

Graph (III)

Luo Tsz Fung {pepper1208}

2025-07-17

Graph (I) & (II) review

- Normally, we use **vector** to store a graph.
- A graph can be either undirected / directed, and weighted / unweighted.
- Tree is a special type of a graph.
- To traverse a graph, we can use DFS or BFS.
 - DFS: traverse to the deepest node
 - BFS: traverse by layer
- Flood fill can be used to detect connected component(s).

Graph (III)

1. Shortest Path Algorithm
 - Bellman-ford Algorithm
 - SPFA (Shortest Path Faster Algorithm)
 - Dijkstra's Algorithm
 - Floyd-Warshall Algorithm
2. Minimum Spanning Tree Algorithm
 - Prim's Algorithm
 - Kruskal's Algorithm

Terminologies

Walk

A walk is a sequence that connect a series of vertices. The length of a walk can be finite or infinite.

Formally, a walk w is the sequence of edges e_1, e_2, \dots, e_k , such that a sequence of vertices $v_0, v_1, v_2, \dots, v_k$ satisfies $e_i = (v_{i-1}, v_i)$, where $i \in [1, k]$. The walk can be also written as $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$.

k is the **length** of the walk. If the graph is a weighted graph, the length of the walk will be the sum of the weights of all edges in the walk.

Terminologies

Trail

For a walk w , if e_1, e_2, \dots, e_k are distinct edges, then w is a trail.

Path (or simple path)

For a walk w , if $v_0, v_1, v_2, \dots, v_k$ are distinct vertices, then w is a path.

Terminologies

Circuit

For a walk w , if $v_0 = v_k$, then w is a circuit.

Cycle (or simple circuit)

For a walk w , if $v_0 = v_k$ and (v_0, v_k) is the only pair of repeated vertices, then w is a circuit.

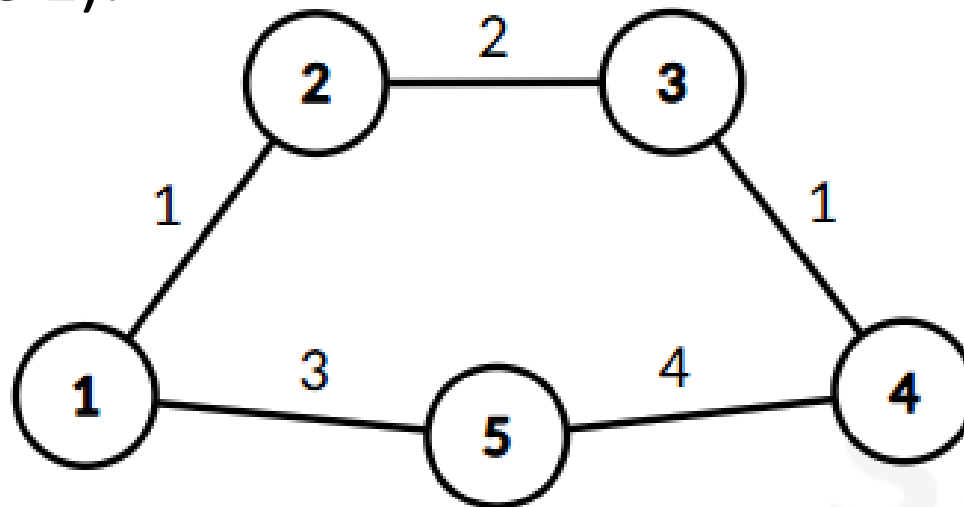
Terminologies

However, the definition of all terminologies can be ambiguous and you should refer to the definition given by the question.

BFS

We can use BFS to find the shortest path on unweighted graphs. Could we use BFS to find the shortest path on weighted graphs? Could you provide a counterexample?

Example (from node 1):



Notations

To be convenient, some notations are defined in this slide:

- n : the number of vertices on the graph
- m : the number of edges on the graph
- s : the source of the shortest path
- $D(u)$: the **actual** shortest path from s to u
- $dis(u)$: the **estimate** shortest path from s to u . After running the shortest path algorithm, $dis(u) = D(u)$ for every u .
- $w(u,v)$: the weight of the edge (u,v)

Bellman-Ford Algorithm

Bellman-Ford algorithm is based on the **relax operation**.

