Advanced Competitive Programming

國立成功大學ACM-ICPC程式競賽培訓隊
nckuacm@imslab.org

Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan

Outline

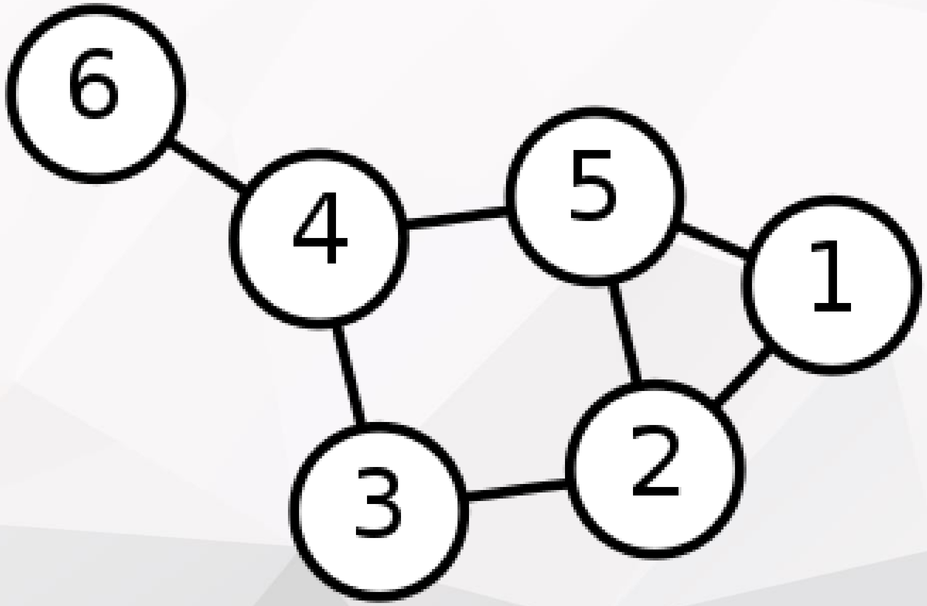
- 術語複習
  - Graph
  - Tree
- 最小生成樹
- A* 搜尋法則
- 單源最短路徑
- 全點對最短路徑

Outline

- 術語複習
  - Graph
  - Tree
- 最小生成樹
- A* 搜尋法則
- 單源最短路徑
- 全點對最短路徑

Graph

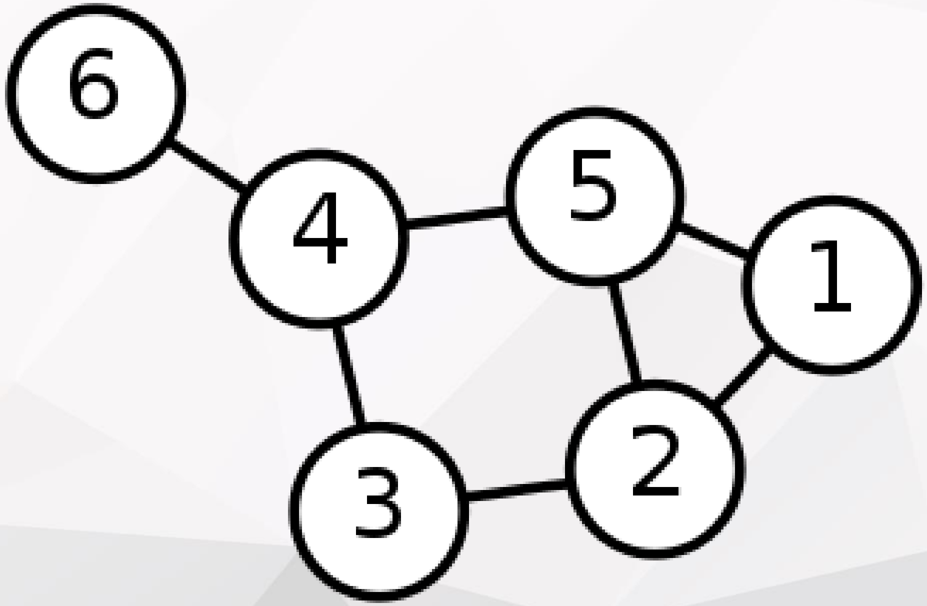
- 圖 (Graph)，是一個由邊 (Edge) 集合與點 (Vertex) 集合所組成的資料結構。



Graph

- 圖 (Graph)，是一個由邊 (Edge) 集合與點 (Vertex) 集合所組成的資料結構。

- G = (E, V) 為圖
- E 為邊集合
- V 為點集合



Graph

- 點 (vertex)： 組成圖的最基本元素

Graph

- 點 (vertex)： 組成圖的最基本元素
- 邊 (edge)： 點與點的關係
  - 通常用 u 表示邊的起端，v 表示邊的終端

Graph

- 點 (vertex)： 組成圖的最基本元素
- 邊 (edge)： 點與點的關係
  - 通常用 u 表示邊的起端，v 表示邊的終端
- 有向圖 (directed graph)： 邊帶有方向性
  - u -> v，表示 u 能走到 v，但 v 不保證走到 u

Graph

- 點 (vertex)： 組成圖的最基本元素
- 邊 (edge)： 點與點的關係
  - 通常用 u 表示邊的起端，v 表示邊的終端
- 有向圖 (directed graph)： 邊帶有方向性
  - u -> v，表示 u 能走到 v，但 v 不保證走到 u
- 無向圖 (undirected graph)： 每條邊都是雙向的
  - u <-> v，表示 v 能到 u，u 能到 v

Graph

- 道路 (walk)： 點邊相間的序列

Graph

- 道路 (walk)： 點邊相間的序列
  e.g. $v_0e_1v_1e_2v_2 \ldots e_nv_n$

Graph

- 道路 (walk)： 點邊相間的序列
  e.g. $v_0e_1v_1e_2v_2...e_nv_n$
- 路徑 (path)： 點不重複的**道路**

Graph

- 道路 (walk)： 點邊相間的序列
  e.g. $v_0e_1v_1e_2v_2 \ldots e_nv_n$
- 路徑 (path)： 點不重複的道路
- 環 (cycle)： 把路徑的**起**點與**終**點連接起來

Graph

- 道路 (walk)： 點邊相間的序列，
  e.g. $v_0e_1v_1e_2v_2\ldots e_nv_n$
- 路徑 (path)： 點不重複的道路
- 環 (cycle)： 把路徑的**起**點與**終**點連接起來
- 走訪/遍歷 (traversal/search)： 走完全部的點或邊

該怎麼表示一張圖呢

Graph 鄰接矩陣

- 用二維陣列表達點與點有無**邊關係**
  - E[u][v] = 1 表示 u 與 v 間有邊
  - E[u][v] = 0 表示 u 與 v 間沒邊

Graph 鄰接矩陣

- 用二維陣列表達點與點有無**邊關係**
  - E[u][v] = 1 表示 u 與 v 間有邊
  - E[u][v] = 0 表示 u 與 v 間沒邊

- 用特殊的值來表示**無法到達**的情況
  - 例如 INT_MAX 或是 -1 或是 INF

Graph 鄰接表

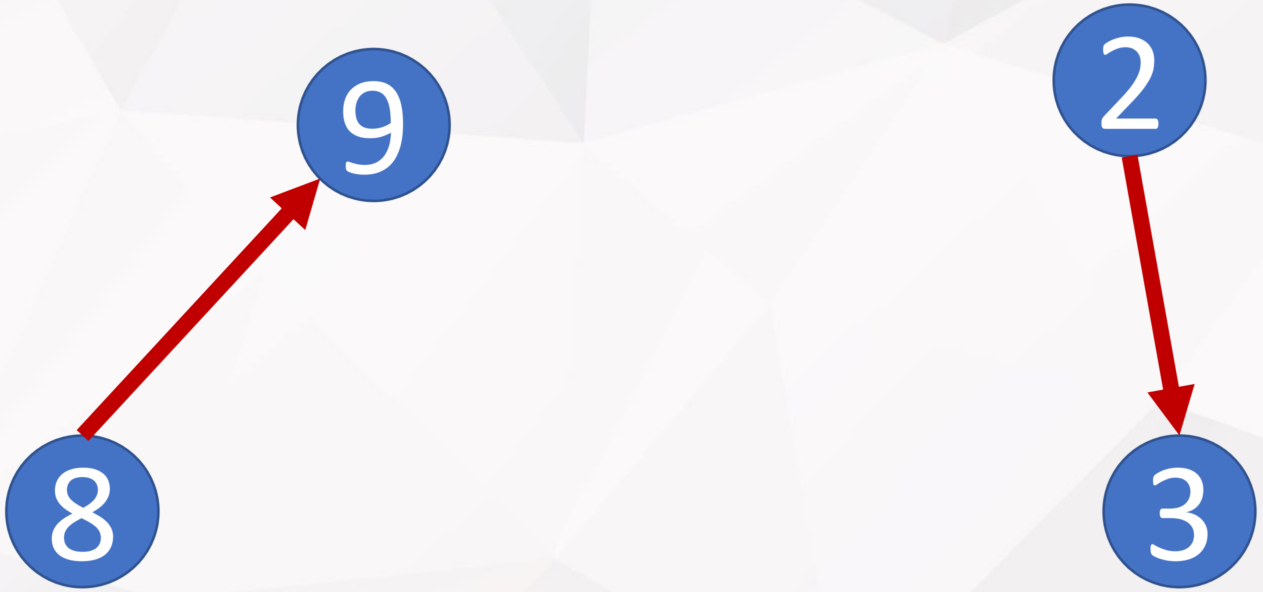
```
vector<int> E[maxv];

to = E[from][0];
```

Graph 鄰接表

```
E[2][0] = 3;
```

Graph 鄰接表

```
E[2][0] = 3;
E[8][0] = 9;
```

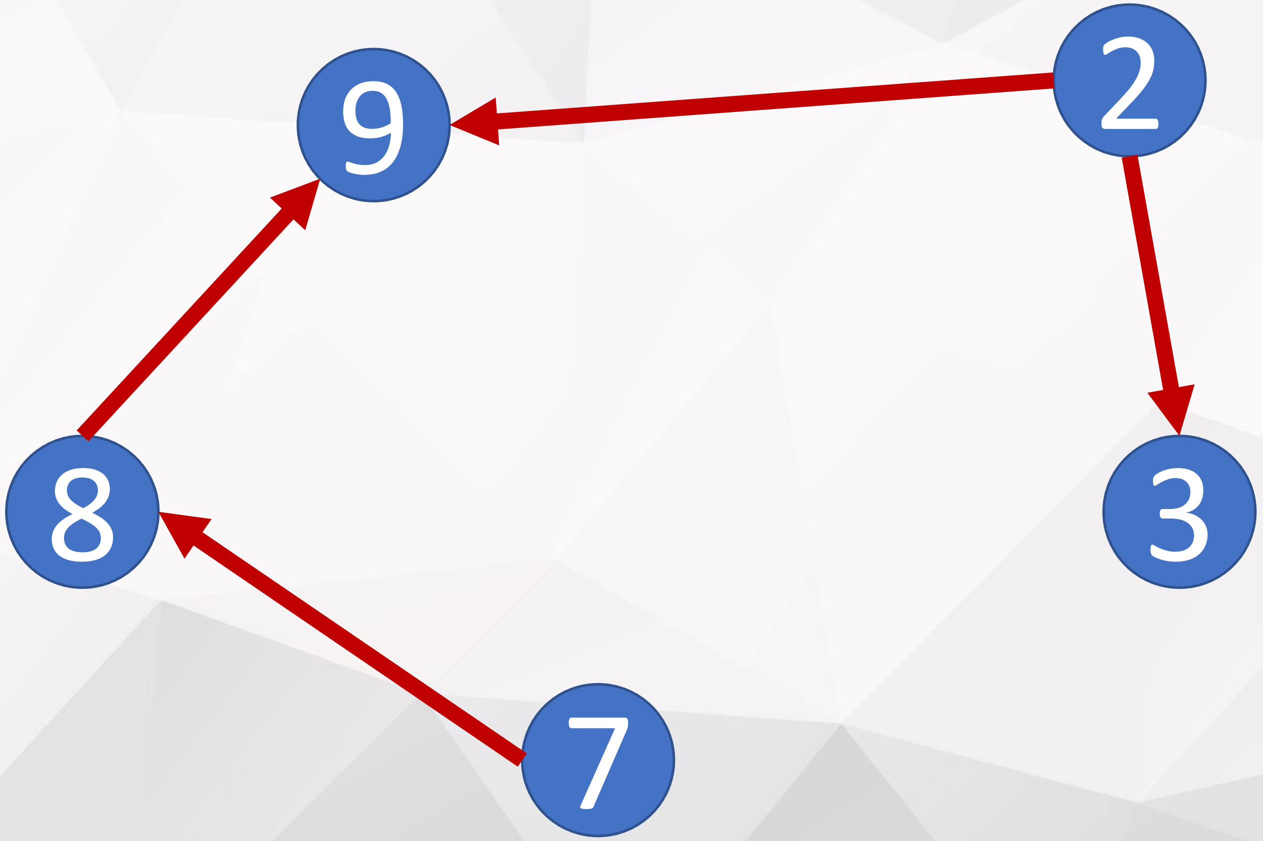


Graph 鄰接表

```
E[2][0] = 3;
E[8][0] = 9;
E[2][1] = 9;
```

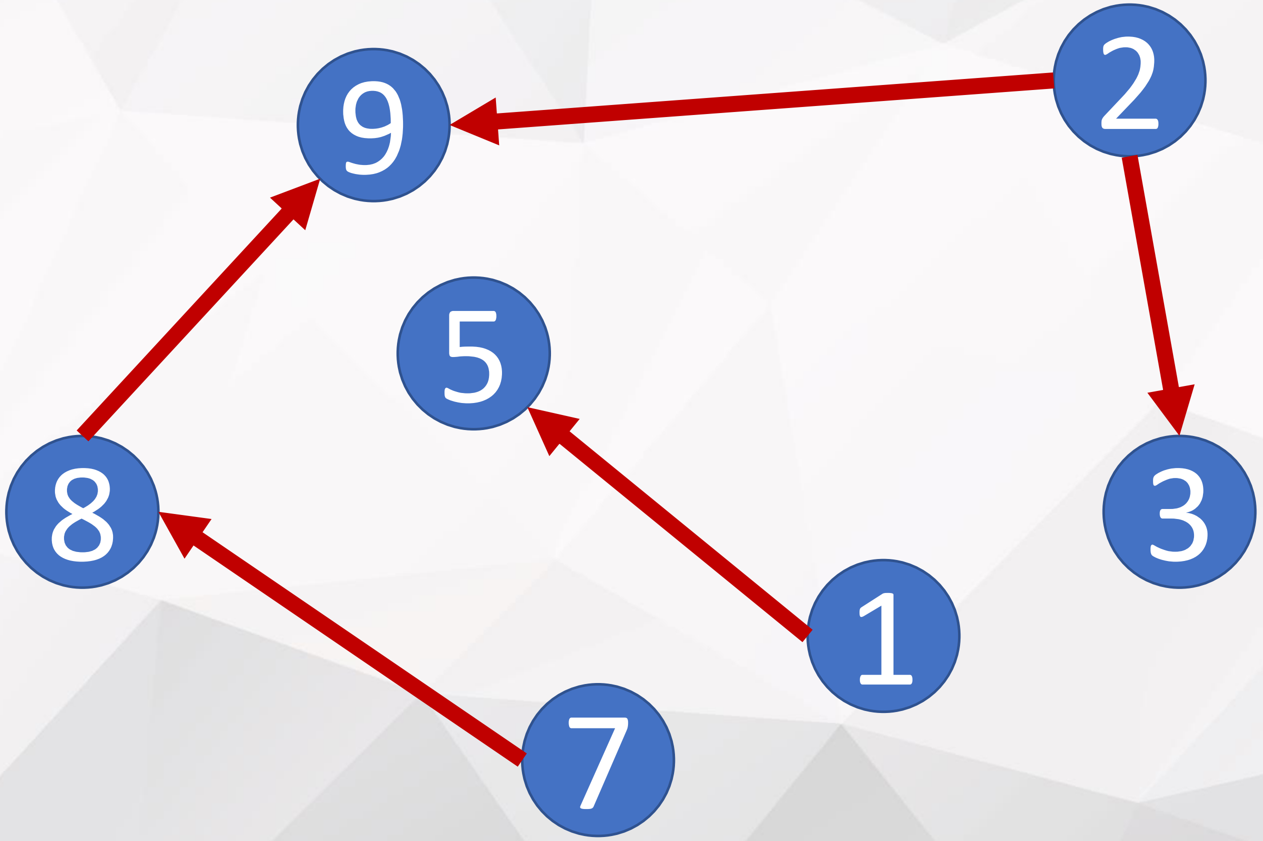
Graph 鄰接表

```
E[2][0] = 3;
E[8][0] = 9;
E[2][1] = 9;
E[7][0] = 8;
```



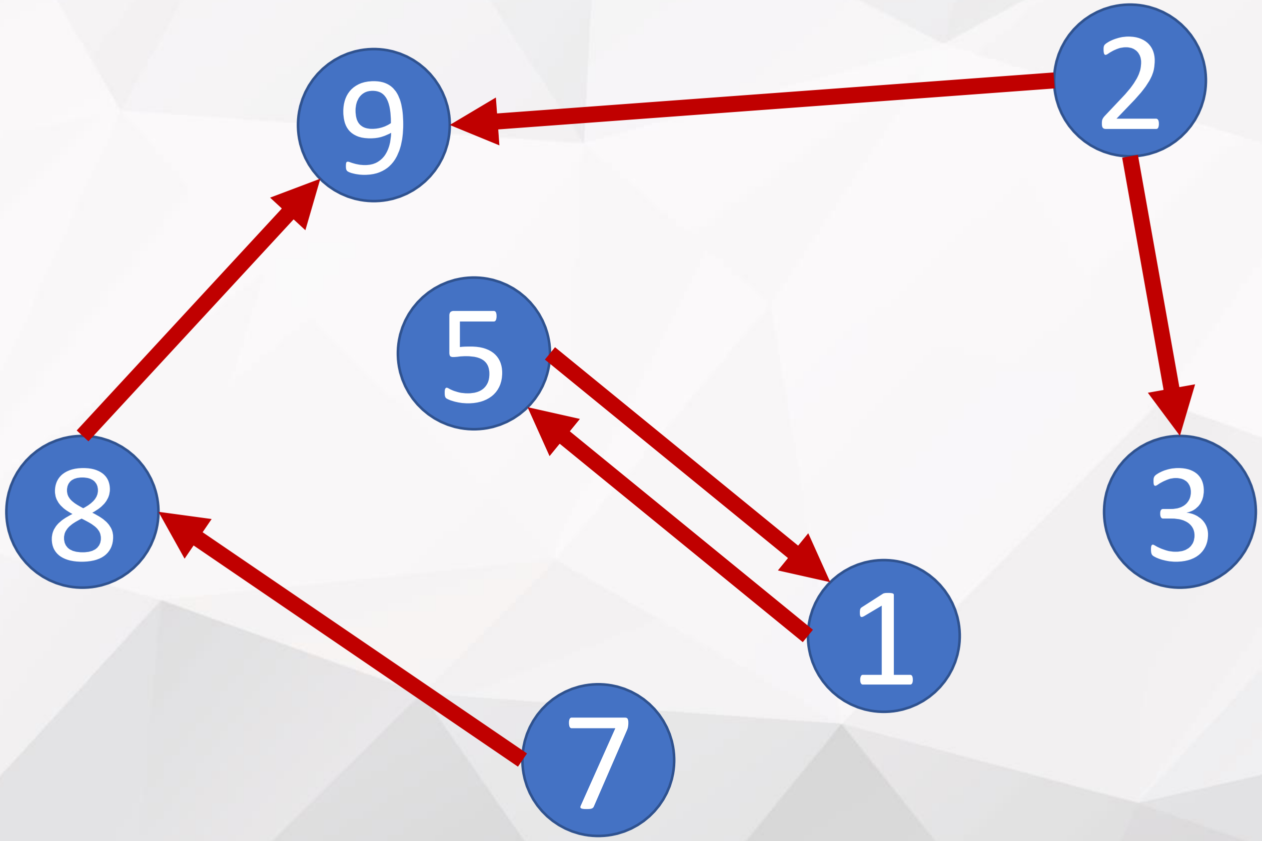
Graph 鄰接表

```
E[2][0] = 3;
E[8][0] = 9;
E[2][1] = 9;
E[7][0] = 8;

E[1][0] = 5;
```



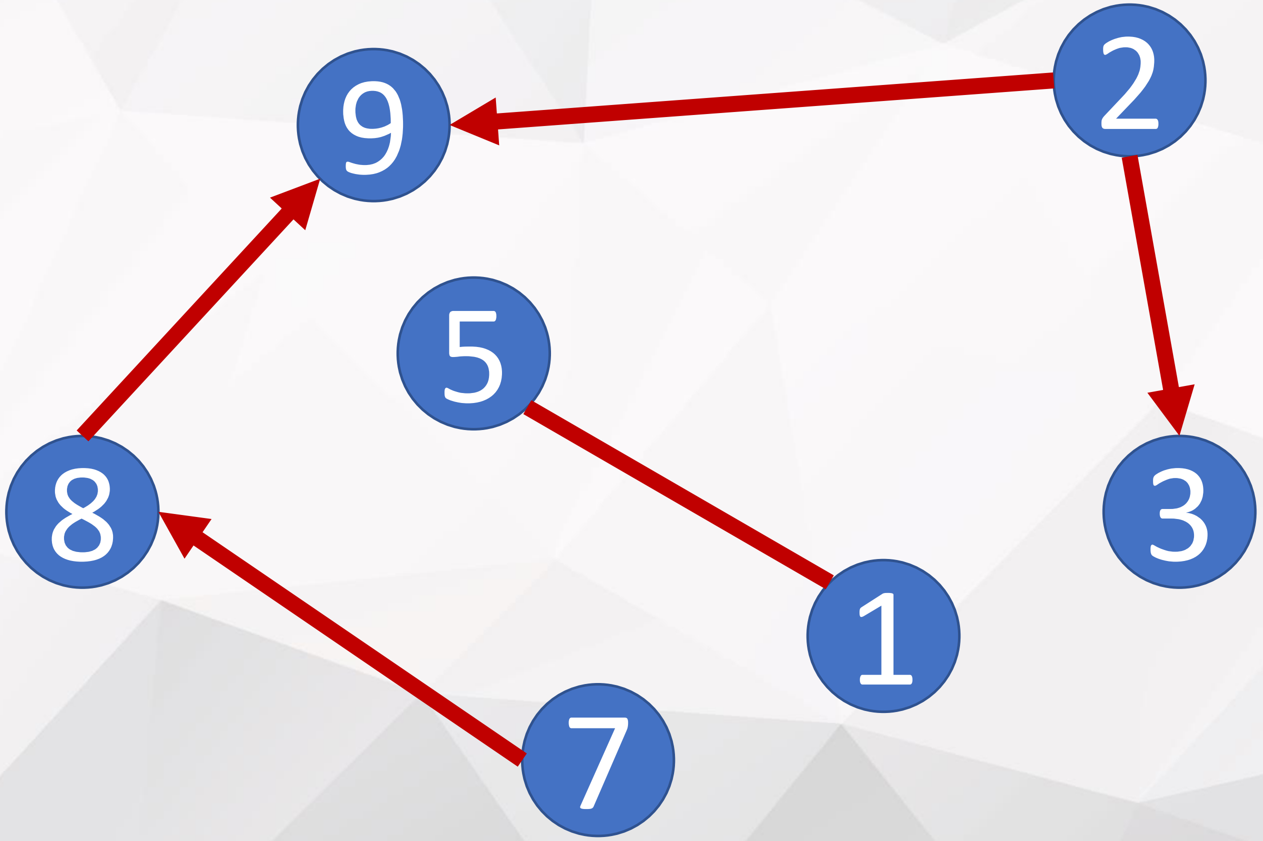
Graph 鄰接表

```
E[2][0] = 3;
E[8][0] = 9;
E[2][1] = 9;
E[7][0] = 8;

E[1][0] = 5;
E[5][1] = 1;
```



