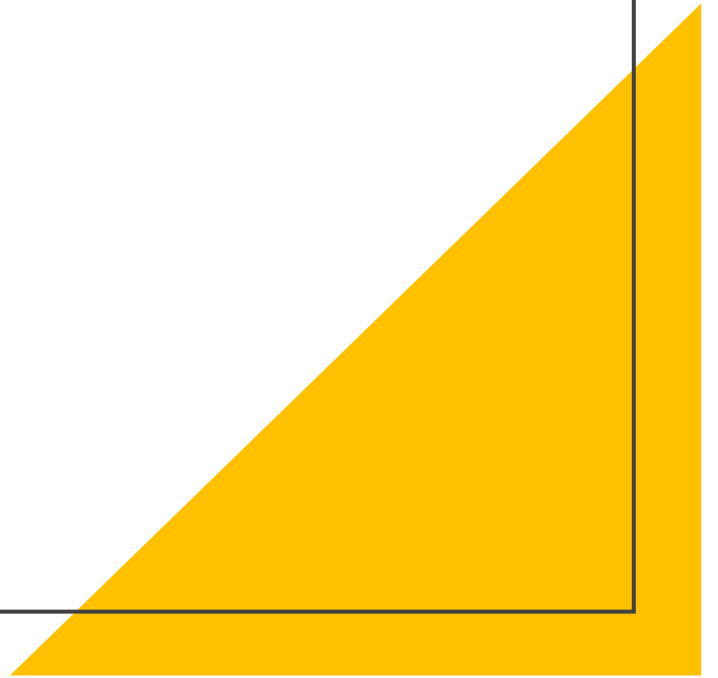
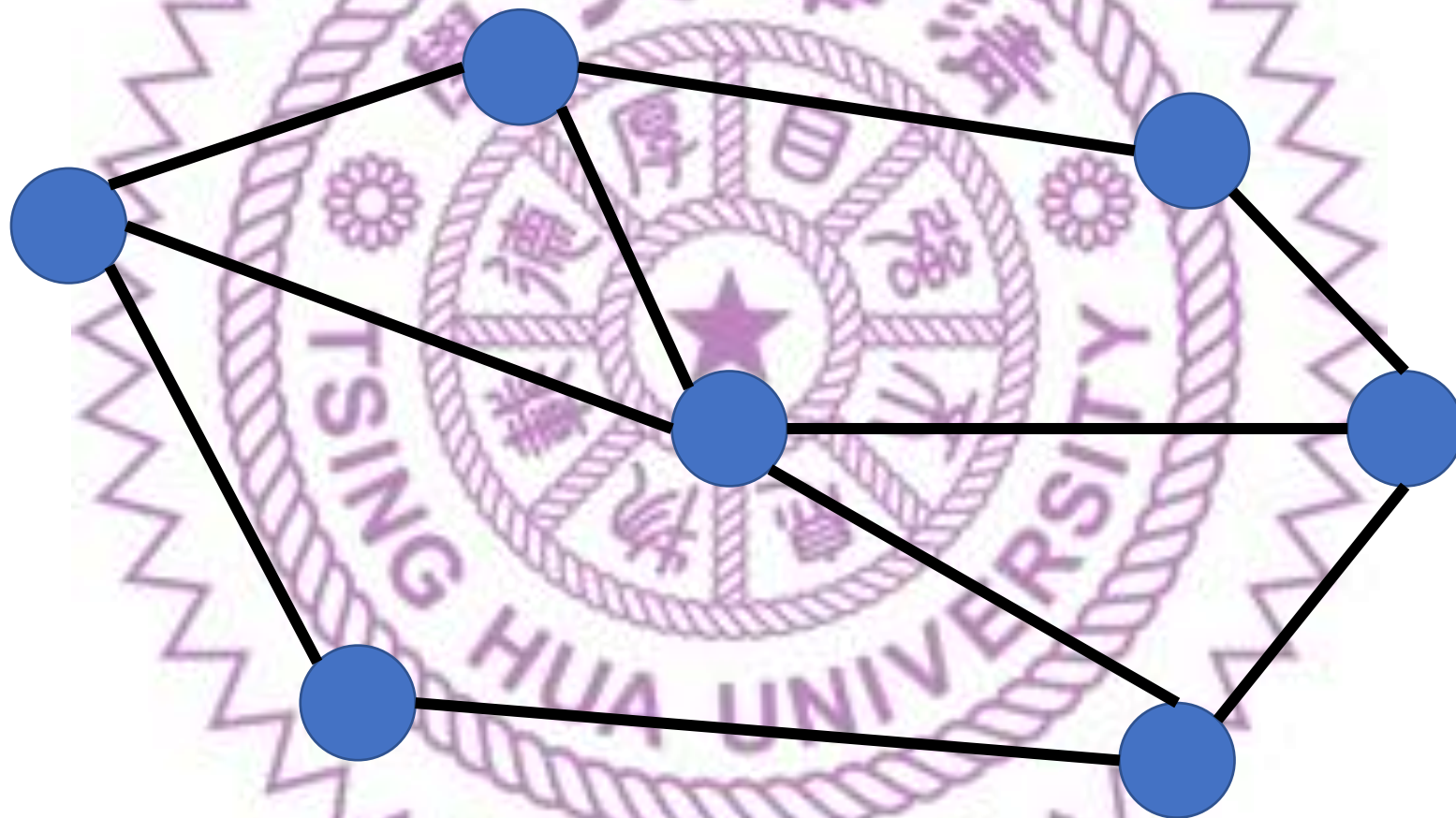


Weighted Graph

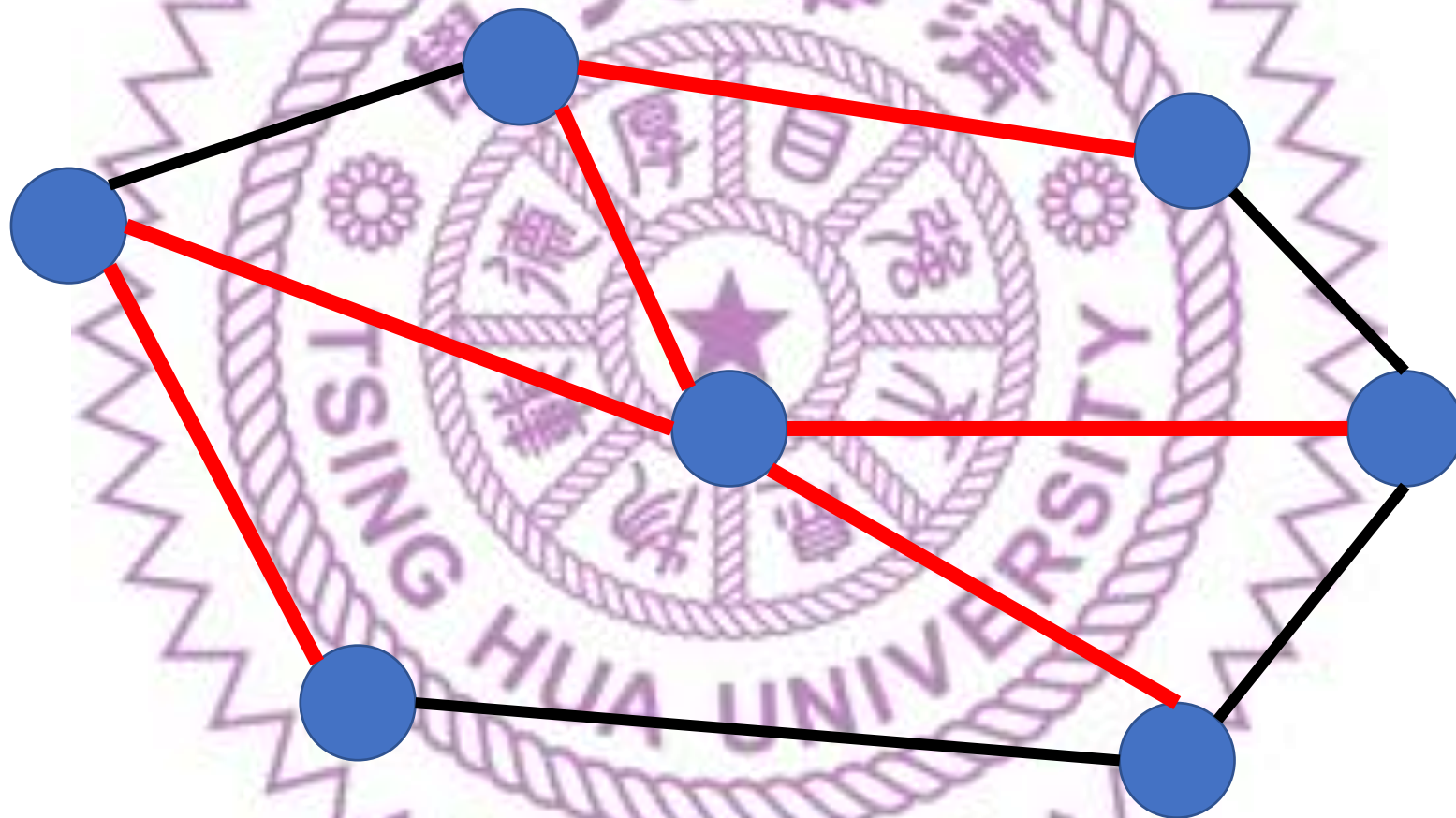
日月卦長



生成樹



生成樹

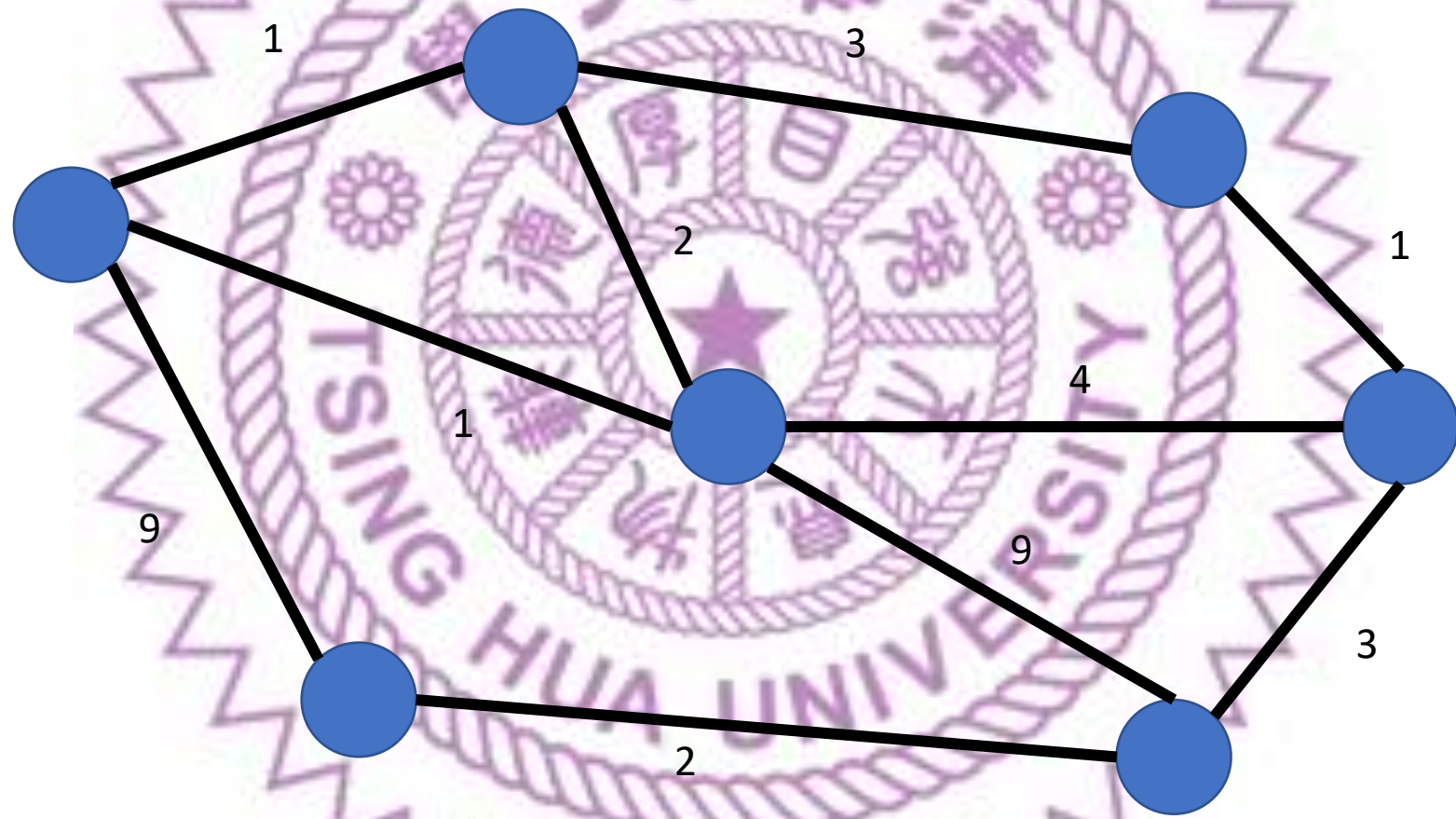


生成樹 Spanning Tree

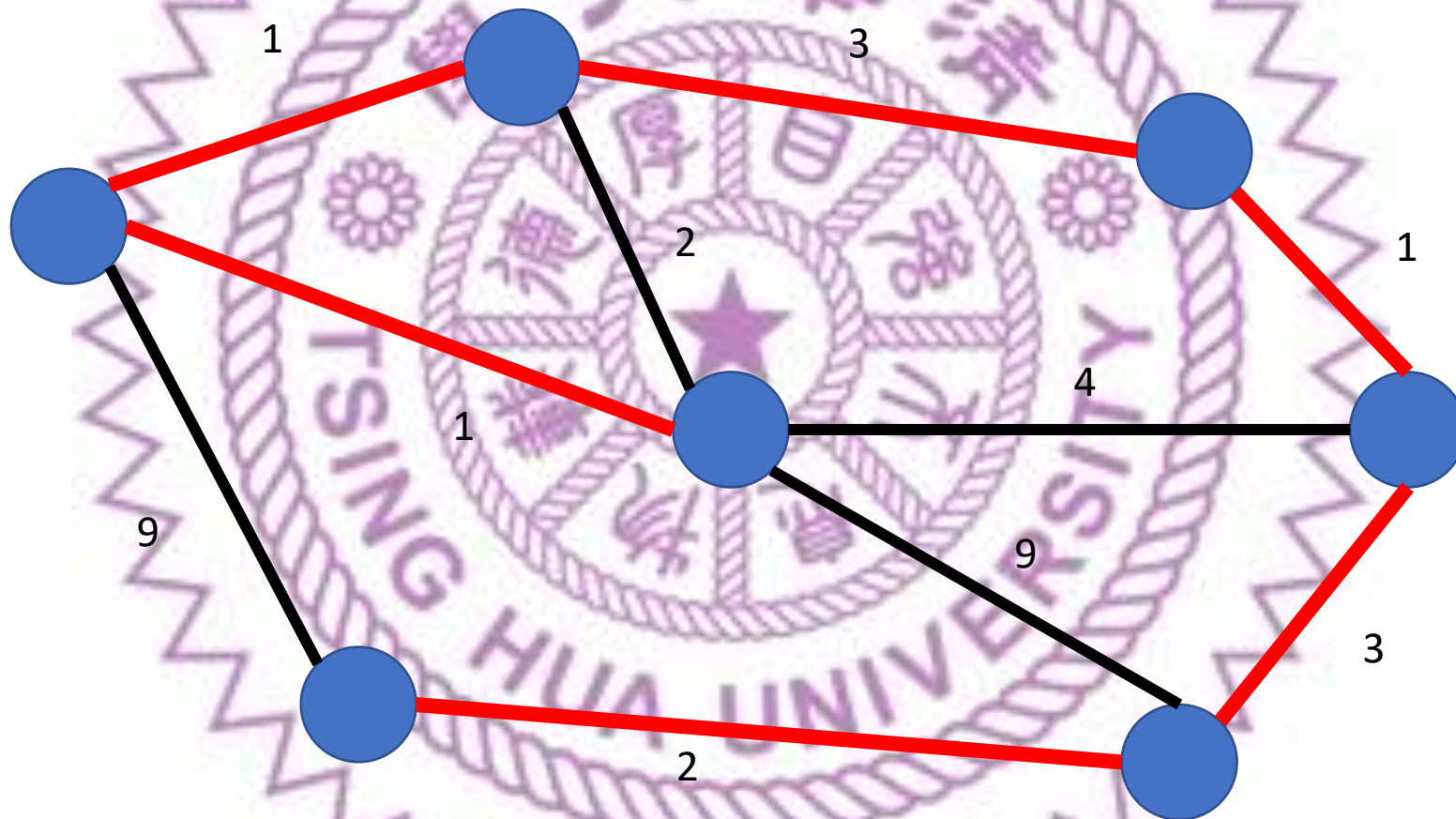
- 無向圖 G 的生成樹 (Spanning Tree) 是具有 G 的全部頂點，但邊數最少的連通子圖。



最小生成樹



最小生成樹



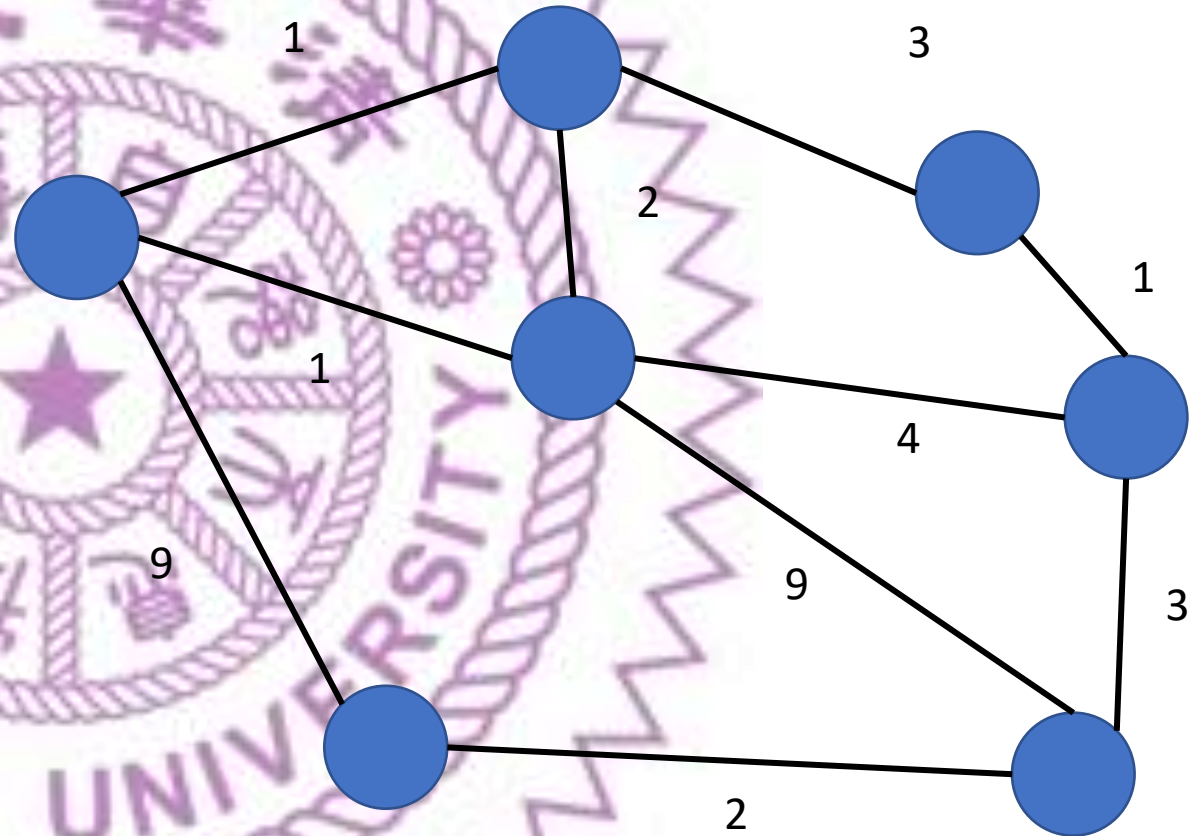
最小生成樹 Minimum Spanning Tree

- 邊權重總合最小的生成樹



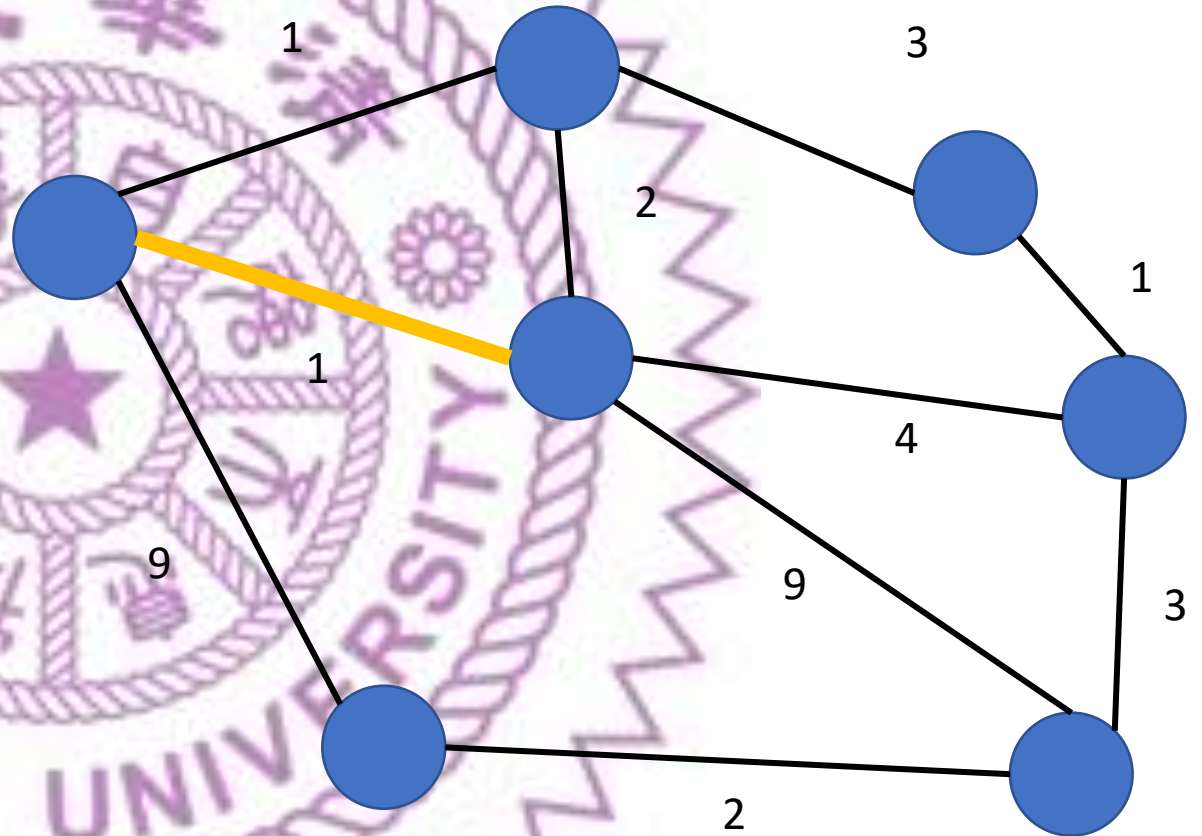
Kruskal's algorithm

- Input $G = (V, E)$
- 設 $Ans = \{ \}$
- While $E \neq \emptyset$:
 - 從 E 中拿出權重最小的邊 e
 - $E \leftarrow E - \{e\}$
 - If Ans 在加入 e 後沒有產生 cycle:
 - $Ans \leftarrow Ans \cup \{e\}$
- 如果 $\|Ans\| = \|V\| - 1$:
 - Ans 就是最小生成樹