Relax operation: for edge (u, v) , the relax operation is the equation below:

$$dis(v) = \min(dis(v), dis(u) + w(u, v))$$


For each relaxation, we try to find the shortest path from s to v by considering the path $s \rightarrow u \rightarrow v$.

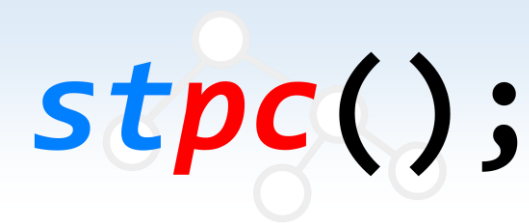
Bellman-Ford Algorithm

We aim to find the single source shortest path for all other vertices.

For each iteration, we try to relax **every edge** on the graph. If there is no valid iteration after a single iteration, the algorithm stops. Trivially, the time complexity for each iteration is $O(m)$.

At most how many iterations will be made? Consider the number of iterations Bellman-Ford Algorithm will be made if the graph is a chain. As shortest paths are all chain-like structures, there will be at most $n - 1$ iterations will be made. The time complexity of Bellman-Ford Algorithm is $O(nm)$.



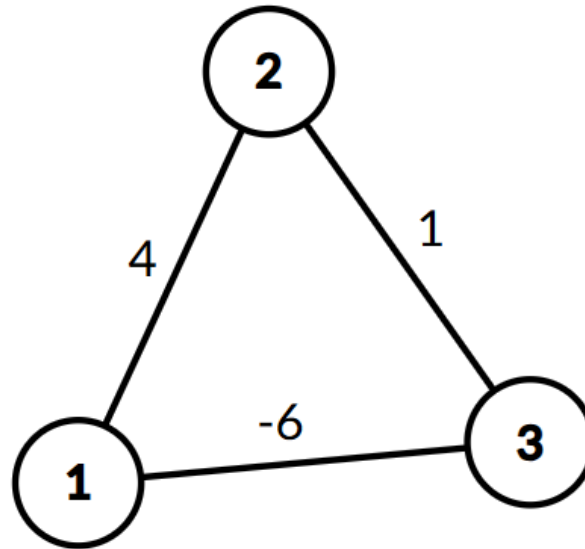
The logo for 'stpc()' features the text 'stpc()' in a stylized font, with 'st' in blue, 'pc' in red, and '()' in black. To the right of the text is a small network diagram consisting of three nodes (circles) connected by lines, with one node at the top and two below it.

```
struct node { int cost, to; };
vector<node> adj[MAXN];
long long dist[MAXN];
int prevnode[MAXN];
void bellman_ford(int s)
{
    for (int i = 1; i <= n; ++i) dist[i] = INT_MAX;
    dist[s] = 0;

    for (int i = 1; i <= n-1; ++i) // traverse n-1 times
    {
        for (int u = 1; u <= n; ++u)
        {
            for (auto [w, v] : adj[u]) // for each edge
            {
                if (dist[v] > dist[u] + w)
                {
                    dist[v] = dist[u] + w;
                    prevnode[v] = u;
                }
            }
        }
    }
}
```

Bellman-Ford Algorithm

Is it possible for the algorithm has n iterations or more? Could you provide an example?



Bellman-Ford Algorithm

Bellman-Ford Algorithm could be used to detect **negative cycle** from a single source, as the algorithm will have infinite iterations when negative cycle(s) exist.

Negative Cycles

Think: how to determine if there exists a negative cycle? (without the constraint that a single source is included in the negative cycle)

SPFA (Shortest Path Faster Algorithm)

SPFA is an “optimized” version of Bellman-Ford algorithm. As there exists ample amount of useless relaxation check, with the observation that: **the vertex that has been relaxed may bring another relaxation.**

Therefore, it uses a queue to reduce the number of relaxation check. However, the worse case complexity of SPFA is still $O(nm)$, which is the same as the original Bellman-Ford algorithm. Details of SPFA will not be discussed here as Bellman-Ford algorithm is sufficient.