Graph 鄰接表

```
E[2][0] = 3;
E[8][0] = 9;
E[2][1] = 9;
E[7][0] = 8;

E[1][0] = 5;
E[5][1] = 1;
```



Tree

- Tree 是一個有向**無環**連通圖

Tree

- Tree 是一個有向無環連通圖
- 節點 (node)： 一般樹上的點

Tree

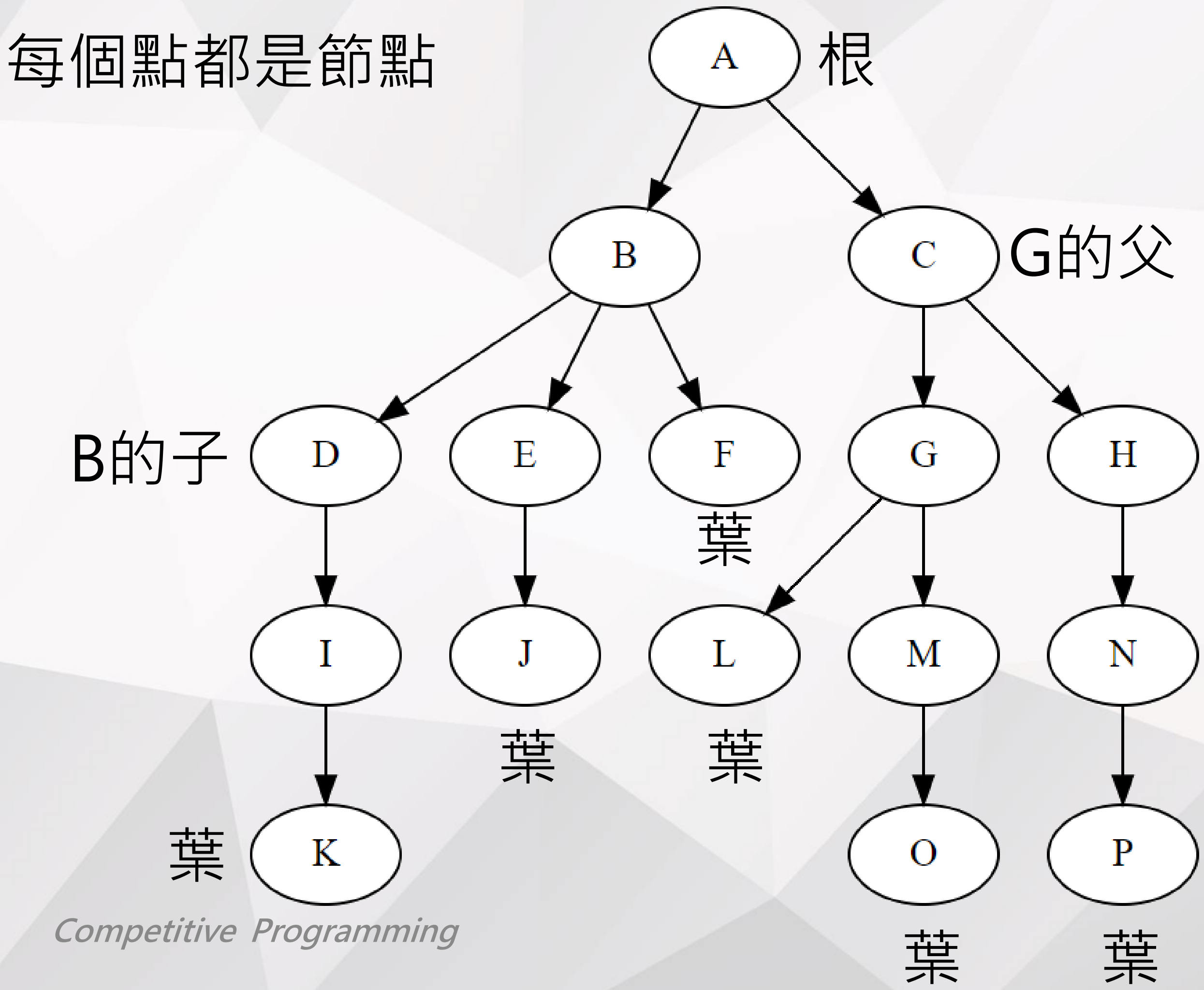
- Tree 是一個有向**無環連通**圖
- 節點 (node)： 一般樹上的點
- 父 (parent)： 節點能**反向**拜訪的**第一**個節點
- 子 (child)： 節點能**正向**拜訪的**第一**個節點

Tree

- Tree 是一個有向**無環連通**圖
- 節點 (node)： 一般樹上的點
- 父 (parent)： 節點能**反向**拜訪的**第一**個節點
- 子 (child)： 節點能**正向**拜訪的**第一**個節點
- 根 (root)： 沒有父節點的節點
- 葉 (leaf)： 沒有子節點的節點

Tree

每個點都是節點



Tree

- 祖先 (ancestor)： 節點能**反向**拜訪的所有節點
- 孫子 (descendant)： 節點能**正向**拜訪的所有節點

Tree

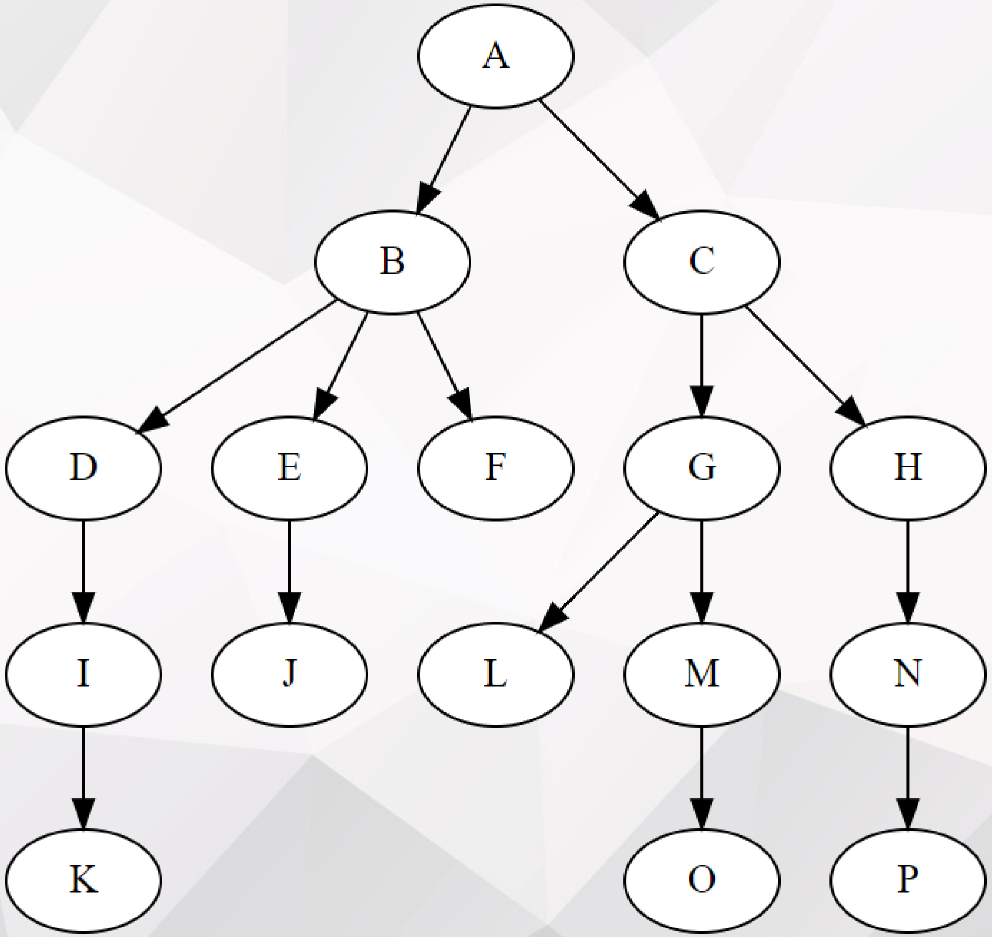
- 祖先 (ancestor)： 節點能反向拜訪的所有節點
- 孫子 (descendant)： 節點能正向拜訪的所有節點
- 深度 (depth)： 從**根**到該節點所**經過的邊數**

Tree

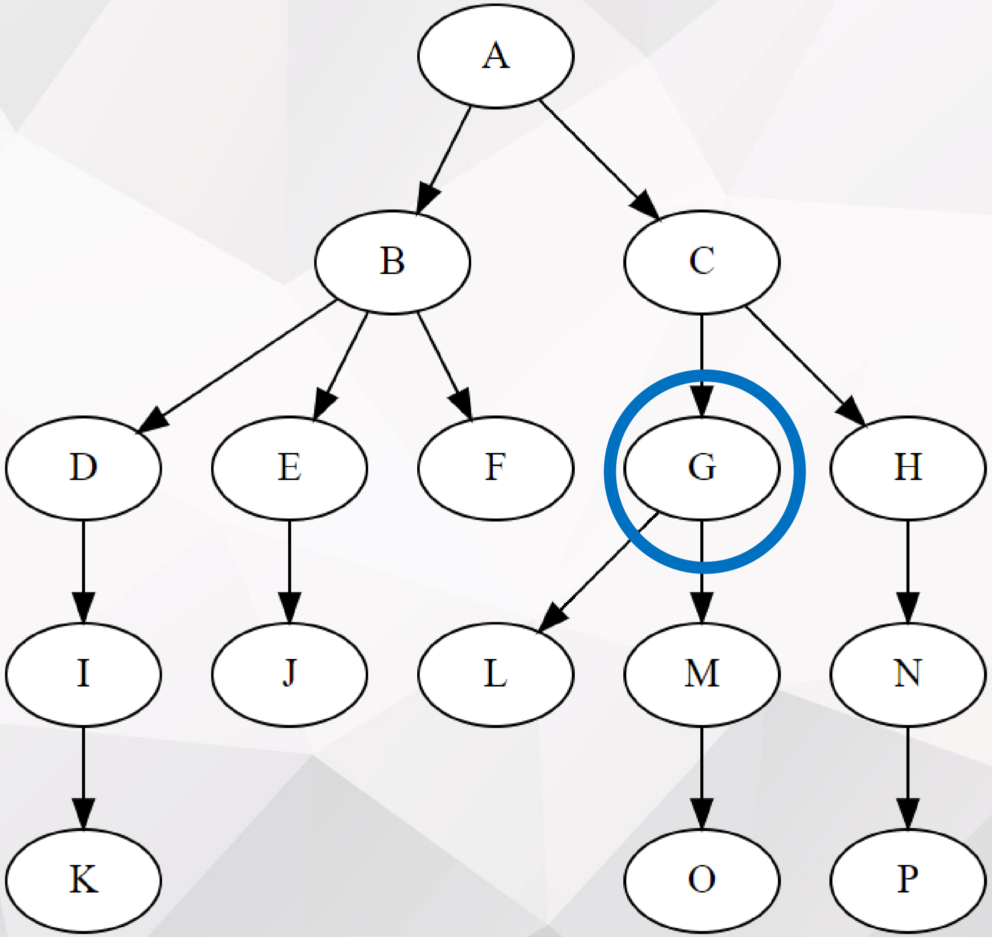
- 祖先 (ancestor)： 節點能反向拜訪的所有節點
- 孫子 (descendant)： 節點能正向拜訪的所有節點
- 深度 (depth)： 從根到該節點所經過的邊數

- 森林 (forest)： 一個集合包含所有**不相交**的 Tree
- 每個非根節點**只有一個**父節點

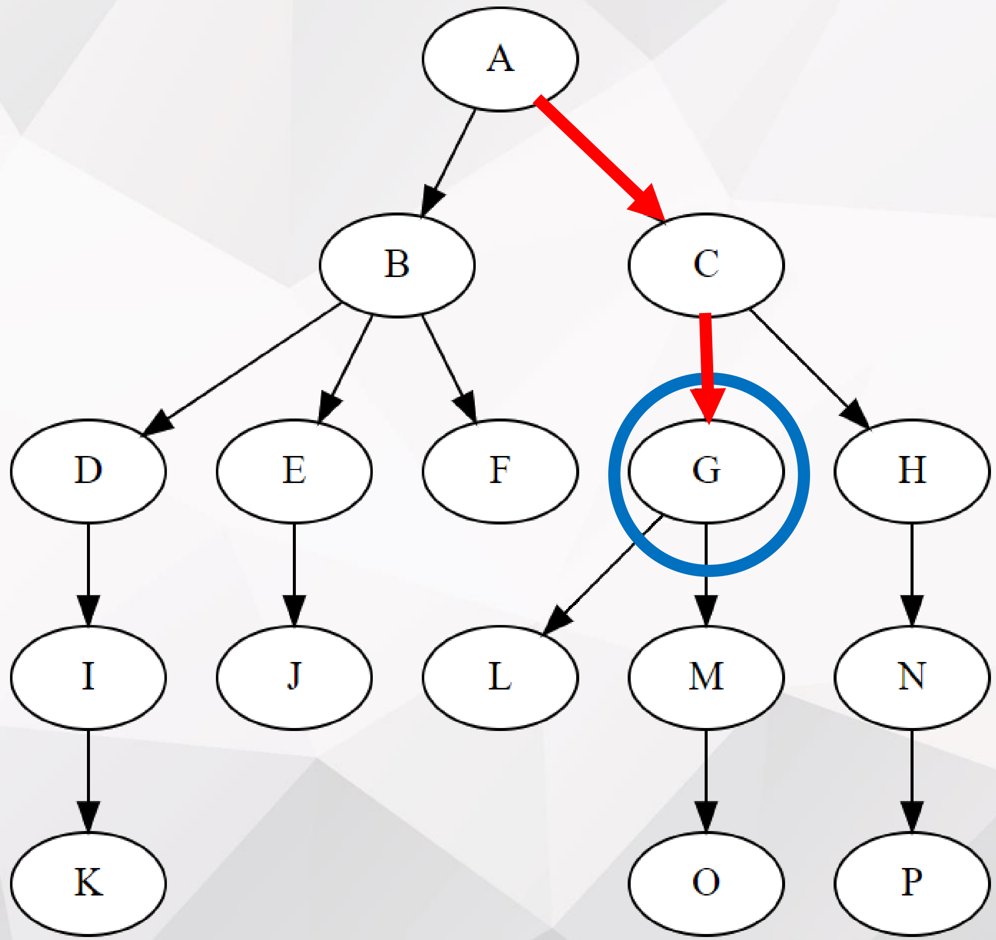
Tree



Tree

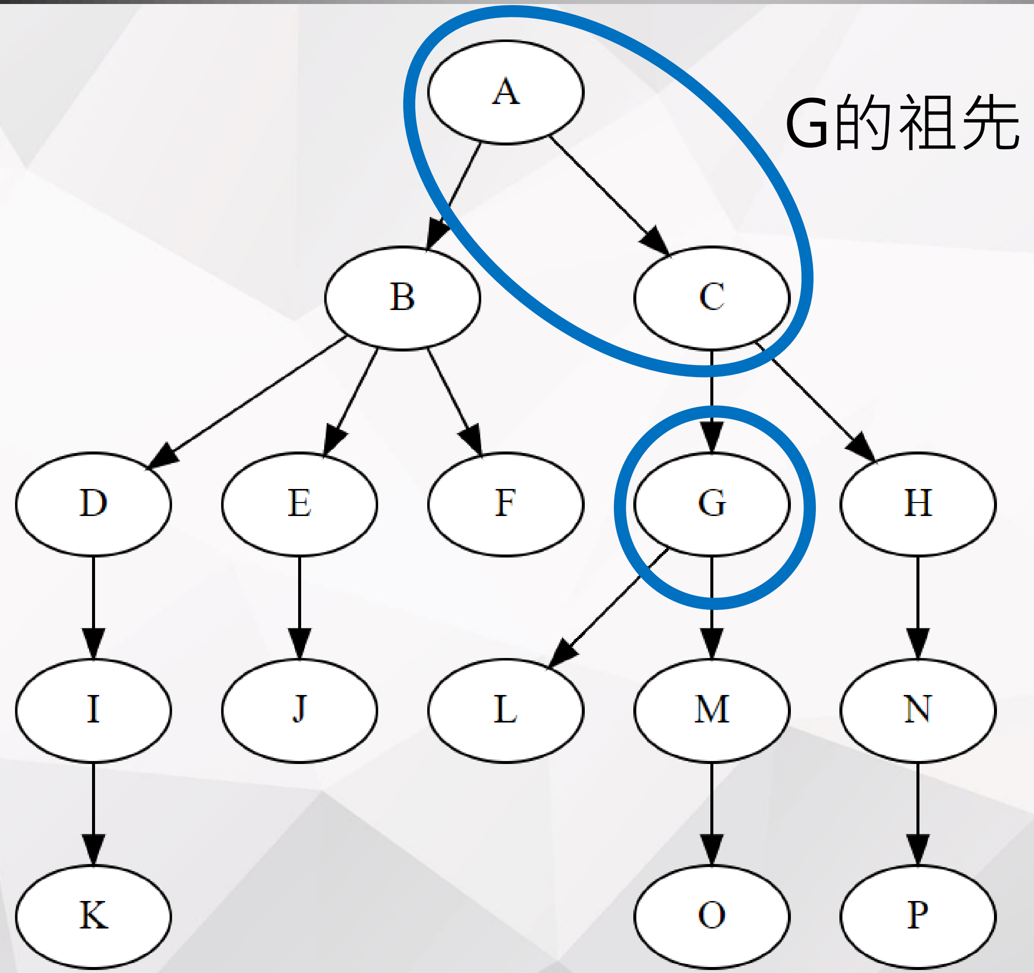


Tree

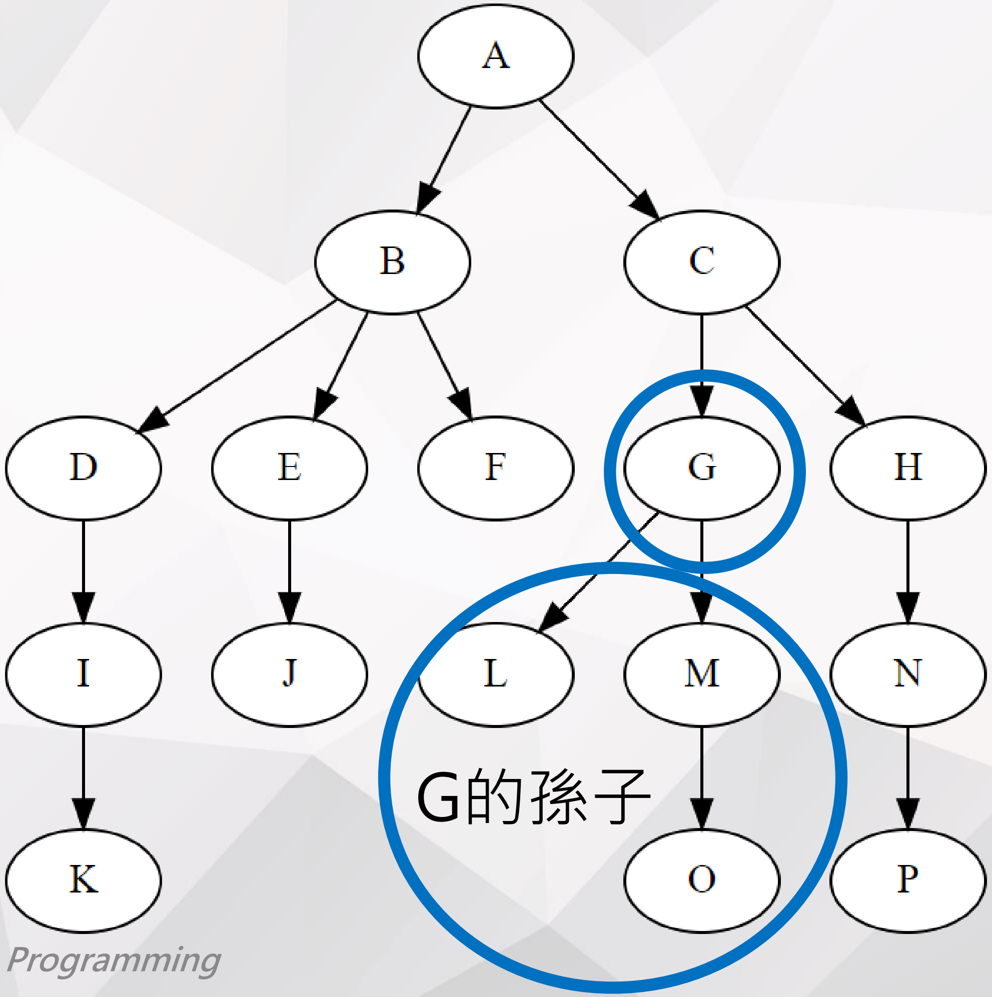


G的Depth為2

Tree



Tree



Outline

- 術語複習
  - Graph
  - Tree
- 最小生成樹
- A* 搜尋法則
- 單源最短路徑
- 全點對最短路徑

Minimum spanning tree

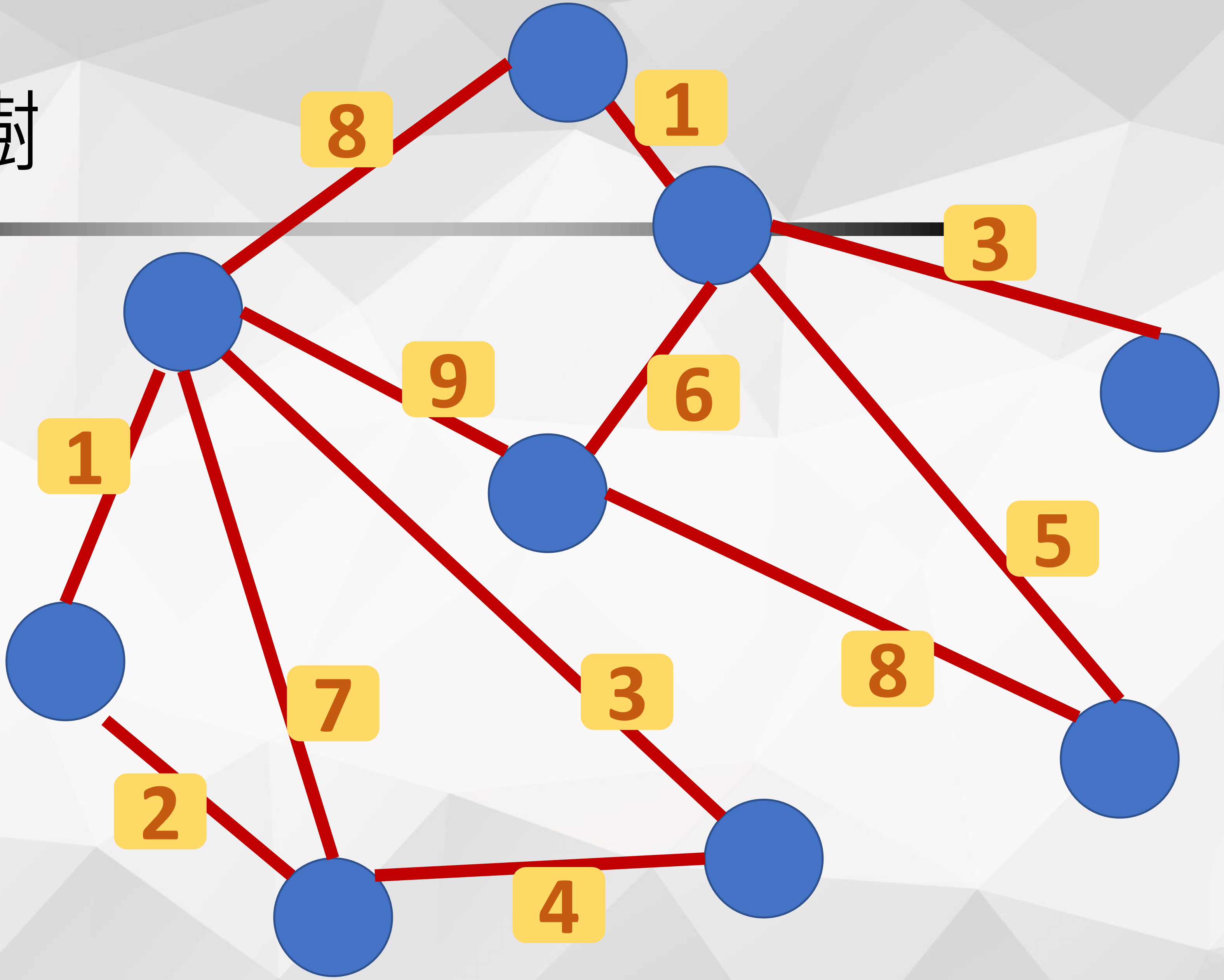
Competitive Programming

Made by 培訓團隊群 CCNS.