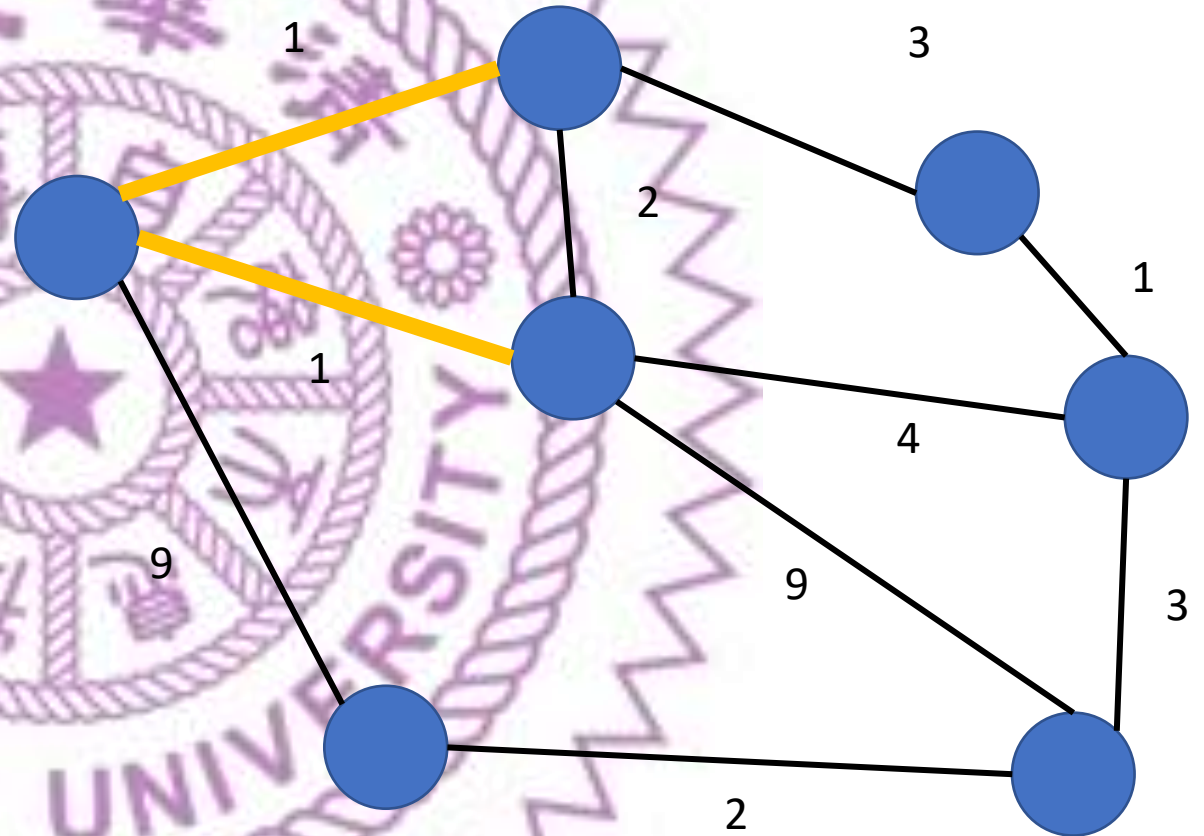
Kruskal's algorithm

- Input $G = (V, E)$
- 設 $Ans = \{ \}$
- While $E \neq \emptyset$:
 - 從 E 中拿出權重最小的邊 e
 - $E \leftarrow E - \{e\}$
 - If Ans 在加入 e 後沒有產生 cycle:
 - $Ans \leftarrow Ans \cup \{e\}$
- 如果 $\|Ans\| = \|V\| - 1$:
 - Ans 就是最小生成樹



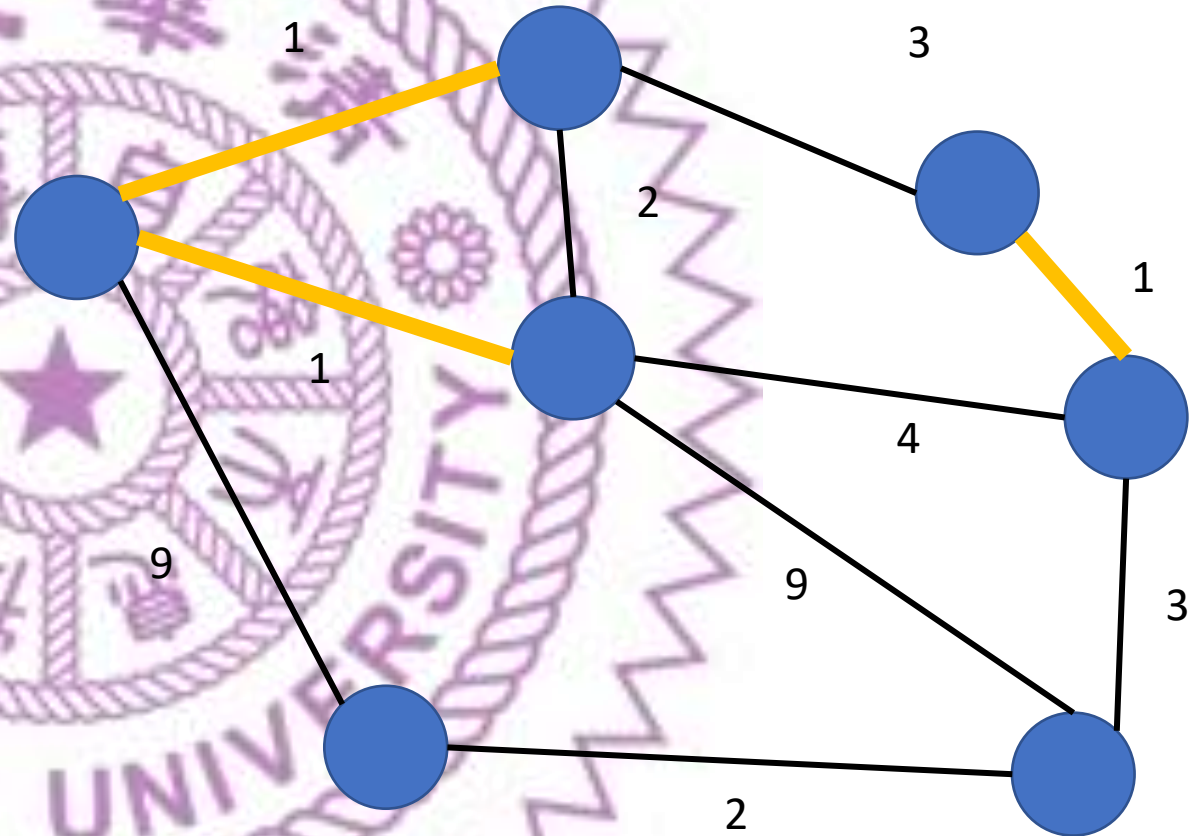
Kruskal's algorithm

- Input $G = (V, E)$
- 設 $Ans = \{ \}$
- While $E \neq \emptyset$:
 - 從 E 中拿出權重最小的邊 e
 - $E \leftarrow E - \{e\}$
 - If Ans 在加入 e 後沒有產生 cycle:
 - $Ans \leftarrow Ans \cup \{e\}$
- 如果 $\|Ans\| = \|V\| - 1$:
 - Ans 就是最小生成樹



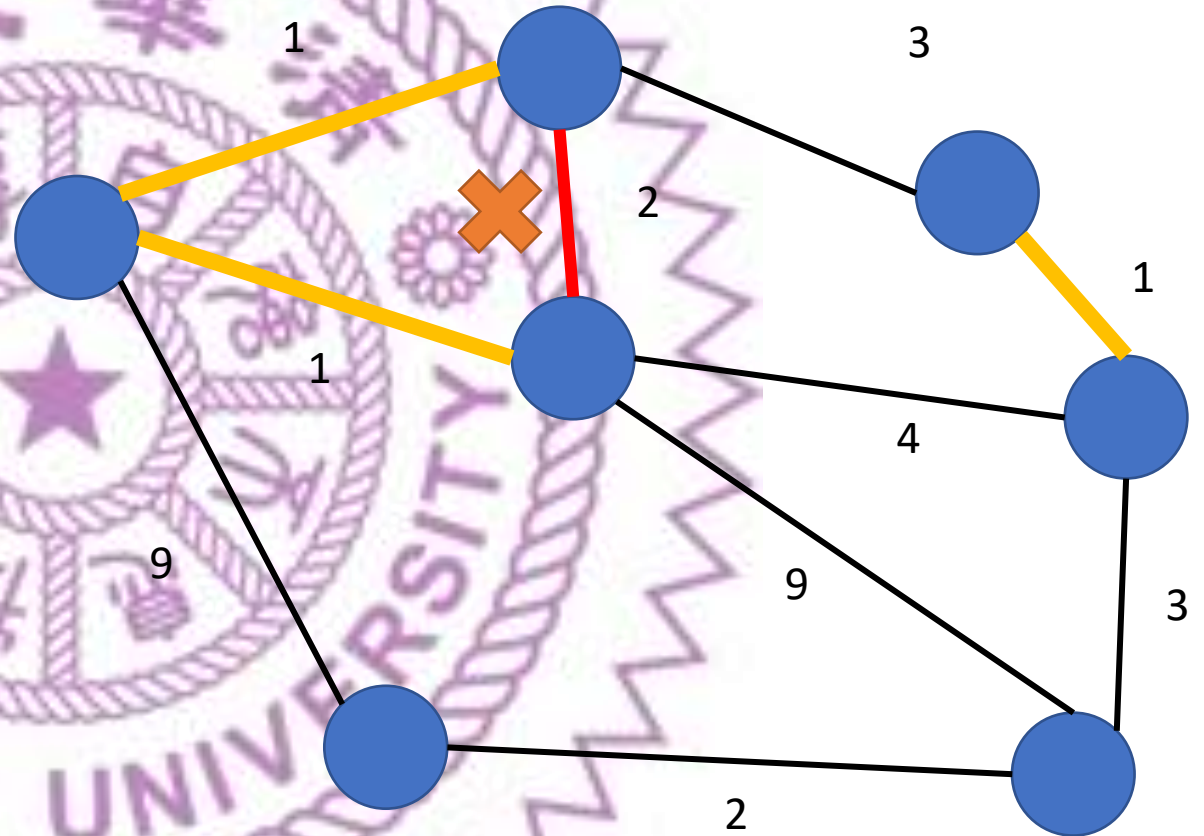
Kruskal's algorithm

- Input $G = (V, E)$
- 設 $Ans = \{ \}$
- While $E \neq \emptyset$:
 - 從 E 中拿出權重最小的邊 e
 - $E \leftarrow E - \{e\}$
 - If Ans 在加入 e 後沒有產生 cycle:
 - $Ans \leftarrow Ans \cup \{e\}$
- 如果 $\|Ans\| = \|V\| - 1$:
 - Ans 就是最小生成樹



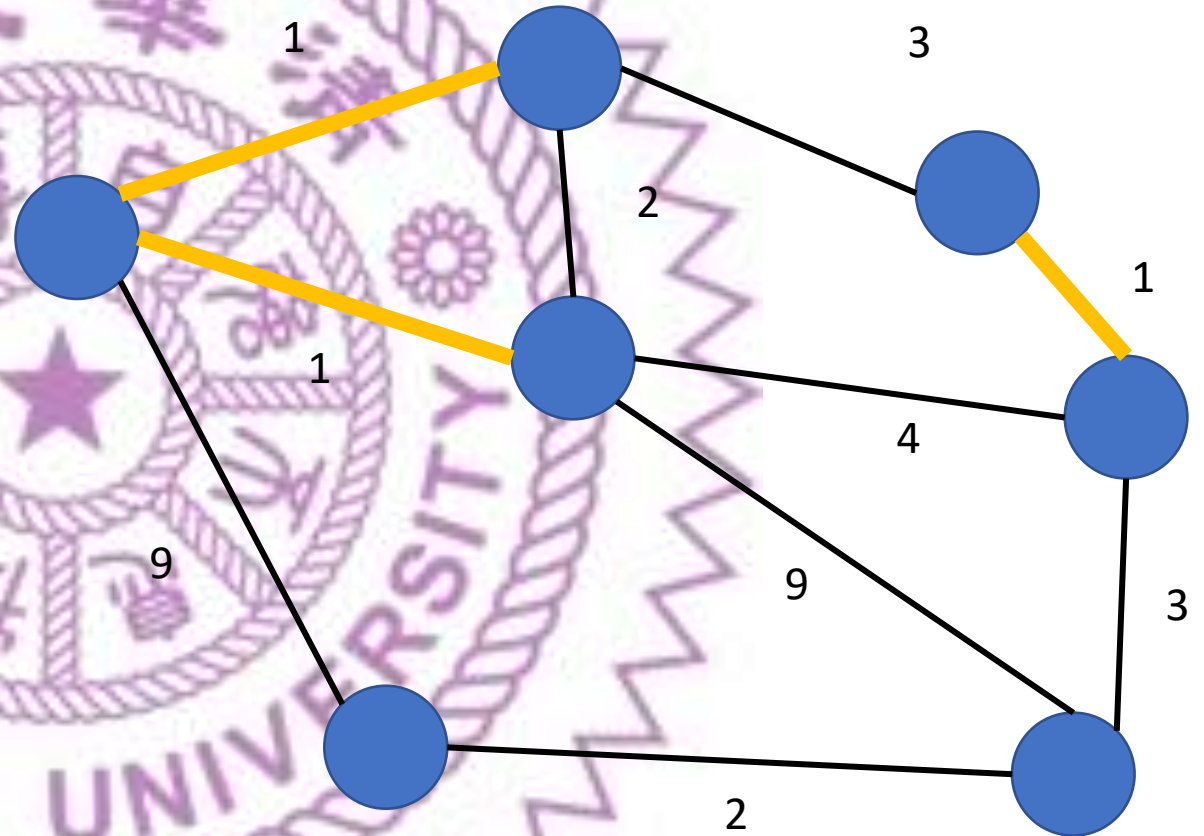
Kruskal's algorithm

- Input $G = (V, E)$
- 設 $Ans = \{ \}$
- While $E \neq \emptyset$:
 - 從 E 中拿出權重最小的邊 e
 - $E \leftarrow E - \{e\}$
 - If Ans 在加入 e 後沒有產生 cycle:
 - $Ans \leftarrow Ans \cup \{e\}$
- 如果 $\|Ans\| = \|V\| - 1$:
 - Ans 就是最小生成樹



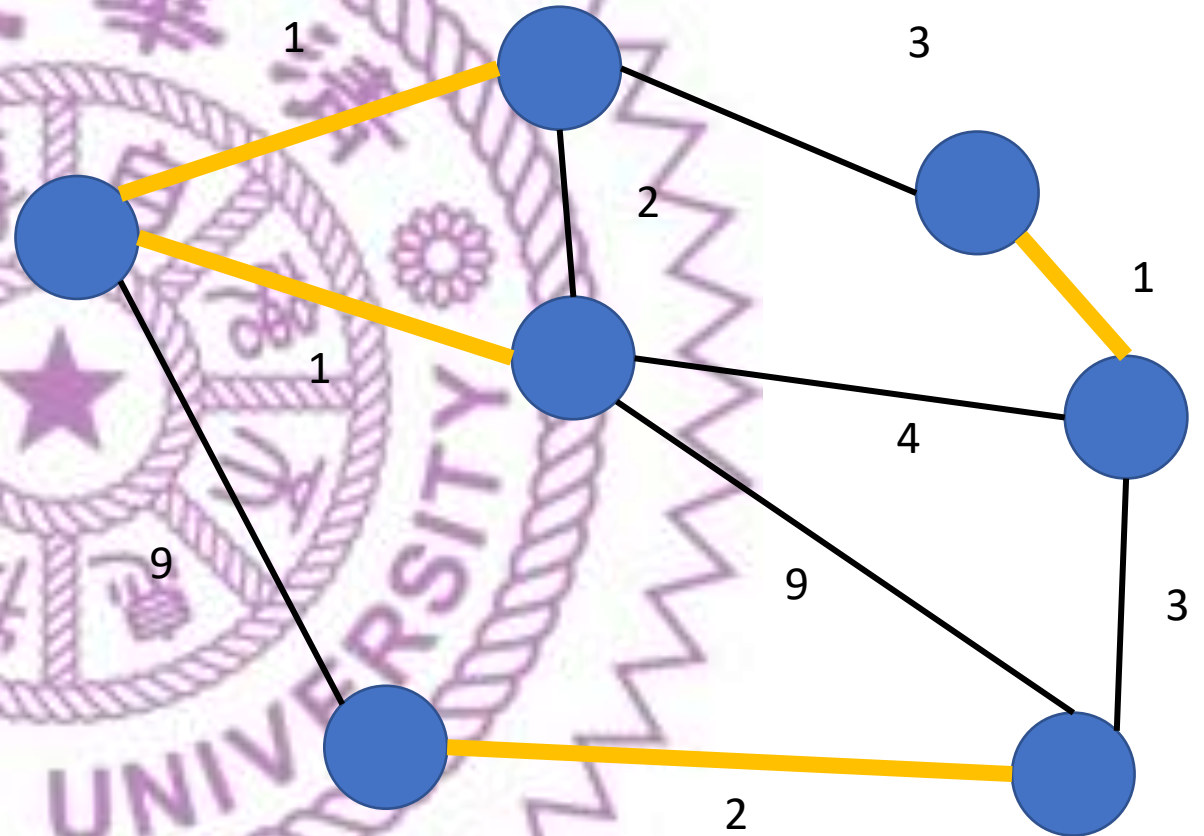
Kruskal's algorithm

- Input $G = (V, E)$
- 設 $Ans = \{ \}$
- While $E \neq \emptyset$:
 - 從 E 中拿出權重最小的邊 e
 - $E \leftarrow E - \{e\}$
 - If Ans 在加入 e 後沒有產生 cycle:
 - $Ans \leftarrow Ans \cup \{e\}$
- 如果 $\|Ans\| = \|V\| - 1$:
 - Ans 就是最小生成樹



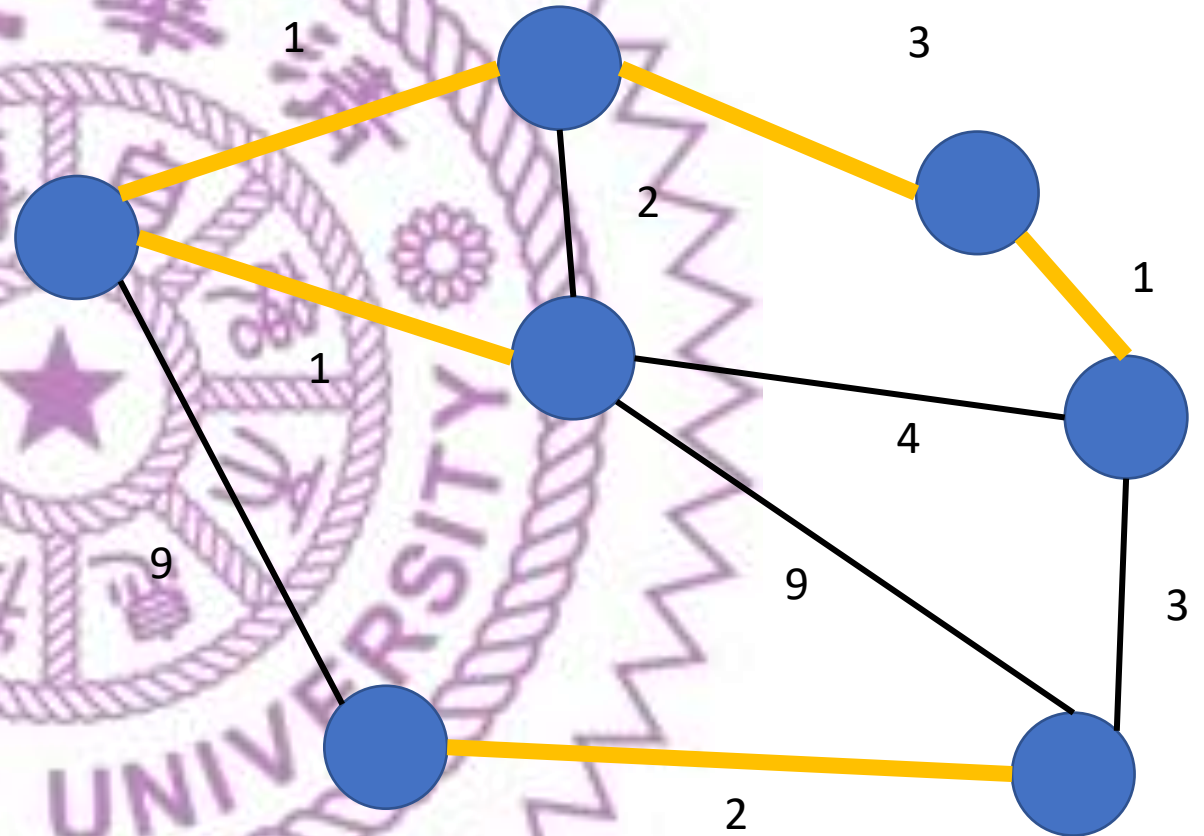
Kruskal's algorithm

- Input $G = (V, E)$
- 設 $Ans = \{ \}$
- While $E \neq \emptyset$:
 - 從 E 中拿出權重最小的邊 e
 - $E \leftarrow E - \{e\}$
 - If Ans 在加入 e 後沒有產生 cycle:
 - $Ans \leftarrow Ans \cup \{e\}$
- 如果 $\|Ans\| = \|V\| - 1$:
 - Ans 就是最小生成樹



