState state() const { return edgeState; }
28     void setState(EdgeState state);
29
30 protected:
31     QRectF boundingRect() const Q_DECL_OVERRIDE;
32     void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget *widget) Q_DECL_OVERRIDE;
33
34 private:
35     Vertex *source, *dest;
36     EdgeState edgeState;
37     QPointF sourcePoint;
38     QPointF destPoint;
39     qreal arrowSize;
40 };
41
42 #endif // EDGE_H
```

Các phương thức và thuộc tính chính là:

Hàm tạo Khởi tạo cạnh từ đỉnh nguồn tới đỉnh đích. Đặt trạng thái mặc định, thêm thông tin về cạnh tới, cạnh ra khỏi đỉnh vào đỉnh nguồn và đỉnh đích tương ứng.

Hàm `setSTate()` Thay đổi trạng thái của cạnh (Init, Inpath, ...)

Hàm `adjust()` Thay đổi cạnh để phù hợp với hiển thị (cạnh nối từ viền của đỉnh chứ không phải từ tâm của đỉnh, nếu hai đỉnh quá sát nhau thì không vẽ cạnh ra).

Các hàm `boundingRect()`, `paint()` Tương tự đối với class *Vertex*.

2.3.3 Class Algorithm

Class *Algorithm* được xây dựng như sau:

```
1  #ifndef ALGORITHM_H
2  #define ALGORITHM_H
3
4  class MainWindow;
5  class GraphScene;
6  class Vertex;
7  class Edge;
8  class QAction;
9  class QAbstractAnimation;
10
11 #include <QVector>
12 #include <QList>
13 #include <QObject>
14 #include <QToolBar>
15 #include <memory>
16
17 class Algorithm: public QObject
18 {
19     Q_OBJECT
20 public:
21     Algorithm(MainWindow *mainWindow);
22     enum AlgType{DFS, BFS, ShortestPath};
23
24     void incEdge() {++nEdge;}
25     void decEdge() {--nEdge;}
26     void clearEdge() {nEdge = 0;}
27
28     QToolBar* getAnimationToolBar();
29
30     virtual Edge* newEdge(Vertex*, Vertex*) = 0;
31
32     void setSelectedVertex(Vertex *);
33
34     static Algorithm *getAlgorithm(AlgType, MainWindow*);
35
36     virtual AlgType type() const = 0;
37
38     friend class GraphScene;
39
40     void resetState();
41 public slots:
42     void setSource();
43     void setDest();
44
45     void addVertex(Vertex *);
46     void removeVertex(Vertex *);
47
48 protected:
49     MainWindow *mainWindow;
50     int nEdge;
51 }
```

```

52     QList<Vertex*> vertexList;
53
54     Vertex *selectedVertex;
55     Vertex *sourceVertex;
56     Vertex *destVertex;
57
58
59
60     void timerEvent(QTimerEvent *) override;
61     virtual void generateAnimationList() = 0;
62     void generateBFSAnimationList(); //debug
63
64     QList<std::shared_ptr<QAbstractAnimation> > animationList;
65     QList<std::shared_ptr<QAbstractAnimation> >::iterator currentAnimation;
66
67     void setDelay(int dl) {delay = dl;}
68
69     void createToolBar();
70     void createActions();
71
72 protected slots:
73     void finishAnimation();
74     void initAllItem();
75
76
77 private:
78     int delay;
79     int timerId;
80
81     bool isPaused;
82     bool isStarted;
83     bool isAutoPlay;
84     bool isForward;
85
86     QAction *playAction;
87     QAction *nextAction;
88     QAction *prevAction;
89     QAction *stopAction;
90
91     QToolBar *animationToolBar;
92 private slots:
93     void play();
94     void next();
95     void prev();
96     void stop();
97 };
98
99 #endif // ALGORITHM_H

```

Trong đó, các phương thức và thuộc tính chính là:

Hàm tạo Tạo action và *Play Toolbar* dùng để chạy thuật toán, cài đặt các tham số khác như đỉnh nguồn mặc định, đỉnh đích mặc định (cho các thuật toán khác sẽ được phát triển tiếp), thời gian delay animation, ...

Hàm getAlgorithm() Khởi động hàm với thuật toán tương ứng *BFS*, *DFS*

Hàm newEdge() Tạo cạnh mới, là phương thức ảo. Mỗi lớp con sẽ định nghĩa phương thức này khác nhau.

Hàm resetState() Khởi động lại trạng thái các đỉnh, cạnh sau mỗi lần duyệt xong (hủy đánh dấu, thay đổi màu sắc).

Thuộc tính vertexList Chứa danh sách các đỉnh, có thể xem dùng như danh sách kề (vì mỗi đỉnh chứa các danh sách outEdges và inEdges).

Hàm generateAnimationList() Phương thức ảo, mỗi lớp con sẽ định nghĩa phương thức này để trình diễn animation tương ứng của thuật toán ra màn hình.

Hàm play() Được kết nối với action Play dùng để khởi động animation.

Các thuộc tính và phương thức khác Thực hiện các chức năng khác hoặc nhằm mục đích phát triển tiếp của chương trình.

2.3.4 Class DFSAnimation, BFSAnimation

Hai class này được cài đặt tương tự nhau, class BFSAnimation được cài đặt như sau:

```
1  #ifndef BFS_H
2  #define BFS_H
3  #include "algorithm.h"
4  #include <QSequentialAnimationGroup>
5  class MainWindow;
6  class Edge;
7  class Vertex;
8
9
10 #include "edge.h"
11
12 class BFSAnimation : public Algorithm
13 {
14     Q_OBJECT
15 public:
16     BFSAnimation(MainWindow *mainWindow);
17
18     Algorithm::AlgType type() const override {return Algorithm::BFS;}
19     Edge* newEdge(Vertex *, Vertex *) override;
20 protected:
21     void generateAnimationList() override;
22 private:
23     void BFSVertex(Vertex *v);
24     //void discoverVertex(Vertex *v);
25
26     QSequentialAnimationGroup *animationGroup;
27
28 };
29
30 #endif // BFS_H
```

Các phương thức và thuộc tính chính:

Hàm tạo Kế thừa từ lớp Algorithm

Hàm newEdge() Định nghĩa lại phương thức ảo từ lớp Algorithm.

Hàm BFSVertex() Thực hiện BFS tại 1 đỉnh.

Hàm generateAnimationList() BFS trên toàn bộ đồ thị, tạo animation.

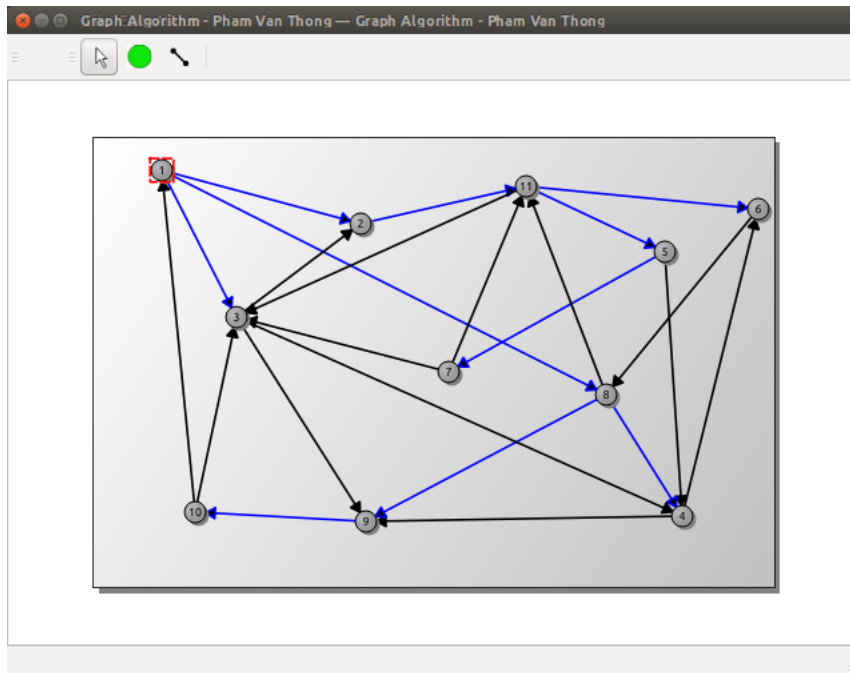
Thuộc tính animationGroup Lưu animation sẽ được trình diễn.

Lớp DFSAnimation được cài đặt hoàn toàn tương tự.

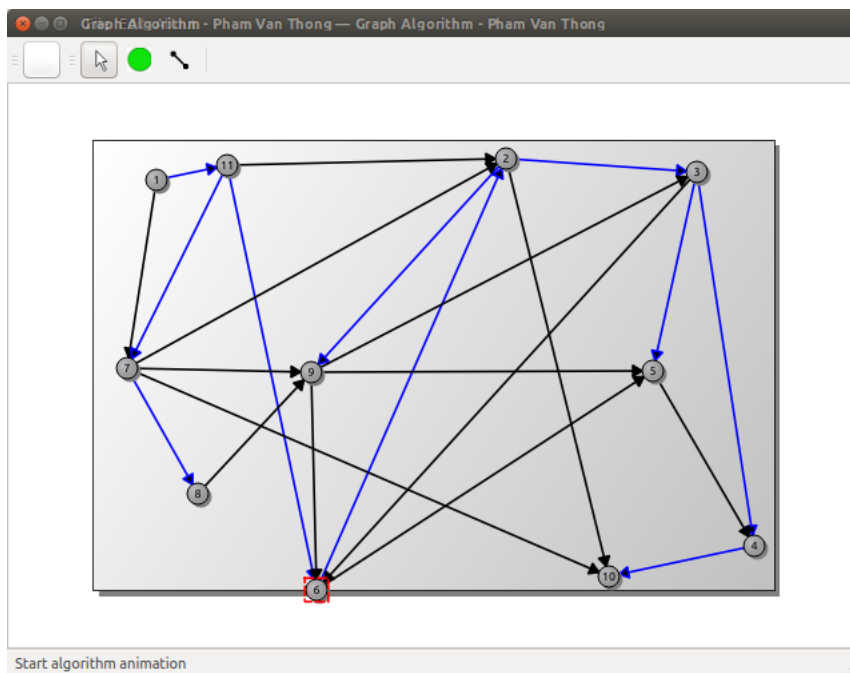
Trong các class trên có dư thừa nhiều phương thức và thuộc tính để nhằm mục đích phát triển cho các thuật toán sau này.

2.4 Kết quả chạy thử

Hình ảnh kết quả chạy thử chương trình:



Hình 3: Kết quả chạy thử BFS Animation



Hình 4: Kết quả chạy thử DFS Animation

3 Thuật toán Bellman-Ford và thuật toán Dijkstra

3.1 Cài đặt

Chương trình này chỉ thực hiện trên giao diện dòng lệnh, chia thành 3 class chính: Graph, Dijkstra, BellmanFord. Class Graph mô tả đối tượng đồ thị, lưu trữ thông tin về số đỉnh, ma trận trọng số của đồ thị, và các phương thức tạo nhập thông tin đồ thị từ file, hiển thị thông tin của đồ thị. Hai class còn lại thừa kế từ lớp Graph và mỗi lớp con thực hiện một thuật toán tương ứng.

Cài đặt cụ thể như sau:

3.1.1 Kết quả chạy thử

Với ma trận kề như tương ứng thì kết quả tìm đường đi ngắn nhất với mỗi thuật toán như sau:

```

1  #ifndef GRAPH_H_
2  #define GRAPH_H_
3
4  #define MAXV 100
5  class Graph
6  {
7  public:
8      int V; //So dinh, so canh
9      int A[MAXV][MAXV]; //Ma tran trong so
10
11      Graph(const char* filename="adjmatrix_input.txt");
12      void info();
13 };
14
15 #endif

```

Hình 5: graph.h

```

1  #include "graph.h"
2  #include <fstream>
3  #include <iostream>
4  #include <iomanip>
5
6  Graph::Graph(const char* filename)
7  {
8      std::ifstream in(filename);
9      in >> V;
10
11      for (int i=0; i<V; i++)
12          for (int j=0; j<V; j++)
13              in >> A[i][j];
14 }
15
16 void Graph::info()
17 {
18     std::cout << "So dinh cua do thi: " << V << std::endl;
19
20     for (int i=0; i<V; i++)
21     {
22         for (int j=0; j<V; j++)
23             std::cout << std::setw(4) << A[i][j];
24         std::cout << std::endl;
25     }
26 }

```

Hình 6: graph.cc

4 Thuật toán cây khung nhỏ nhất

Các thuật toán Kruskal và Prim được cài đặt như sau:

Kết quả chạy thử Với đồ thị đầu vào như sau:

Kết quả thu được như sau:

```

1  #ifndef DIJKSTRA_H_
2  #define DIJKSTRA_H_
3
4  #include "graph.h"
5  #define MAXW 1000
6
7  class Dijkstra:public Graph
8  {
9  private:
10     int s, t;
11
12     int Truoc[MAXV];
13     int d[MAXV];
14     bool final[MAXV];
15 public:
16     void printResult();
17     void dijkstraAlgorithm(int s, int t);
18
19 };
20
21
22 #endif

```

Hình 7: dijkstra.h

5 Kết luận

5.1 Những kết quả đạt được

- Những kiến thức tổng quan, cơ bản về đồ thị và một số thuật toán quan trọng trên đồ thị.
- Xây dựng chương trình mô phỏng thuật toán DFS và BFS trên giao diện đồ họa.
- Cài đặt các thuật toán tìm đường đi ngắn nhất trên đồ thị: Bellman-Ford và Dijkstra.
- Cài đặt các thuật toán tìm cây khung nhỏ nhất của đồ thị: Prim và Kruskal.

5.2 Hạn chế

- Chương trình mô phỏng còn rất đơn giản, ít chức năng.
- Chưa xây dựng được nhiều thuật toán trên giao diện đồ họa


```

1  #include "dijkstra.h"
2  #include <iostream>
3
4
5  void Dijkstra::printResult()
6  {
7      if (d[t] >= MAXW)
8          std::cout << "\nKhong co duong di tu " << s << " den " << t << std::endl;
9      else
10     {
11         std::cout << "\nKhoang cach tu " << s << " den " << t << " la: " << d[t] << std::endl;
12         std::cout << "Duong di ngan nhat tu " << s << " den " << t << std::endl;
13
14
15         int i = t;
16         while (i != s)
17         {
18             std::cout << i << " <- ";
19             i = Truoc[i];
20         }
21         std::cout << s << std::endl;
22     }
23 }
24
25
26
27 void Dijkstra::dijkstraAlgorithm(int s, int t)
28 {
29     this->s = s; this->t = t;
30     int minp, u, v;
31     //Khoi tao nhan
32     for (v=1; v<V; v++)
33     {
34         d[v] = A[s][v];
35         Truoc[v] = s;
36         final[v] = false;
37     }
38
39     Truoc[s] = -1;
40     d[s] = 0;
41     final[s] = true;
42
43     while (!final[t])
44     {
45         //Tim dinh co nhan tam thoi nho nhat
46         minp = MAXW;
47         u = -1;
48         for (v=0; v<V; v++)
49             if (!final[v] && minp>d[v])
50             {
51                 u = v;
52                 minp = d[v];
53             }
54         final[u] = true;
55
56         if (u==-1)
57             break;
58
59         for (v=0; v<V; v++)
60             if (!final[v] && (d[v] > d[u] + A[u][v]))
61             {
62                 d[v] = d[u] + A[u][v];
63                 Truoc[v] = u;
64             }
65     }

```

```

1  #ifndef BELLMANFORD_H_
2  #define BELLMANFORD_H_
3
4  #include "graph.h"
5
6  #define MAXW 1000
7
8  class BellmanFord:public Graph
9  {
10 private:
11     int s, t;
12
13     int Truoc[MAXV];
14     int d[MAXV];
15 public:
16     void printResult();
17     void bellmanFordAlgorithm(int s, int t);
18 };
19
20 #endif //BELLMANFORD_H_

```

Hình 9: bellmanford.h