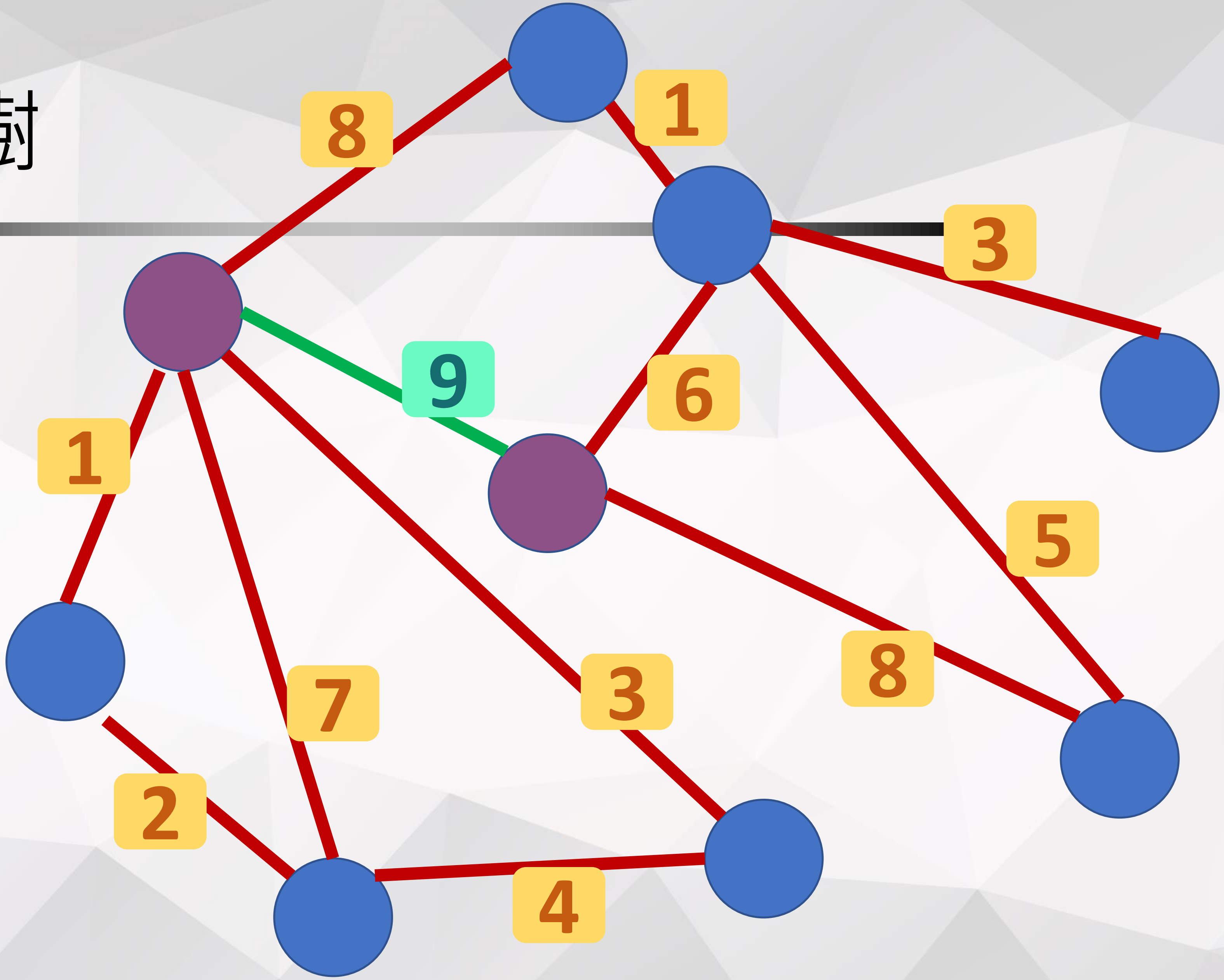
生成樹

- 給定**連通圖** G = (E, V)
- 使用 E 子集**連接所有的點** (屬於 V) 所得到的**樹**

生成樹



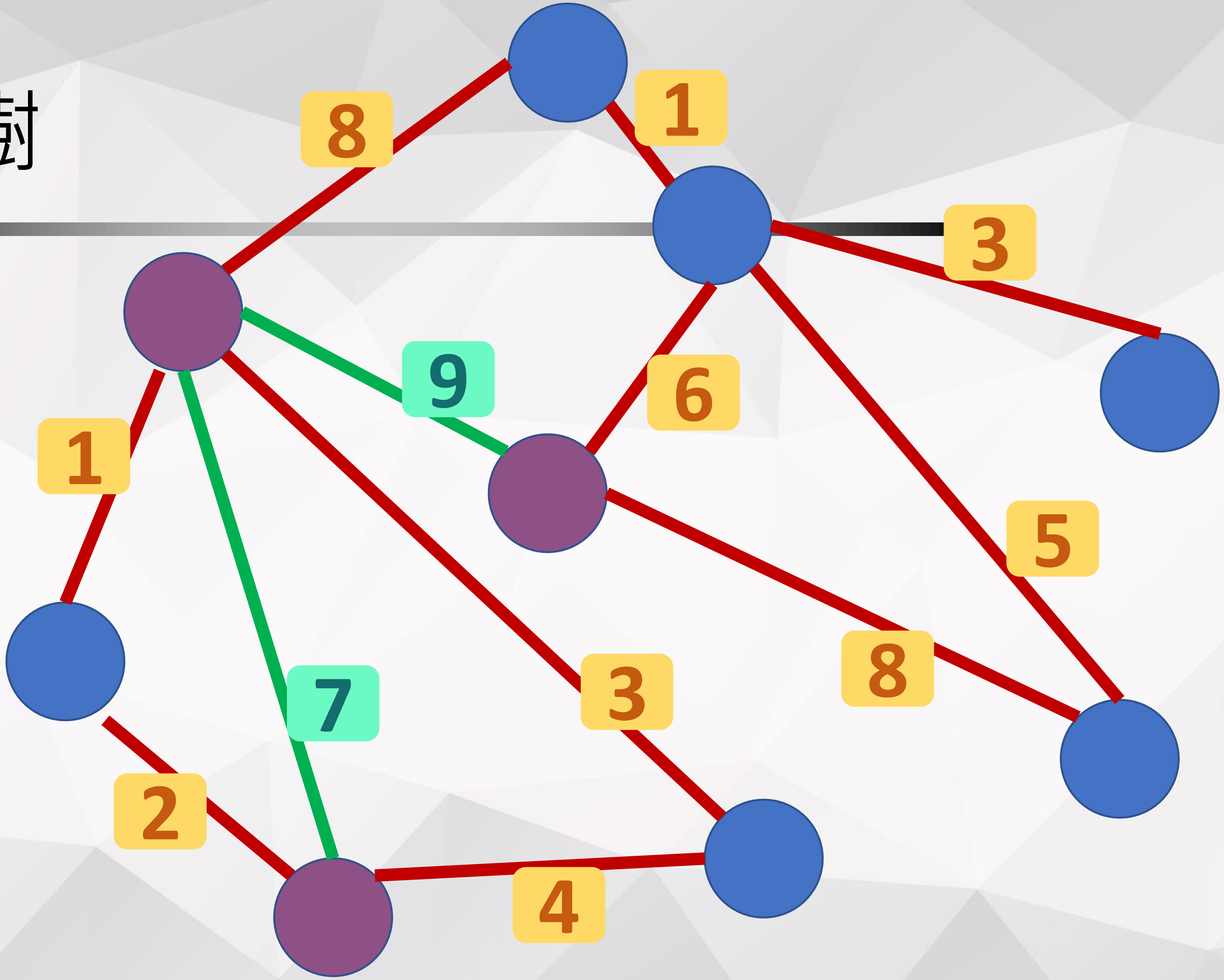
生成樹



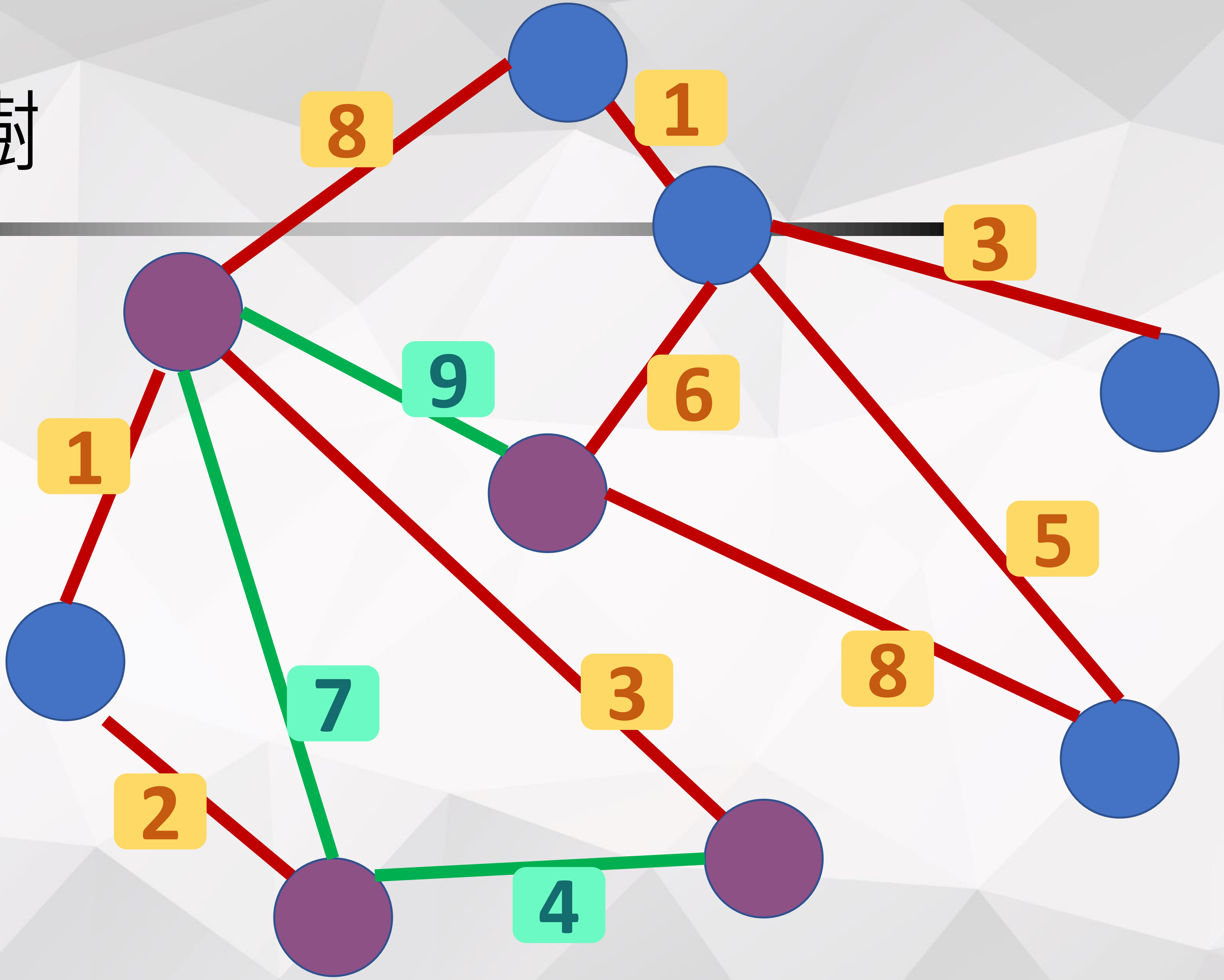
9

Competitive Programming

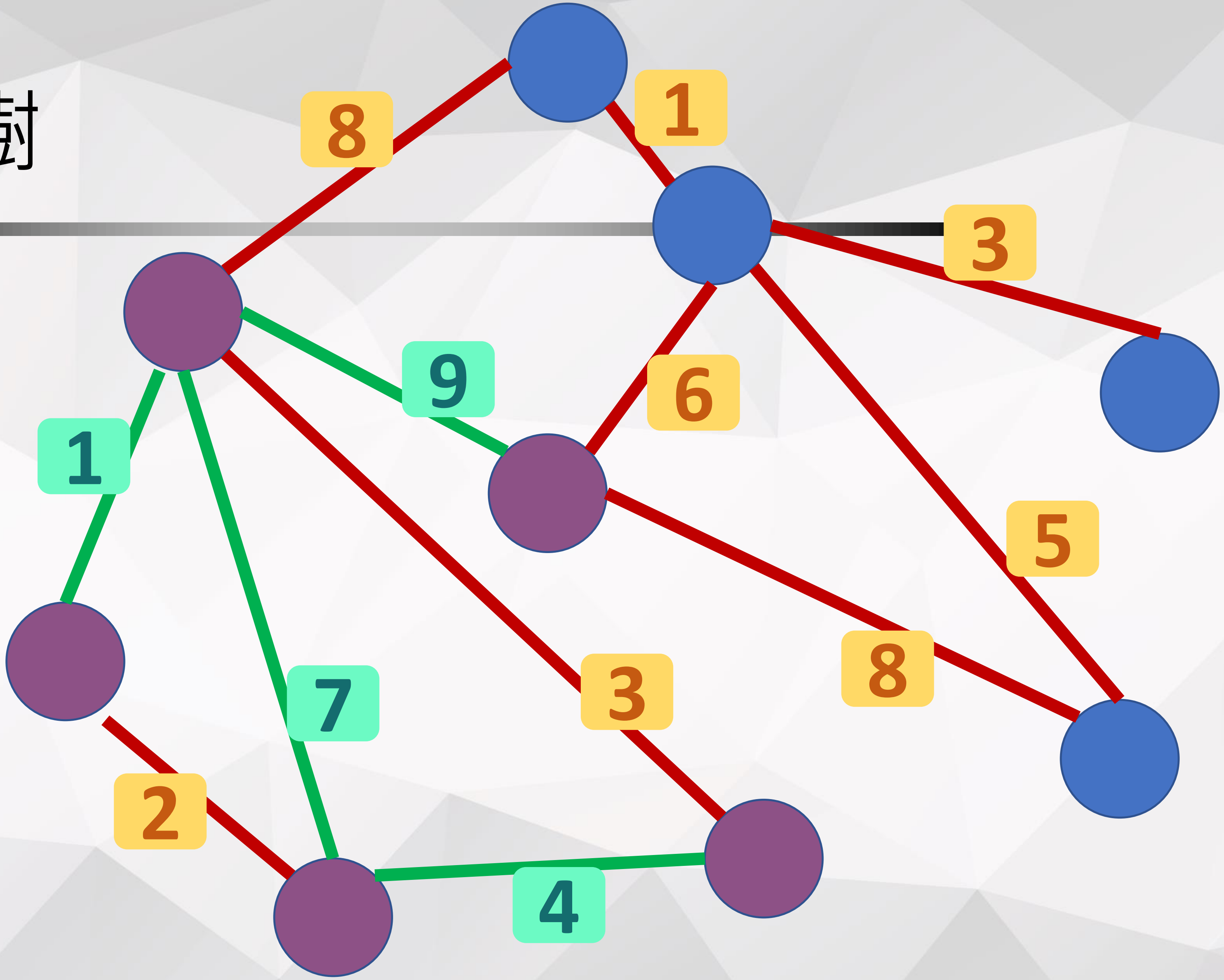
生成樹



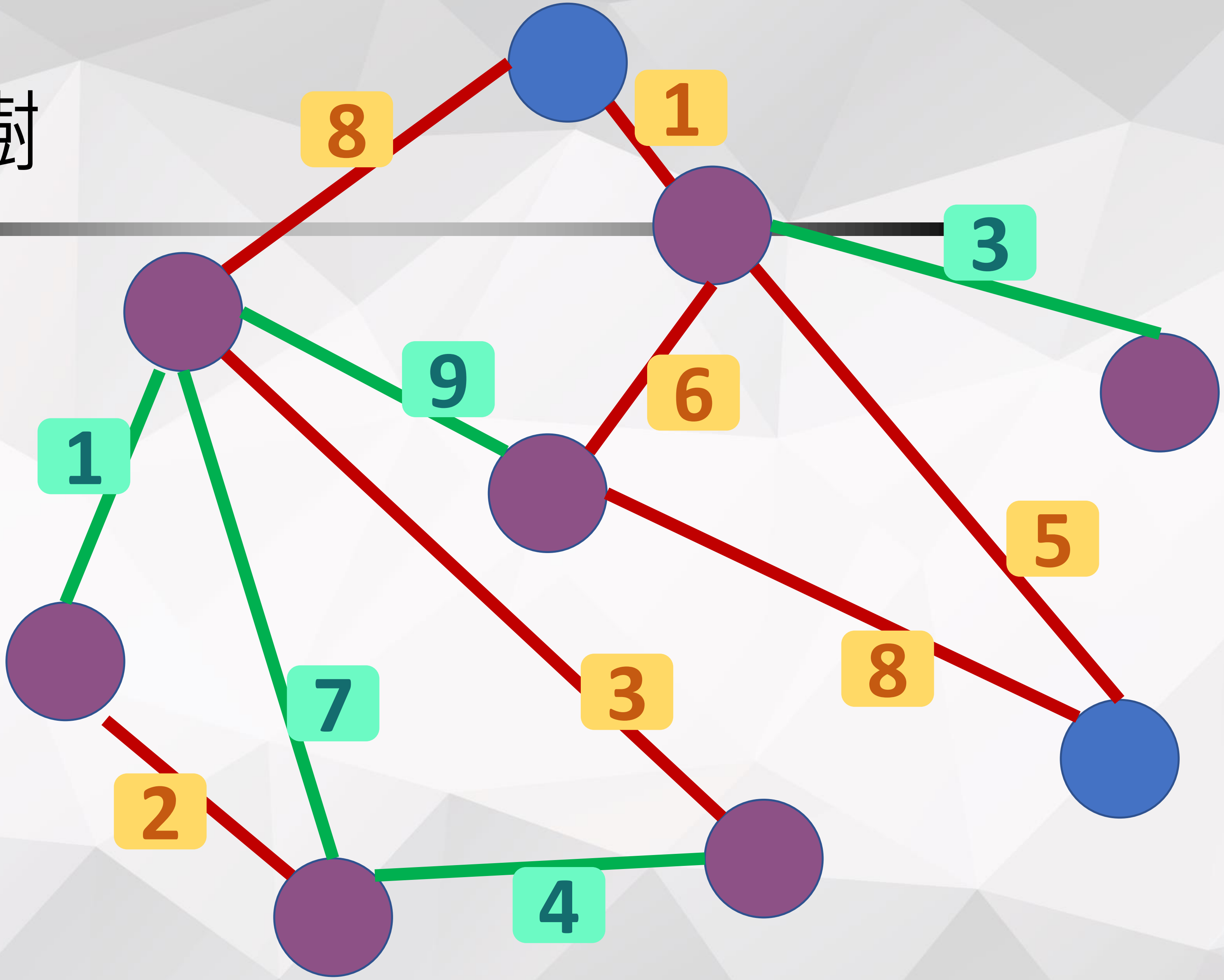
生成樹



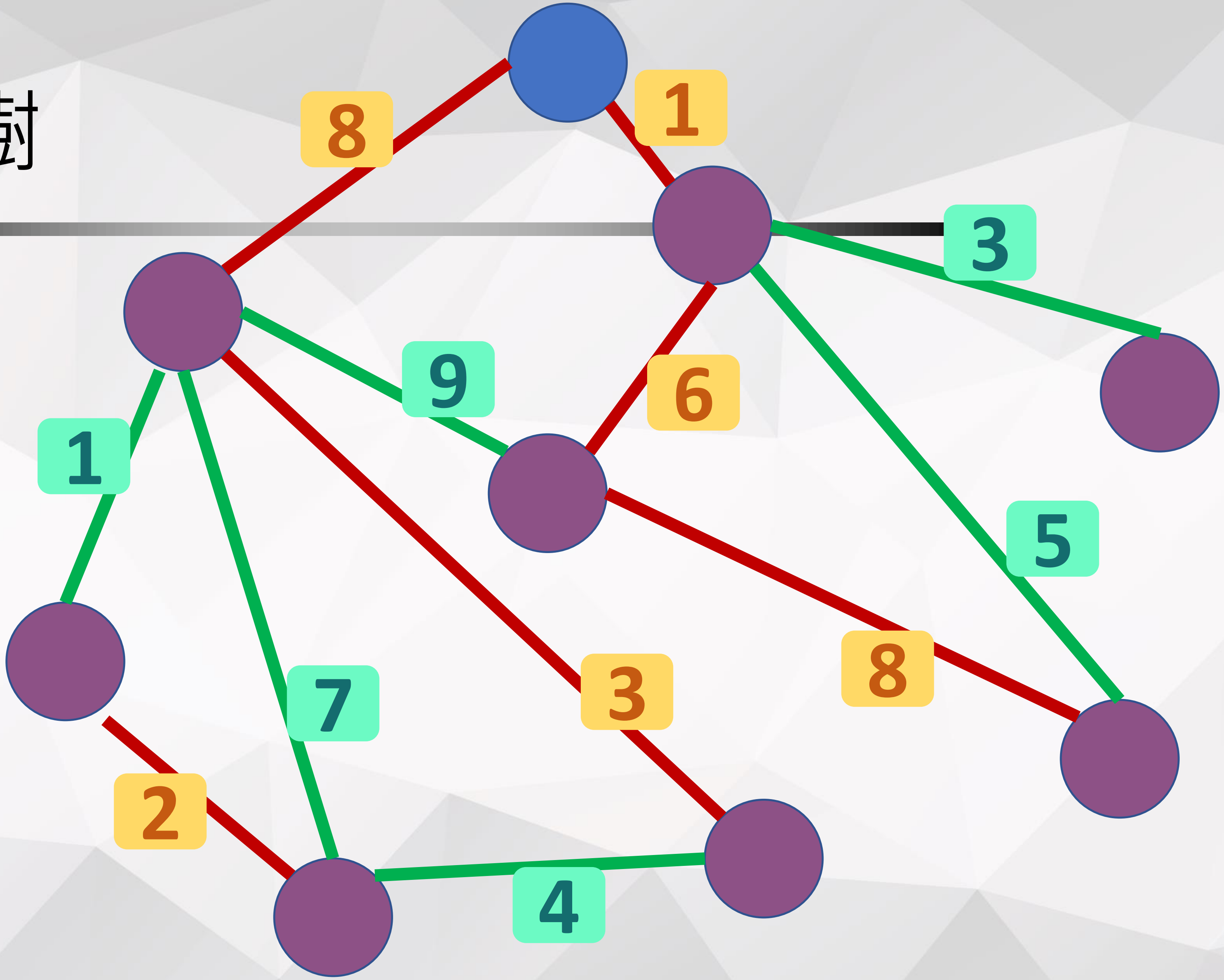
生成樹



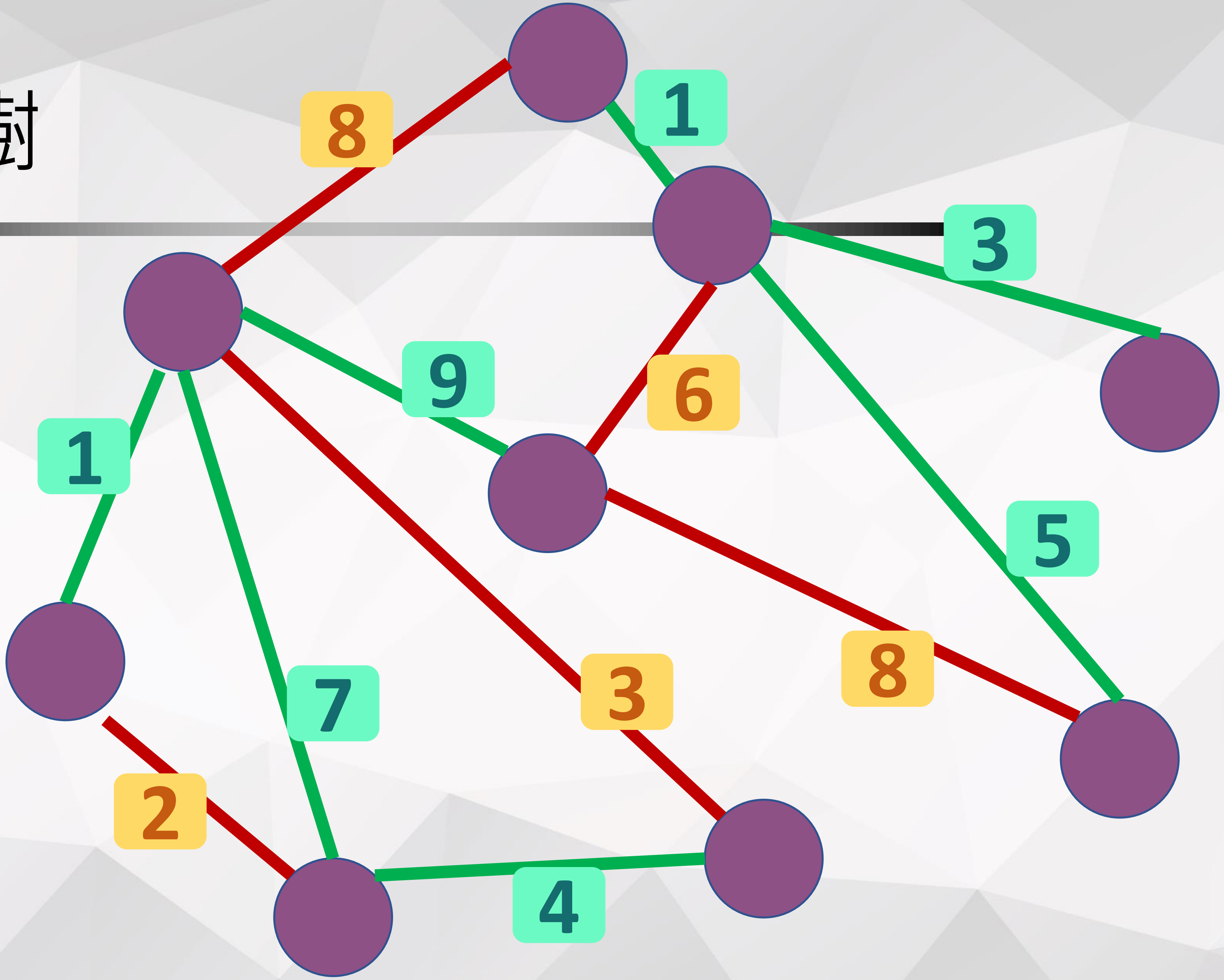
生成樹



