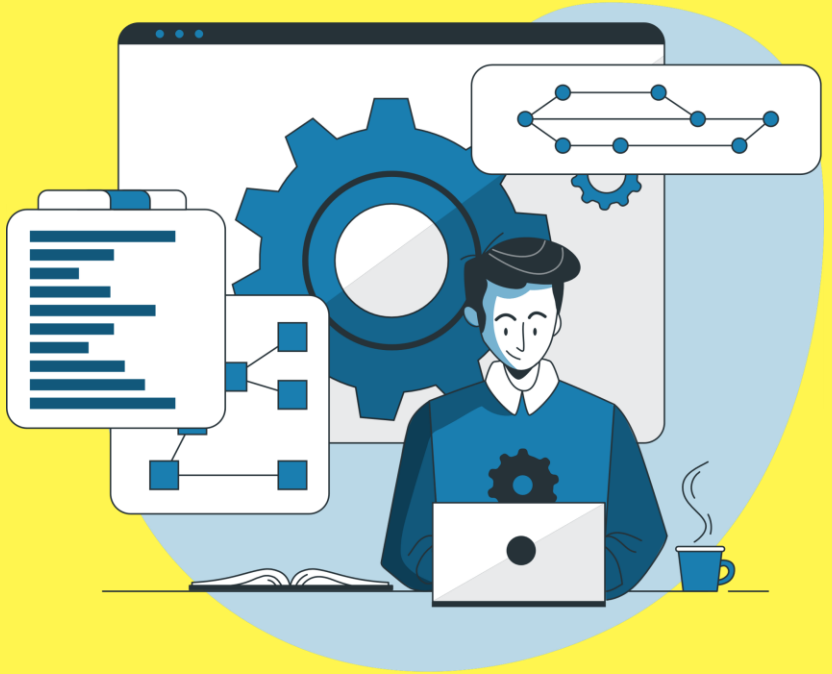
TÌM ĐƯỜNG ĐI NGẮN NHẤT



1. Thuật toán Dijkstra:

Thuật toán Dijkstra được sử dụng để tìm đường đi ngắn nhất từ 1 đỉnh tới mọi đỉnh còn lại trên đồ thị, có thể áp dụng cho cả đồ thị có hướng và vô hướng không chứa trọng số âm. **Độ phức tạp:** O((E + V)logV)

Ý tưởng thuật toán:

Thuật toán duy trì tập S chứa những đỉnh đã xác định được đường đi ngắn nhất từ đỉnh nguồn, thuật toán lần lượt lựa chọn đỉnh u thuộc tập V - S có đường đi tới đỉnh nguồn là ngắn nhất, sau đó đưa đỉnh u vào tập S và cập nhật (relax) đường đi ngắn nhất cho các đỉnh kề với đỉnh u.

Để có thể lựa chọn nhanh đỉnh u có đường đi ngắn nhất tính từ đỉnh nguồn ta sử dụng hàng đợi ưu tiên trong C++.

Mã giả:

```
Dijkstra (G, w, s){
    Initialize - Single - Source(G,s)
    S =  Ø;
    Q = G.V
    while (Q ≠ Ø){
        u = Extract - Min(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
            Relax(u, v, w)
}
```

1. Thuật toán Dijkstra:

Ta sử dụng hàng đợi ưu tiên Q lưu pair với first lưu khoảng cách đường đi ngắn nhất từ đỉnh nguồn, và second lưu đỉnh. Q sẽ sắp xếp các phần tử theo khoảng cách đường đi tăng dần, nếu 2 đỉnh có cùng khoảng cách đường đi thì sắp xếp theo số thứ tự đỉnh tăng dần.

Ban đầu Q chỉ bao gồm cặp {0, s} với s là đỉnh bắt đầu của thuật toán. Thuật toán sẽ được lặp lại khi Q còn chưa rỗng, mỗi lần lặp sẽ lấy ra cặp (d, u) ở đỉnh hàng đợi Q tương ứng với đỉnh u là đỉnh có đường đi ngắn nhất tính từ đỉnh nguồn s. Nếu d > dist[u] với dist[u] là khoảng cách đường đi ngắn nhất ghi nhận được cho đỉnh u thì ta sẽ bỏ qua u.