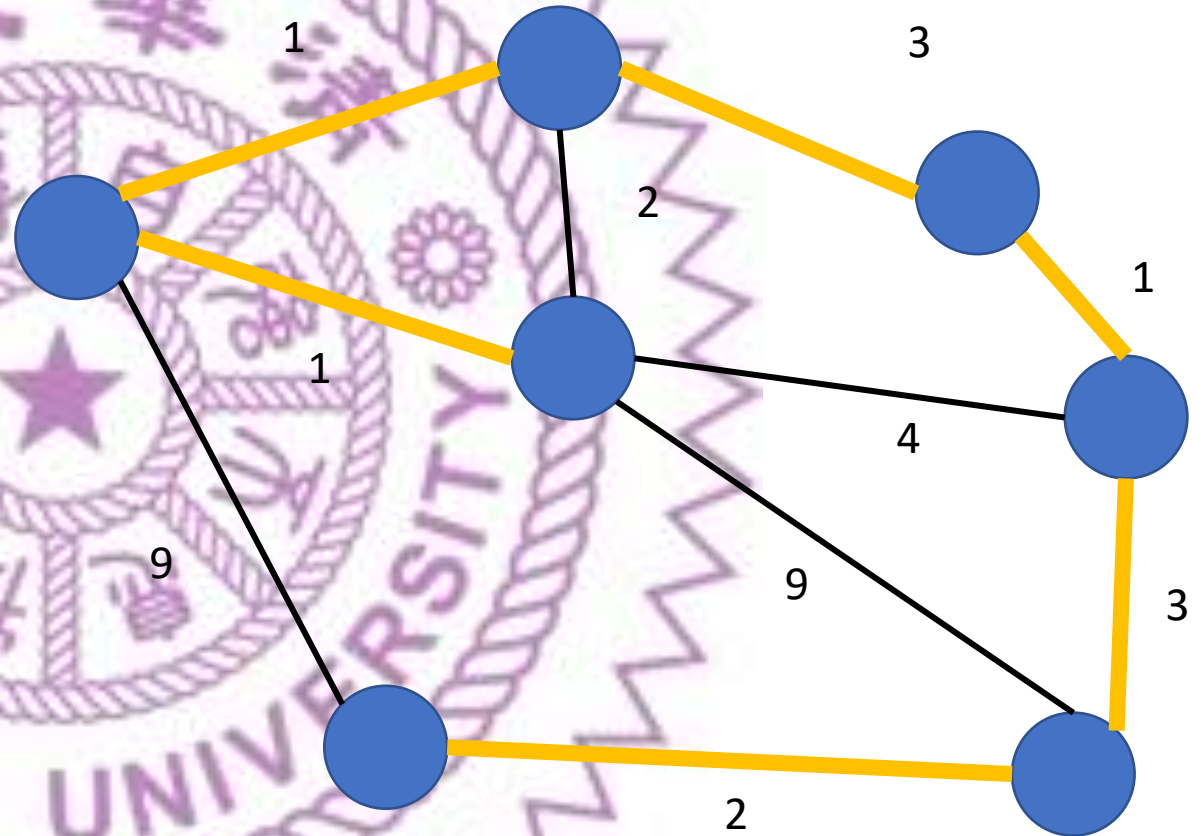
Kruskal's algorithm

- Input $G = (V, E)$
- 設 $Ans = \{ \}$
- While $E \neq \emptyset$:
 - 從 E 中拿出權重最小的邊 e
 - $E \leftarrow E - \{e\}$
 - If Ans 在加入 e 後沒有產生 cycle:
 - $Ans \leftarrow Ans \cup \{e\}$
- 如果 $\|Ans\| = \|V\| - 1$:
 - Ans 就是最小生成樹



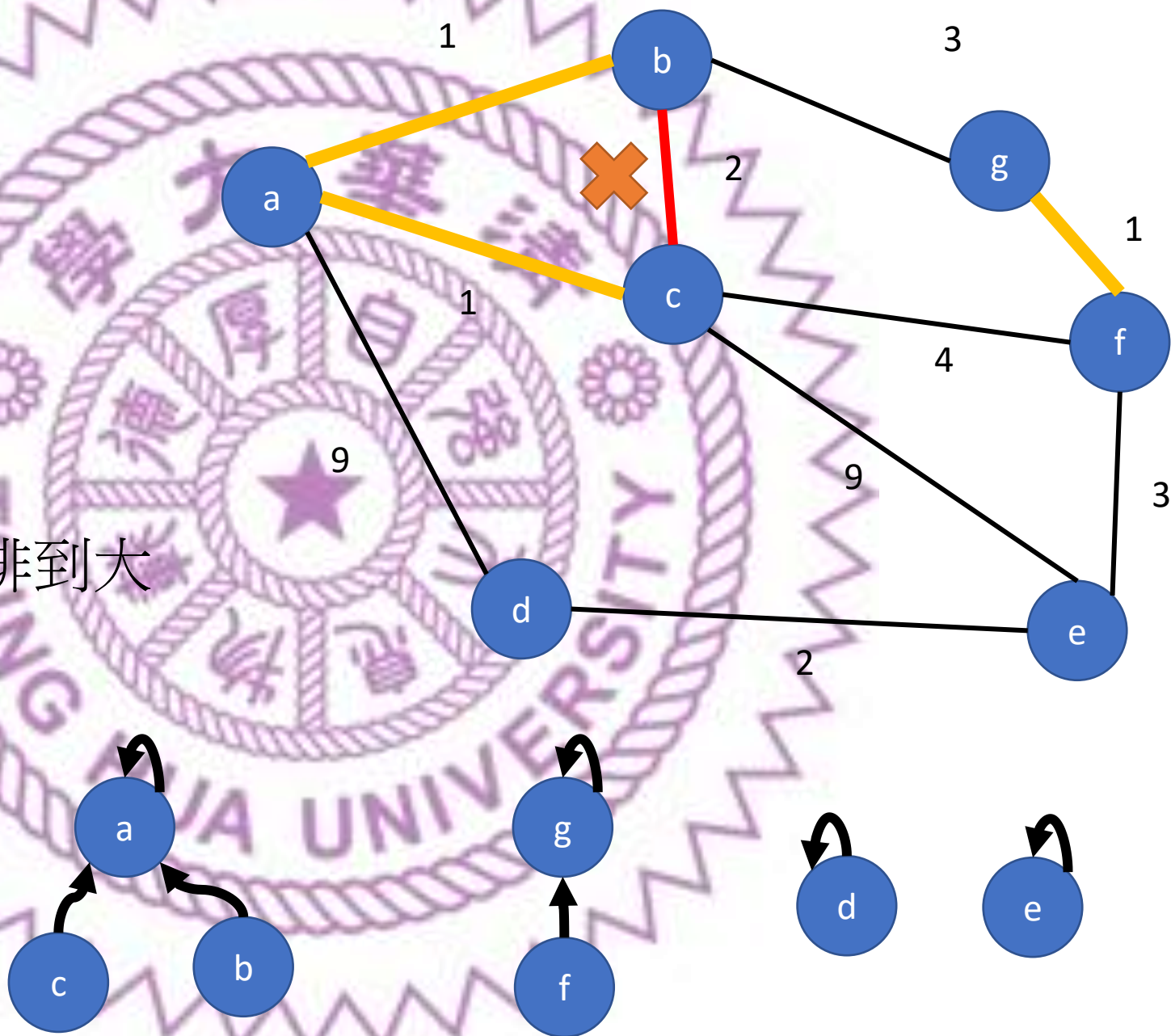
Kruskal's algorithm

- Input $G = (V, E)$
- 設 $Ans = \{ \}$
- While $E \neq \emptyset$:
 - 從 E 中拿出權重最小的邊 e
 - $E \leftarrow E - \{e\}$
 - If Ans 在加入 e 後沒有產生 cycle:
 - $Ans \leftarrow Ans \cup \{e\}$
- 如果 $\|Ans\| = \|V\| - 1$:
 - Ans 就是最小生成樹



實作細節

- 如何判斷環?
➔ Disjoint Set
- 集合處理很慢
➔ 直接將edge由小排到大



最小生成樹只需要 Edge 的資訊

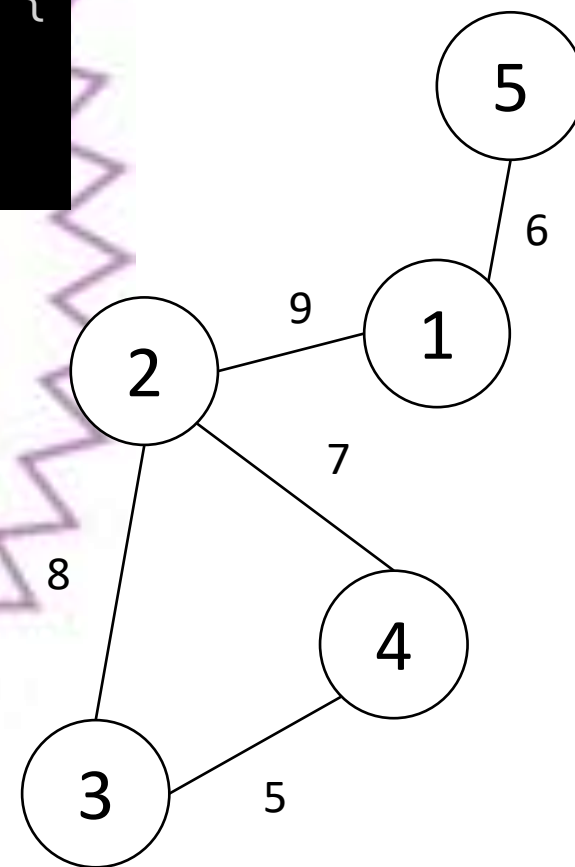
```
struct Edge {  
    int u, v, cost;  
    bool operator<(const Edge &other) const {  
        return cost < other.cost;  
    }  
};
```

n 個點, m 條邊

m 條邊

5	5	
1	2	9
2	3	8
2	4	7
1	5	6
3	4	5

```
vector<Edge> E;  
int n, m;  
cin >> n >> m;  
E.resize(m);  
for (int i = 0; i < m; ++i) {  
    cin >> E[i].u >> E[i].v >> E[i].cost;  
}
```



實作很簡單

```
vector<Edge> E;

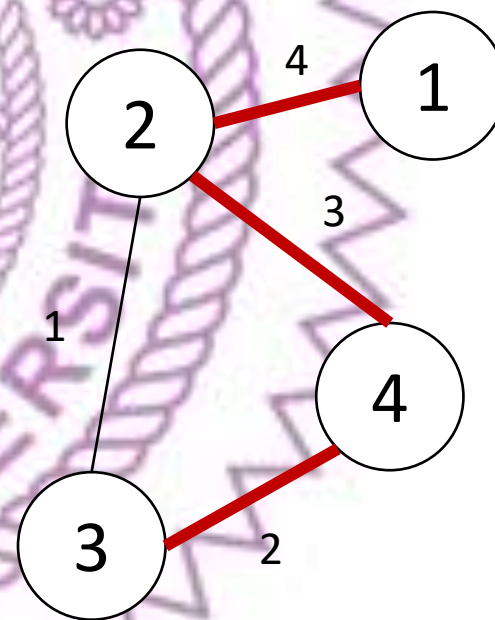
long long MST(int n) {
    sort(E.begin(), E.end());
    DisjointSet DS(n);
    long long ans = 0;
    int cnt = 0;
    for (auto &e : E) {
        if (DS.Same(e.u, e.v)) continue;
        DS.Union(e.u, e.v);
        ans += e.cost;
        ++cnt;
    }
    if (cnt < n - 1) cout << "not connected\n";
    return ans;
}
```

最小瓶頸路徑

- $a \rightarrow b$ 的最小瓶頸路徑
是邊權最大值最小的那條路徑
可以有好多條

- Ex: $1 \rightarrow 3$ 的最小瓶頸路徑有

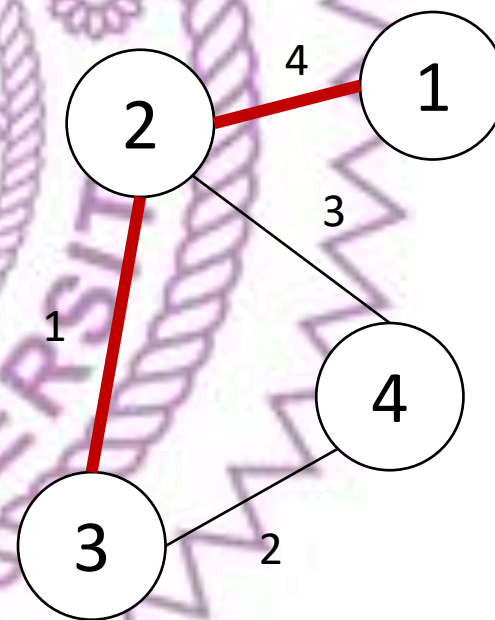
- $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$
- $1 \rightarrow 2 \rightarrow 3$



最小瓶頸路徑

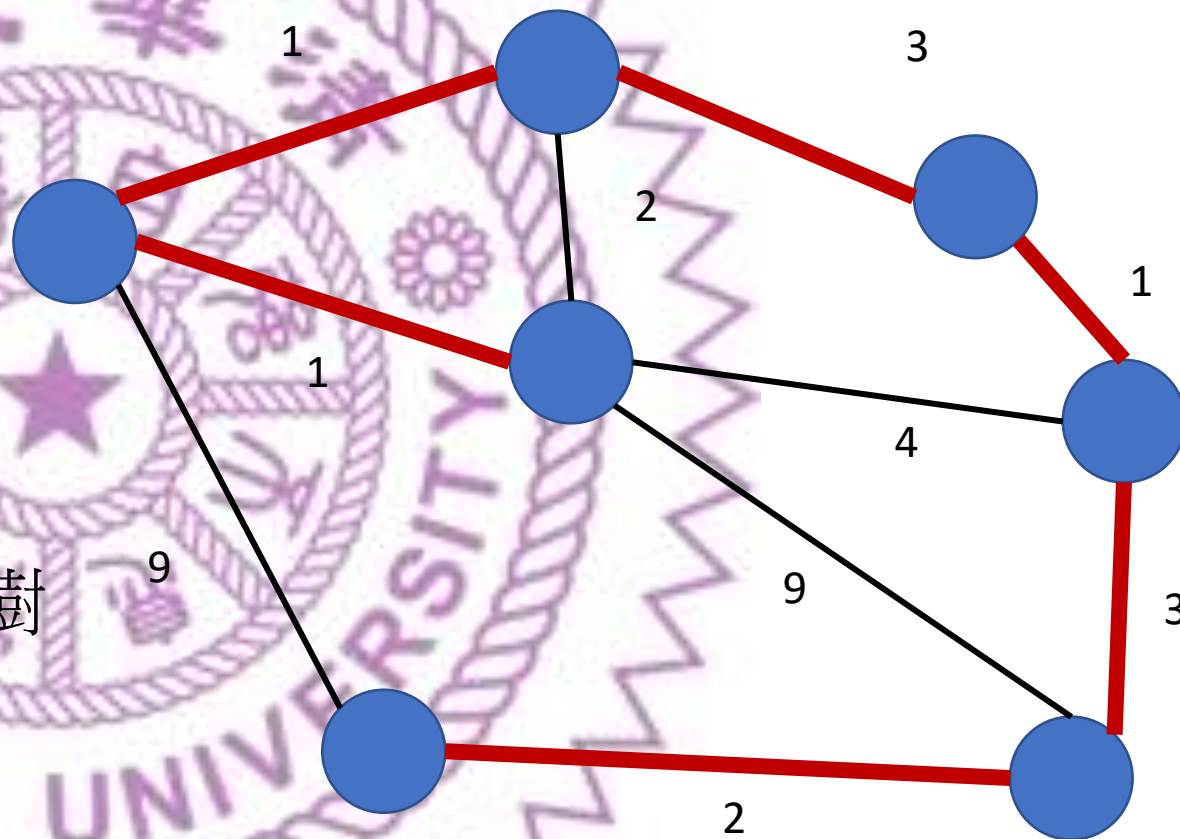
- $a \rightarrow b$ 的最小瓶頸路徑
是邊權最大值最小的那條路徑
可以有好多條

- Ex: $1 \rightarrow 3$ 的最小瓶頸路徑有
 - $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$
 - $1 \rightarrow 2 \rightarrow 3$