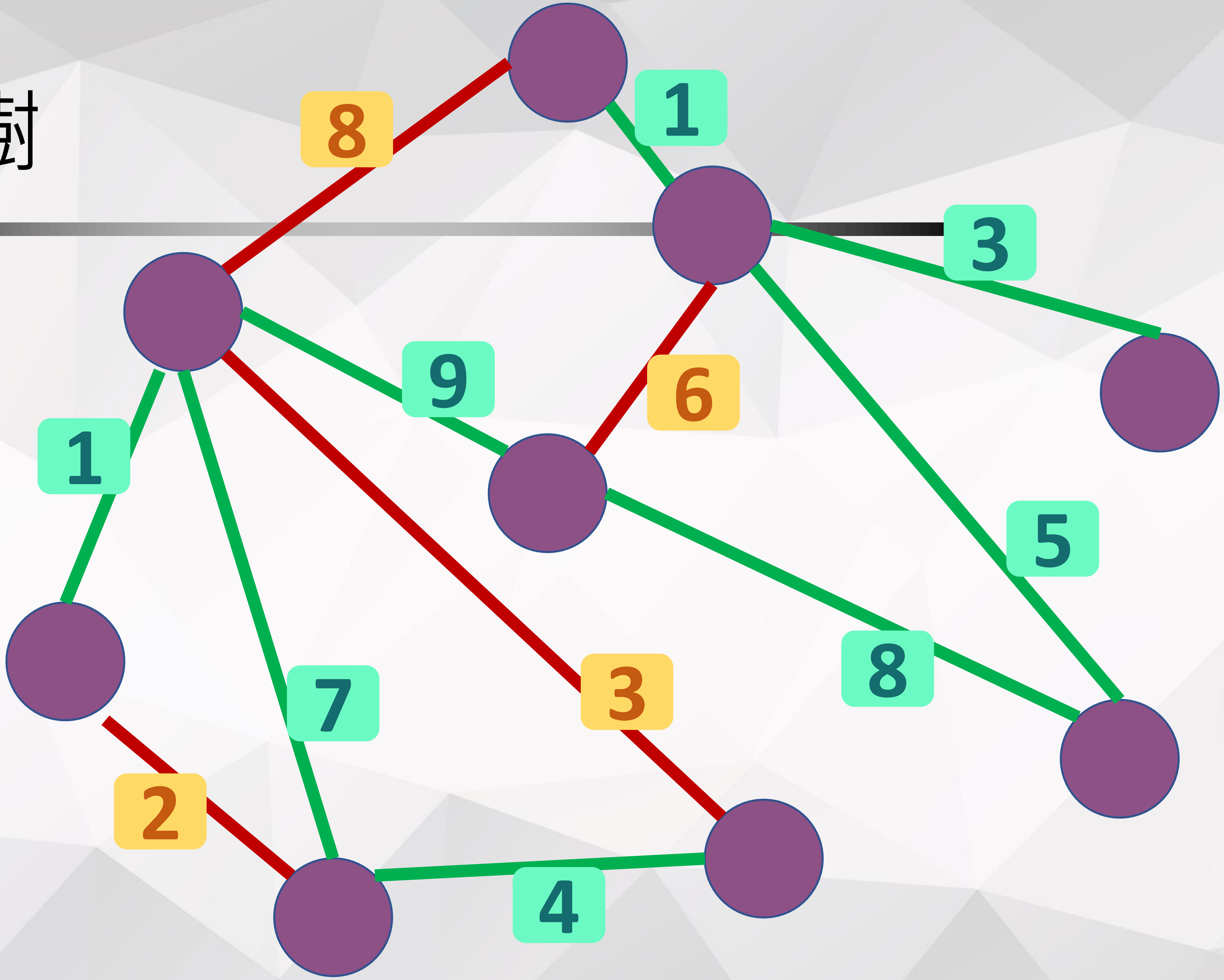
生成樹



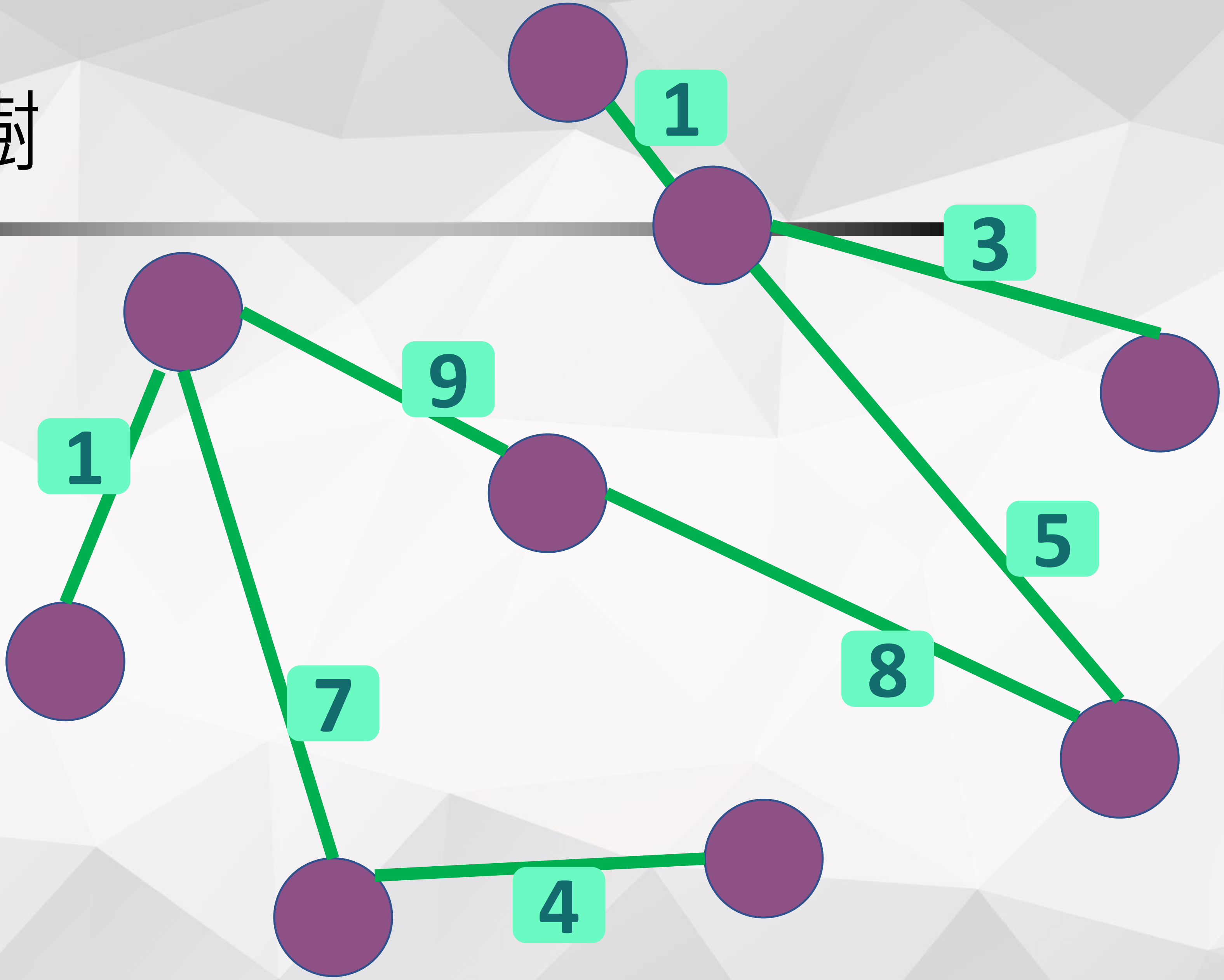
生成樹



生成樹



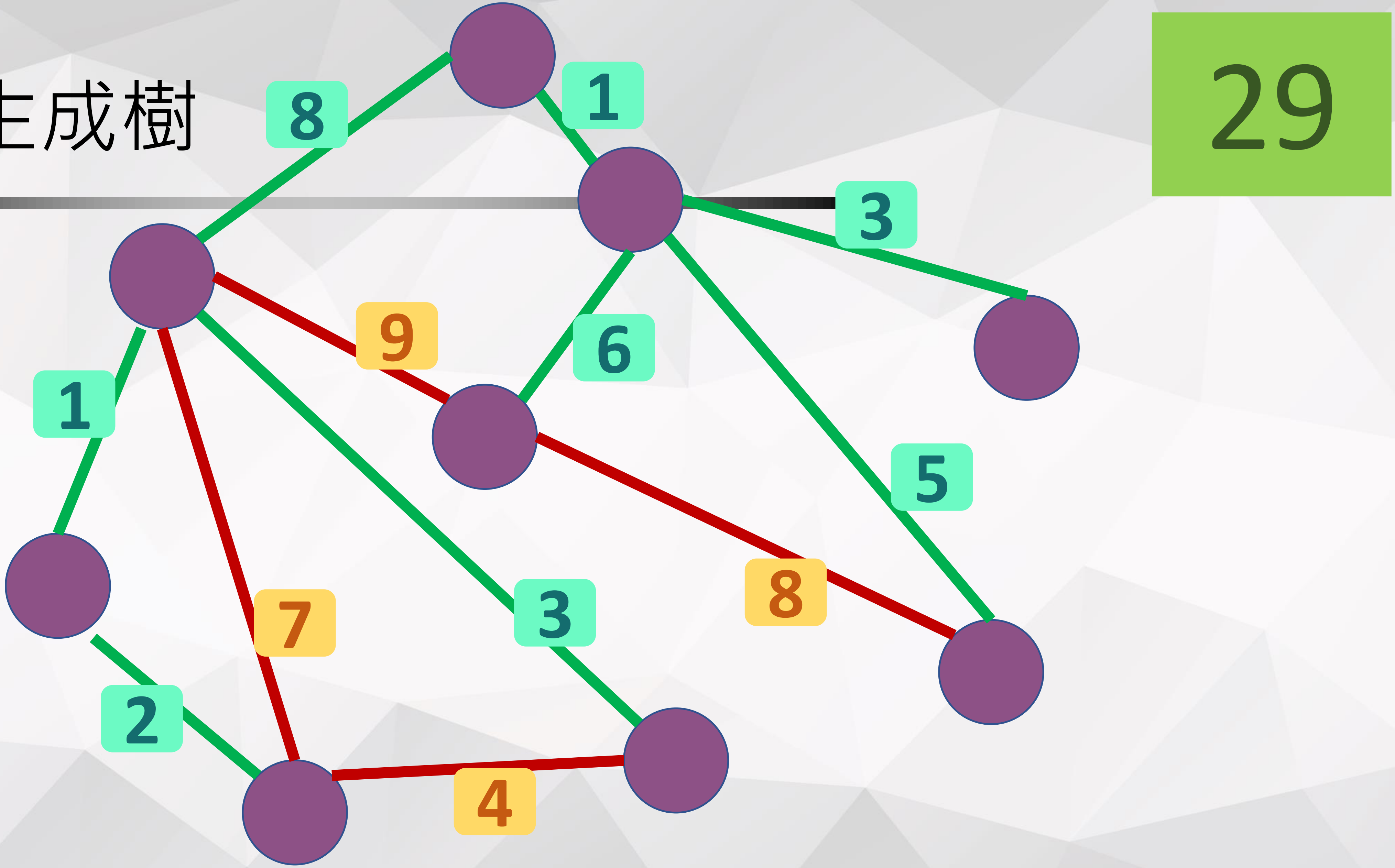
生成樹



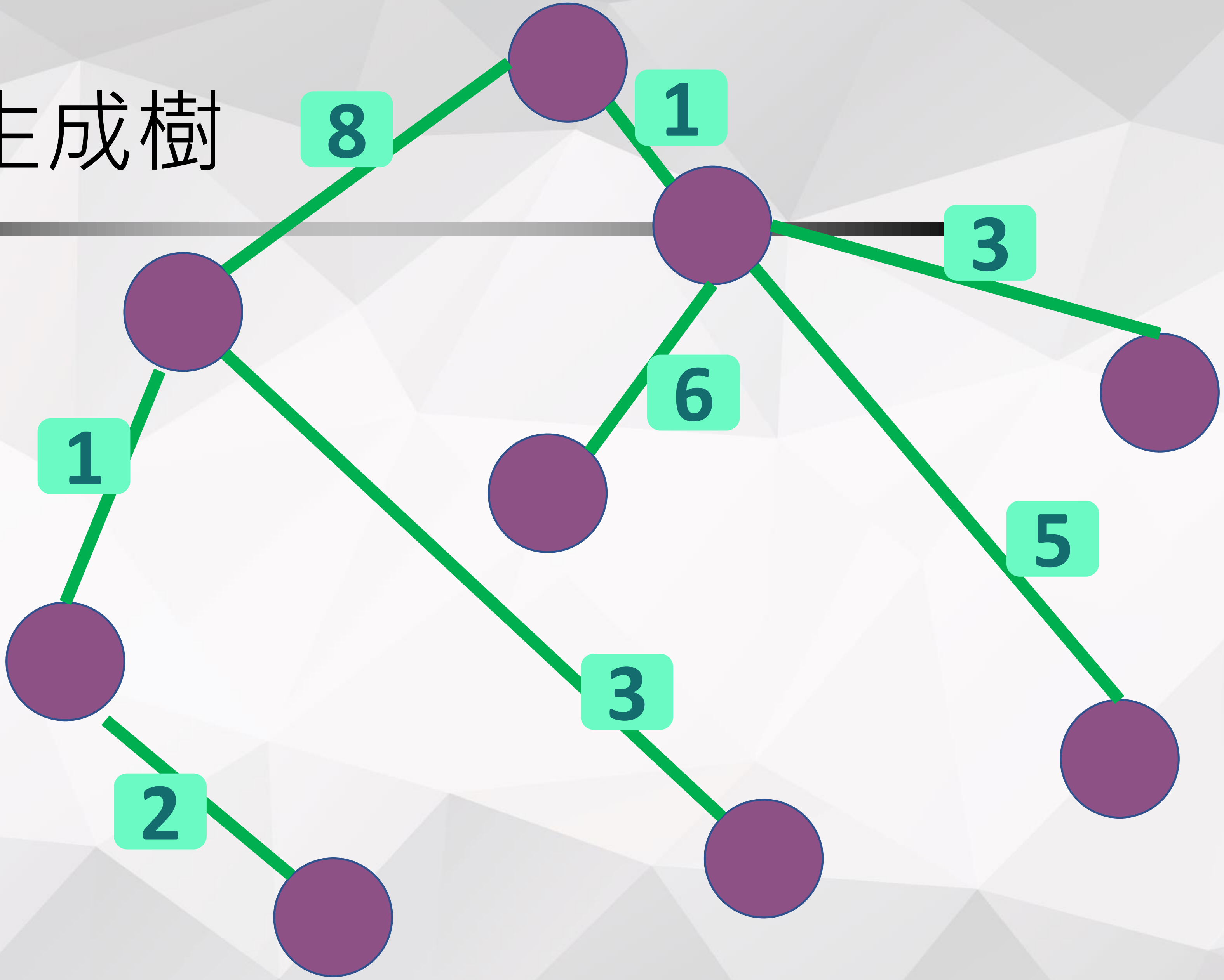
最小生成樹

- 給定連通圖 G = (E, V)
- 在所有生成樹中，找到**邊權重總和最小**的生成樹

最小生成樹



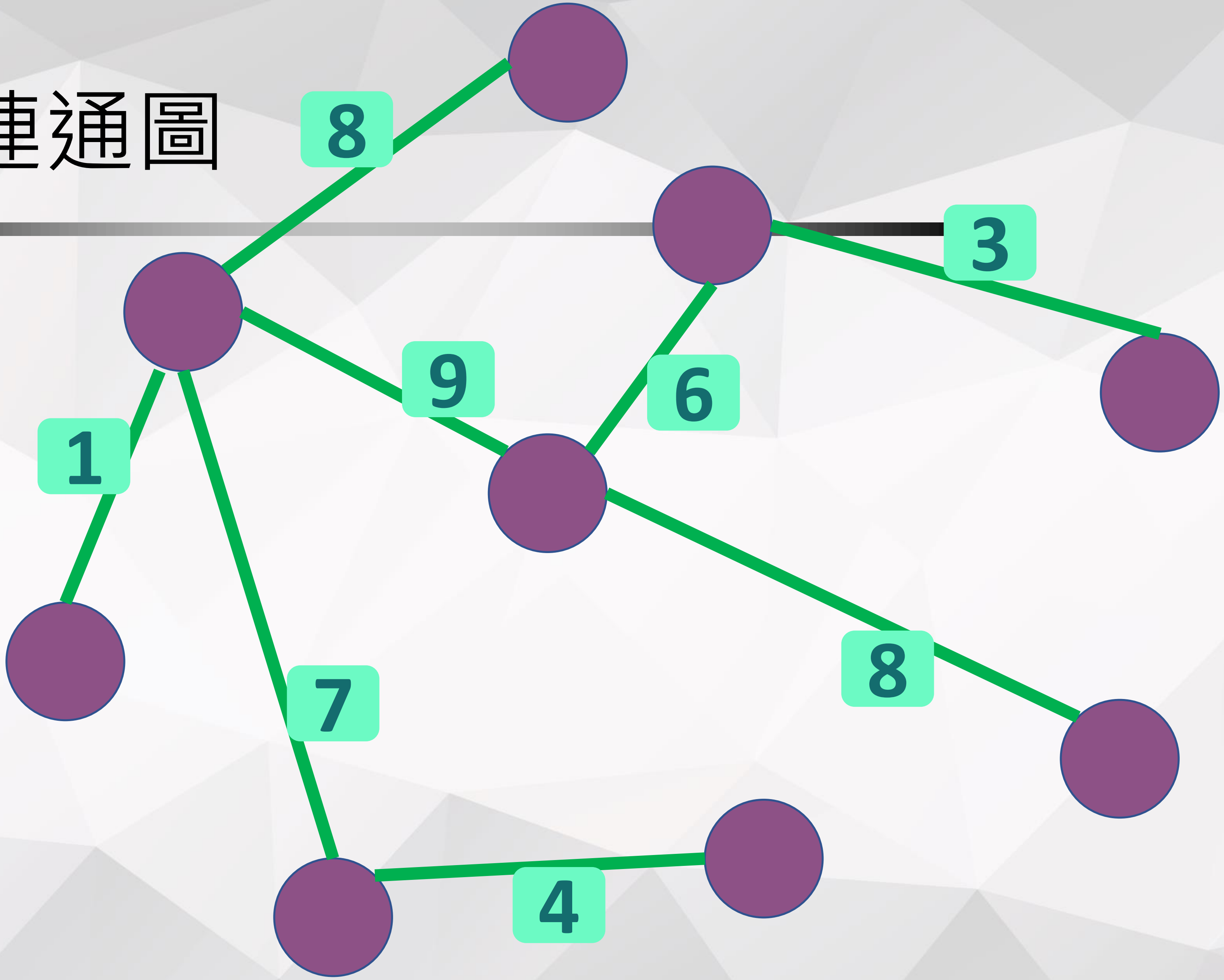
最小生成樹



兩個重要前提

- 樹是**無環**的**連通**圖
- 若圖只有點無任何邊，那每點都是彼此獨立**連通塊**

無環連通圖



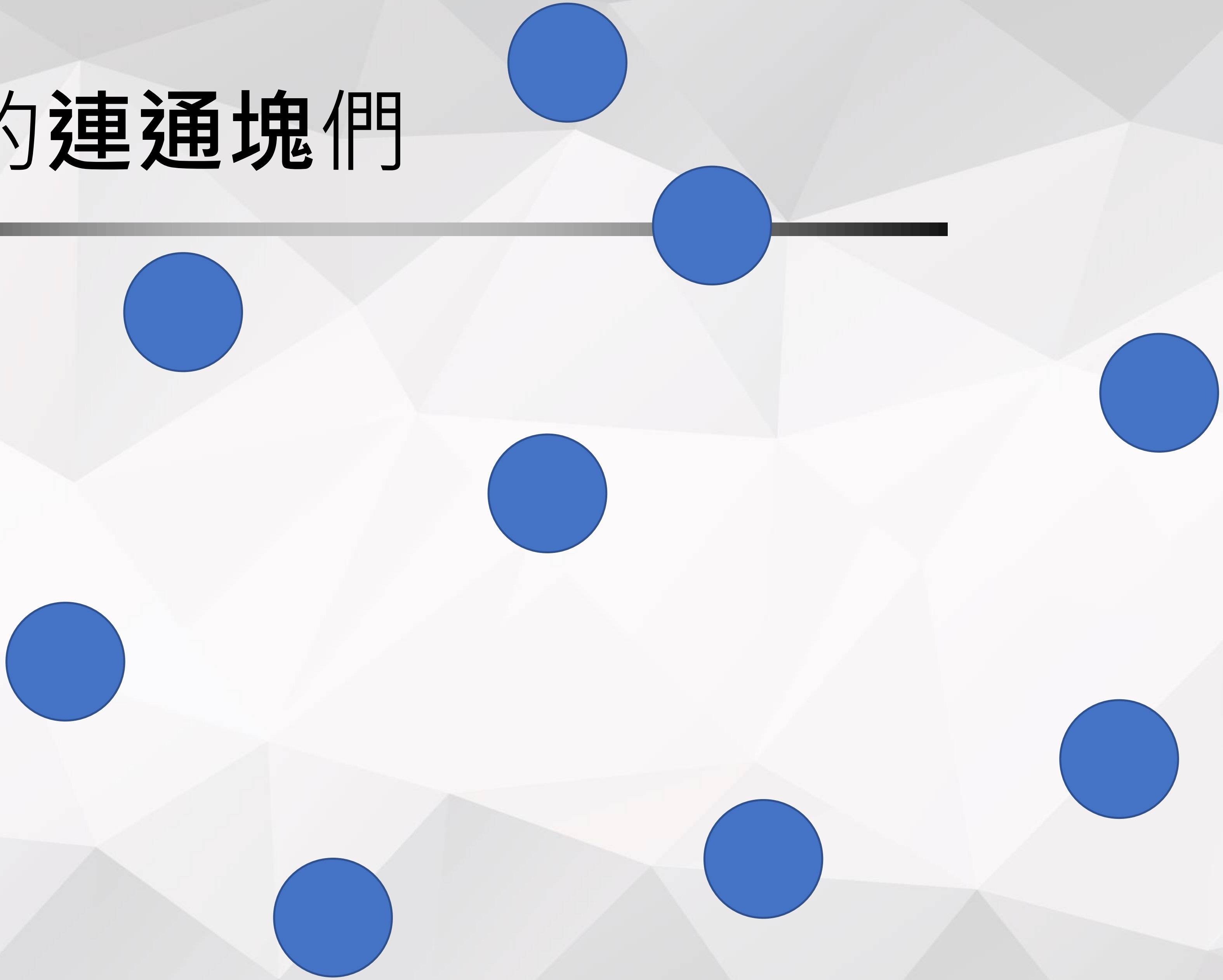
兩個重要前提

- 樹是**無環**的**連通**圖