```

1  #include "bellmanford.h"
2  #include "graph.h"
3  #include <iostream>
4
5  void BellmanFord::printResult()
6  {
7      if (d[t] == MAXW)
8          std::cout << "\nKhong co duong di tu " << s << " den " << t << std::endl;
9      else
10     {
11         std::cout << "\nKhoang cach tu " << s << " den " << t << " la: " << d[t] << std::endl;
12         std::cout << "Duong di ngan nhat tu " << s << " den " << t << std::endl;
13
14
15         int i = t;
16         while (i != s)
17         {
18             std::cout << i << " <- ";
19             i = Truoc[i];
20         }
21         std::cout << s << std::endl;
22     }
23 }
24
25
26 void BellmanFord::bellmanFordAlgorithm(int s, int t)
27 {
28     this->s = s, this->t = t;
29
30     int u, v, count;
31     bool stop;
32     //Khoi tao
33     for (v=1; v<V; v++)
34         d[v] = MAXW;
35     d[s] = 0;
36
37     for (count = 0; count < V-1; count++)
38     {
39         stop = true;
40         for (u=0; u<V; u++)
41             for (v=0; v<V; v++)
42                 if (d[v] > d[u] + A[u][v])
43                 {
44                     d[v] = d[u] + A[u][v];
45                     Truoc[v] = u;
46                     stop = false;
47                 }
48         if (stop)
49             break;
50     }
51 }

```

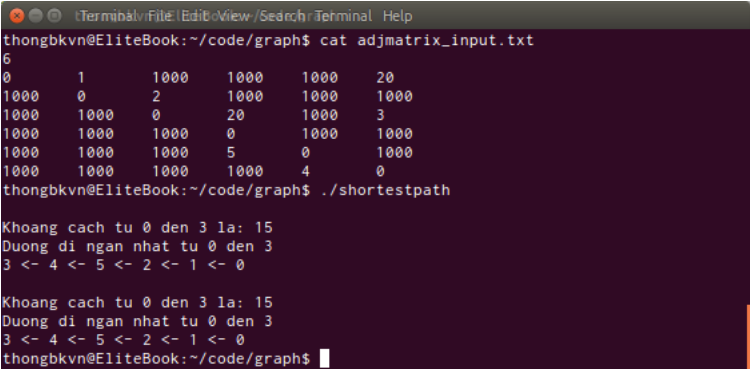
Hình 10: bellmanford.cc

```

1  #include "graph.h"
2  #include "dijkstra.h"
3  #include "bellmanford.h"
4  #include <iostream>
5  int main()
6  {
7      Dijkstra a;
8      BellmanFord b;
9      a.dijkstraAlgorithm(0,3);
10     a.printResult();
11     b.bellmanFordAlgorithm(0,3);
12     b.printResult();
13     return 0;
14 }

```

Hình 11: main.cc



```

thongbkvn@EliteBook:~/code/graph$ cat adjmatrix_input.txt
6
0      1      1000    1000    1000    20
1000    0      2      1000    1000    1000
1000    1000    0      20     1000    3
1000    1000    1000    0      1000    1000
1000    1000    1000    5      0      1000
1000    1000    1000    1000    4      0
thongbkvn@EliteBook:~/code/graph$ ./shortestpath

Khoang cach tu 0 den 3 la: 15
Duong di ngan nhat tu 0 den 3
3 <- 4 <- 5 <- 2 <- 1 <- 0

Khoang cach tu 0 den 3 la: 15
Duong di ngan nhat tu 0 den 3
3 <- 4 <- 5 <- 2 <- 1 <- 0
thongbkvn@EliteBook:~/code/graph$

```

Hình 12: Kết quả chạy thử chương trình tìm đường đi ngắn nhất

```

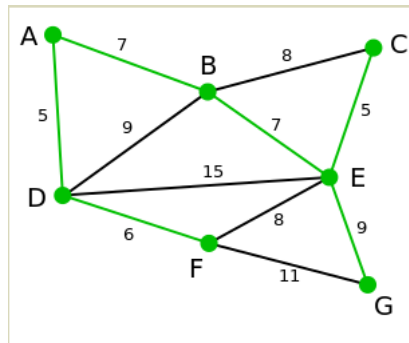
1  #include <iostream>
2  #include <fstream>
3  #include <iomanip>
4
5  #define MAX 100
6  #define INT_MAX 1000
7
8  int V, //So dinh
9      A[MAX][MAX], //Ma tran trong so
10     MST[MAX][MAX]; //Do thi chua cay bao trum nho nhat
11
12 //Khoi tao cac bien can thiet cho thuat toan Prim
13 int parent[MAX], key[MAX];
14 bool Free[MAX];
15
16 int extractMin();
17 void Prim(int start)
18 {
19     //Khoi tao cac bien
20     //parent: nut cha cua nut hien taij
21     //free: nut do da duoc tham hay chua
22     //key: gia tri
23     for (int i=0; i<V; i++)
24     {
25         parent[i] = -1;
26         Free[i] = true;
27         key[i] = INT_MAX;
28     }
29
30     //Bat dau qua trinh tim kiem
31     key[start] = 0;
32
33     for (int i=0; i<V-1; i++)
34     {
35         //Tim dinh tiep theo tu tat ca cac dinh da tim duoc
36         //sao cho dinh moi tim duoc la canh nho nhat
37
38         int u=extractMin();
39         //Thong bao da tim duoc
40         Free[u] = false;
41         //Duyet het tat ca cac nut
42         for (int v=0; v<V; v++)
43             //Neu nut do chua duoc tim thay
44             if (Free[v] && A[u][v] != 0 && A[u][v] < key[v])
45             {
46                 key[v] = A[u][v];
47                 parent[v] = u;
48             }
49     }
50
51     //MST la cay vua tim duoc
52     //dua vao parent de truy vet va tim ra cay bao trum nho nhat
53
54     for (int i=0; i<V; i++)
55         MST[parent[i]][i] = MST[i][parent[i]] = A[parent[i]][i];
56 }
57
58 int extractMin()
59 {
60     int min = INT_MAX, u;
61     for (int i=0; i<V; i++)
62         if (Free[i] && key[i] < min)
63         {
64             min = key[i];
65             u = i;
66         }
67 }

```