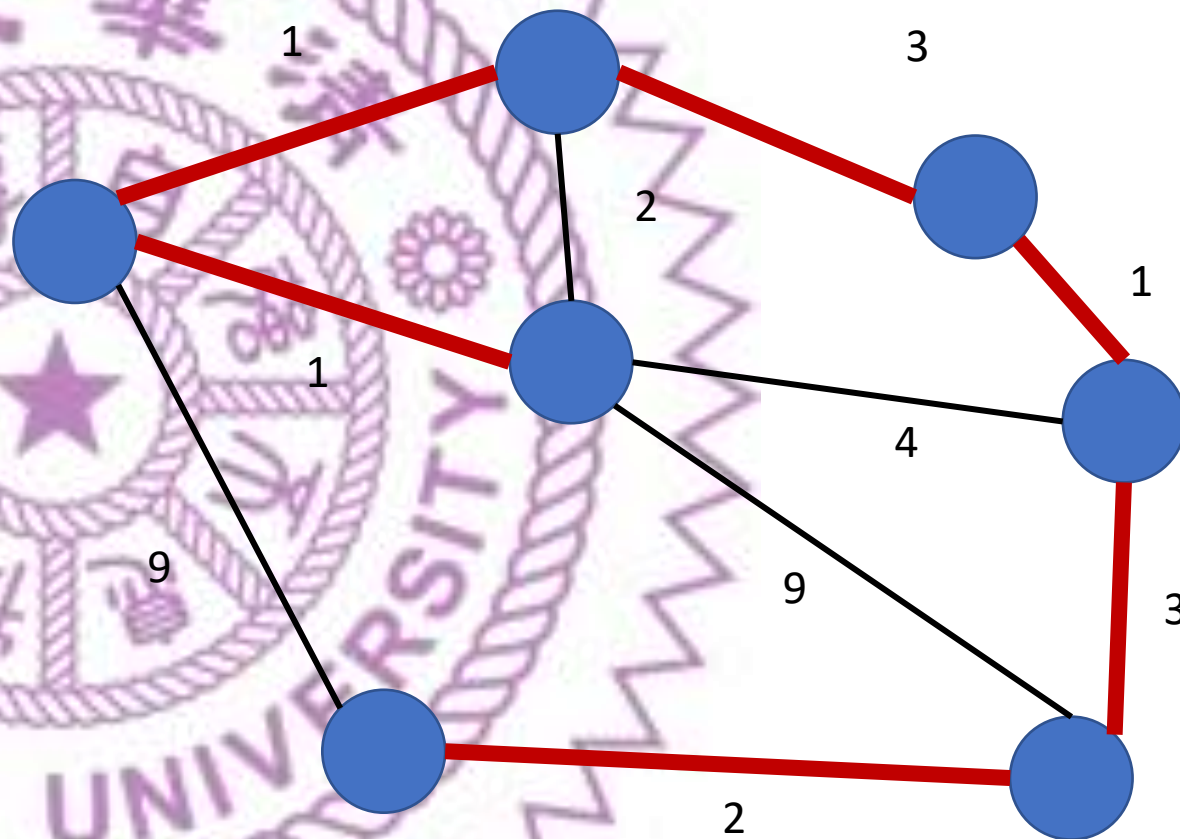
最小瓶頸生成樹

- 圖 G 的最小瓶頸生成樹 T
- 滿足：
 - T 上權重最大的邊是所有生成樹中最小的
- 最小生成樹 **是** 最小瓶頸生成樹
 - 反過來不一定成立



最小瓶頸生成樹

- 圖 G 的最小生成樹 T
- 滿足：
 - T 上任兩點 a, b 的路徑同時是圖 G 中 $a \rightarrow b$ 的最小瓶頸路徑



最短路徑

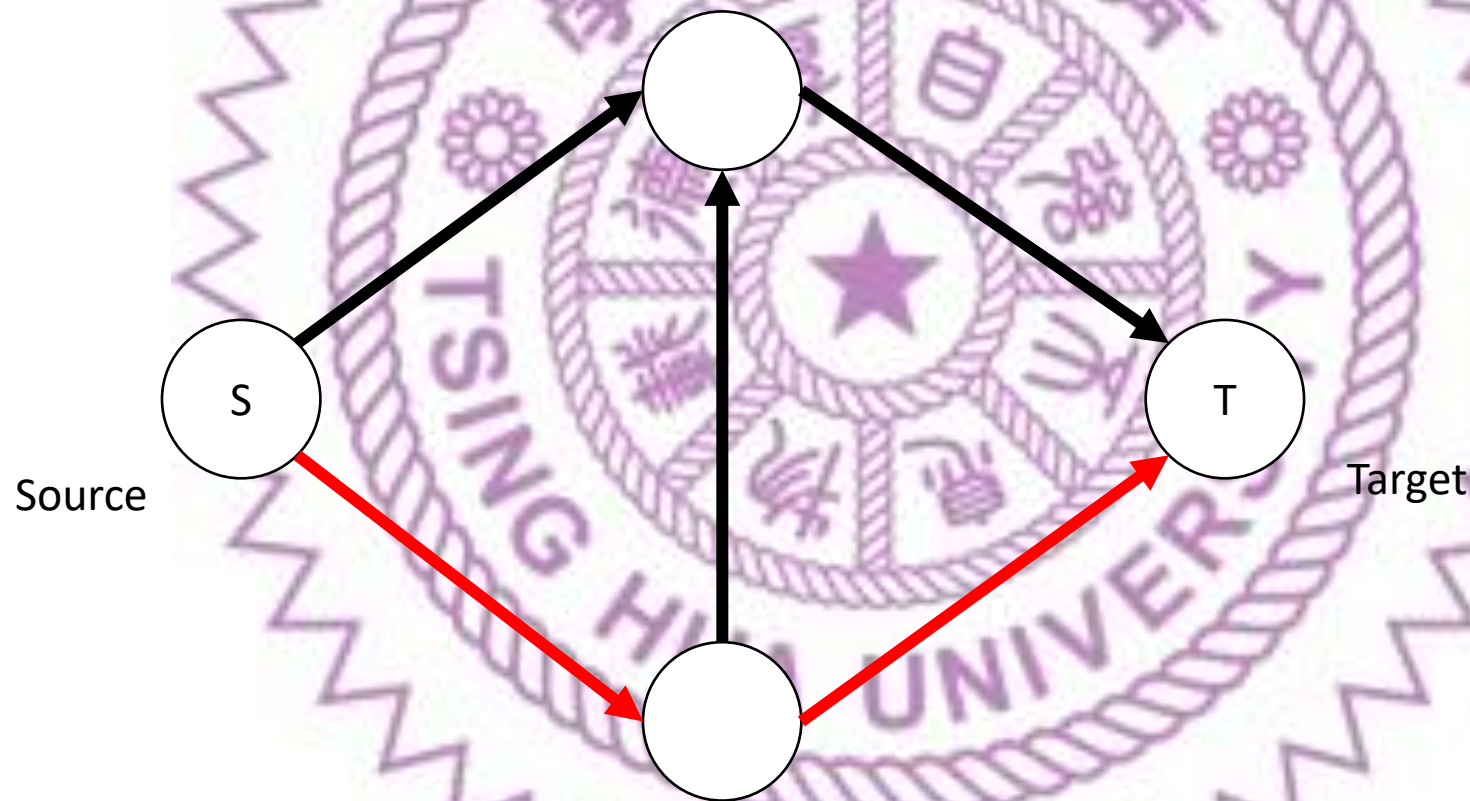
Shortest Path

三大演算法

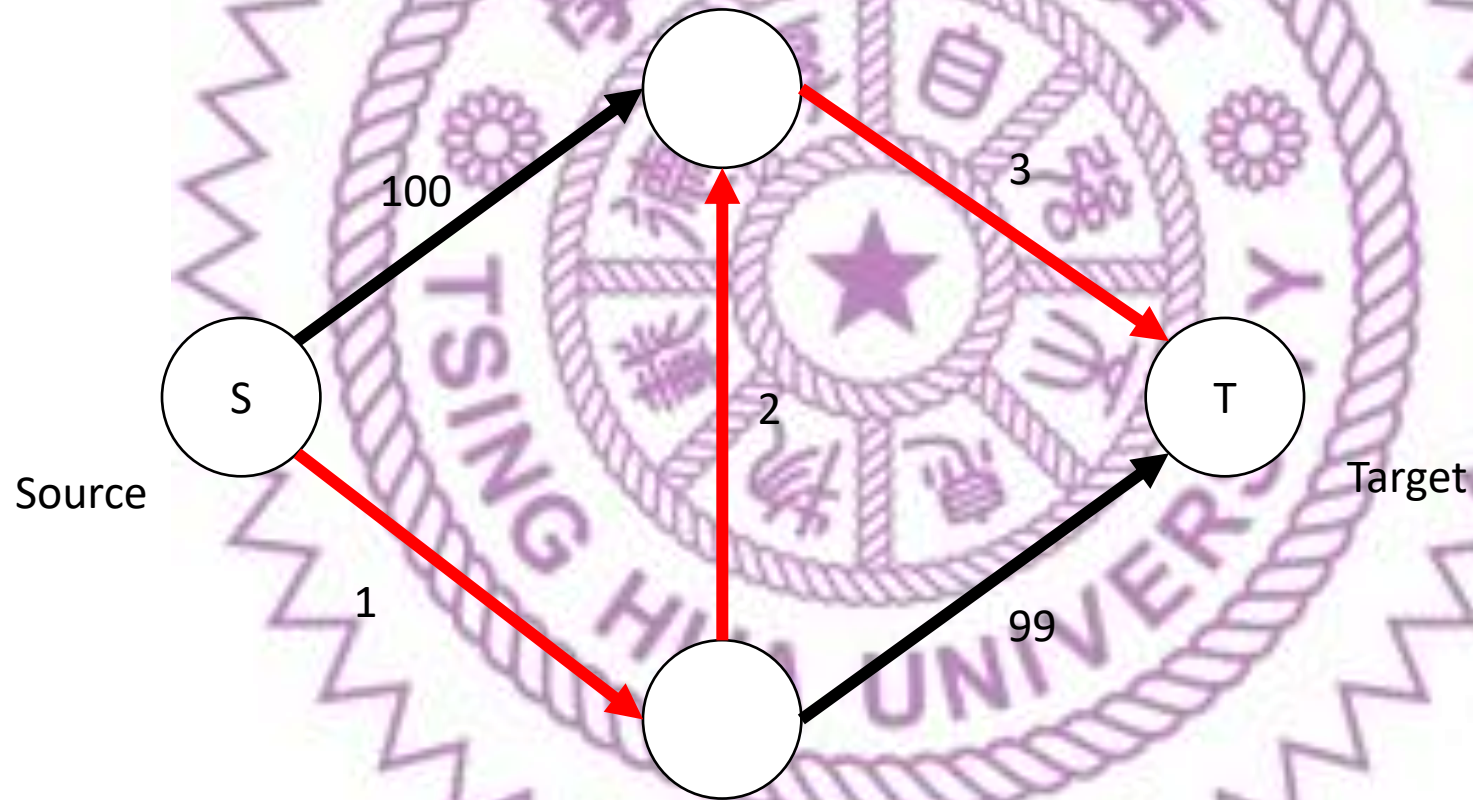
最常見應用：google map



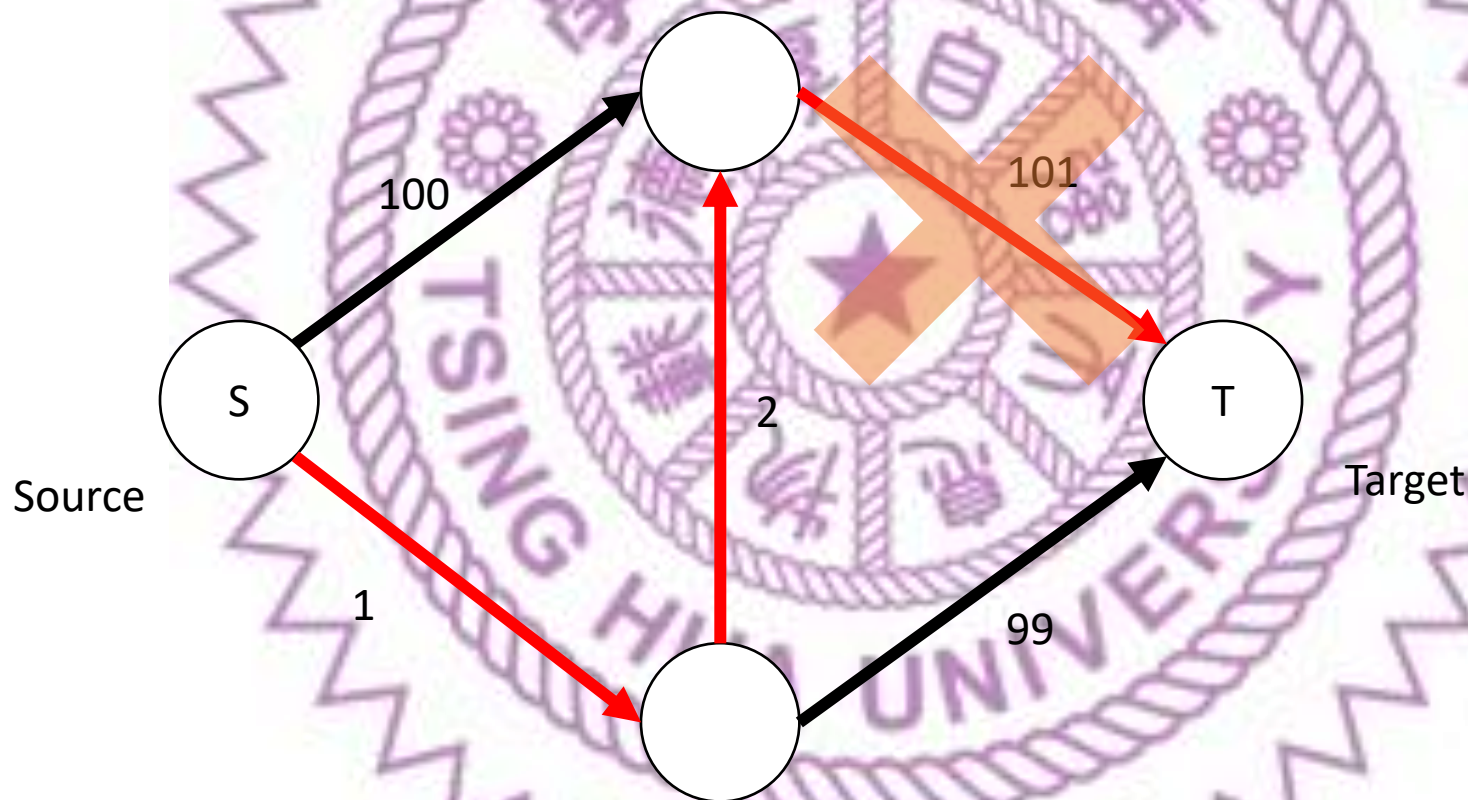
圖論上的最短路徑(BFS)



圖論上的最短路徑



貪心法顯然不行



Dijkstra

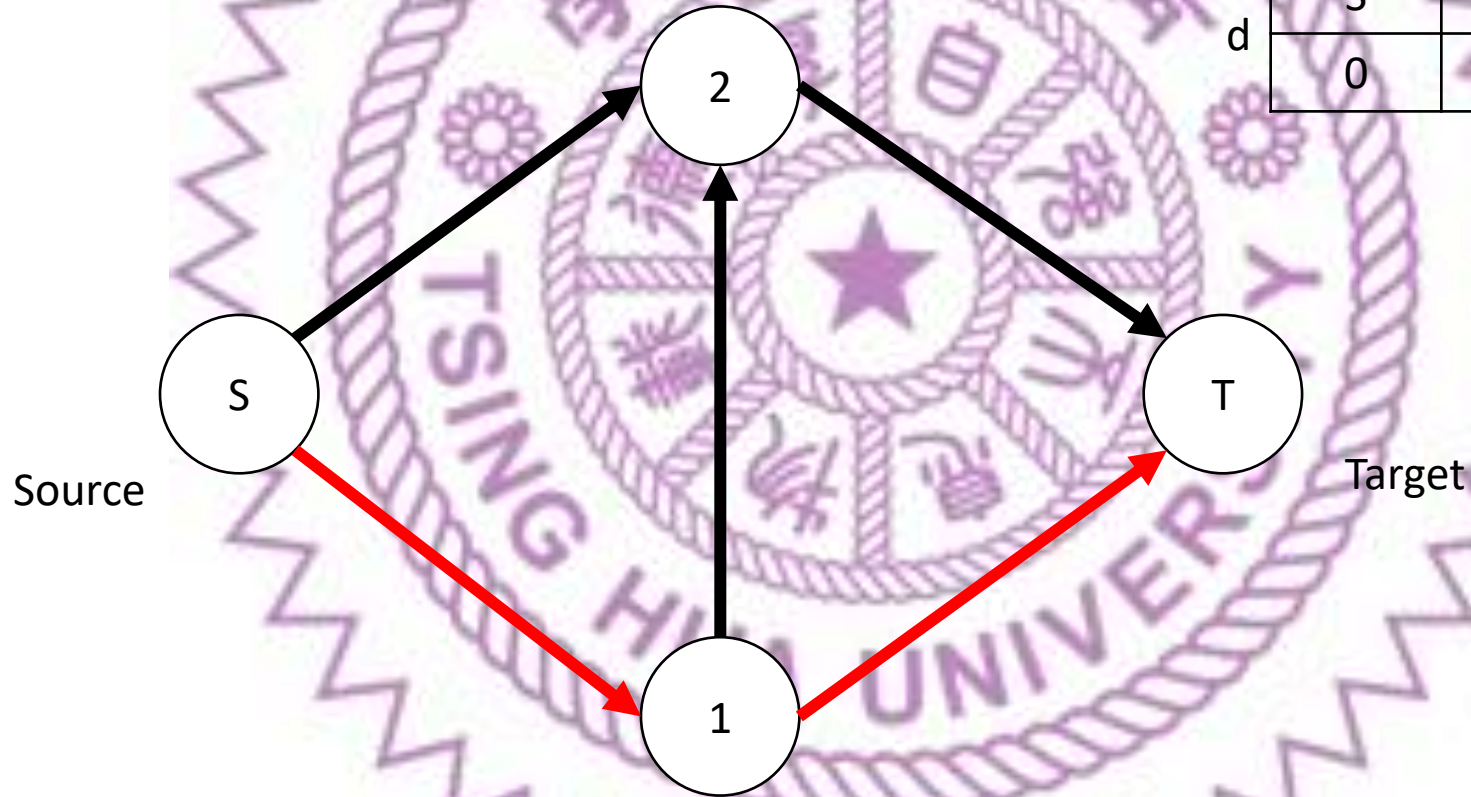
限制條件：邊的權重大於0



無權單源最短路徑

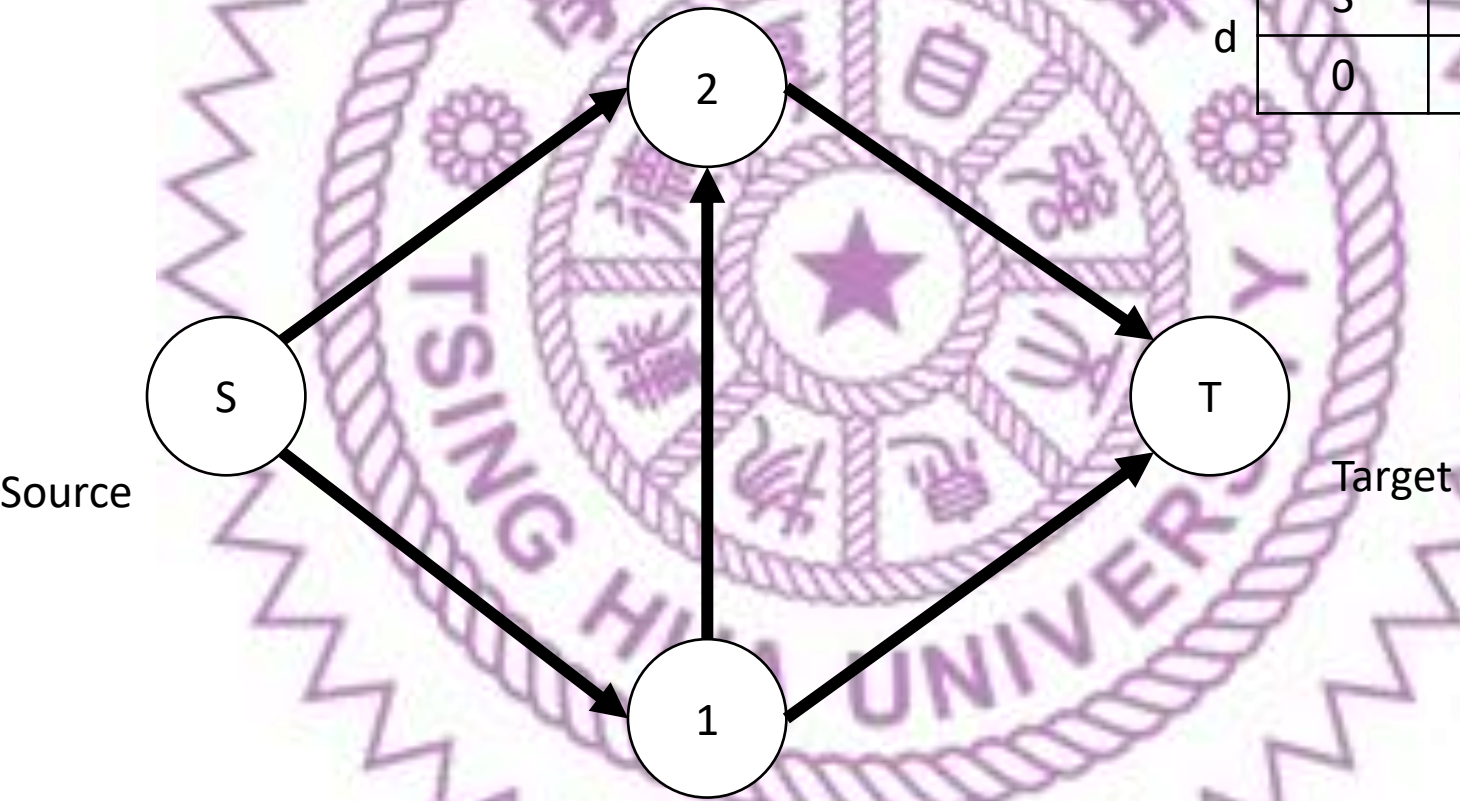
目標：
算出這表格d

S	1	2	T
0	1	1	2



無向圖單源最短路徑

一開始除了起點
其他點都是無限大



d	S	1	2	T
	0	∞	∞	∞

直接 BFS

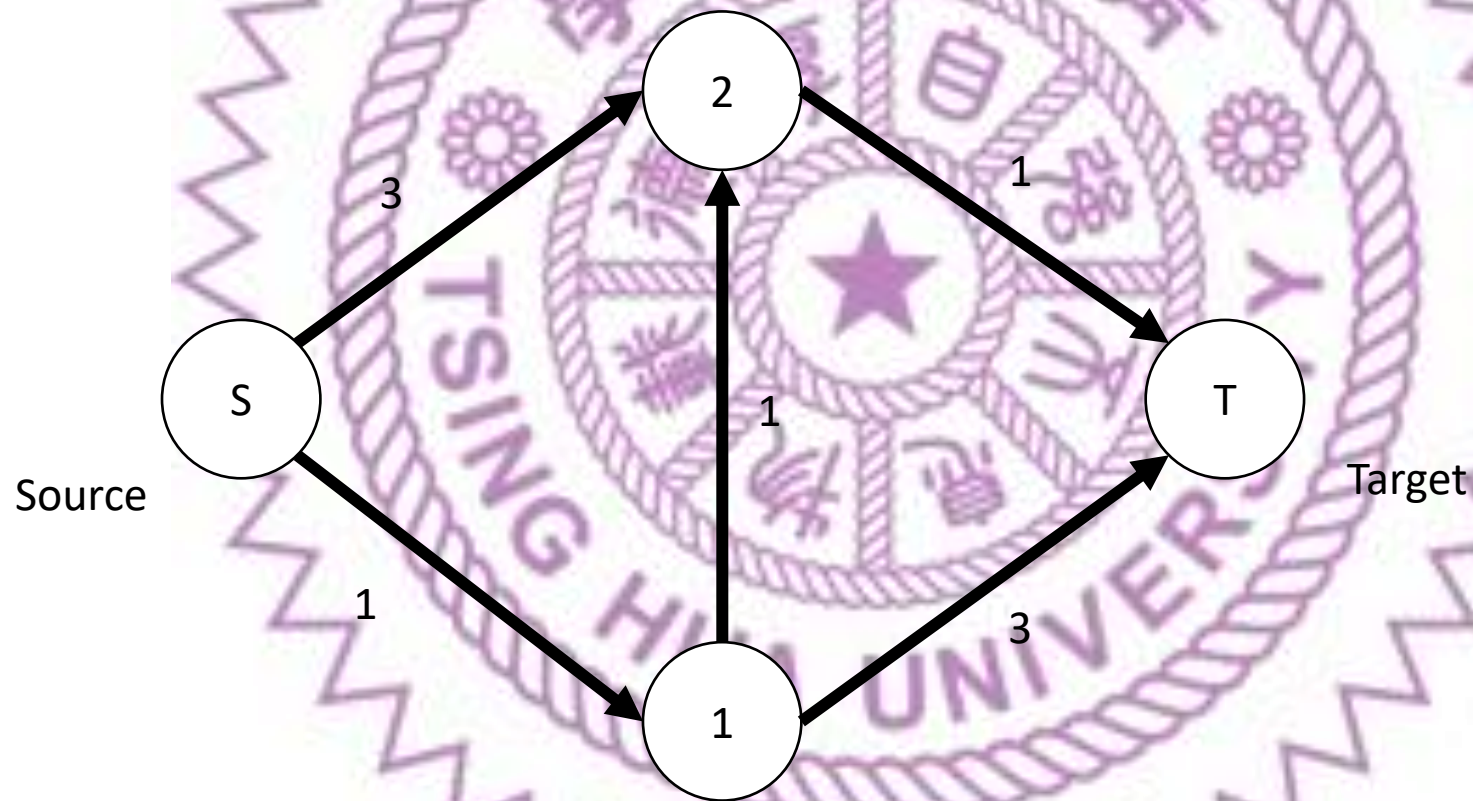
注意此時從queue拿出來時
d[u]已經是最短路徑了

```
vector<long long> BFS(const vector<vector<int>> &G, int S) {  
    int n = G.size(); // 假設點的編號為 0 ~ n-1  
    vector<long long> d(n, INF);  
    vector<bool> visited(n, false);  
    queue<int> Q;  
    Q.emplace(S);  
    d[S] = 0; // 起點設 0  
    while (Q.size()) {  
        int u = Q.front();  
        Q.pop();  
        if (visited[u]) continue;  
        visited[u] = true;  
        for (auto v : G[u]) {  
            if (d[v] > d[u] + 1)  
                d[v] = d[u] + 1;  
            Q.emplace(v);  
        }  
    }  
    return d;  
}
```