- 若圖只有點無任何邊，那每點都是彼此獨立**連通塊**

獨立的**連通塊**們

0



最小生成樹

- Kruskal 演算法
- Prim 演算法

最小生成樹

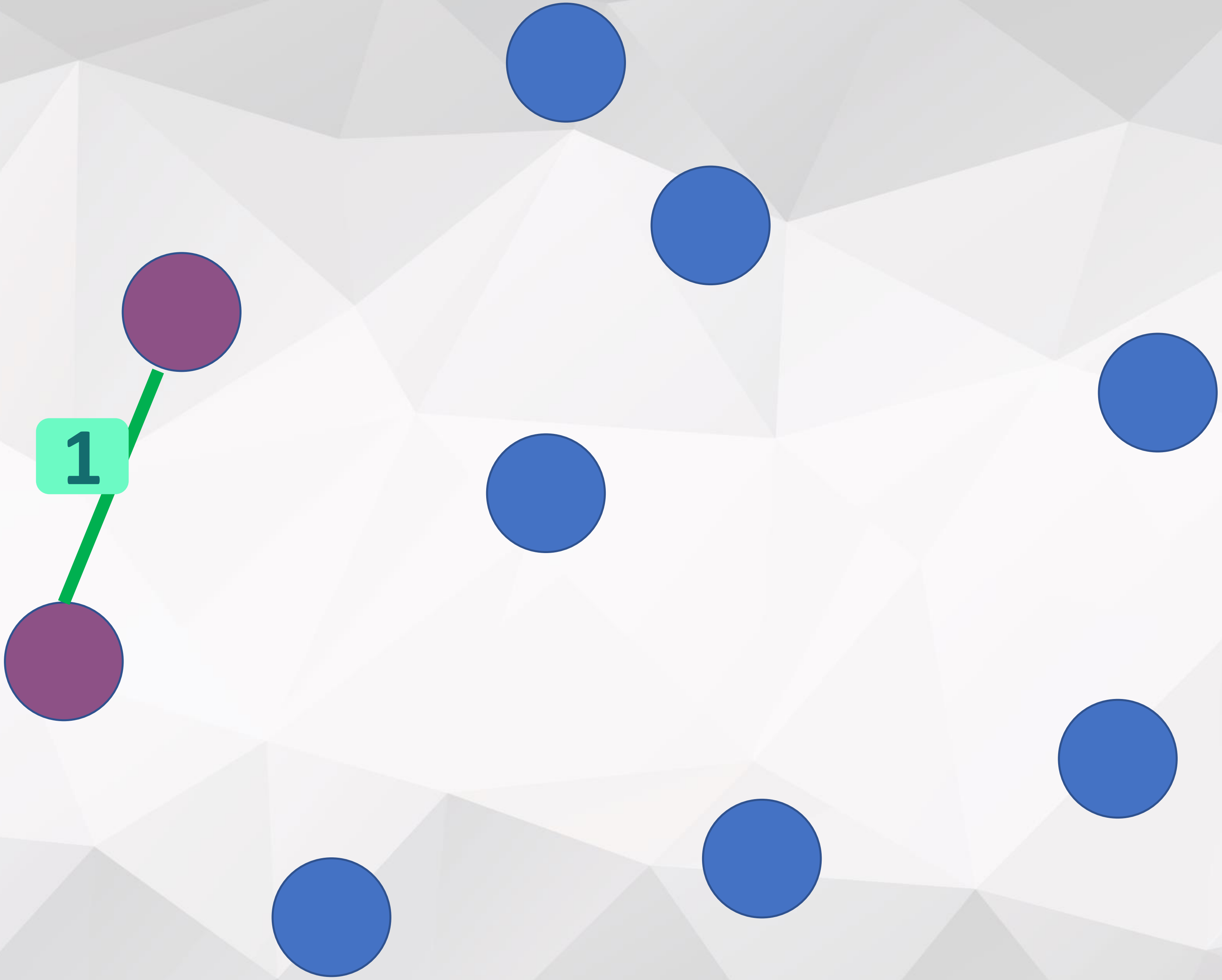
- Kruskal 演算法
- Prim 演算法

Kruskal 演算法

- 在產生生成樹以前，所有點都為獨立的連通塊
- 若兩個獨立的連通塊相連，整個圖就少一連通塊

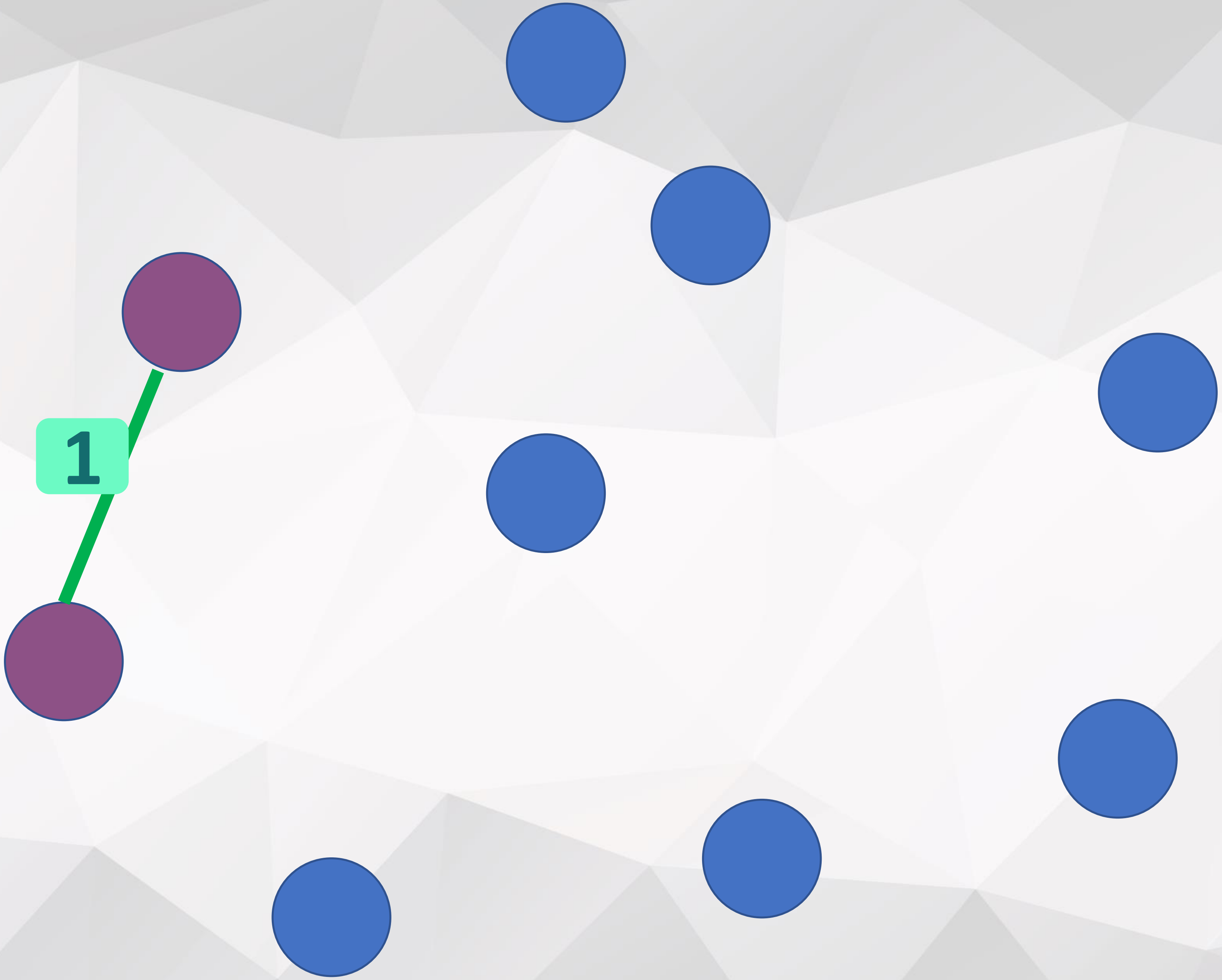
0

1

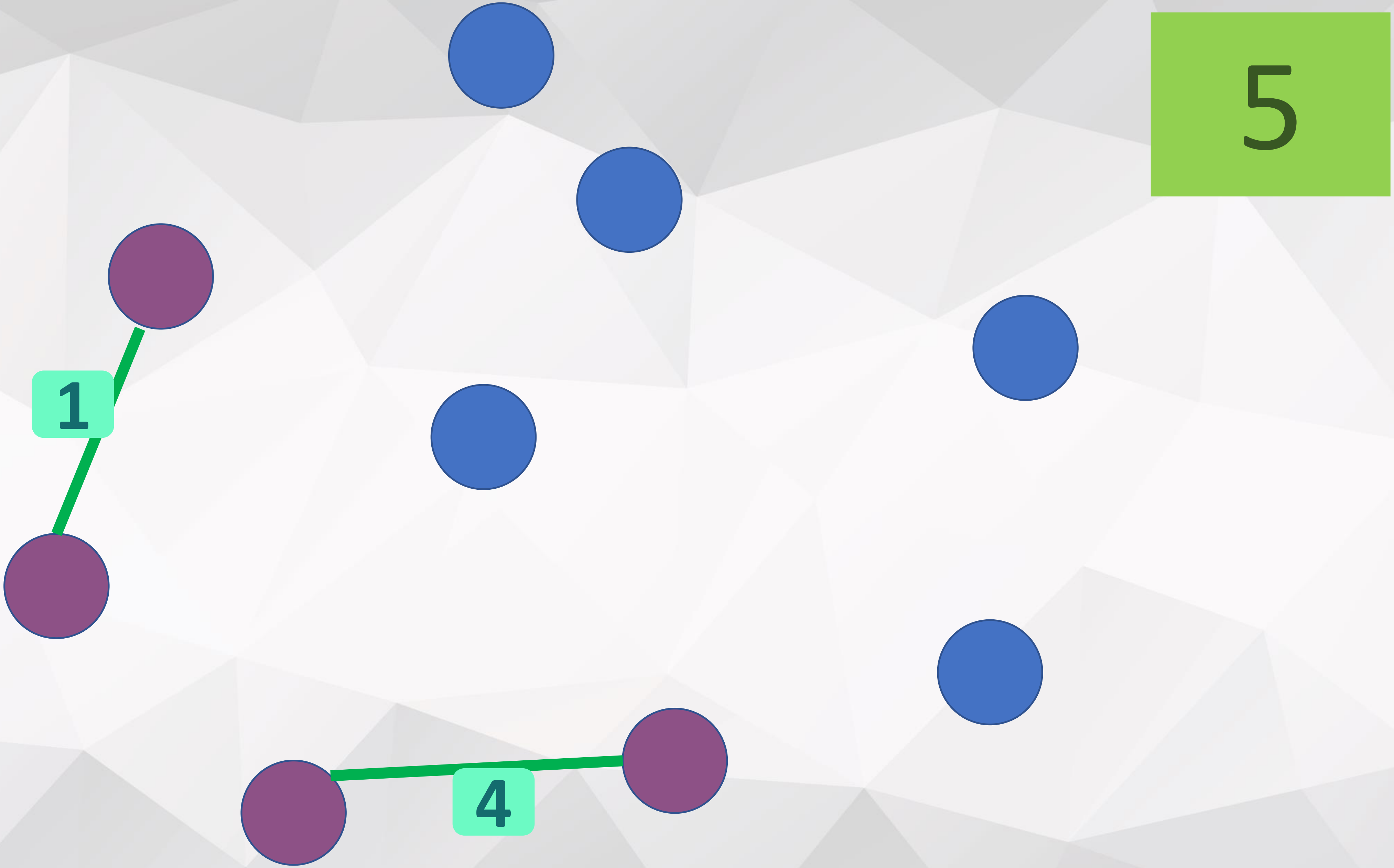


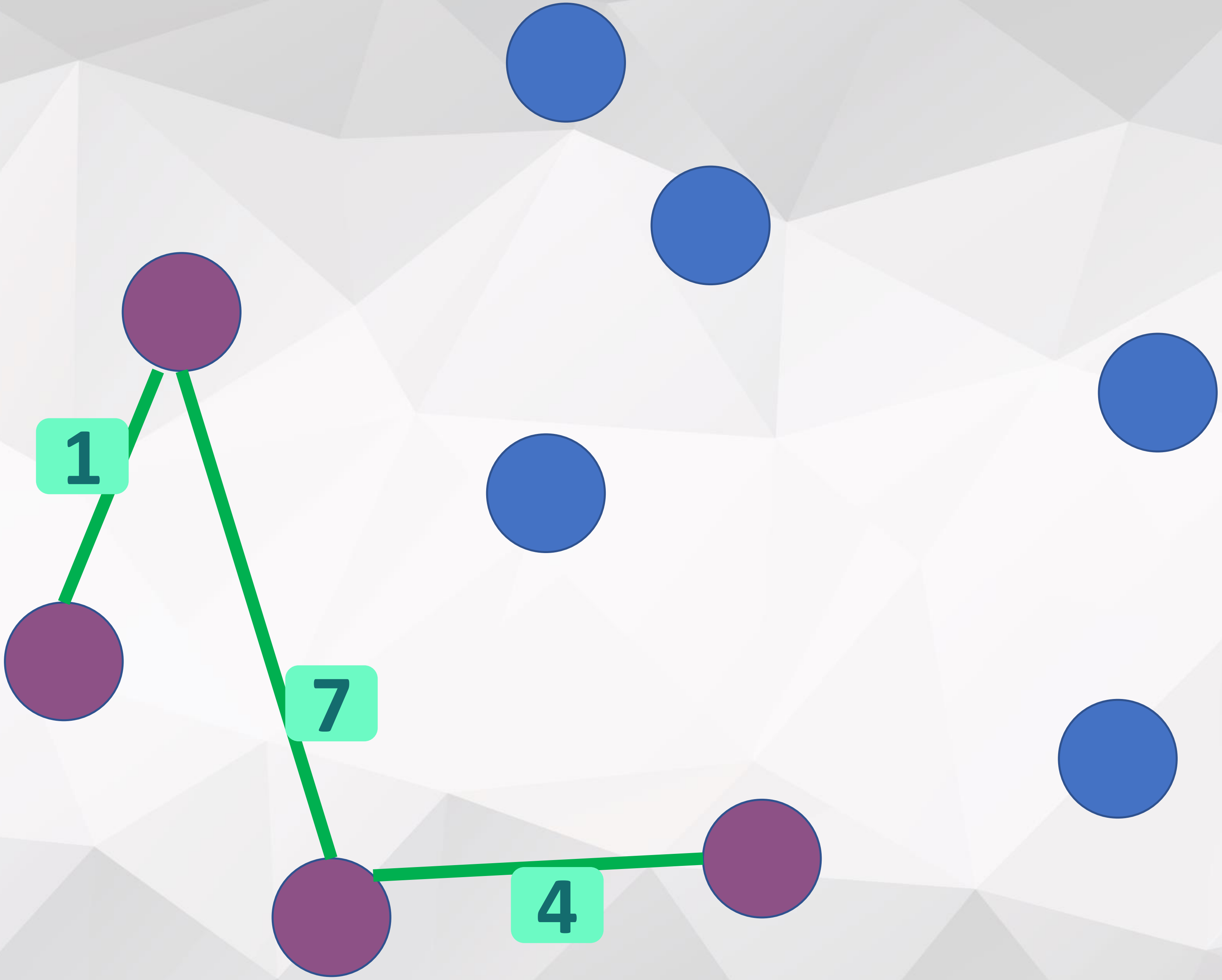
如何產生生成樹

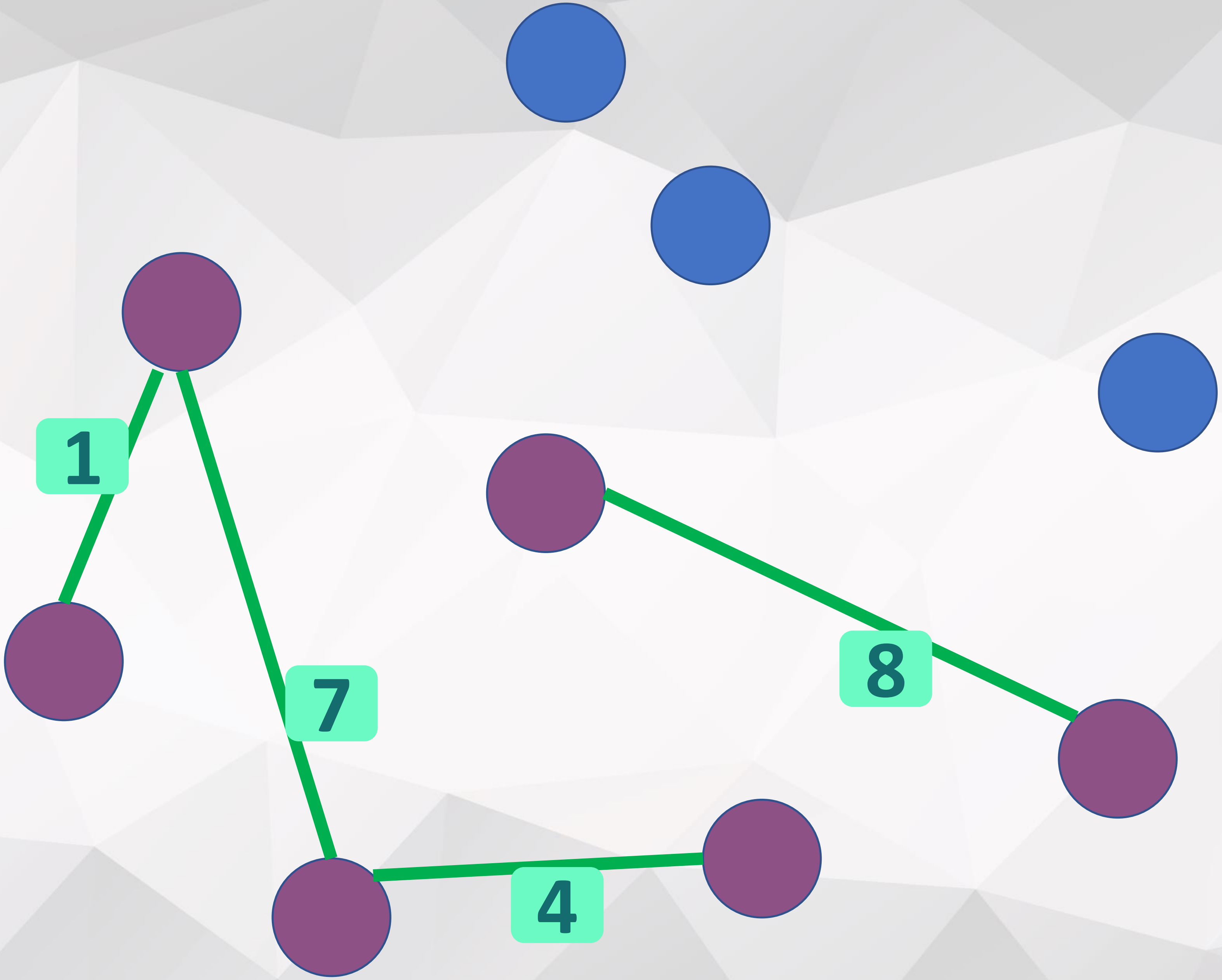
- 在連通塊 A 與連通塊 B 相連時確保不會產生環
- 最終就能得到一棵生成樹