Khi đỉnh u được xét nó sẽ cố gắng cập nhật khoảng cách ngắn nhất cho các đỉnh v kề với u, sau đó tiếp tục đẩy đỉnh v và dist[v] vào hàng đợi, điều này có thể dẫn tới nhiều phiên bản khác nhau của cùng 1 đỉnh với khoảng cách khác nhau trong hàng đợi. Tuy nhiên ta sẽ luôn chọn phiên bản có khoảng cách ngắn nhất để xét trước.

1. Thuật toán Dijkstra:

Code

```
typedef pair<int, int> ii;
int n, m, s;
vector<ii> adj[1005];

void nhap(){
   cin >> n >> m >> s;
   for(int i = 0; i < m; i++){
   int x, y, w; cin >> x >> y >> w;
   adj[x].push_back({y, w});
   adj[y].push_back({x, w});
   }
}
```

```
void dijkstra(){
   priority_queue<ii, vector<ii>, greater<ii>> q;
   q.push({0, s});
   vector<int> d(n + 1, 1e9); d[s] = 0;
   while(!q.empty()){
      ii t = q.top(); q.pop();
      int dis = t.first, u = t.second;
      if(dis > d[u]) continue;
      for(ii e : adj[u]){
         int v = e.first, w = e.second;
         if(d[v] > d[u] + w){
            d[v] = d[u] + w;
            q.push({d[v], v});
         }
      }
   }
   for(int i = 1; i <= n; i++){
      cout << d[i] << ' ';
   }
}
```

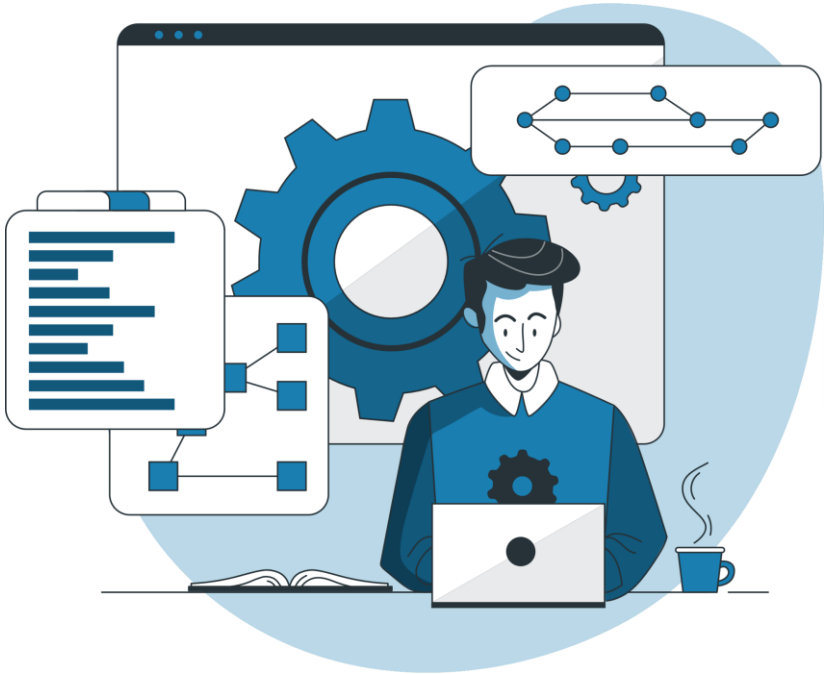
2. Thuật toán Bellman-Ford:

Thuật toán Bellman-Ford được sử dụng đề tìm đường đi ngắn nhất từ 1 đỉnh tới mọi đỉnh còn lại trên đồ thị, khác với Dijkstra thì Bellman-Ford có thể áp dụng cho đồ thị với cạnh có trọng số âm, tuy nhiên không thể áp dụng nếu đồ thị có chu trình âm. Thuật toán này có thể sử dụng để xác định rằng đồ thị có chu trình âm.
Độ phức tạp: O(V * E).