关于SPFA

• 它死了


stpc();

```
queue<int> Q;
bool inq[V];
int dist[V];
int parent[V];

for (int i = 1; i <= n; ++i)
{
    inq[i] = false;
    dist[i] = INF;
    parent[i] = i;
}

dist[source] = 0;
Q.push(source); inq[source] = true;
while (!Q.empty())
{
    int curr = Q.front();
    Q.pop(); inq[curr] = false;
    for (int i = 0; i < int(adj[curr].size()); ++i)
    {
        edge e = adj[curr][i];
        if (dist[e.to] > dist[curr] + e.cost)
        {
            dist[e.to] = dist[curr] + e.cost;
            parent[e.to] = curr;
            if (inq[e.to] == false)
            {
                Q.push(e.to); inq[e.to] = true;
            }
        }
    }
}
```

Dijkstra's Algorithm

Dijkstra's Algorithm finds the single source shortest path on **non-negative weighted graphs**. The time complexity of it is better than Bellman-Ford algorithm on this type of graph.

Dijkstra's Algorithm

Dijkstra's Algorithm finds the single source shortest path on **non-negative weighted graphs**. The time complexity of it is better than Bellman-Ford algorithm on this type of graph.

Note: See Appendix I for the proof of Dijkstra's Algorithm

Dijkstra's Algorithm

Divide all vertices into two sets: S and T , where the shortest path from s to i is determined if vertex i is in S , and vertex i is in T otherwise.

Initialize $dis(s) = 0$ and $dis(i) = \infty$ for all other vertices.

Repeat the followings:

1. Select a vertex v in T such that the shortest path from s to v is the shortest among all vertices in T .
2. Relax all edges that has been added in S .

until T is empty.

Naïve time complexity: $O(n^2 + m)$

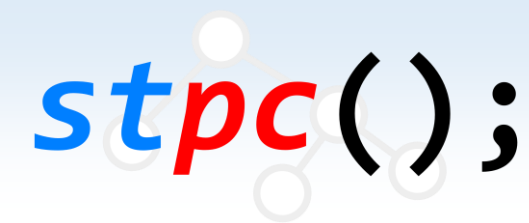
Dijkstra's Algorithm

We can use a heap to maintain the Dijkstra's Algorithm.

For each relaxation, we can insert the vertex into the heap. When the heap is not empty, check whether the top vertex of the heap has been relaxed and skip the required vertex.

The time complexity is $O(m \log n)$.

Actually, there are various versions of Dijkstra's Algorithm with different time complexities. For the current stage, it is sufficient for you to learn a single maintenance method of Dijkstra's Algorithm using heap.



stpc();

```
struct node { int cost, to; };
vector<node> adj[MAXN];
long long dist[MAXN];
int prevnode[MAXN];
void dijkstra(int s)
{
    for (int i = 1; i <= n; ++i) dist[i] = INT_MAX;
    priority_queue<pair<int, int>, vector<pair<int, int>>,
                    greater<pair<int, int>>> PQ;
    PQ.push({0, s});
    dist[s] = 0;
    while (!PQ.empty())
    {
        int u = PQ.top().second;
        int w = PQ.top().first;
        PQ.pop();
        if (dist[u] != w) continue;
        for (node e : adj[u])
        {
            if (dist[e.to] > e.cost + dist[u])
            {
                dist[e.to] = e.cost + dist[u];
                prevnode[e.to] = u;
                PQ.push({dist[e.to], e.to});
            }
        }
    }
}
```

Dijkstra's Algorithm

Single Source Shortest Path

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm is used to find the shortest path between any two vertices on a graph with no negative cycles.

Define the state $dis[k][x][y]$ to be the shortest path from vertex x to vertex y in the subgraph $V' = \{1, 2, \dots, k\}$. Note that x and y are not required to be in the subgraph.

Initially, $dis[0][x][x] = 0$, $dis[0][x][y] = w(x, y)$, $dis[0][x][y] = \infty$ if x and y are not connected by an edge.

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm is used to find the shortest path between any two vertices on a graph with no negative cycles.

Define the state $dis[k][x][y]$ to be the shortest path from vertex x to vertex y in the subgraph $V' = \{1, 2, \dots, k\}$. Note that x and y are not required to be in the subgraph.

Initially, $dis[0][x][x] = 0$, $dis[0][x][y] = w(x, y)$, $dis[0][x][y] = \infty$ if x and y are not connected by an edge.

Obviously, our goal is to find the value stored in $dis[n][x][y]$.