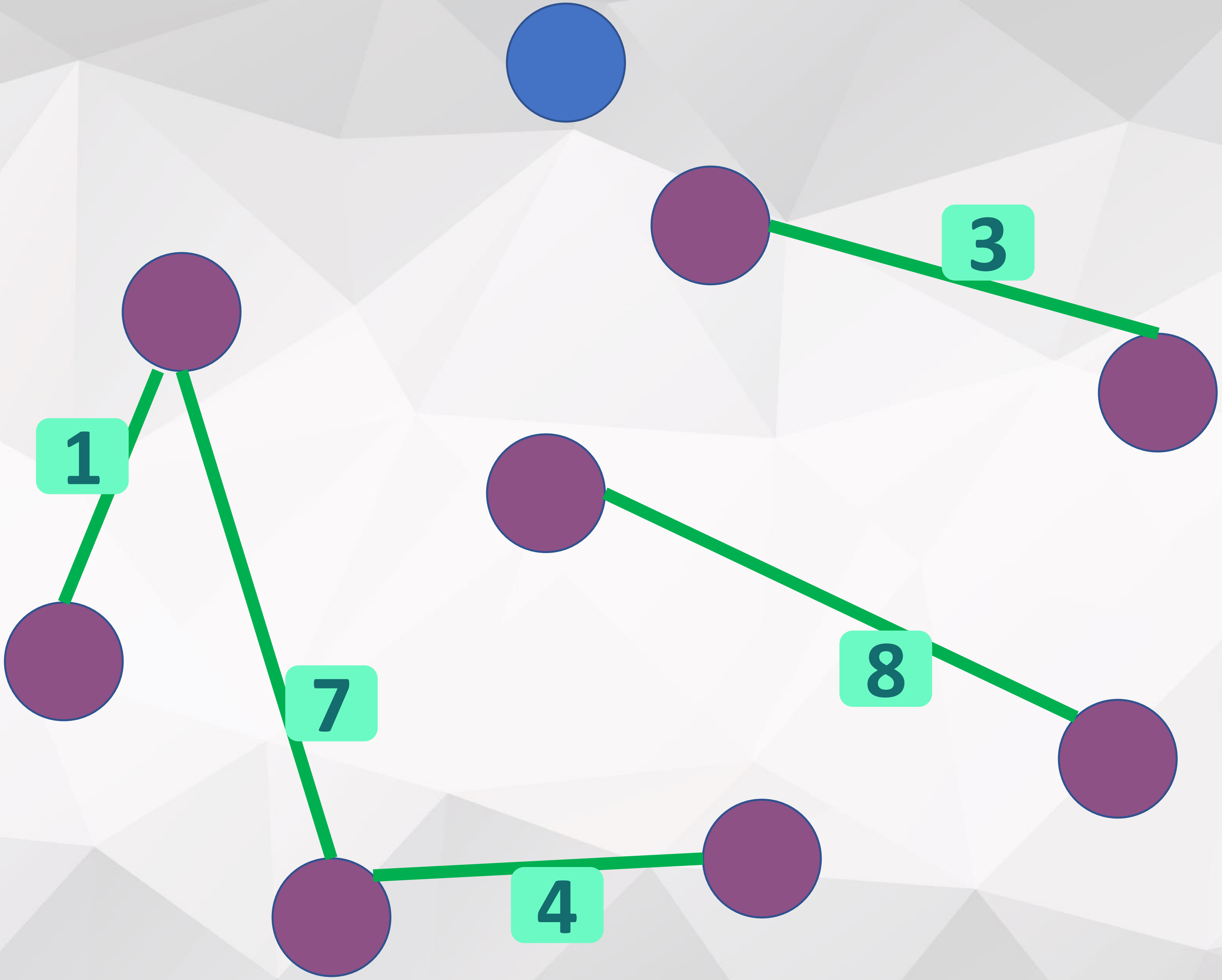
1

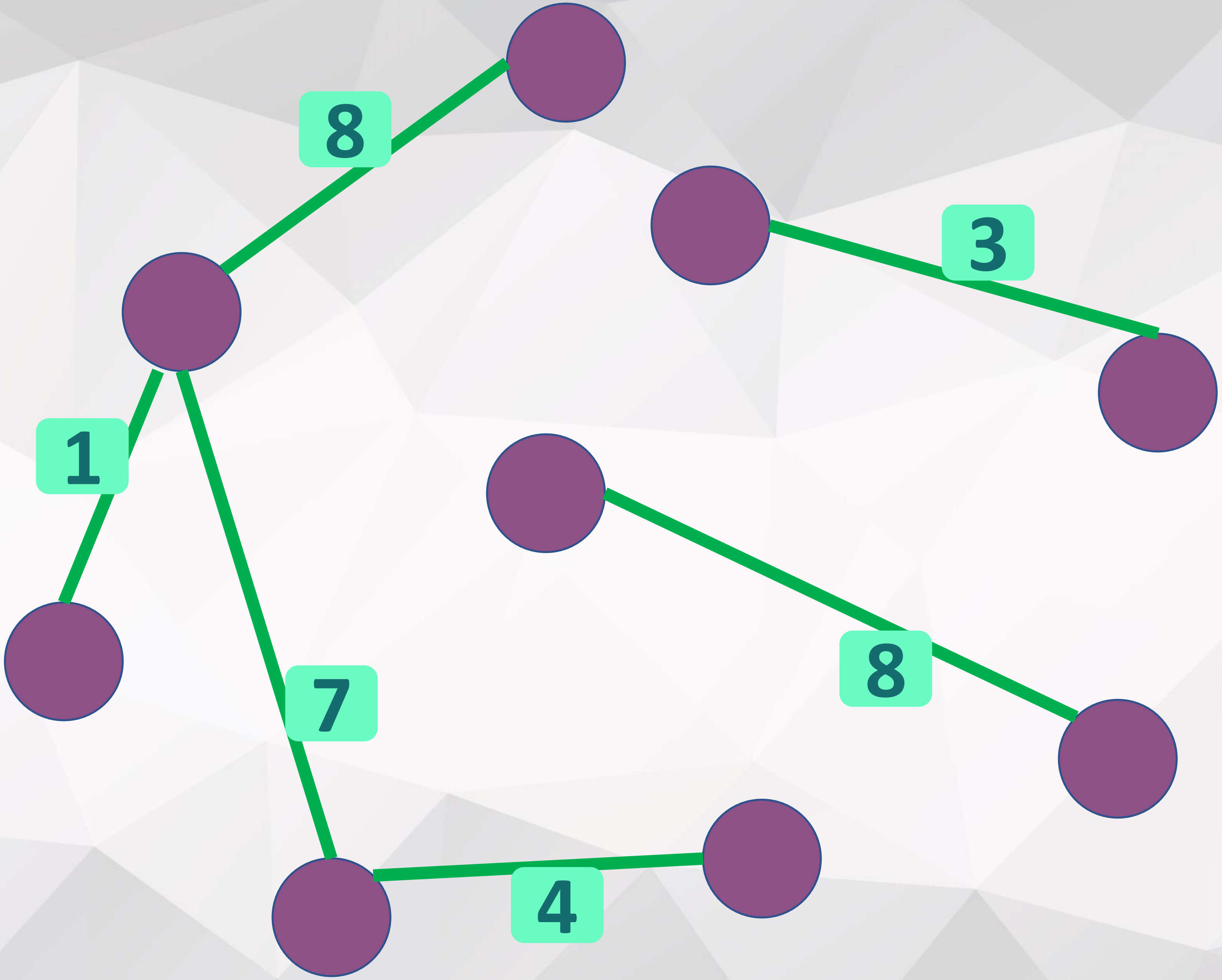

5
1
4

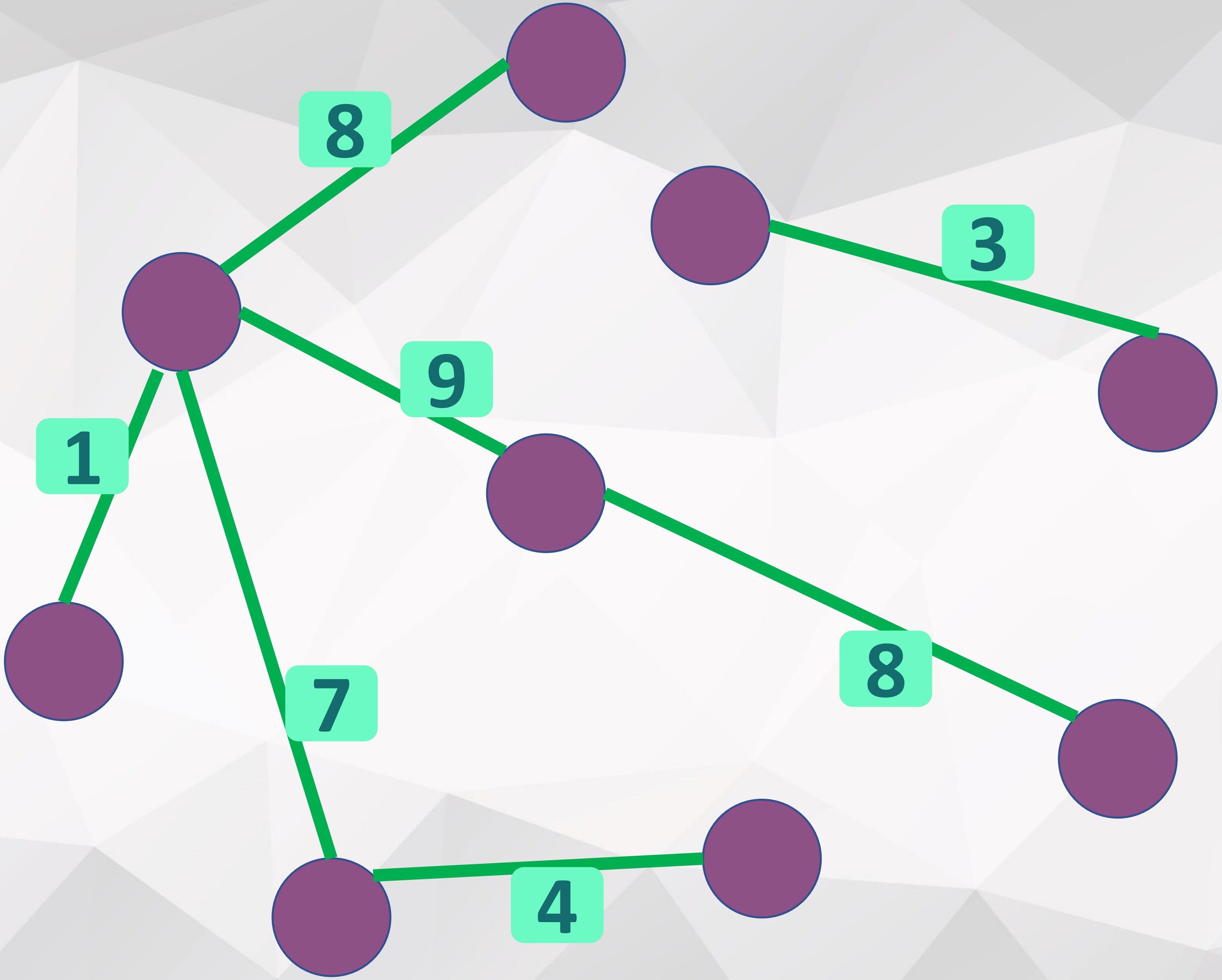

1
7
4


1
7
8
4

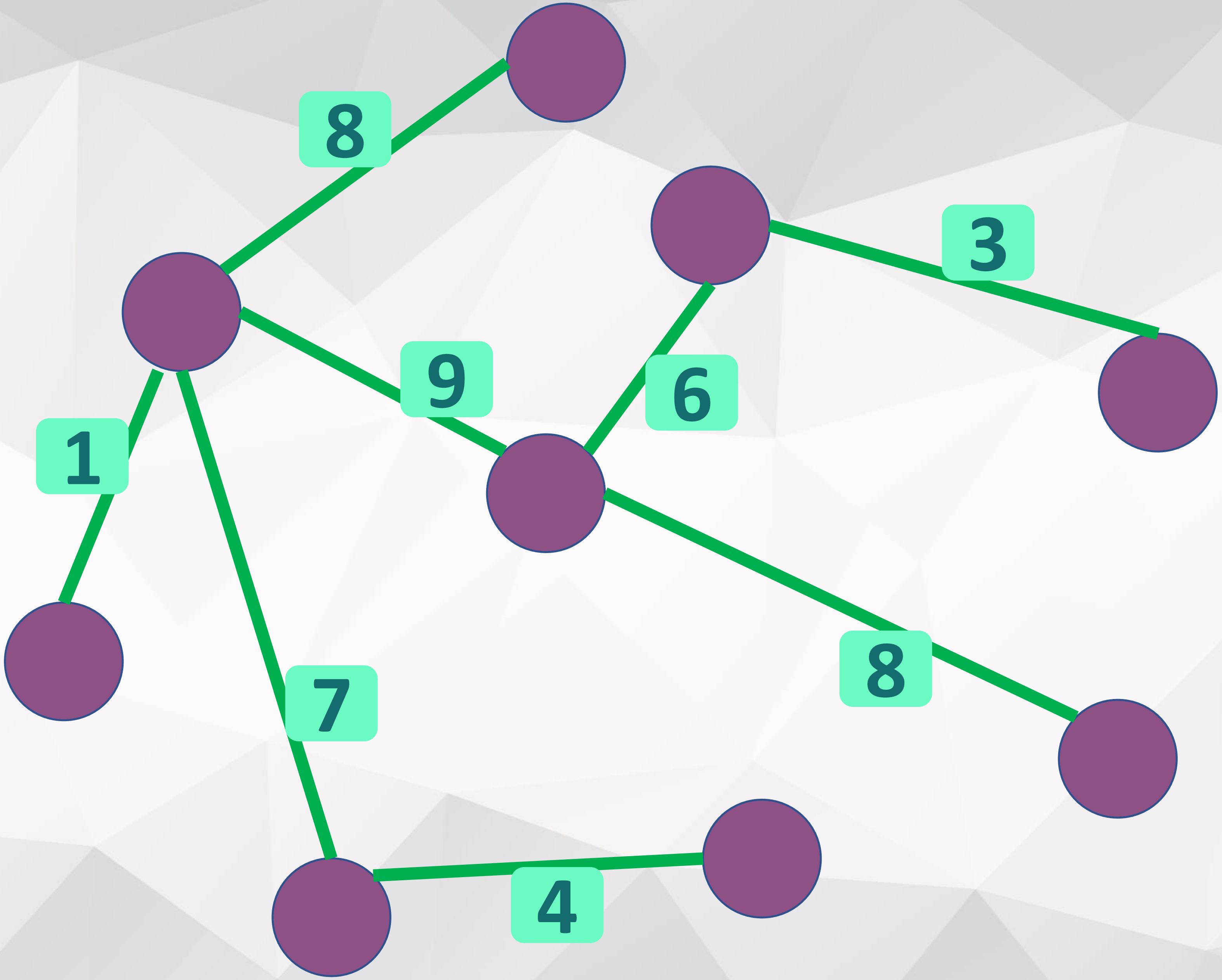
23




8
3
1
8
7
4



40



46

Competitive Programming

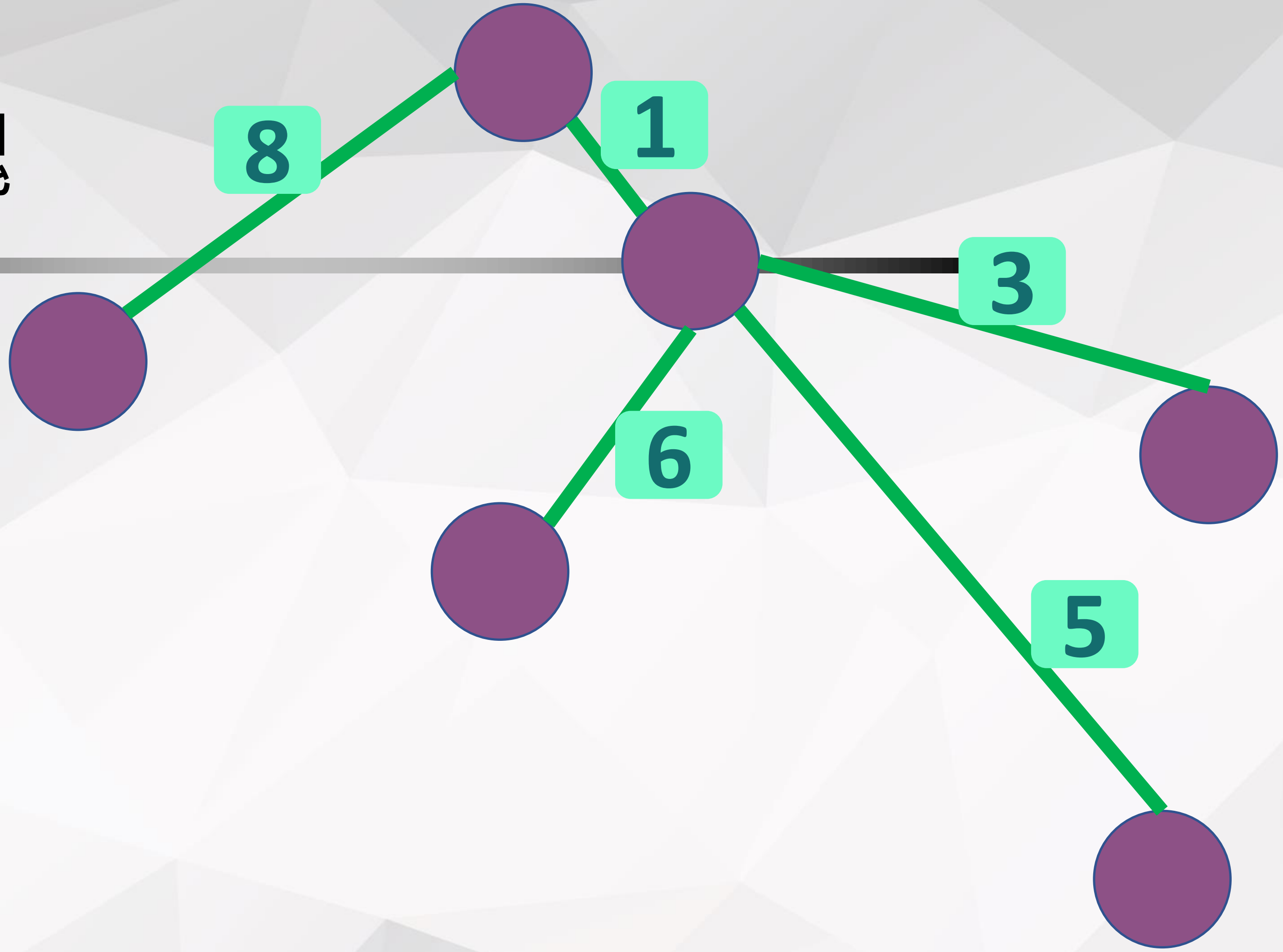
Made by 培訓團隊群 CCNS

怎樣不產生環

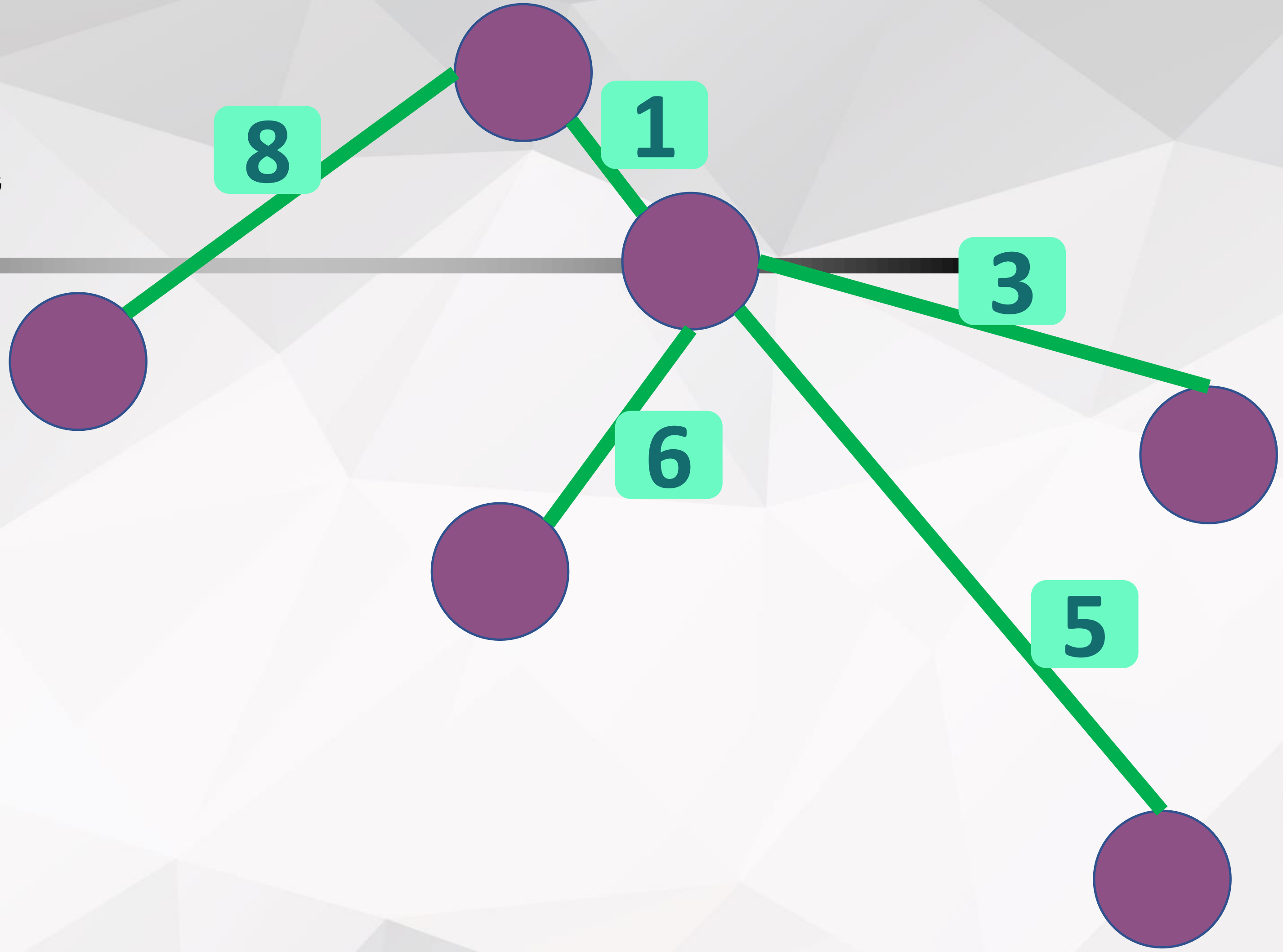
- 在連通塊 A 與連通塊 B 相連時確保不會產生環

$A \neq B \Leftrightarrow$ 加入新的邊不產生環

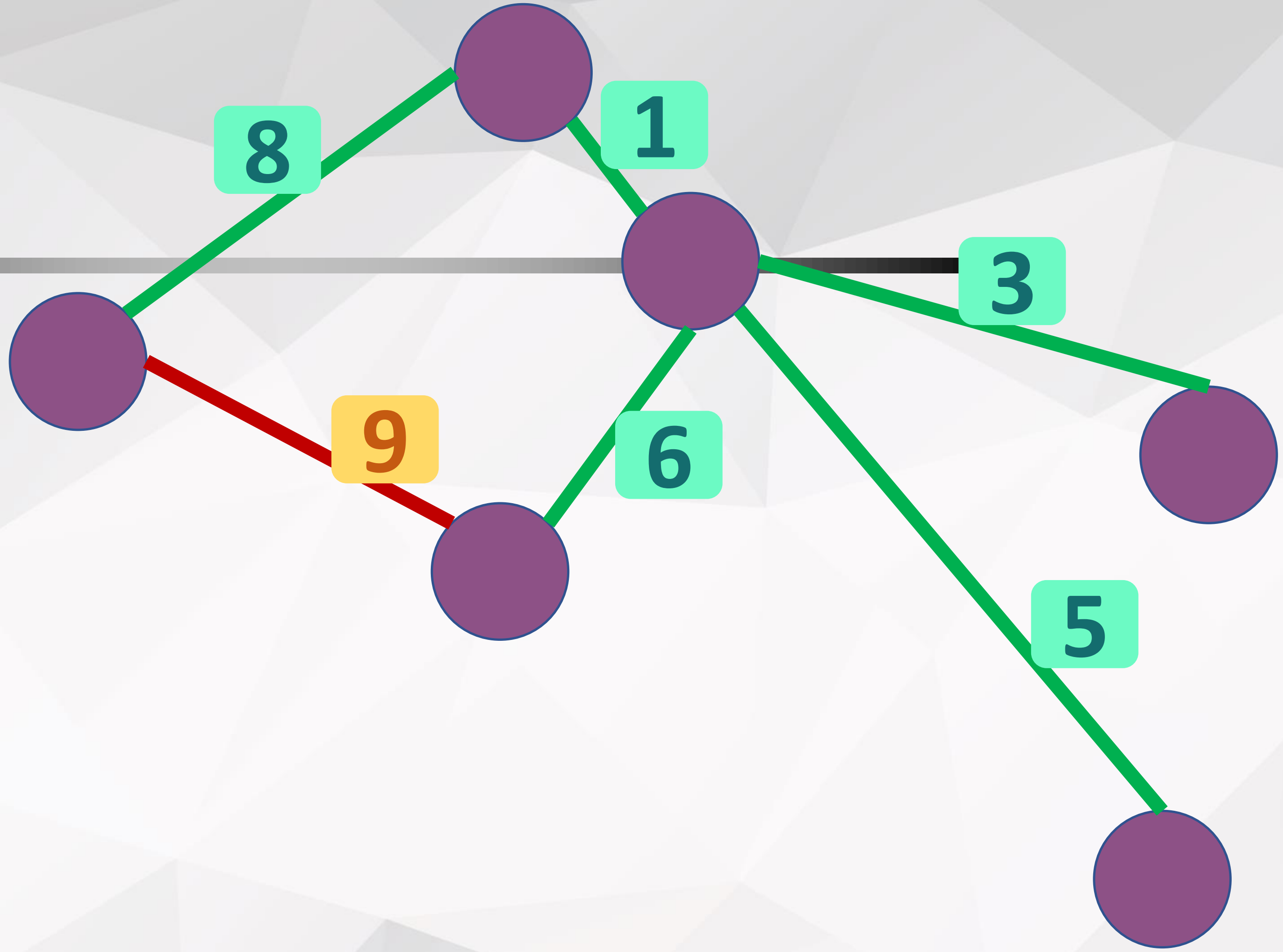
A 連通塊



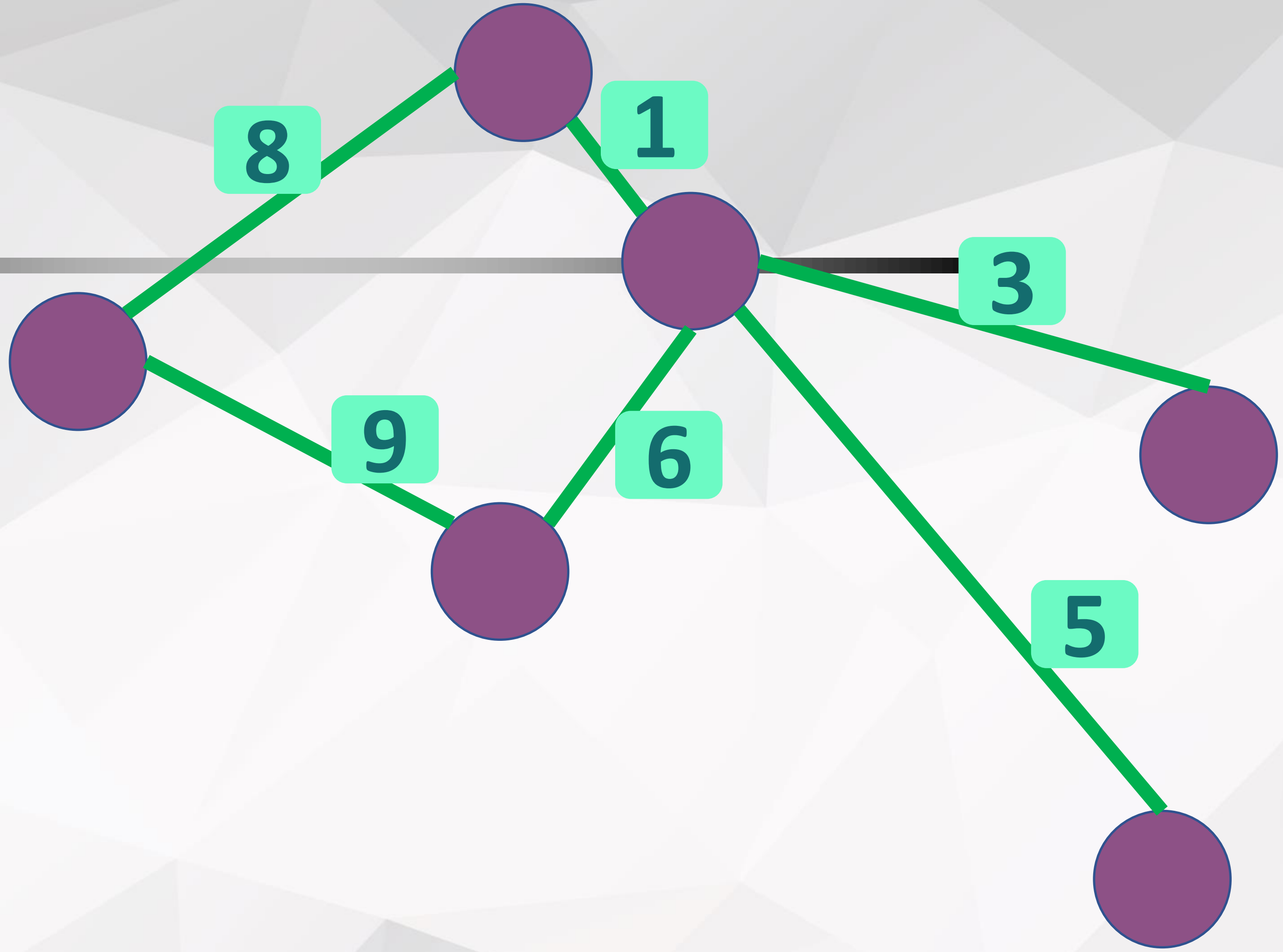
B 連通塊



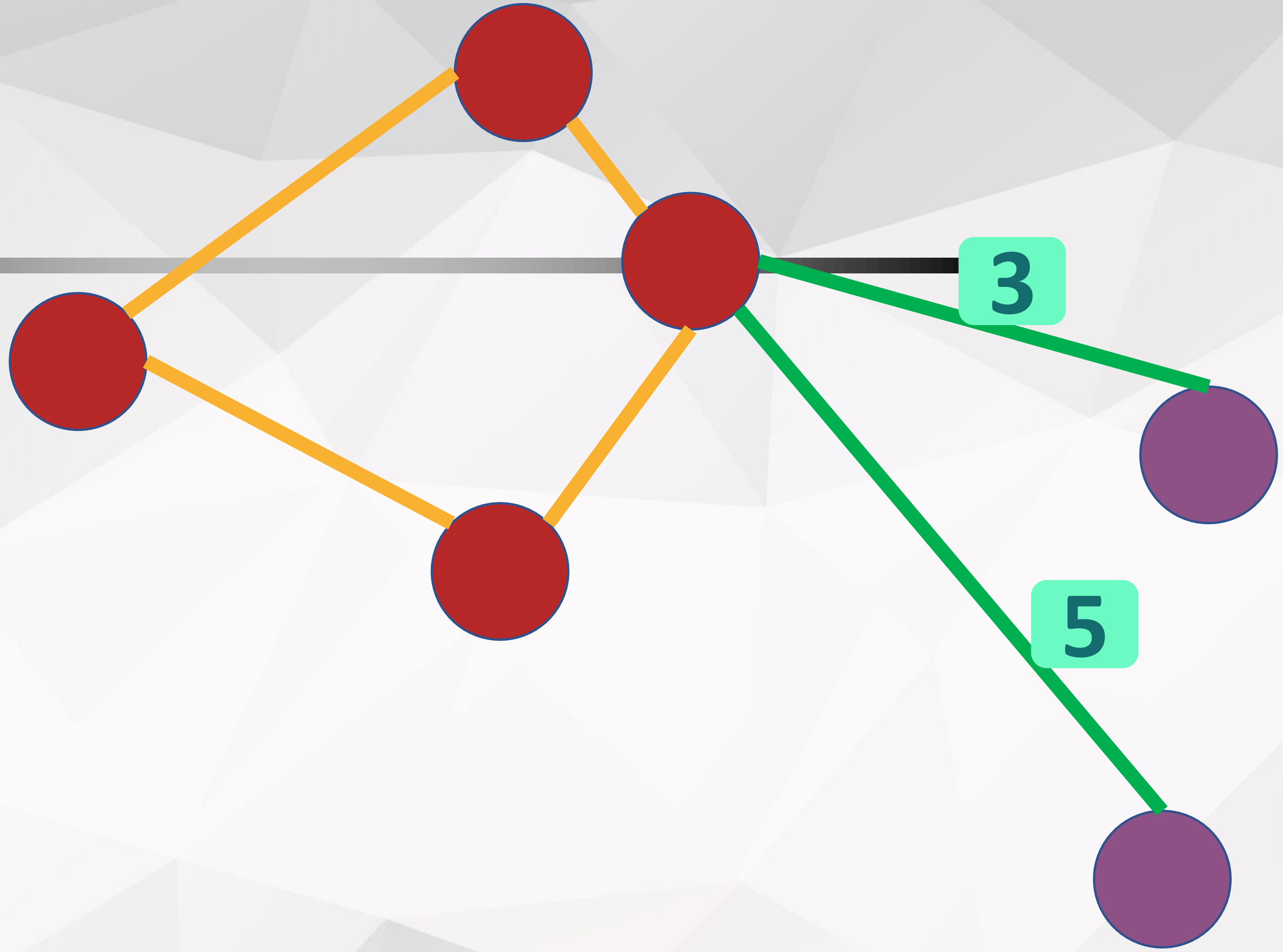
相連



相連



產生環



怎樣不產生環

- 在連通塊 A 與連通塊 B 相連時確保不會產生環