Ý tưởng thuật toán:

Ban đầu ta sử dụng một mảng d[ ] để lưu khoảng cách từ đỉnh nguồn s tới mọi đỉnh còn lại trên đồ thị, d[s] = 0 và d[u] = INF (vô cùng lớn) với mọi đỉnh u còn lại trên đồ thị

Thuật toán lặp n - 1 bước, mỗi bước sẽ xét tất cả các cặp cạnh (u, v) có trọng số w. Nếu d[v] > d[u] + w thì sẽ cập nhật d[v].



2. Thuật toán Bellman-Ford:

Code

```
struct edge{
    int x, y, w;
};
const int INF = (int)1e9;
int n, m;
vector<edge> E;
int d[1005];

void nhap(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        int x, y, w; cin >> x >> y >> w;
        E.push_back((edge){x, y, w});
    }
}
```

```
void BellmanFord(int s){
    fill(d + 1, d + n + 1, INF);
    d[s] = 0;
    for(int i = 1; i <= n - 1; i++){
        for(edge e : E){
            int u = e.x, v = e.y, w = e.w;
            if(d[u] < INF){
                d[v] = min(d[v], d[u] + w);
            }
        }
    }
    for(int i = 1; i <= n; i++){
        cout << d[i] << ' ';
    }
}
```

2. Thuật toán Bellman-Ford:

Kiểm tra chu trình âm trên đồ thị

```
bool check(int s){
   d[s] = 0;
   for(int i = 1; i <= n - 1; i++){
      for(edge e : E){
         if(d[e.x] < INF)
            d[e.y] = min(d[e.y], d[e.x] + e.w);
      }
   }
   bool ok = false;
   for(edge e : E){
      if(d[e.x] < INF){
         if(d[e.y] > d[e.x] + e.w){
            ok = true; break;
         }
      }
   }
   return ok;
}
```

```
bool negativeCycle(){
   fill(d + 1, d + n + 1, INF);
   bool res = false;
   for(int i = 1; i <= n; i++){
      if(d[i] == INF){
         if(check(i)){
            res = true; break;
         }
      }
   }
   return res;
}
```

Chú ý: Trong trường hợp đồ thị có nhiều mảnh liên thông thì bạn phải check tất cả mọi mảnh liên thông.

3. Thuật toán Floyd Warshall:

Khác với thuật toán Dijkstra vs Bellman-Ford áp dụng để tìm đường đi ngắn nhất từ 1 đỉnh tới mọi đỉnh còn lại trên đồ thị (SSSP - Single source shortest path), **Floyd** được áp dụng để tìm đường đi ngắn nhất giữa mọi cặp đỉnh trên đồ thị (All-pair Shortest Path). Thuật toán áp dụng cho đồ thị có hướng, vô hướng không có chu trình âm (có thể có cạnh âm). **Độ phức tạp:** O(V^3)

Code

```
int Mat[1005][1005];
//Mat[i][j] : Khoảng cách đường đi ngắn nhất từ i tới j
int n; //V
void Floyd(){
   for(int k = 1; k <= n; k++){
      for(int i = 1; i <= n; i++){
         for(int j = 1; j <= n; j++){
            Mat[i][j] = min(Mat[i][j], Mat[i][k] +  Mat[k][j]);
         }
      }
   }
}
```

Chú ý: Thông thường Floyd chỉ áp dụng được với đồ thị nhỏ có V <= 400.

4. Note về các thuật toán tìm đường đi ngắn nhất:

Đồ thị/Tính chất	BFS	Dijkstra	Bellman Ford	Floyd Warshall
Độ phức tạp	O(V + E)	O((V + E)logV)	O(V * E)	O(V^3)
Max size	V, E <= 10M	V, E <= 300K	V * E <= 10M	V<=400
Không trọng số	Tốt nhất	Tốt	Tệ	Nhìn chung tệ
Có trọng số	WA	Tốt nhất	Tốt	Nhìn chung tệ
Có trọng số âm	WA	Cần biến đổi	Tốt	Nhìn chung tệ
Có chu trình âm	Không thể xác định	Không thể xác định	Có thể xác định	Có thể xác định
Đồ thị nhỏ	WA nếu có trọng số	Đúng nhưng không phù hợp	Đúng nhưng không phù hợp	Tốt nhất