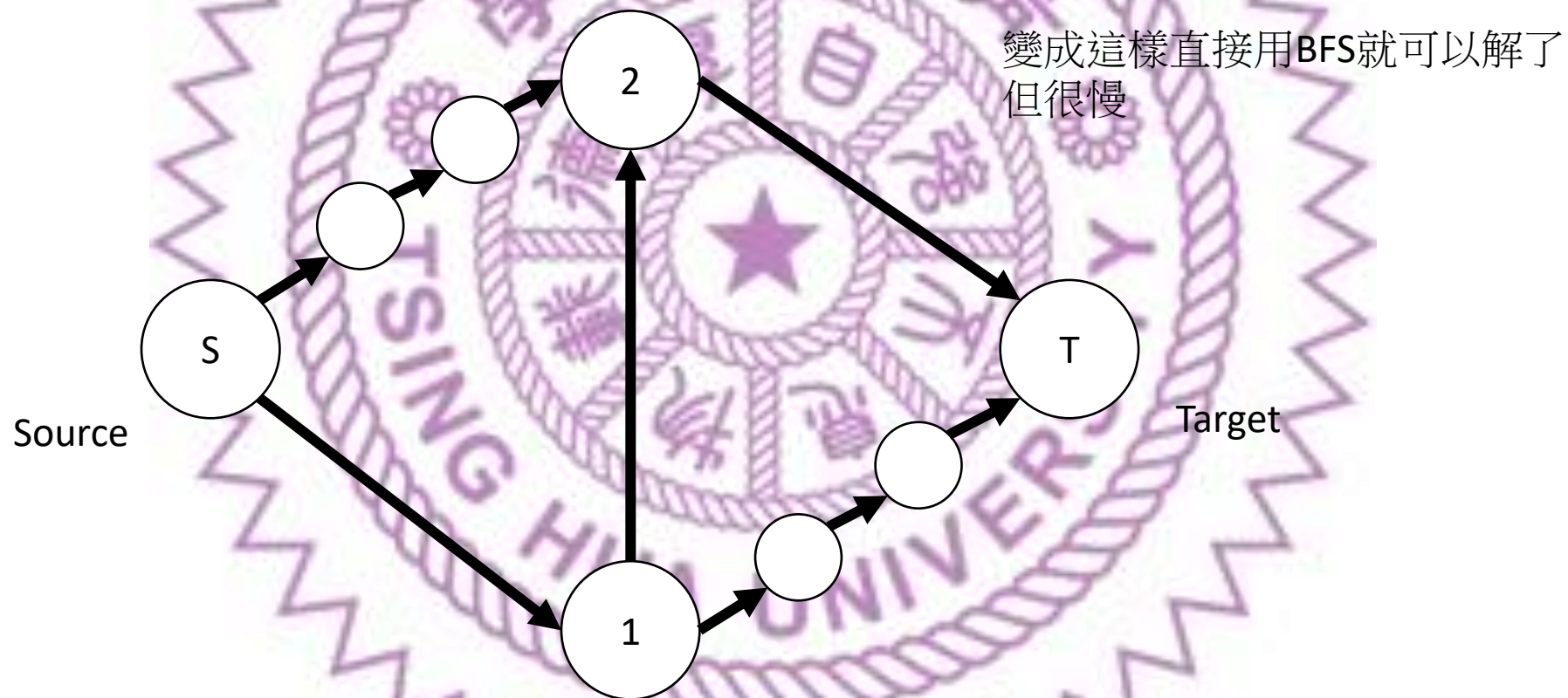

加速：
進到 if
才放進
queue

```
vector<long long> BFS(const vector<vector<int>> &G, int S) {  
    int n = G.size(); // 假設點的編號為 0 ~ n-1  
    vector<long long> d(n, INF);  
    vector<bool> visited(n, false);  
    queue<int> Q;  
    Q.emplace(S);  
    d[S] = 0; // 起點設 0  
    while (Q.size()) {  
        int u = Q.front();  
        Q.pop();  
        if (visited[u]) continue;  
        visited[u] = true;  
        for (auto v : G[u])  
            if (d[v] > d[u] + 1) {  
                d[v] = d[u] + 1;  
                Q.emplace(v);  
            }  
    }  
    return d;  
}
```

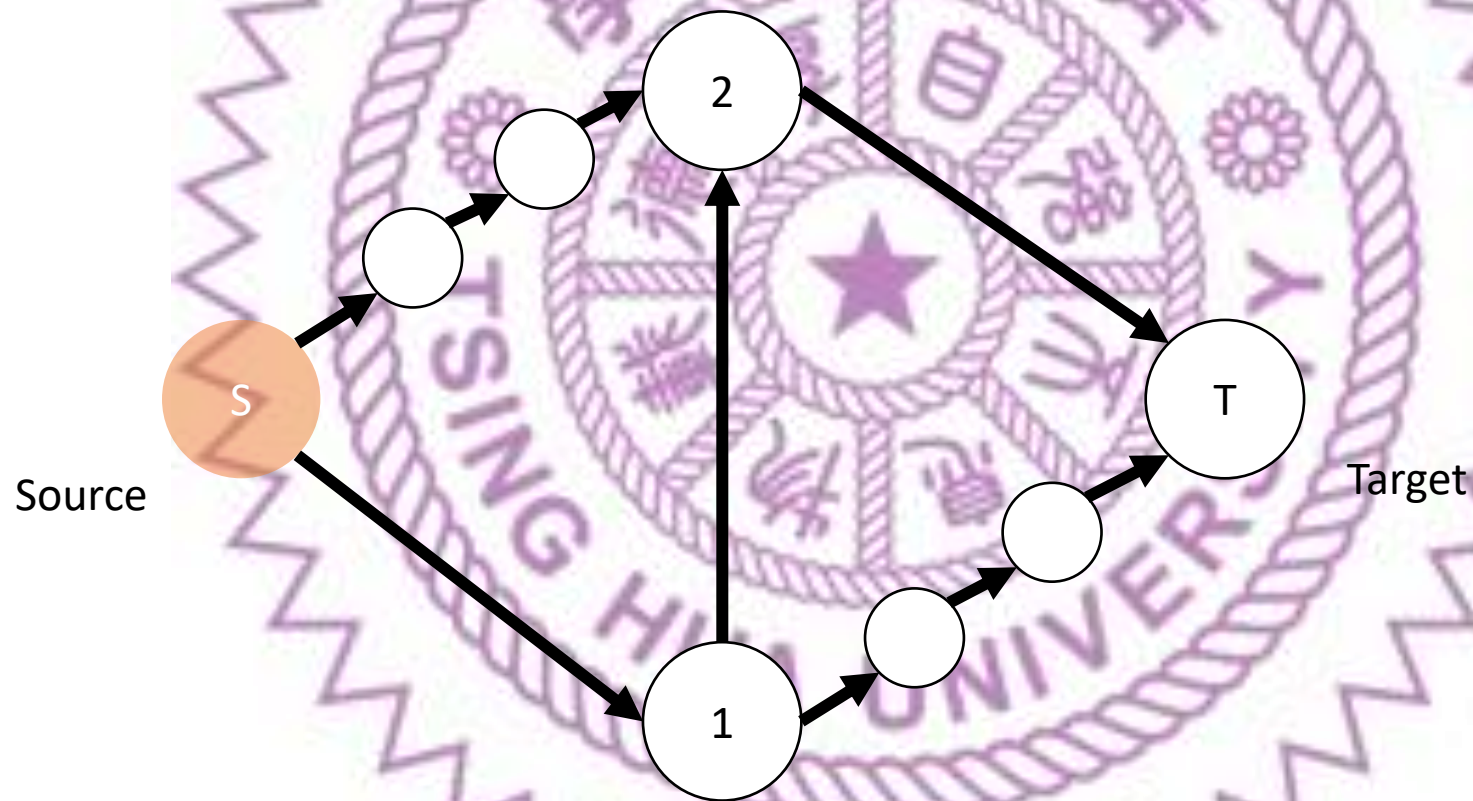
單源最短路徑



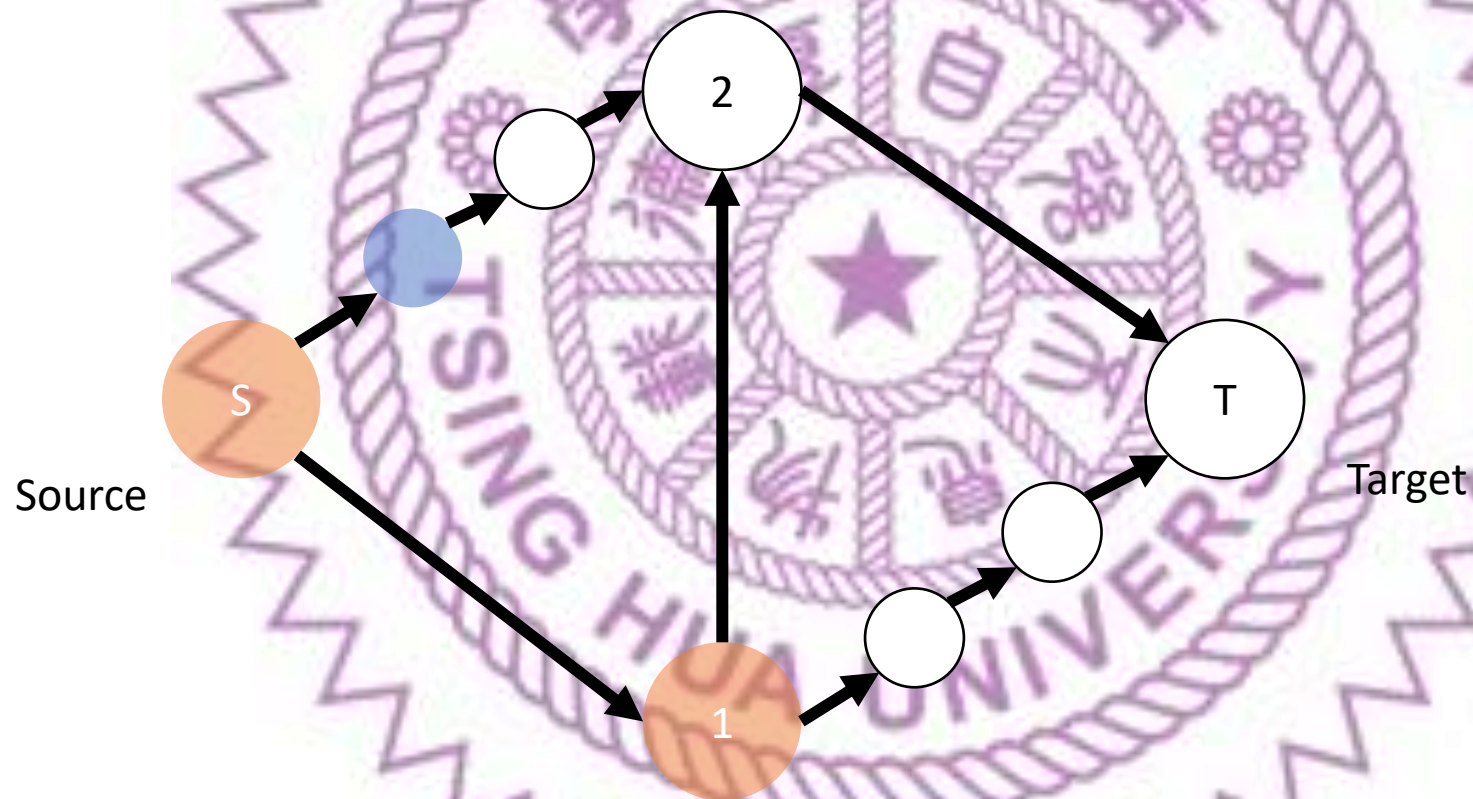
單源最短路徑



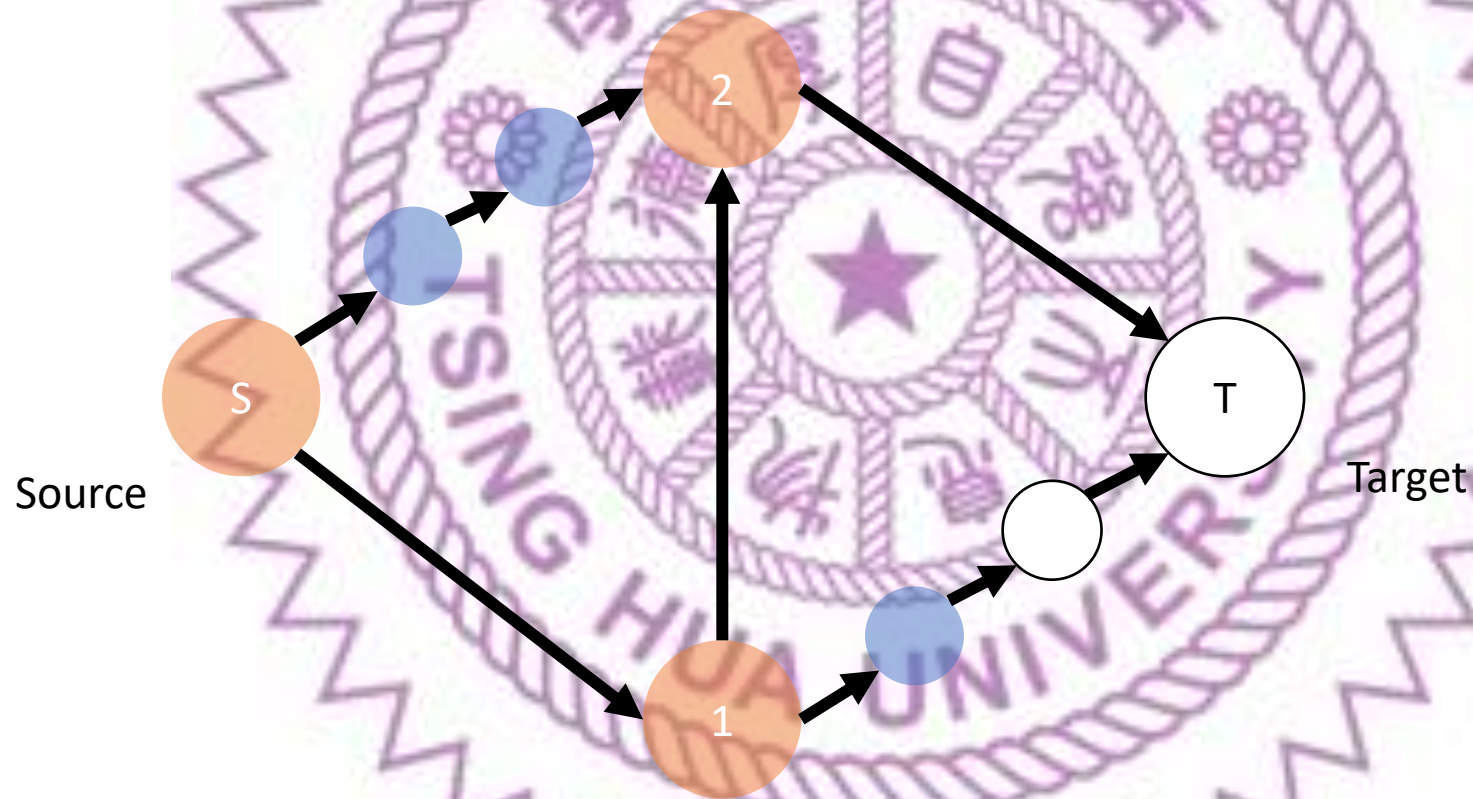
單源最短路徑



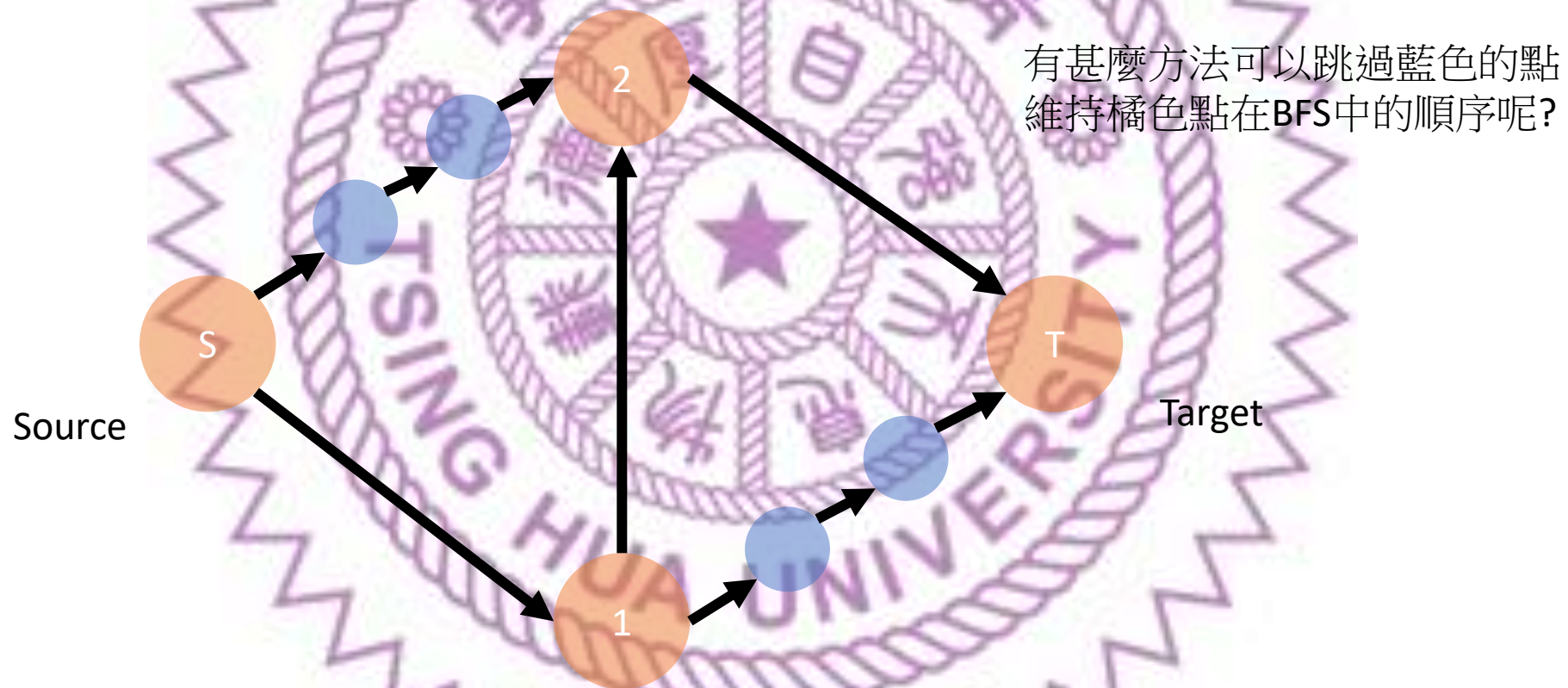
單源最短路徑



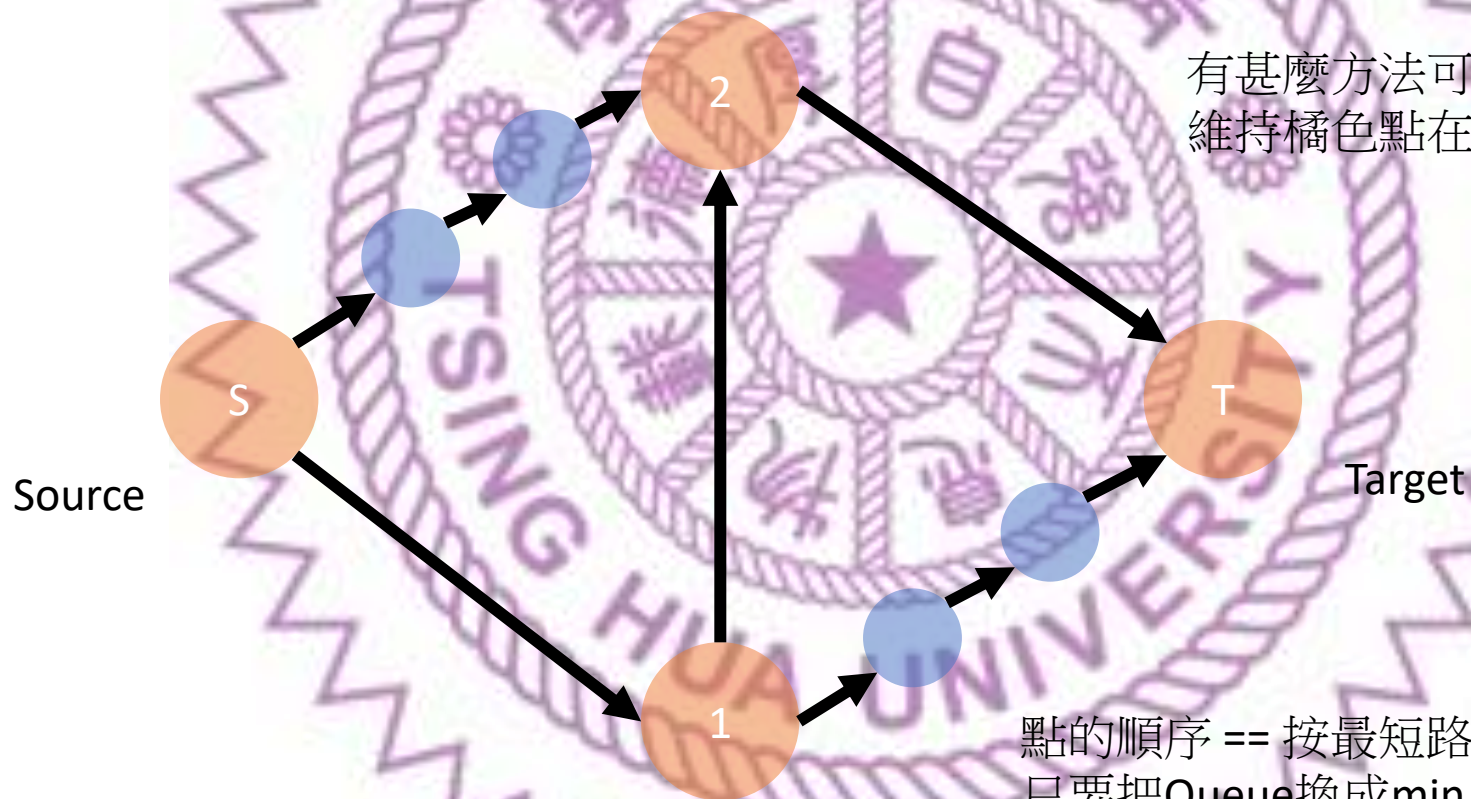
單源最短路徑



單源最短路徑



單源最短路徑



有甚麼方法可以跳過藍色的點
維持橘色點在BFS中的順序呢?