$A \neq B \Leftrightarrow$ 加入新的邊不產生環

怎樣不產生環

- 在連通塊 A 與連通塊 B 相連時確保不會產生環

$$A \neq B \Leftrightarrow \text{加入新的邊不產生環}$$

- 所以過程中要確保挑的**連通塊互為獨立**

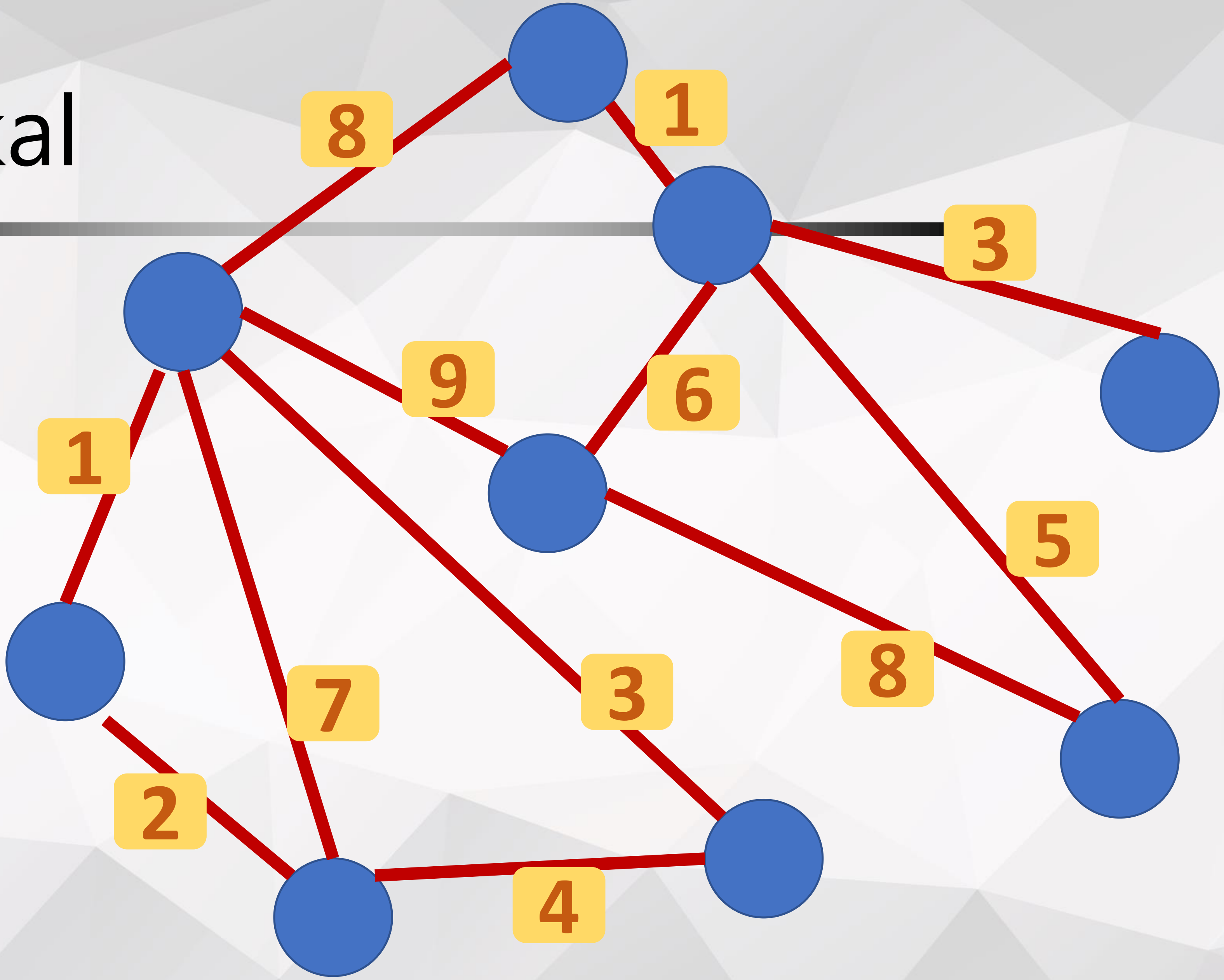
Kruskal 演算法

- 直覺的，每次相連選一個**合法**且**權重最小**的邊

- 最終得到的生成樹，就是最小生成樹

Kruskal

0

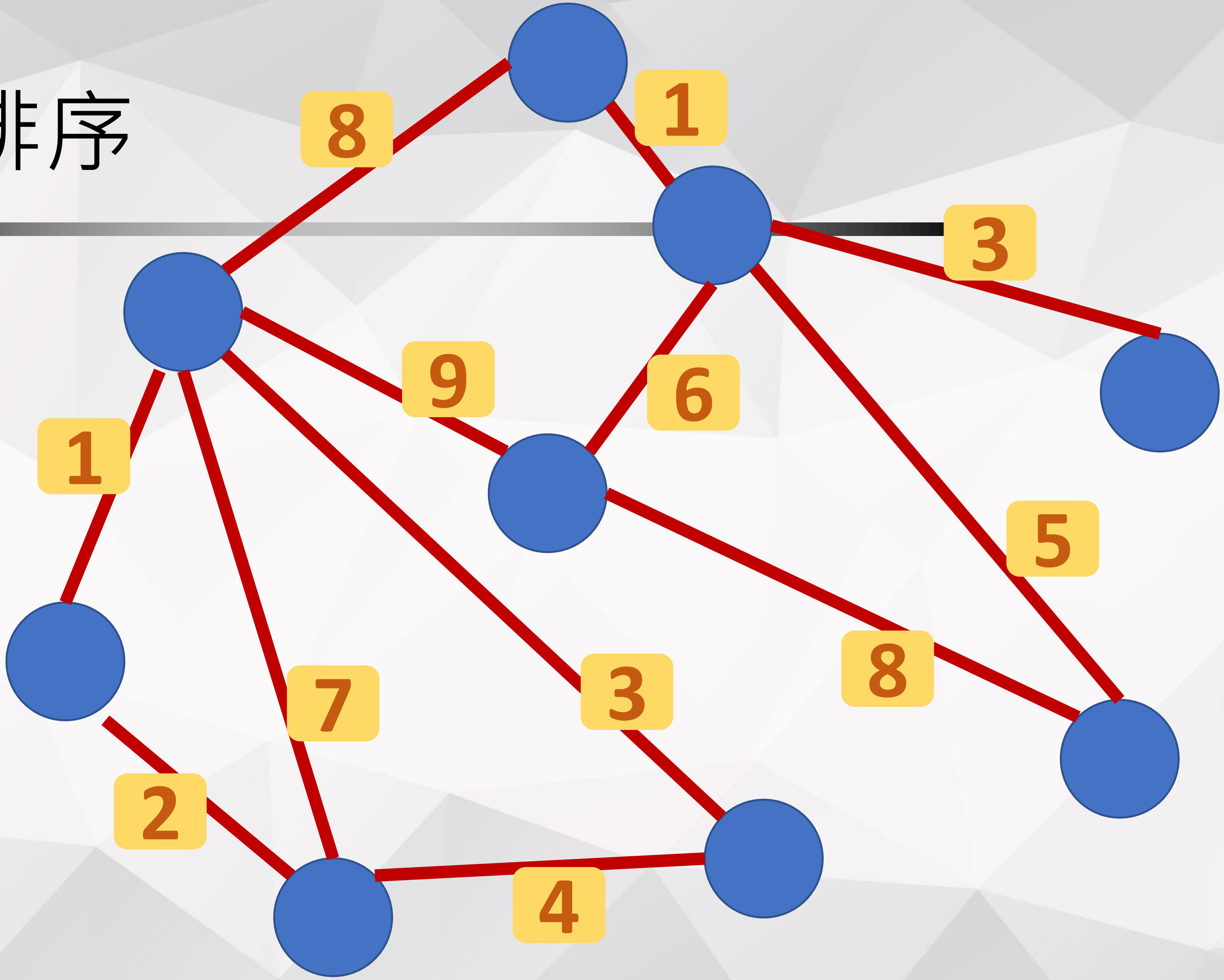


Competitive Programming

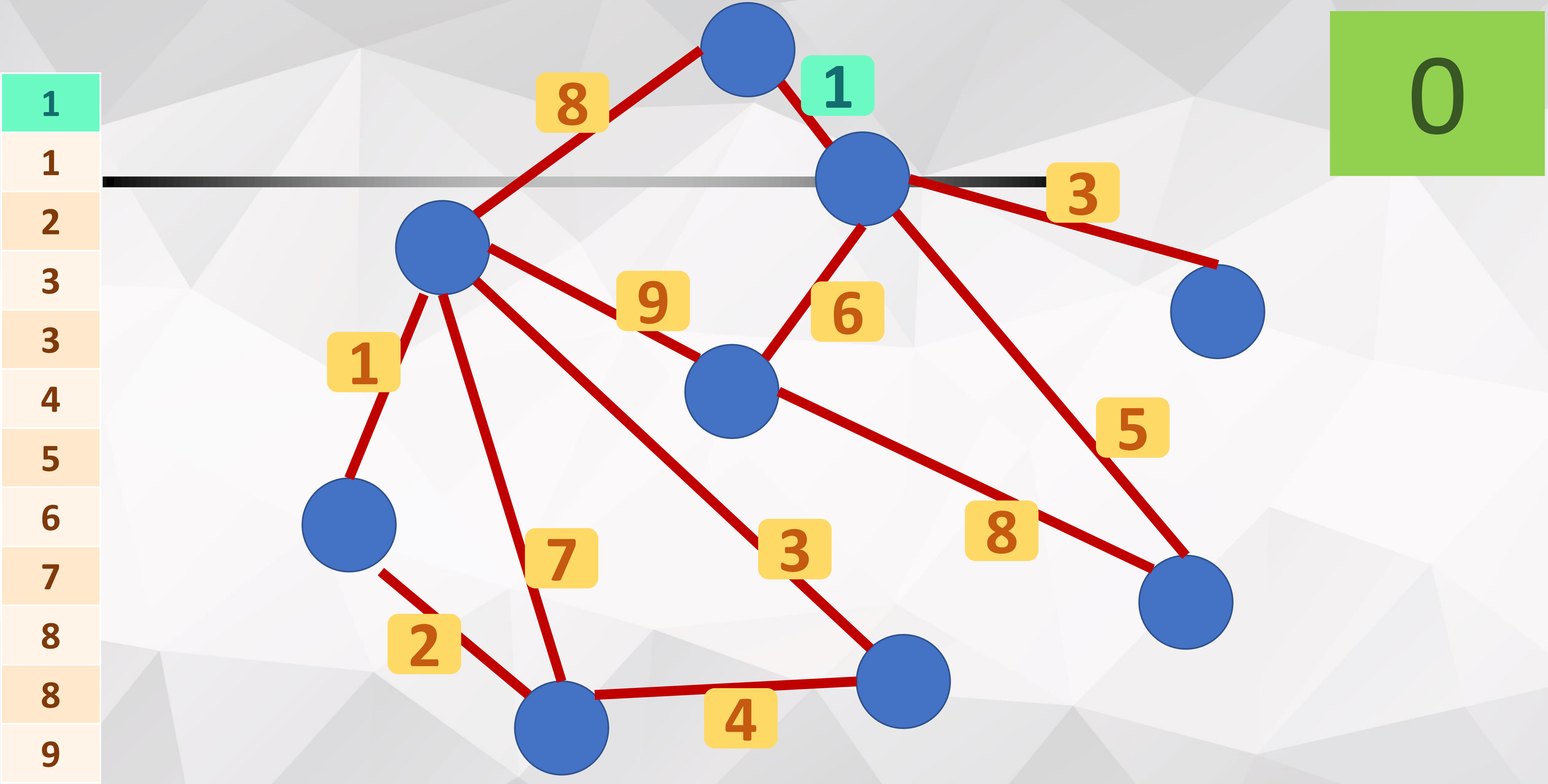
權重排序

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

0



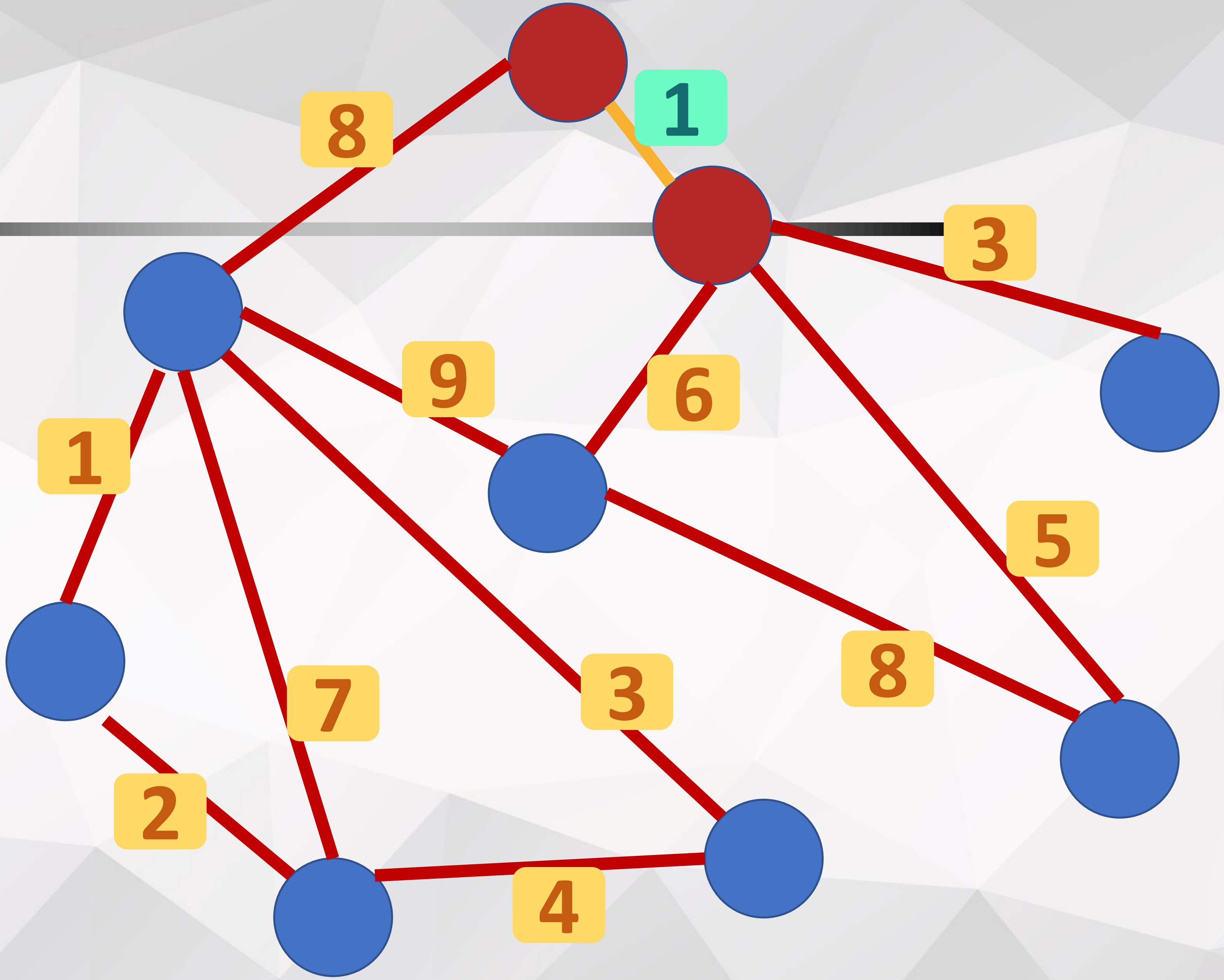
Competitive Programming



0
1
1
2
3
3
4
5
6
7
8
8
9
8
1
3
9
6
1
5
7
3
8
2
4

檢查

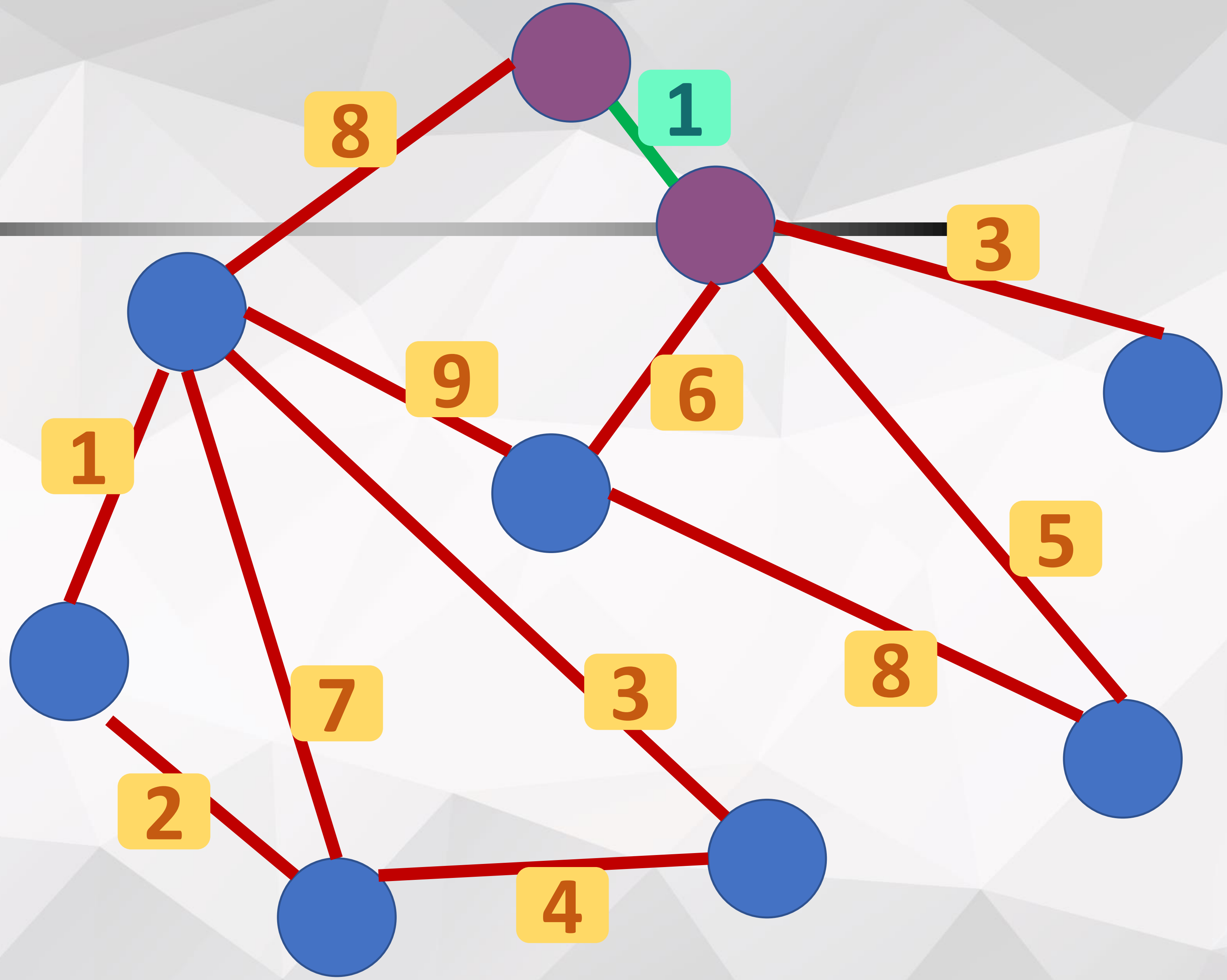
1
1
2
3
3