```

1  #include <iostream>
2  #include <fstream>
3  #include <iomanip>
4
5
6  #define max 100
7  #define INT_MAX 1000
8  int V, nEdge, A[max][max], MST[max][max], parent[max];
9
10
11 //Cau truc 1 canh
12 struct Edge
13 {
14     int u, v, weight;
15 };
16
17 //Khai bao mang cac canh
18 Edge edge[10*max];
19
20 void Sort();
21 int Find_Set(int);
22 void Union(int, int);
23
24 void Kruskal()
25 {
26     //Sap xep cac canh theo chieu tang dan cua trong so
27     Sort();
28
29
30     //Duyet tat ca cac canh
31     for (int i=0; i<nEdge; i++)
32     {
33         //u va v la 2 dau mut cua canh i
34
35         //tim nut goc cua dinh u
36         int u = Find_Set(edge[i].u);
37         int v = Find_Set(edge[i].v);
38
39         //Neu u giống v thì u và v có cùng dinh goc
40         //vi vậy cần kiểm tra u và v không cùng goc
41         if (u != v)
42         {
43             int x = edge[i].u;
44             int y = edge[i].v;
45             //Luu lai cay bao trum
46             MST[x][y] = MST[y][x] = edge[i].weight;
47             //Ket noi dinh u va dinh v
48             Union(u,v);
49         }
50     }
51 }
52
53 //Ham sap xep canh theo trong so
54 void Sort()
55 {
56     for (int i=0; i<nEdge; i++)
57     for (int j=nEdge-1; j>i; j--)
58         if (edge[j-1].weight > edge[j].weight)
59         {
60             Edge tmp = edge[j-1];
61             edge[j-1] = edge[j];
62             edge[j] = tmp;
63         }
64 }
65

```



Hình 15: Đồ thị đầu vào và kết quả mẫu

```

thongbkvn@EliteBook:~/code/graph$ cat mst_matrix.txt
7
0 7 1000 5 1000 1000 1000
7 0 8 9 7 1000 1000
1000 8 0 1000 5 1000 1000
5 9 1000 0 15 6 1000
1000 7 5 15 0 8 9
1000 1000 1000 6 8 0 11
1000 1000 1000 1000 9 11 0
thongbkvn@EliteBook:~/code/graph$ g++ kruskal.cc
thongbkvn@EliteBook:~/code/graph$ ./a.out
0 7 0 5 0 0 0
7 0 0 0 7 0 0
0 0 0 0 5 0 0
5 0 0 0 0 6 0
0 7 5 0 0 0 9
0 0 0 6 0 0 0
0 0 0 0 9 0 0
thongbkvn@EliteBook:~/code/graph$ g++ prim.cc
thongbkvn@EliteBook:~/code/graph$ ./a.out
0 7 0 5 0 0 0
7 0 0 0 7 0 0
0 0 0 0 5 0 0
5 0 0 0 0 6 0
0 7 5 0 0 0 9
0 0 0 6 0 0 0
0 0 0 0 9 0 0
thongbkvn@EliteBook:~/code/graph$

```

Hình 16: Kết quả thu được với mỗi thuật toán