Floyd-Warshall Algorithm

The transitional formula is stated below:

$$dis[k][x][y] = \min(dis[k-1][x][y], dis[k-1][x][k] + dis[k-1][k][y])$$

where $dis[k-1][x][y]$ means the shortest path from x to y not passing through k and $dis[k-1][x][k] + dis[k-1][k][y]$ means the shortest path from x to y passing through k .

Time complexity: $O(n^3)$

[Floyd-Warshall Algorithm](#)

```
for (int i = 1; i <= n; ++i)
{
    for (int j = 1; j <= n; ++j)
    {
        if (i == j) dist[i][j] = 0;
        else if (adj[i][j] != 0) dist[i][j] = adj[i][j];
        else dist[i][j] = INF;
    }
}

for (int k = 1; k <= n; ++k)
{
    for (int i = 1; i <= n; ++i)
    {
        for (int j = 1; j <= n; ++j)
        {
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
        }
    }
}
```

stpc();

Extensive Reading

- Cycle with minimum length
- Transitive closure
- Johnson's algorithm
- 0-1 BFS
- Multi-layer shortest path algorithm with graph modelling
- System of Difference Constraints
- k -shortest path
- Shortest path on modular graph

Practice Problems

- [\[JLOI 2011\] 飛行路線](#)
- [最短路計數](#)
- [\[NOIP-S 2009\] 最優貿易](#)
- [災後重建](#)
- [\[NOIP-J 2002\] 產生數](#)
- [\[NOI 2007\] 社交網路](#)

Minimum Spanning Tree

Spanning Tree

A subgraph with $n - 1$ edges and all vertices in the original graph, and the subgraph is a single connected component.

Minimum Spanning Tree (MST)

A spanning tree with the sum of weights of all edges is the minimum.

Minimum Spanning Tree

Kruskal's Algorithm

- 1 **Input.** The edges of the graph e , where each element in e is (u, v, w) denoting that there is an edge between u and v weighted w .
- 2 **Output.** The edges of the MST of the input graph.
- 3 **Method.**
- 4 $result \leftarrow \emptyset$
- 5 sort e into nondecreasing order by weight w
- 6 **for** each (u, v, w) in the sorted e
- 7 **if** u and v are not connected in the union-find set
- 8 connect u and v in the union-find set
- 9 $result \leftarrow result \cup \{(u, v, w)\}$
- 10 **return** $result$

Kruskal's Algorithm

Obviously, DSU (disjoint set union find) is used to maintain the Kruskal's Algorithm.

Time complexity: $O(m \log m)$

Note: See Appendix II for the proof of Kruskal's Algorithm

Prim's Algorithm

It shares similar idea with Dijkstra's Algorithm.

Note that it only outputs the sum of weights of the MST.

```
1  Input. The nodes of the graph  $V$  ; the function  $g(u, v)$  which  
    means the weight of the edge  $(u, v)$ ; the function  $adj(v)$  which  
    means the nodes adjacent to  $v$ .  
2  Output. The sum of weights of the MST of the input graph.  
3  Method.  
4   $result \leftarrow 0$   
5  choose an arbitrary node in  $V$  to be the root  
6   $dis(root) \leftarrow 0$   
7  for each node  $v \in (V - \{root\})$   
8       $dis(v) \leftarrow \infty$   
9   $rest \leftarrow V$   
10 while  $rest \neq \emptyset$   
11      $cur \leftarrow$  the node with the minimum  $dis$  in  $rest$   
12      $result \leftarrow result + dis(cur)$   
13      $rest \leftarrow rest - \{cur\}$   
14     for each node  $v \in adj(cur)$   
15          $dis(v) \leftarrow \min(dis(v), g(cur, v))$   
16 return  $result$ 
```

Prim's Algorithm

Using heap to maintain Prim's algorithm,
Time complexity: $O((n + m) \log m)$

Note: See Appendix III for the proof of Prim's Algorithm

Extensive Reading

- Boruvka's Algorithm
- Uniqueness of MST
- Second minimum spanning tree
- Kruskal reconstruction tree

Practice Problems

- [買禮物](#)
- [口袋的天空](#)
- [逐個擊破](#)
- [\[APIO 2008\] 免費道路](#)

Q&A