點的順序 == 按最短路徑由小排到大
只要把Queue換成min heap紀錄當前的最短路徑！

Adjacency List + 紀錄邊權重

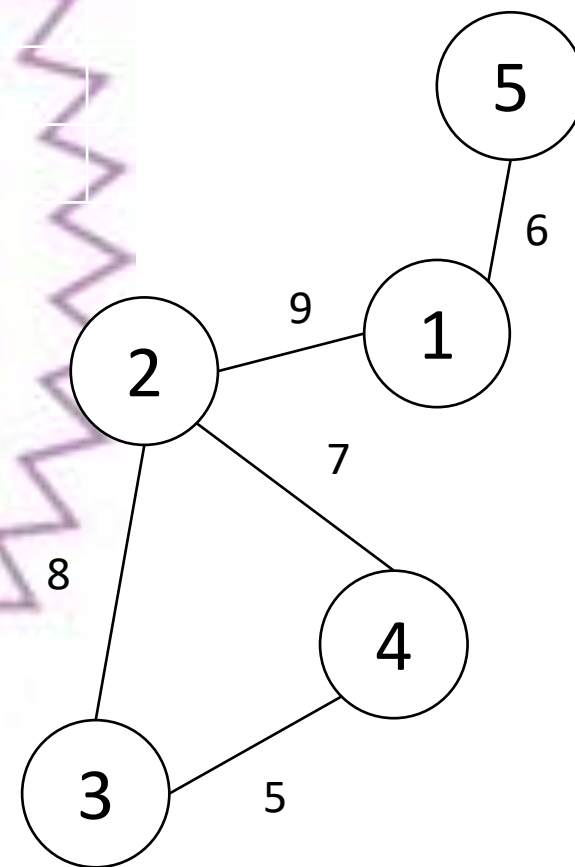
n 個點, m 條邊

m 條邊

5	5	
1	2	9
2	3	8
2	4	7
1	5	6
3	4	5

1	2,9	5,6
2	3,8	4,7
3	4,5	
4		
5		

```
vector<vector<pair<int, int>>>
G;
int n, m;
cin >> n >> m;
G.assign(n + 1, {});
while (m--) {
    int u, v, cost;
    cin >> u >> v >> cost;
    G[u].emplace_back(v, cost);
}
```



Dijkstra

```
vector<long long> dijkstra(const vector<vector<pair<int, int>>> &G, int S) {  
    int n = G.size(); // 假設點的編號為 0 ~ n-1  
    vector<long long> d(n, INF);  
    vector<bool> visited(n, false);  
    using QueuePair = pair<long long, int>;  
    priority_queue<QueuePair, vector<QueuePair>, greater<QueuePair>> Q;  
    d[S] = 0; // 起點設 0  
    Q.emplace(d[S], S);  
    while (Q.size()) {  
        int u = Q.top().second;  
        Q.pop();  
        if (visited[u]) continue;  
        visited[u] = true;  
        for (auto [v, cost] : G[u])  
            if (d[v] > d[u] + cost) {  
                d[v] = d[u] + cost;  
                Q.emplace(d[v], v);  
            }  
    }  
    return d;  
}
```

Dijkstra 不使用 visited

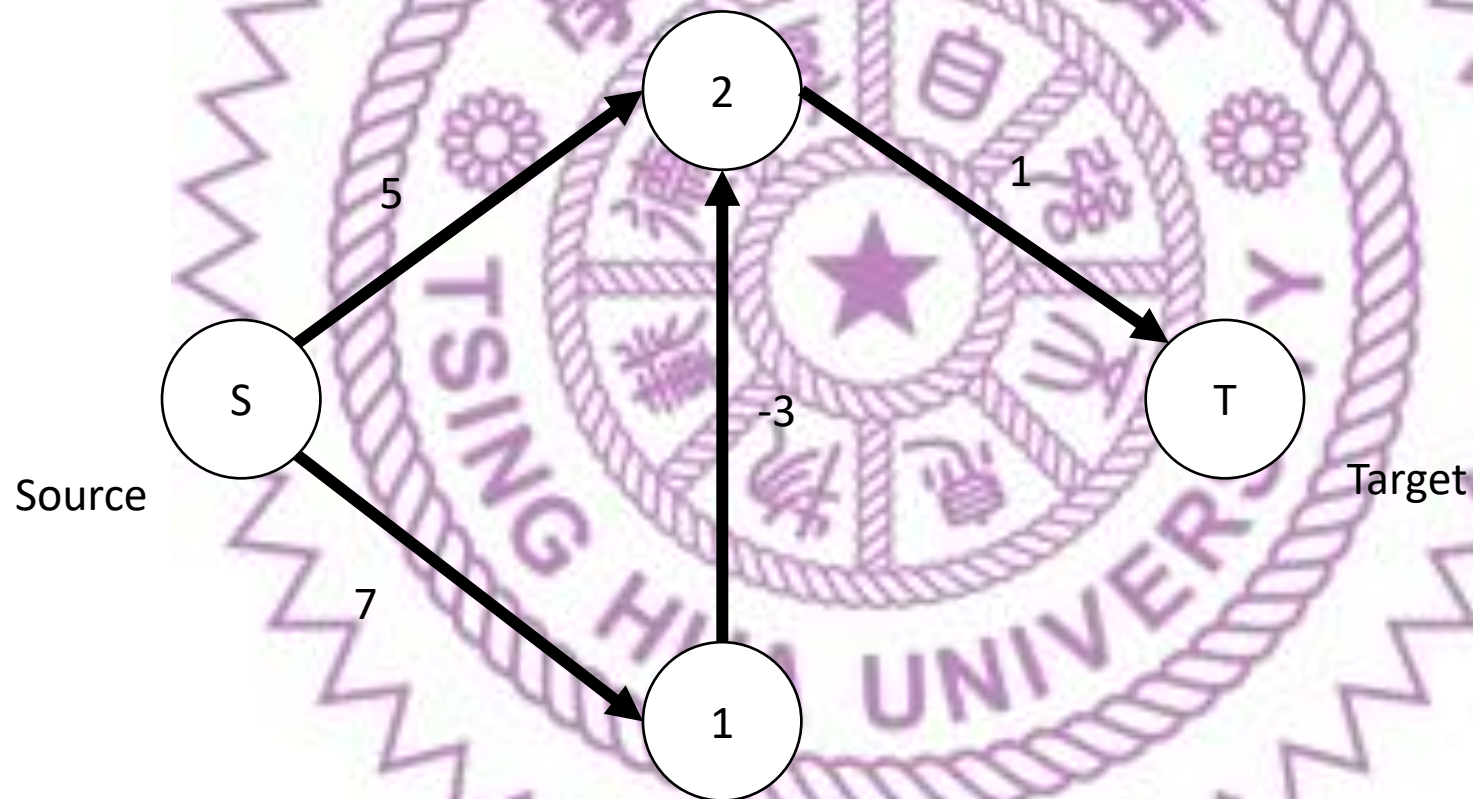
```
vector<long long> dijkstra(const vector<vector<pair<int, int>>> &G, int S) {  
    int n = G.size(); // 假設點的編號為 0 ~ n-1  
    vector<long long> d(n, INF);  
    using QueuePair = pair<long long, int>;  
    priority_queue<QueuePair, vector<QueuePair>, greater<QueuePair>> Q;  
    d[S] = 0; // 起點設 0  
    Q.emplace(d[S], S);  
    while (Q.size()) {  
        auto [u_dis, u] = Q.top();  
        Q.pop();  
        if (d[u] < u_dis) continue;  
        for (auto [v, cost] : G[u])  
            if (d[v] > d[u] + cost) {  
                d[v] = d[u] + cost;  
                Q.emplace(d[v], v);  
            }  
    }  
    return d;  
}
```

時間複雜度

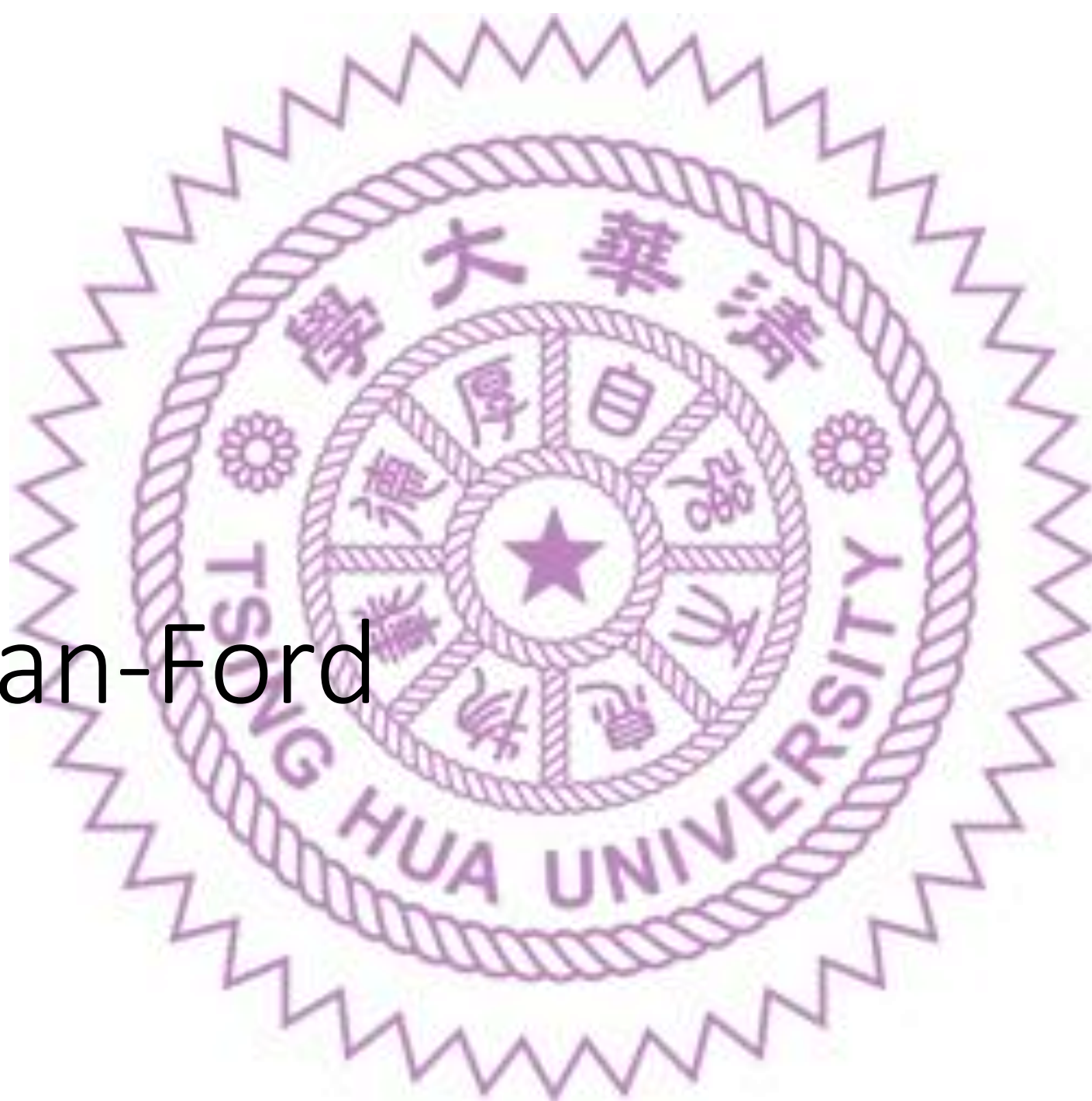
- 每條邊都有機會被丟進heap
- $O(|E|\log |V|)$
- 用費波納契堆
- $O(|E| + |V|\log |V|)$



會爛掉的例子



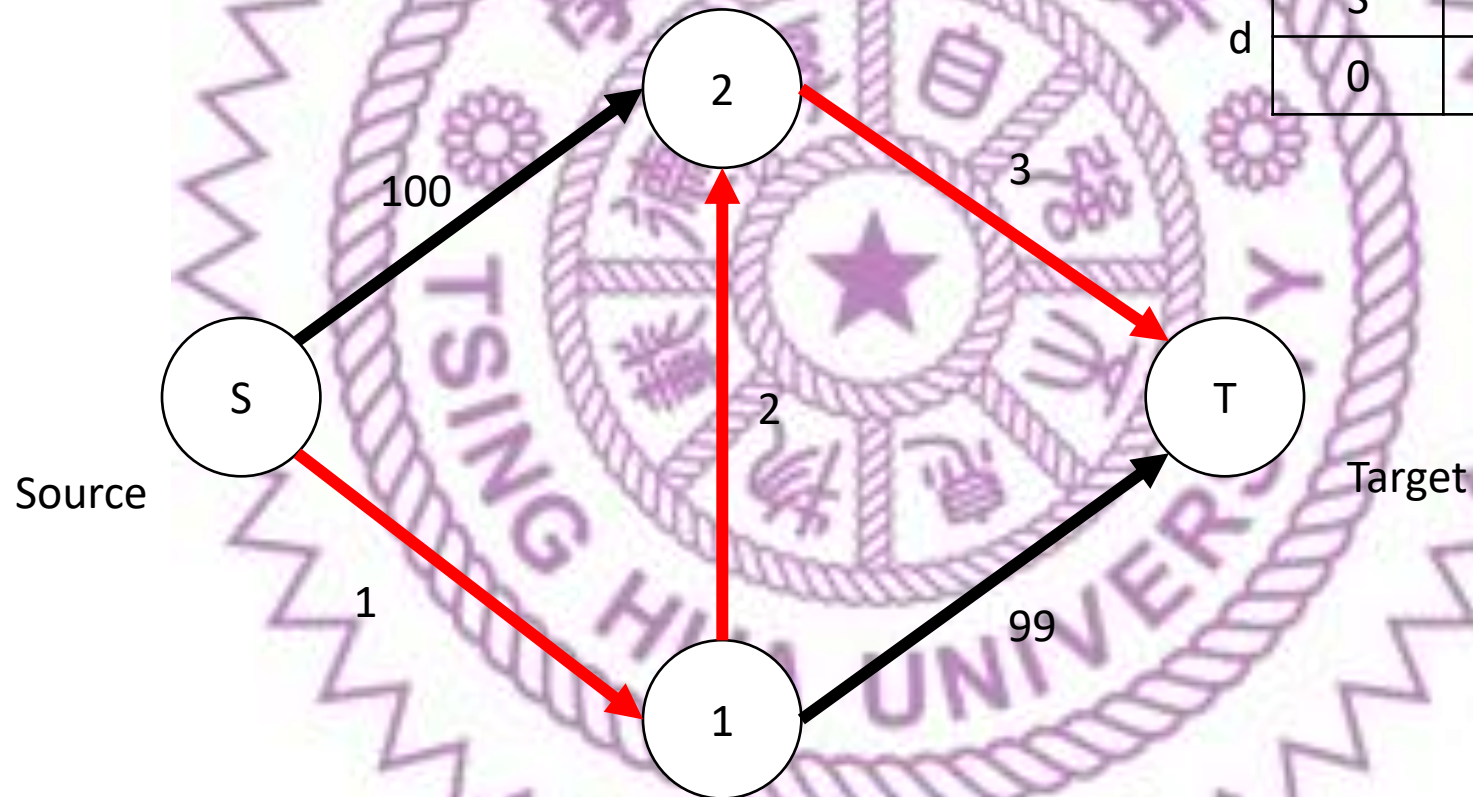
Bellman-Ford



單源最短路徑

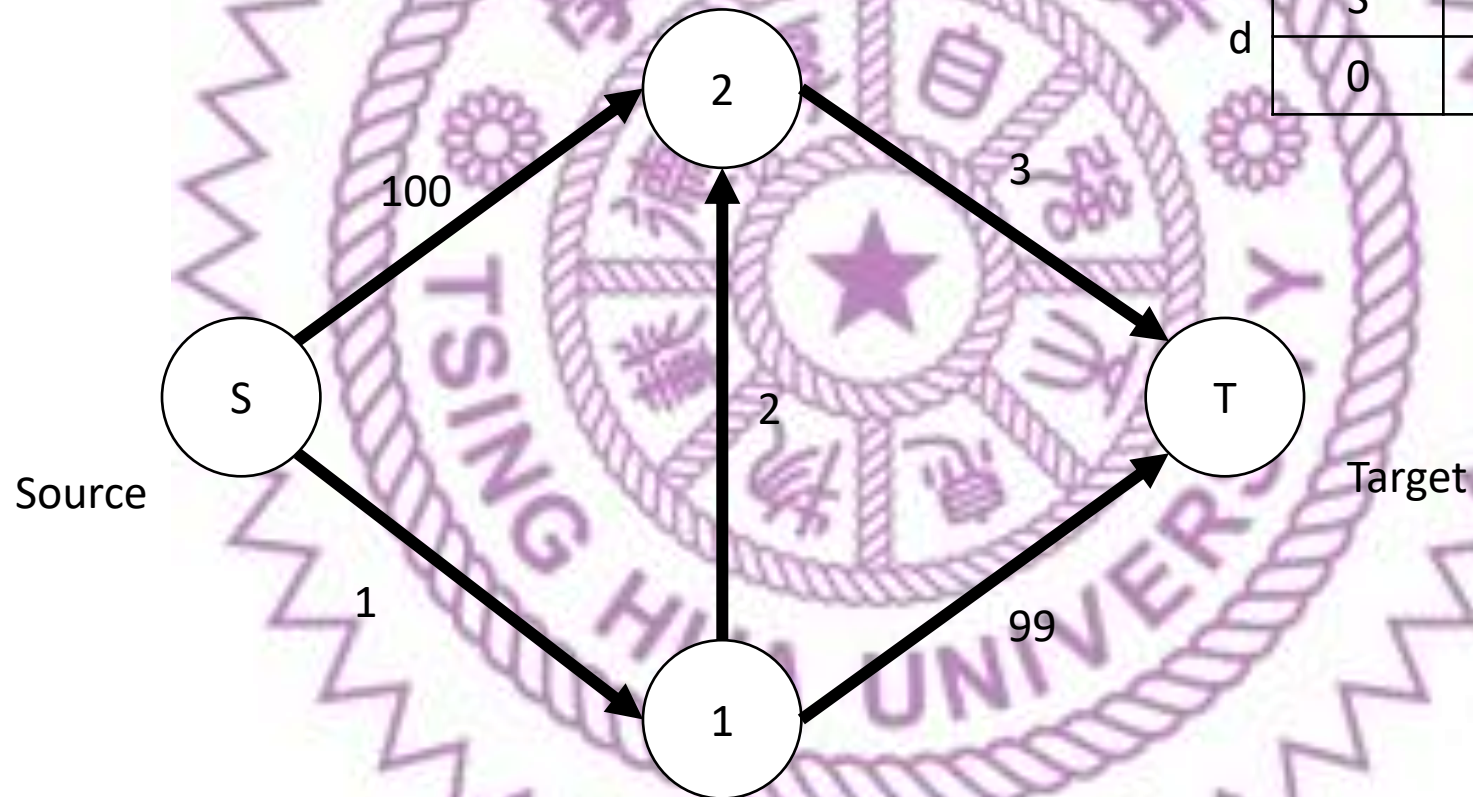
目標：
算出這表格d

S	1	2	T
0	1	3	6



單源最短路徑

一開始除了起點
其他點都是無限大



	S	1	2	T
d	0	∞	∞	∞

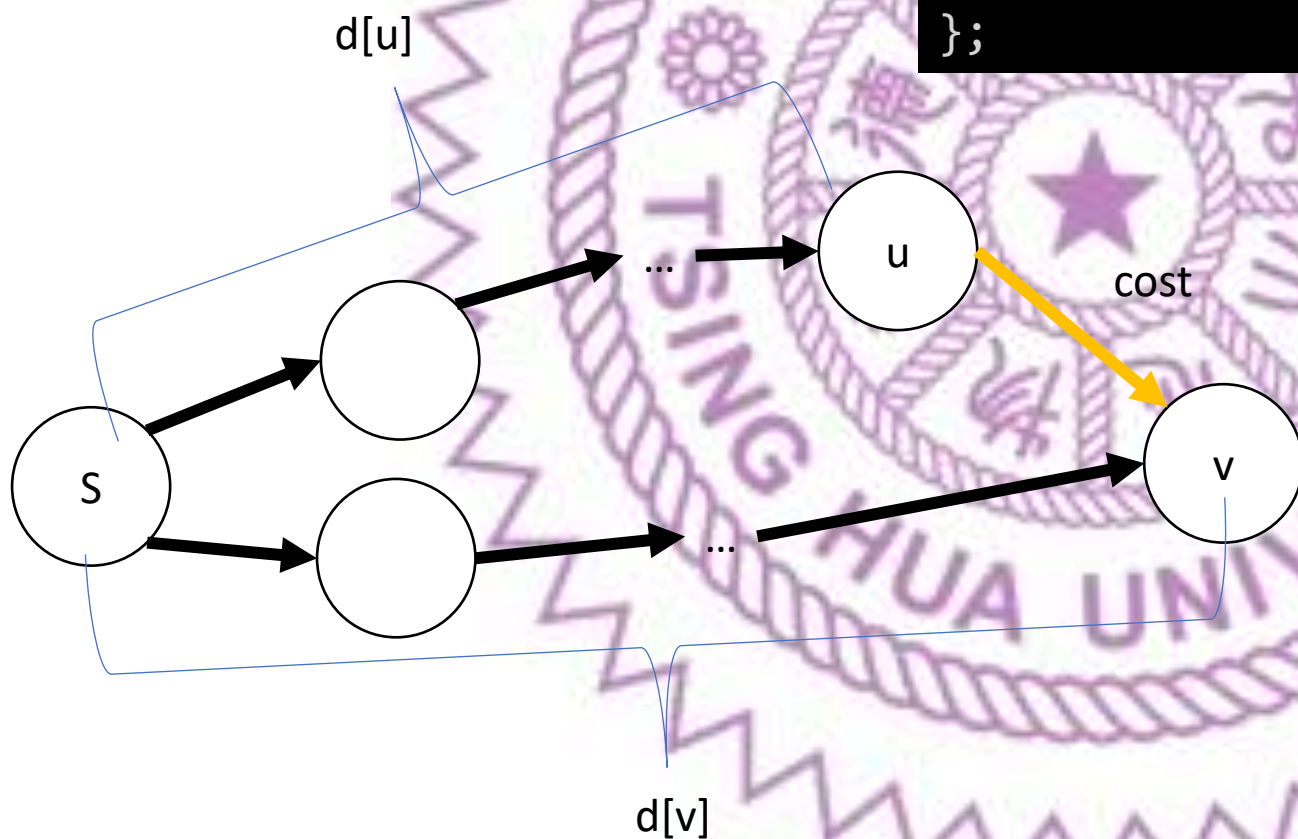
邊的資料結構

```
struct Edge {  
    int u, v;  
    int cost;  
};
```



鬆弛操作

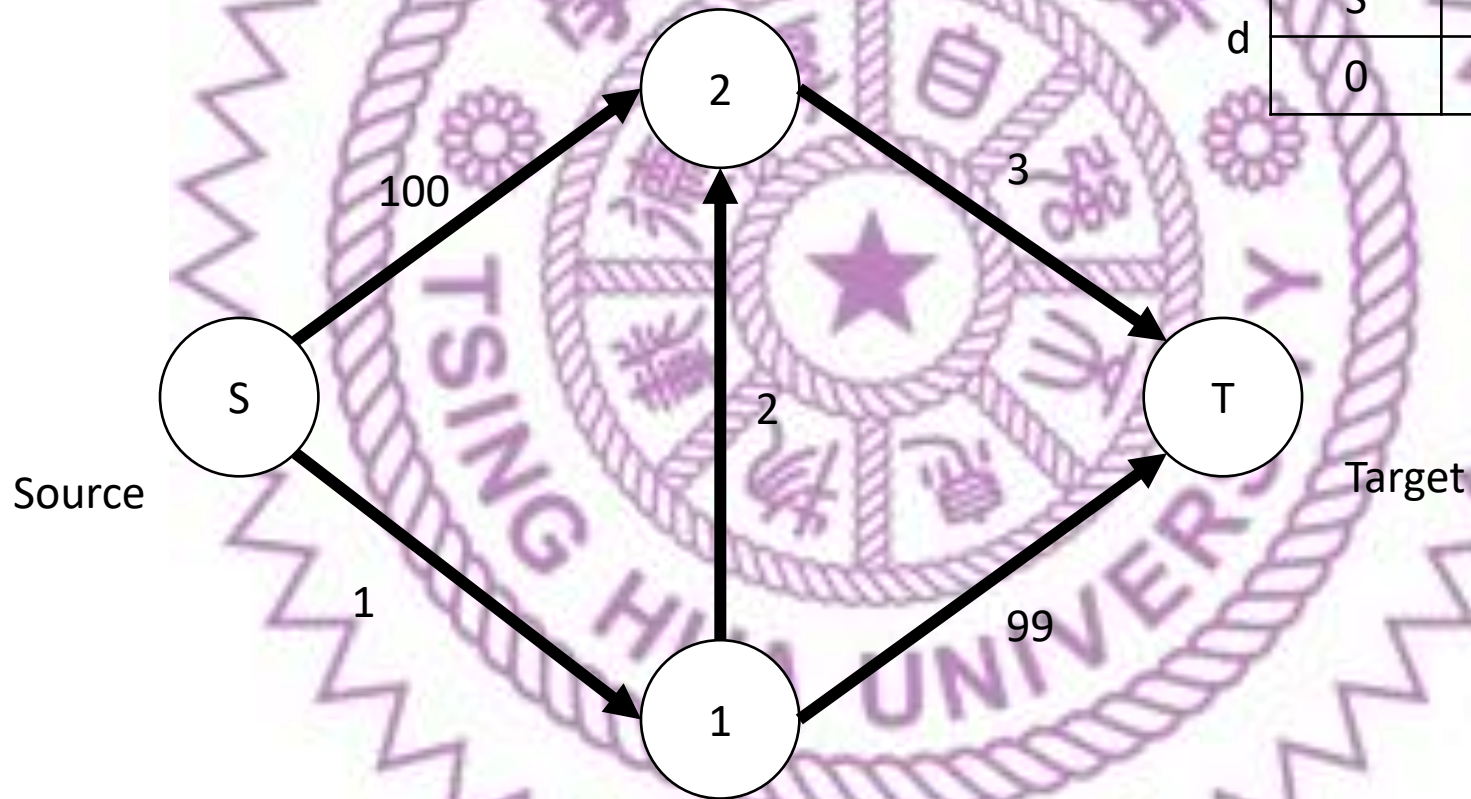
```
auto relax = [&](Edge e) {  
    if (d[e.v] > d[e.u] + e.cost) {  
        d[e.v] = d[e.u] + e.cost;  
        return true;  
    }  
    return false;  
};
```



暴力法：做到無法鬆弛為止

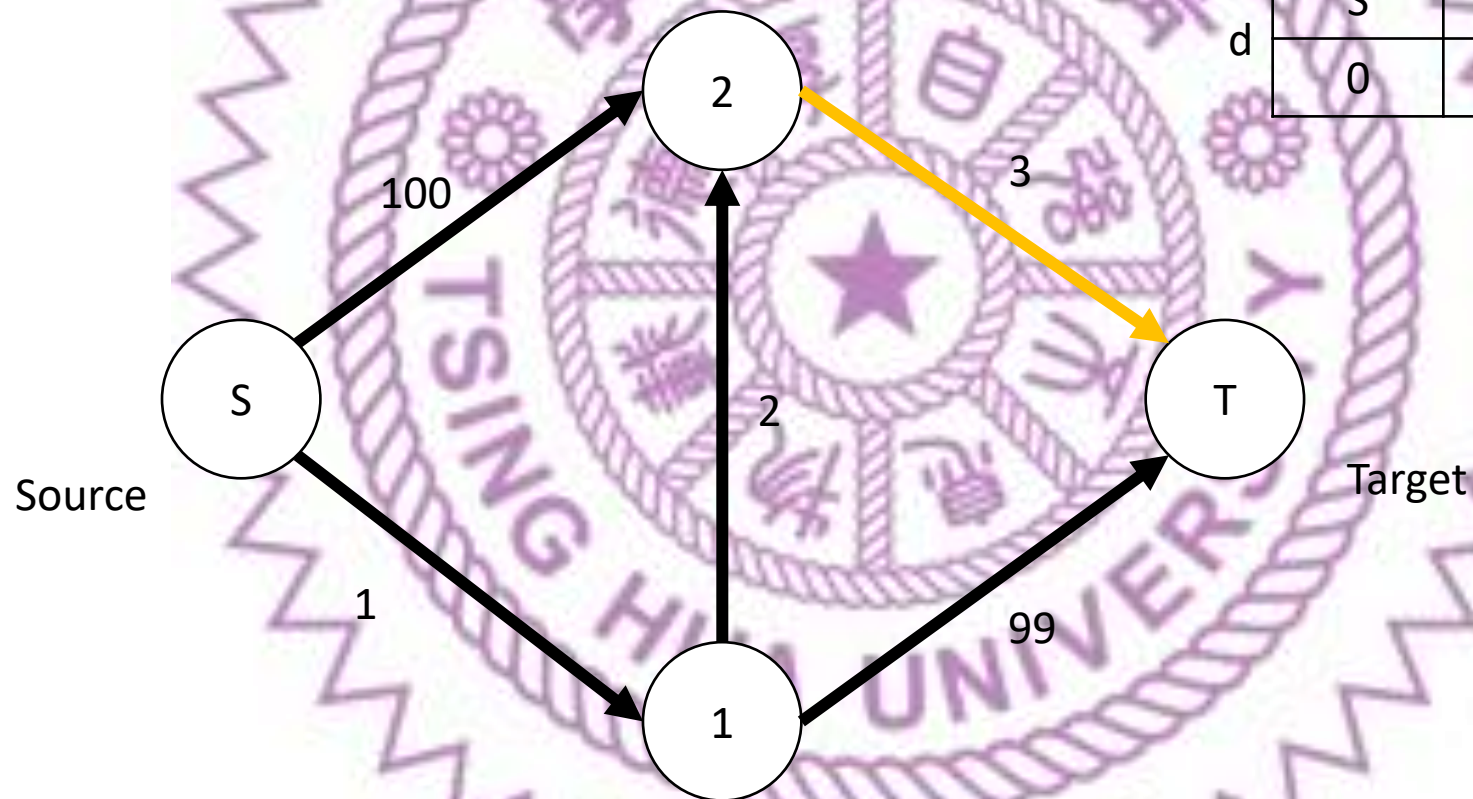
```
vector<long long> bellman_ford(vector<Edge> E, int n, int S) {  
    vector<long long> d(n, INF); // 假設點的編號為 0 ~ n-1  
    d[S] = 0; // 起點設 0  
    auto relax = [&](Edge e) { ... };  
    for (;;) {  
        bool update = false;  
        for (auto &e : E)  
            update |= relax(e);  
        if (!update) break;  
    }  
    return d;  
}
```

單源最短路徑



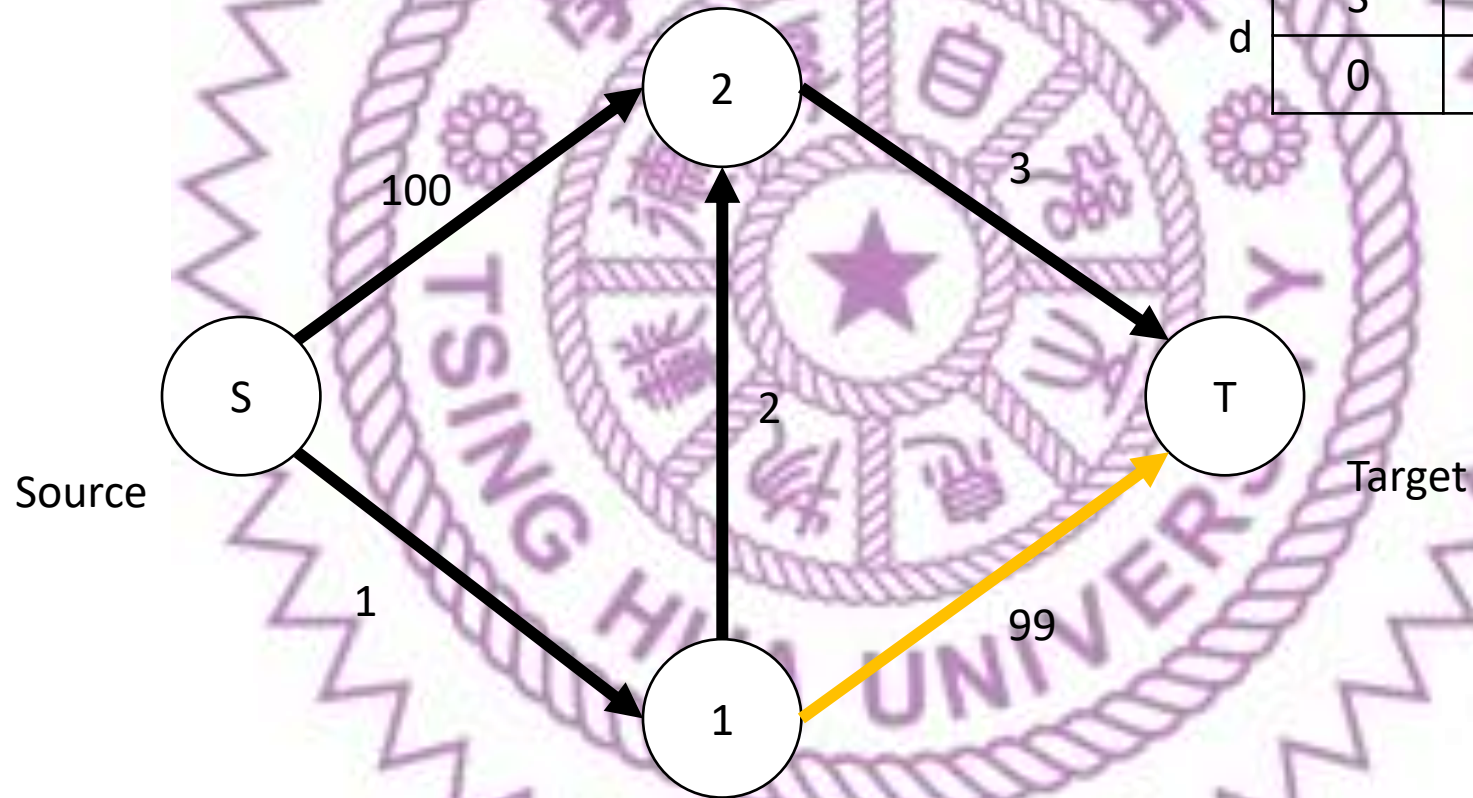
	S	1	2	T
d	0	∞	∞	∞

單源最短路徑



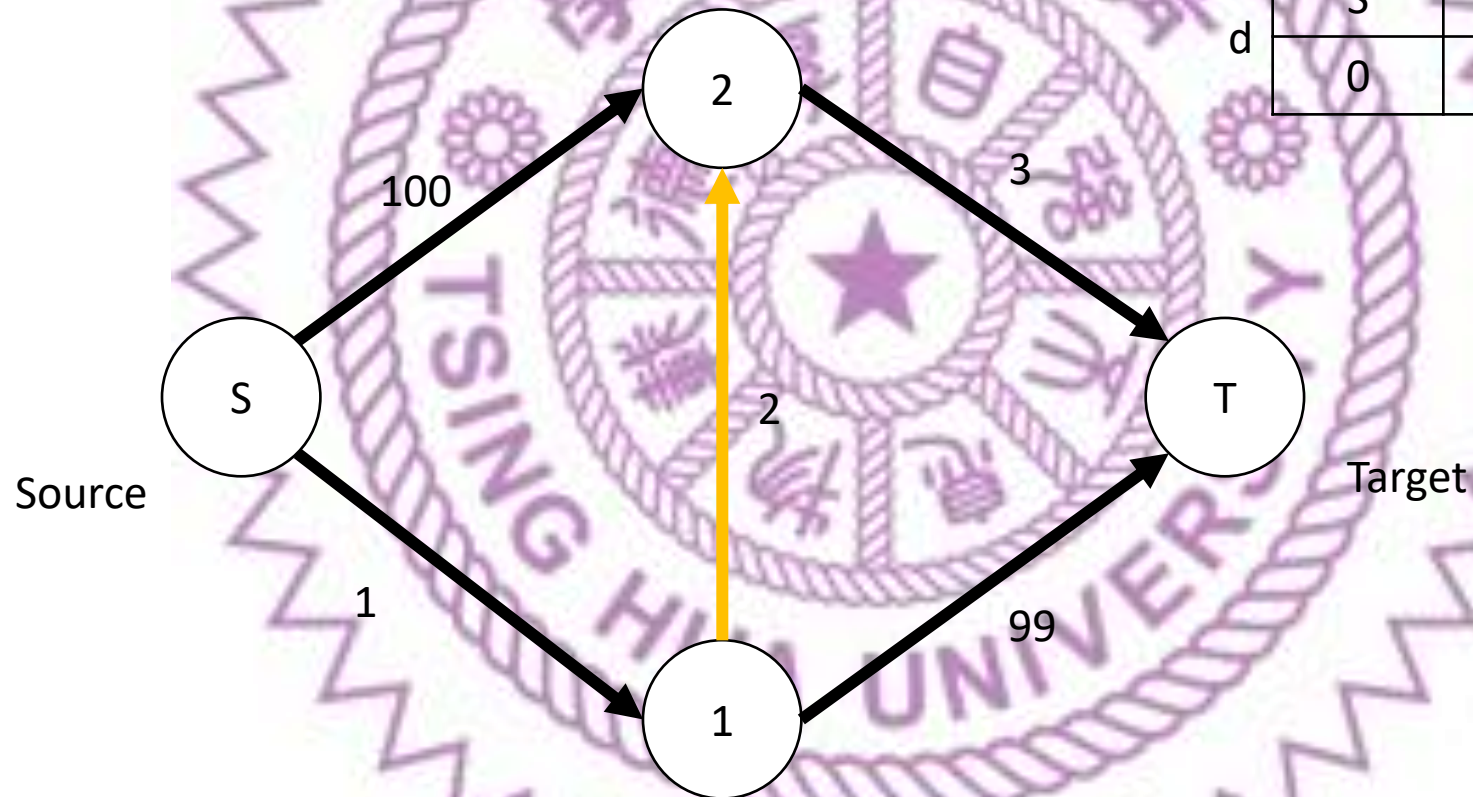
	S	1	2	T
d	0	∞	∞	∞

單源最短路徑



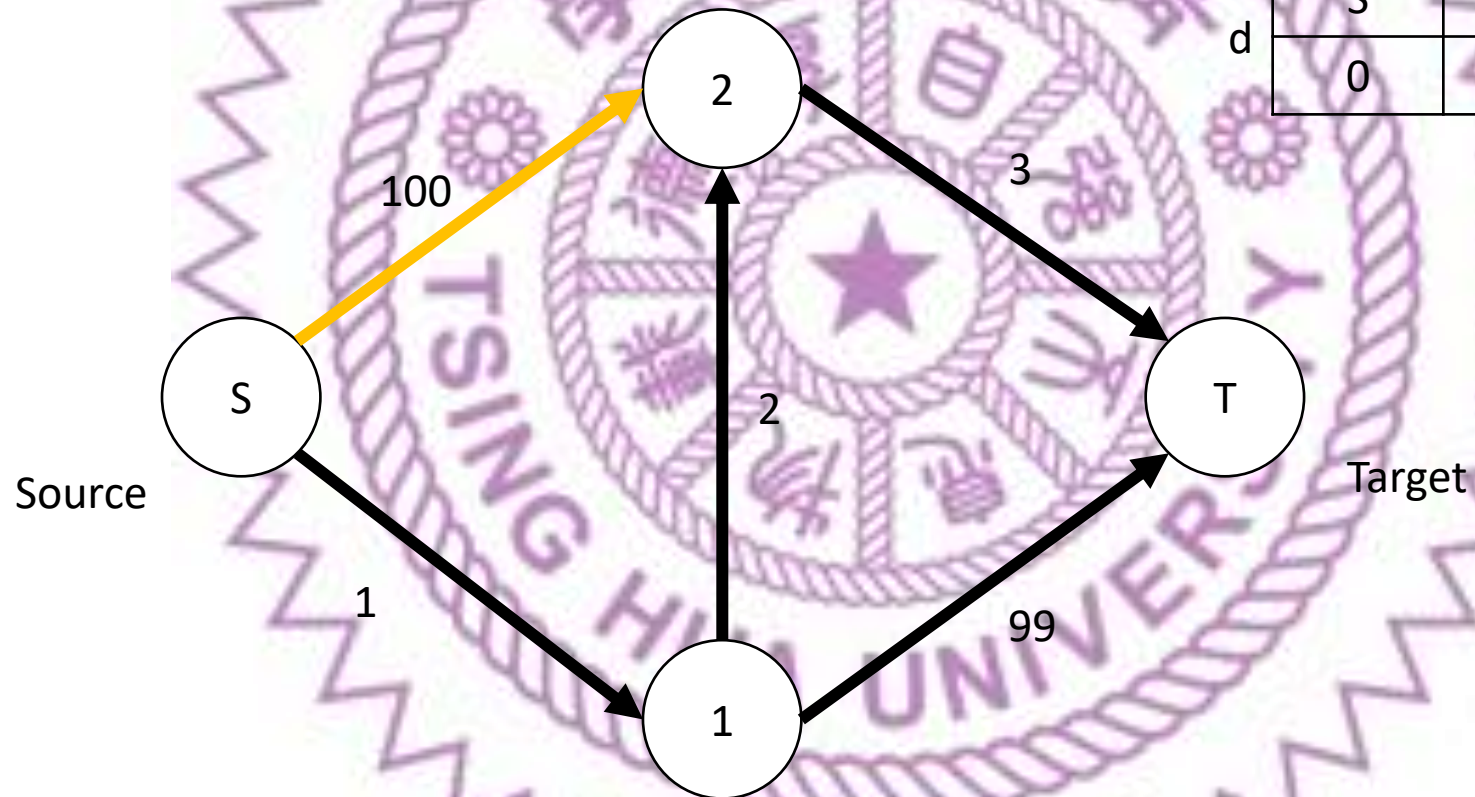
d	S	1	2	T
	0	∞	∞	∞

單源最短路徑



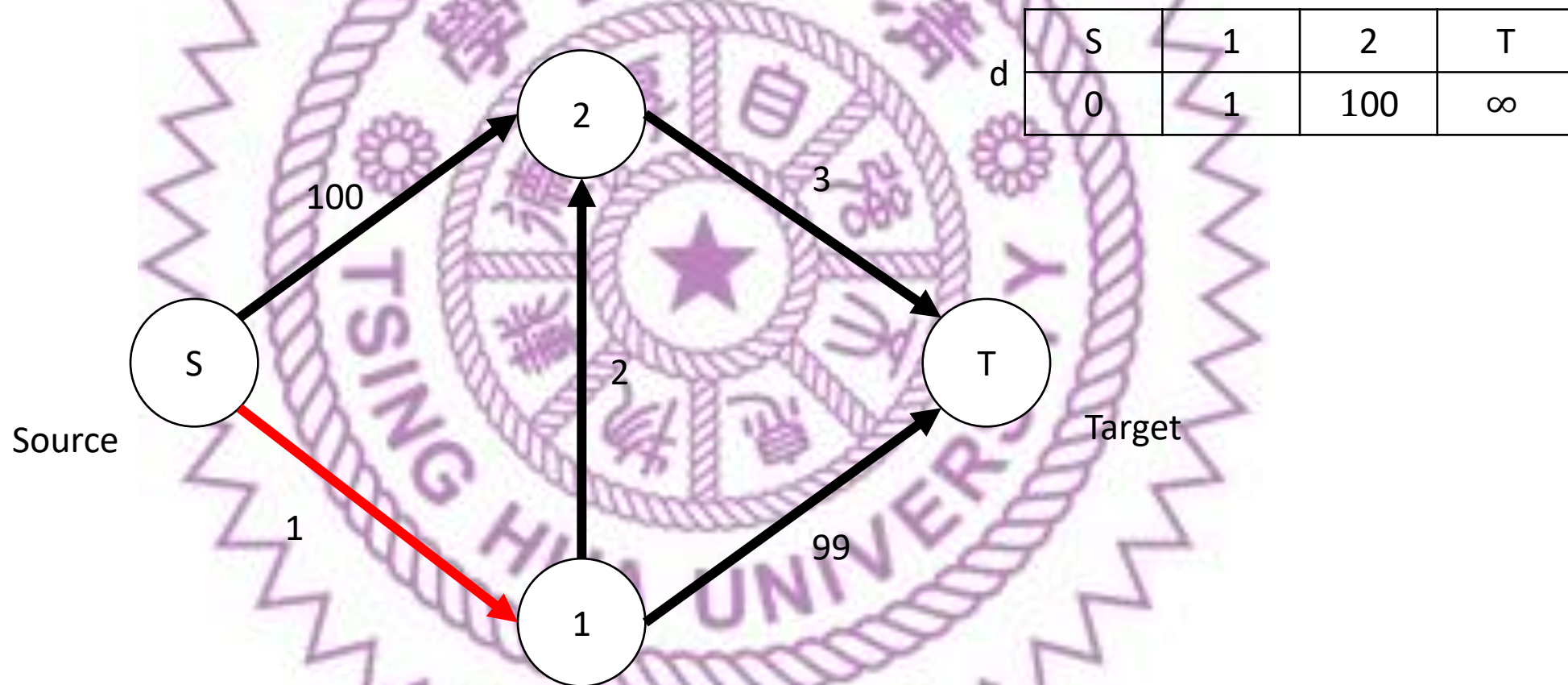
	S	1	2	T
d	0	∞	∞	∞

單源最短路徑



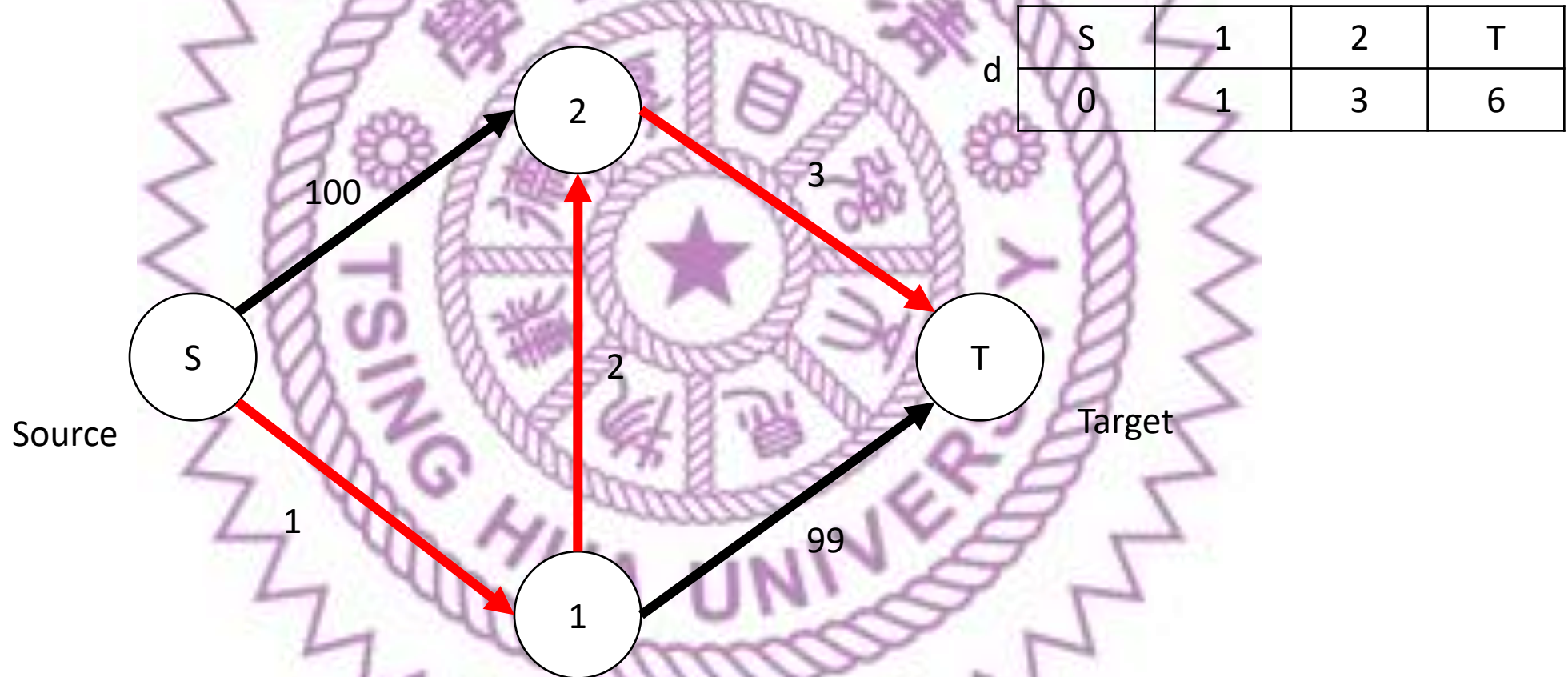
	S	1	2	T
d	0	∞	100	∞

單源最短路徑



最差情況下
枚舉完所有邊可以保證有一個點的最短路徑被找到

單源最短路徑



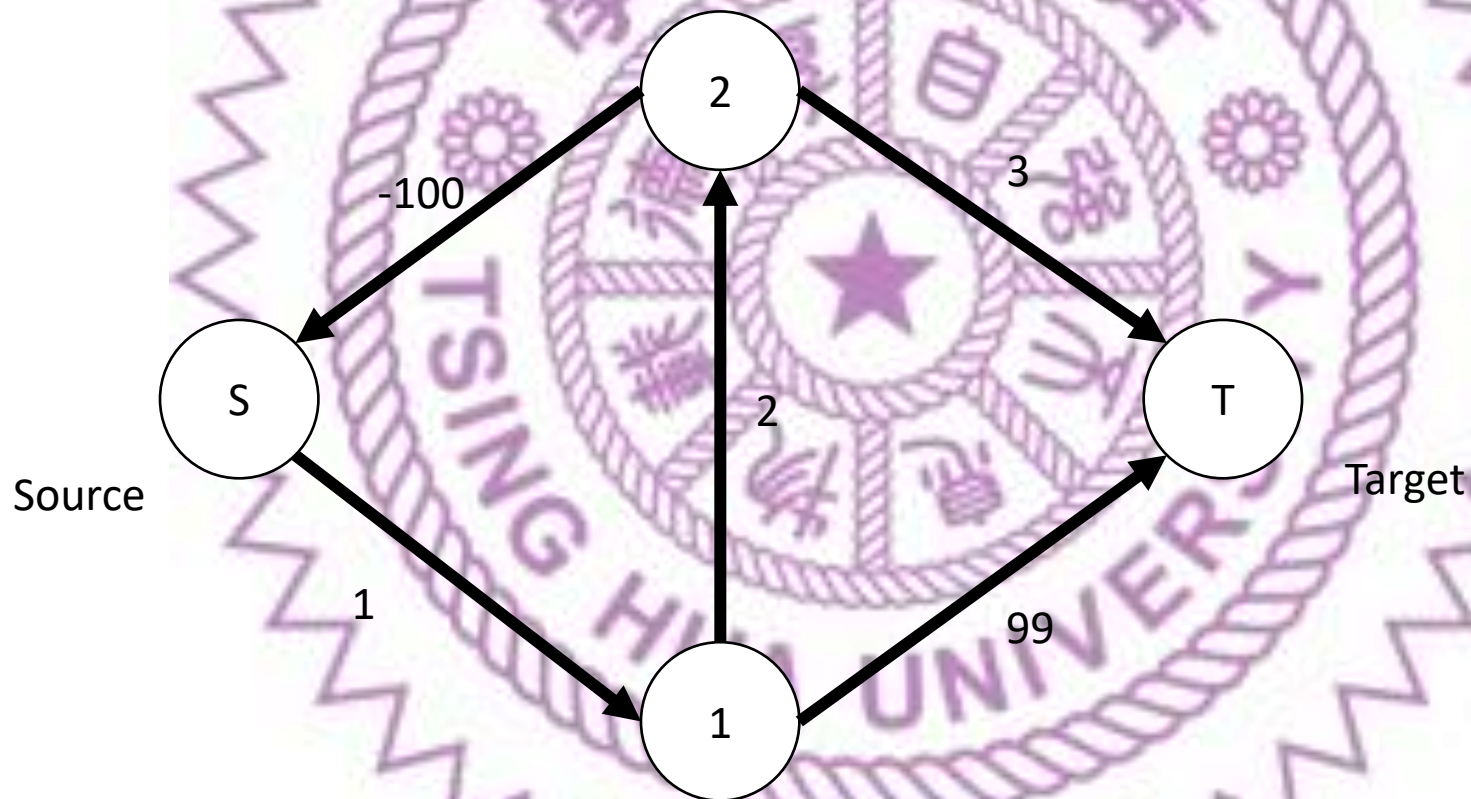
最短路徑只有 $n-1$ 條邊(最短路徑樹)

Bellman Ford 最差複雜度

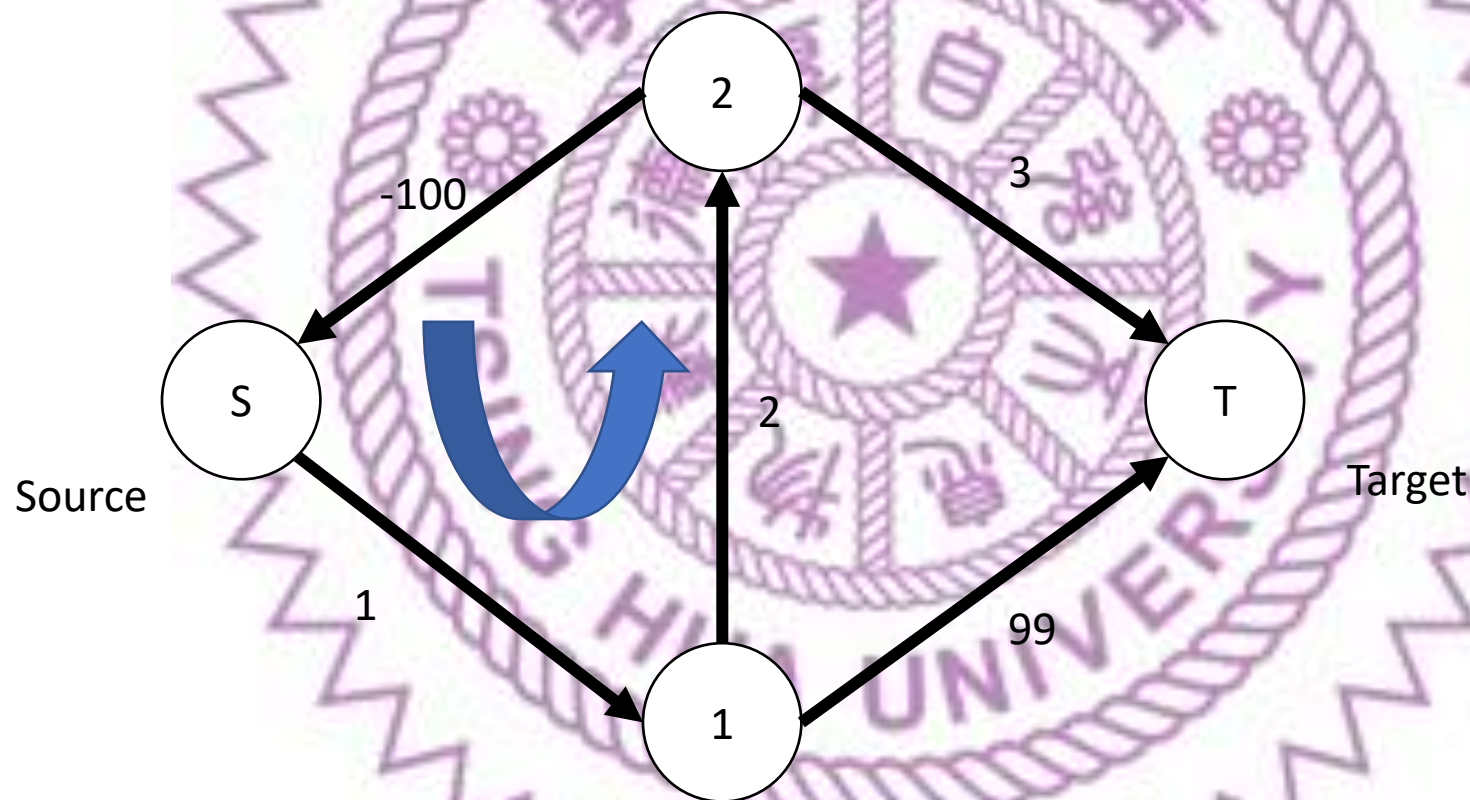
- 最短路徑的邊數 \times 枚舉所有邊的時間
 - $(|V| - 1) \times |E|$
- $O(|V||E|)$

```
vector<long long> bellman_ford(vector<Edge> E, int n, int S) {  
    vector<long long> d(n, INF); // 假設點的編號為 0 ~ n-1  
    d[S] = 0; // 起點設 0  
    auto relax = [&](Edge e) { ... };  
    for (int t = 1; t <= n - 1; ++t) {  
        for (auto &e : E) relax(e);  
    }  
    return d;  
}
```


等等，這樣會發生甚麼事



只要一直繞路徑長就會變成負無限大



改良算法

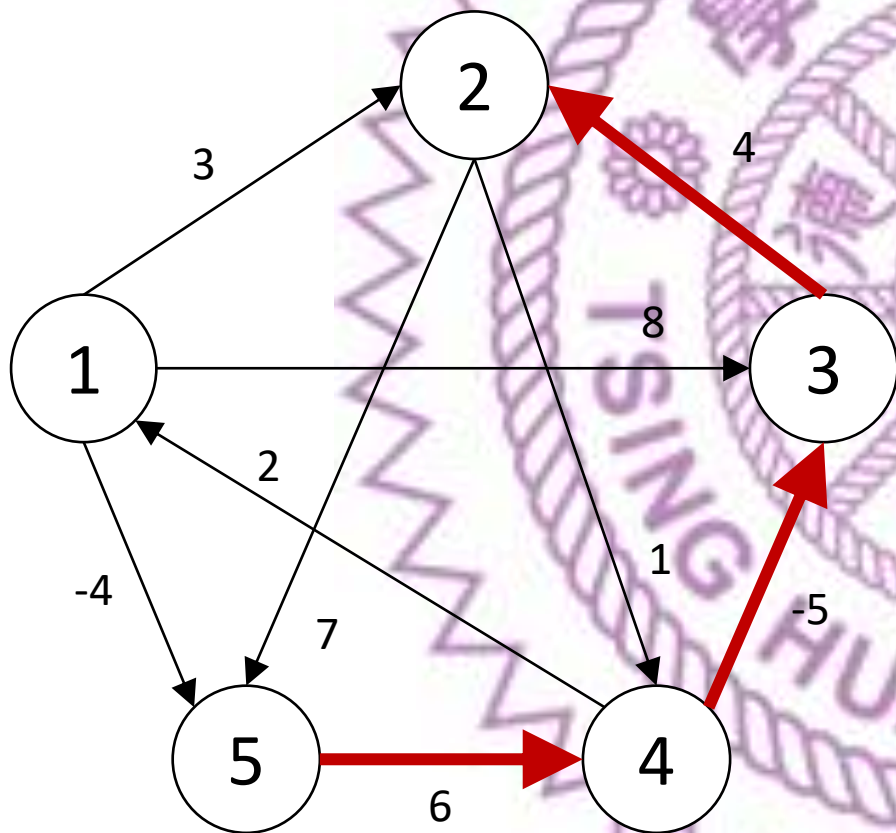
```
vector<long long> bellman_ford(vector<Edge> E, int n, int S) {  
    vector<long long> d(n, INF); // 假設點的編號為 0 ~ n-1  
    d[S] = 0; // 起點設 0  
    auto relax = [&](Edge e) { ... };  
    for (int t = 1; t <= n; ++t) {  
        bool update = false;  
        for (auto &e : E)  
            update |= relax(e);  
        if (t == n && update) return {};  
    }  
    return d;  
}
```


Floyd Warshall

全點對最短路徑



目標輸出



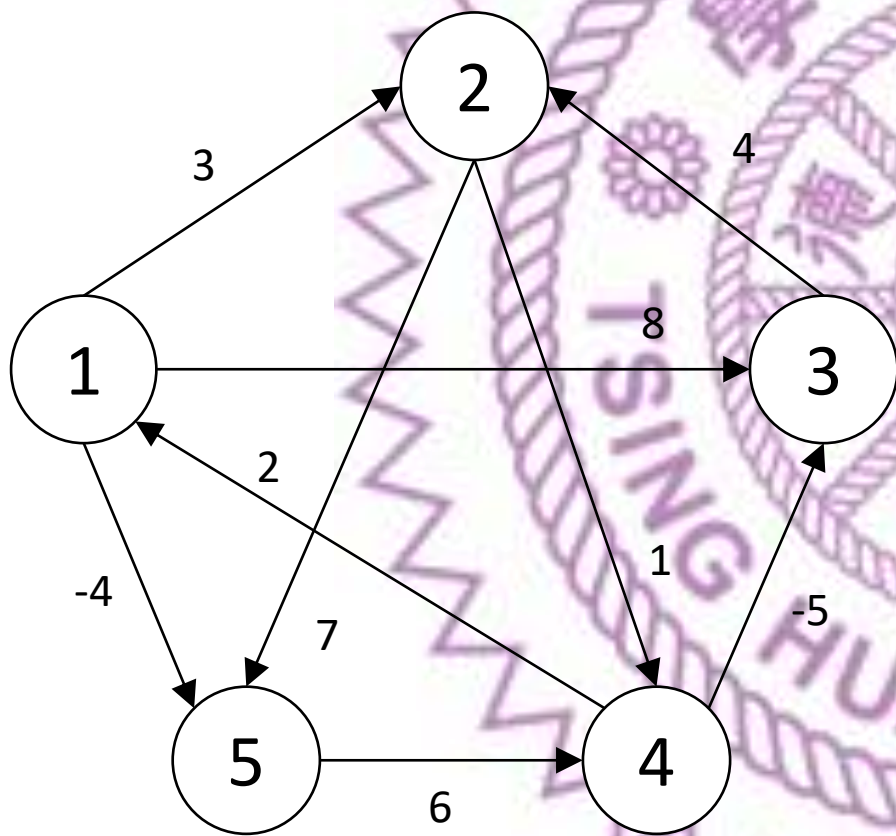
表格 D

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

定義

- 假設點的編號為 $0 \sim n - 1$
- $D(k, u, v)$:
從 u 點走到 v 點，途中能經過的點編號都小於 k
這種情況下的最短路徑
- $D(0, u, v)$:
 - 如果 u, v 有邊聯通 $\rightarrow (u, v)$ 的權重
 - Else ∞

輸入格式 – 直接用 Adjacency Matrix

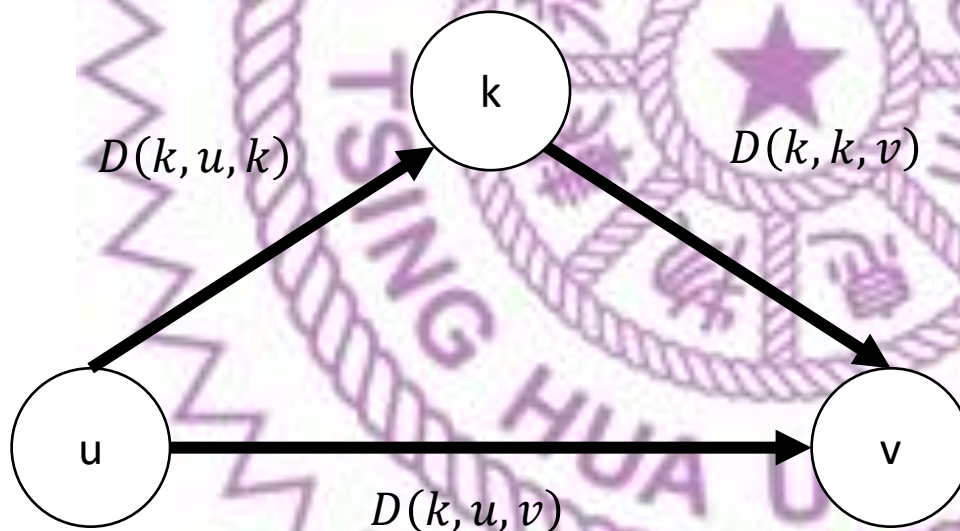


表格 D_0

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

計算 $D(k + 1, u, v)$

- $D(k + 1, u, v) = \min(D(k, u, v), D(k, u, k) + D(k, k, v))$



造著公式寫

```
for (int k = 0; k < n; ++k)
  for (int u = 0; u < n; ++u)
    for (int v = 0; v < n; ++v)
      D[k + 1][u][v] = min(D[k][u][v], D[k][u][k] + D[k][k][v]);
```


空間優化

```
void floyd_warshall(vector<vector<long long>> &D) {  
    int n = D.size(); // 假設點的編號為 0 ~ n-1  
    for (int k = 0; k < n; ++k)  
        for (int u = 0; u < n; ++u)  
            for (int v = 0; v < n; ++v)  
                D[u][v] = min(D[u][v], D[u][k] + D[k][v]);  
}
```

被封印的第四演算法 – SPFA

- Shortest Path Faster Algorithm
 - 中國人在 1994 年取的名子
- Algorithm D
 - Edward F. Moore 在 1959 年發表
- 本質上是 Bellman Ford，所以支援負邊
- 「平均」時間 $O(|E|)$
- 但能構造出一個稀疏圖但需要 $O(|V|^2)$ 的例子
- 各位請自行學習，斟酌使用，進階課程有機會用到

SPFA

```
vector<long long> spfa(vector<vector<pair<int, int>>> G, int S) {  
    int n = G.size(); // 假設點的編號為 0 ~ n-1  
    vector<long long> d(n, INF);  
    vector<bool> in_queue(n, false);  
    vector<int> cnt(n, 0);  
    queue<int> Q;  
    d[S] = 0;  
    auto enqueue = [&](int u) {  
        in_queue[u] = true; Q.emplace(u);  
    };  
    enqueue(S);  
    while (Q.size()) {  
        int u = Q.front(); Q.pop();  
        in_queue[u] = false;  
        for (auto [v, cost] : G[u])  
            if (d[v] > d[u] + cost) {  
                if (++cnt[u] >= n) return {}; // 存在負環  
                d[v] = d[u] + cost;  
                if (!in_queue[v]) enqueue(v);  
            }  
    }  
    return d;  
}
```