4
5
6
7
8
8
9



0

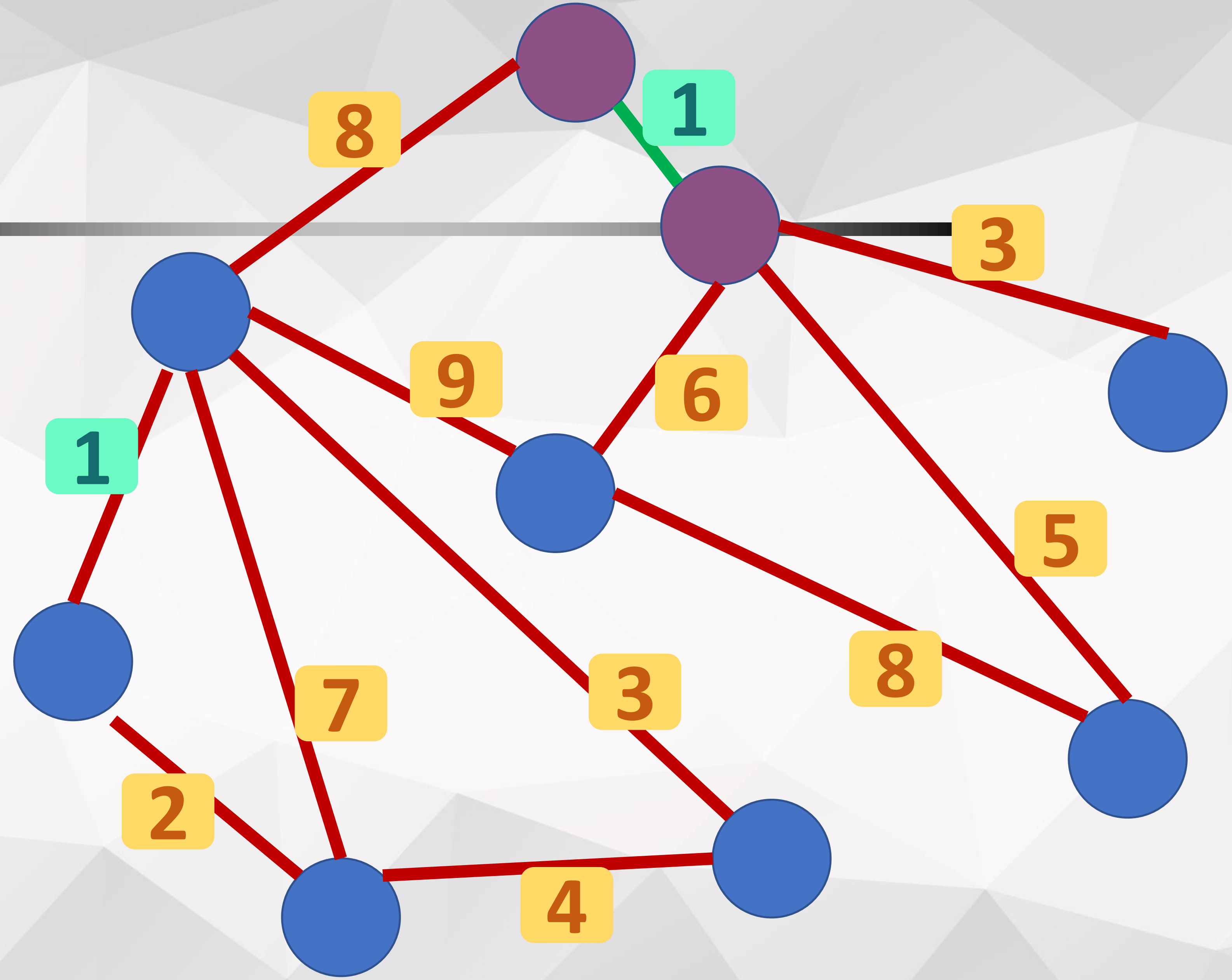
相連

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



1

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

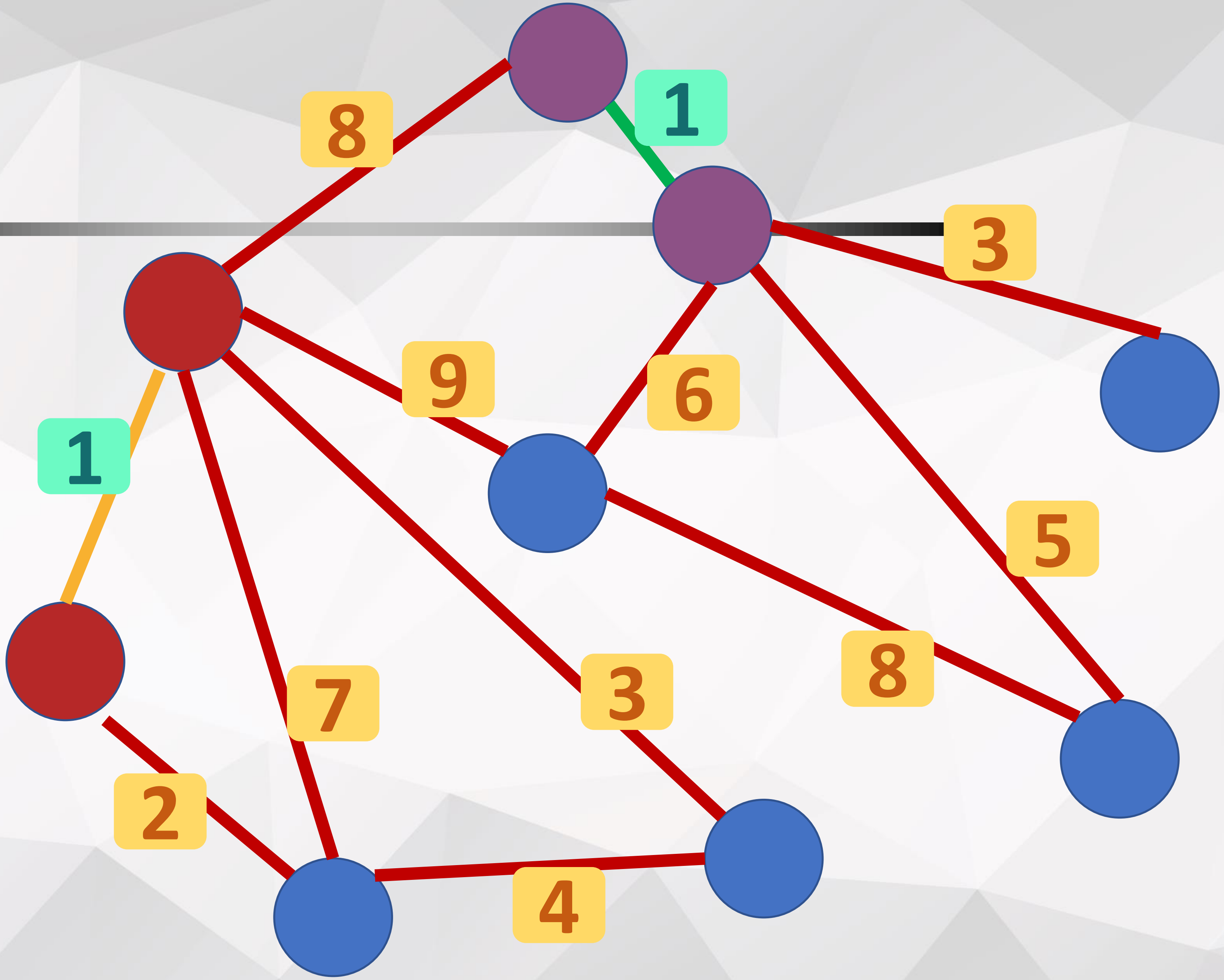


1

Competitive Programming

檢查

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

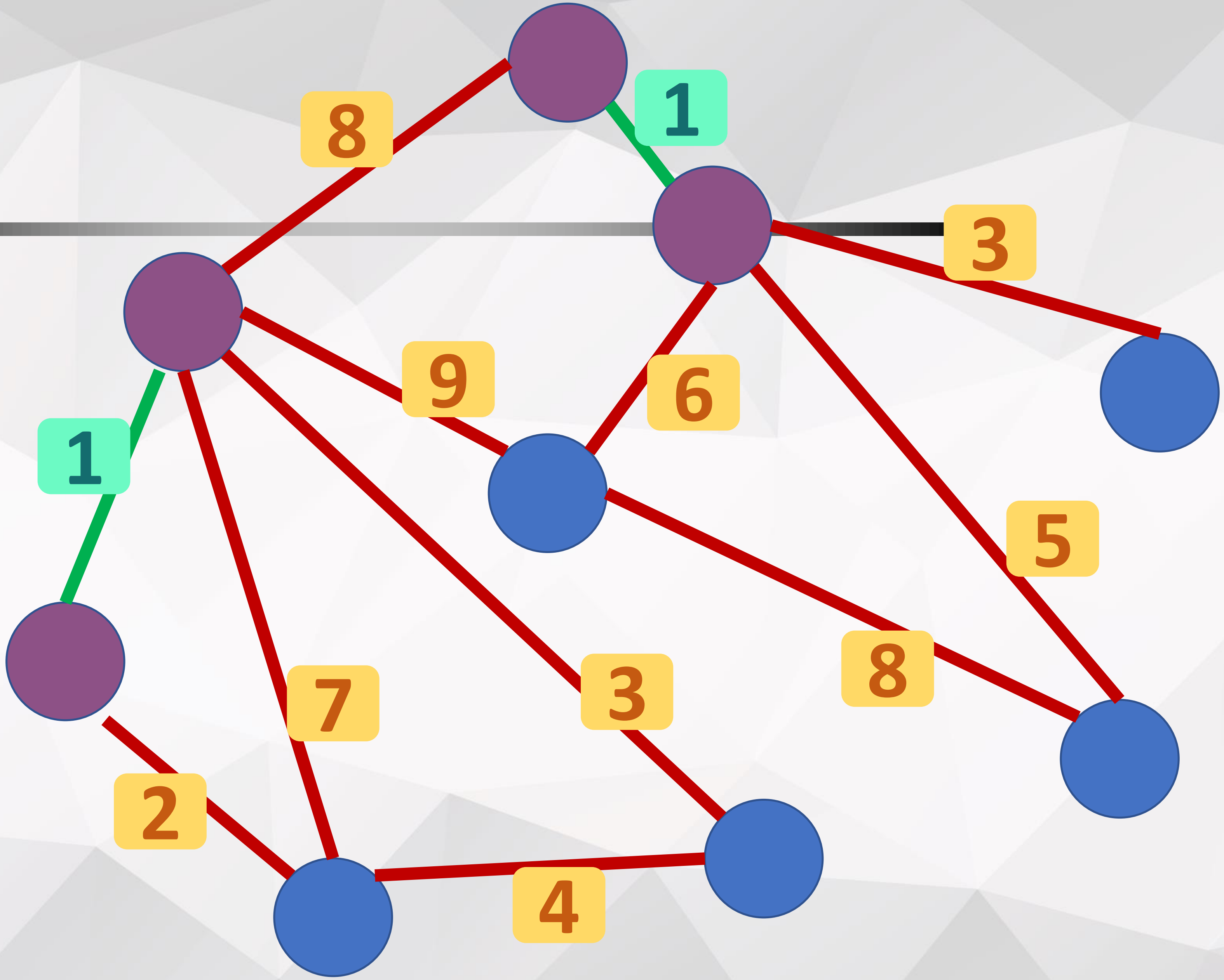


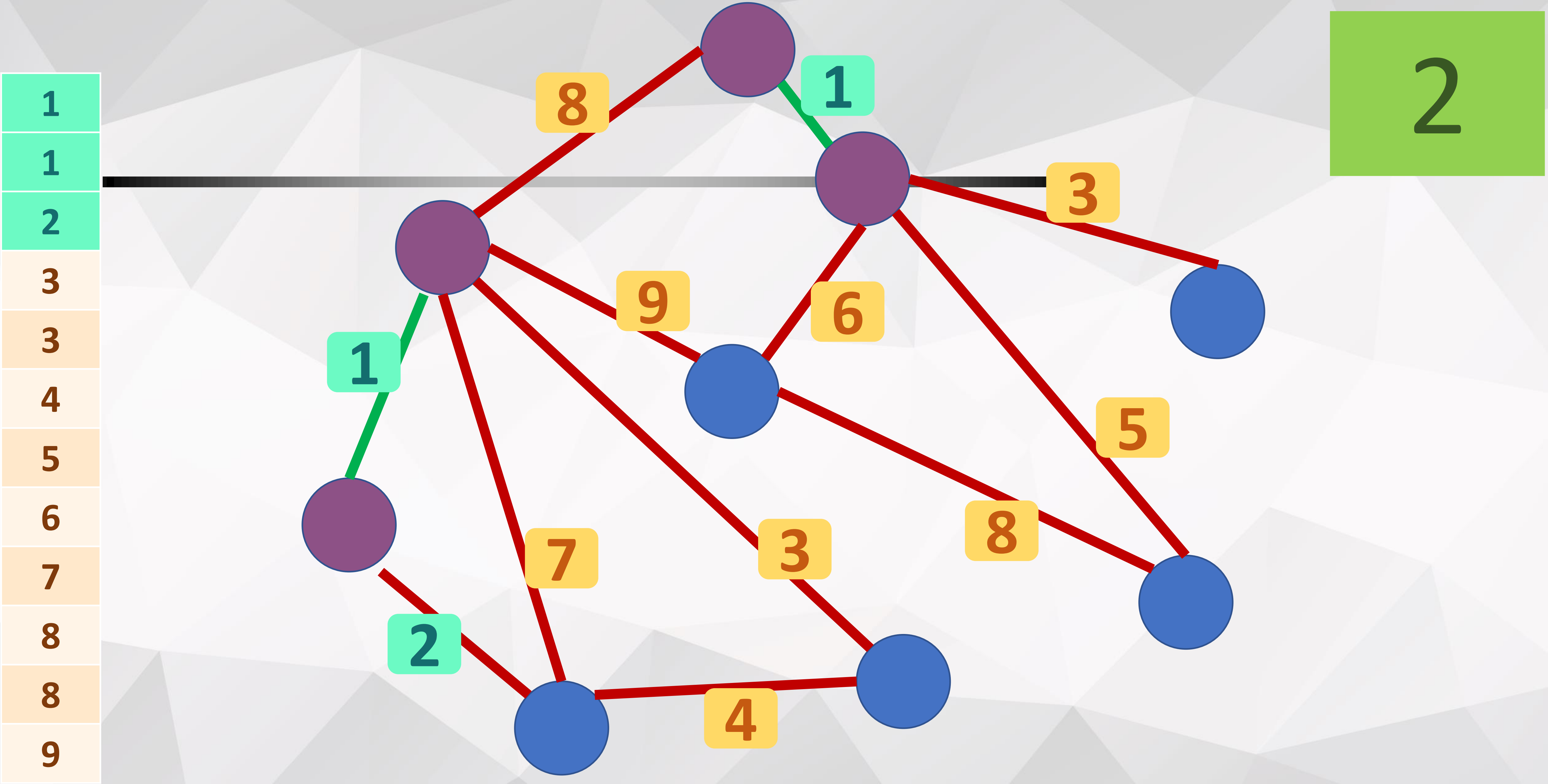
1

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

相連

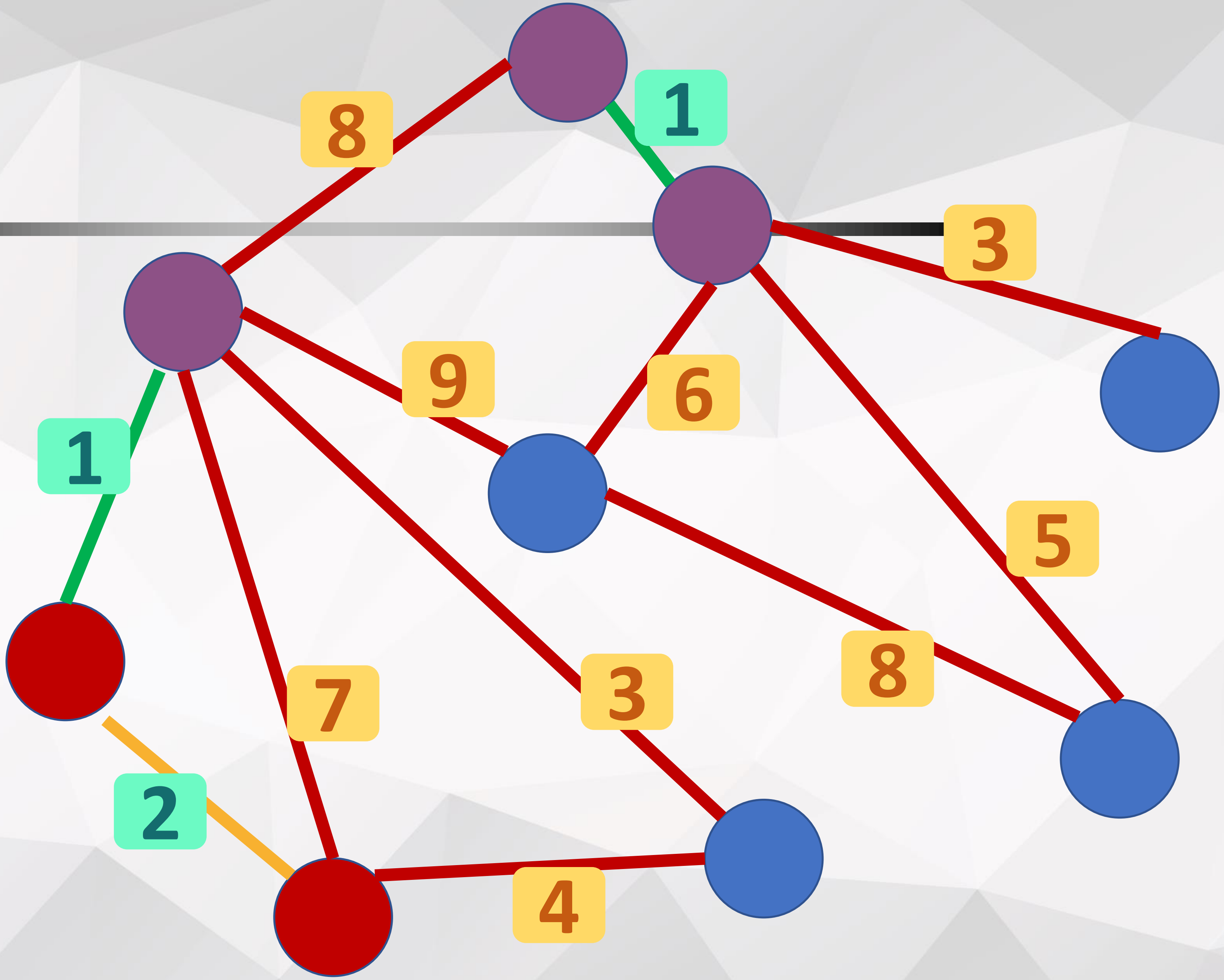
2




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

檢查

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

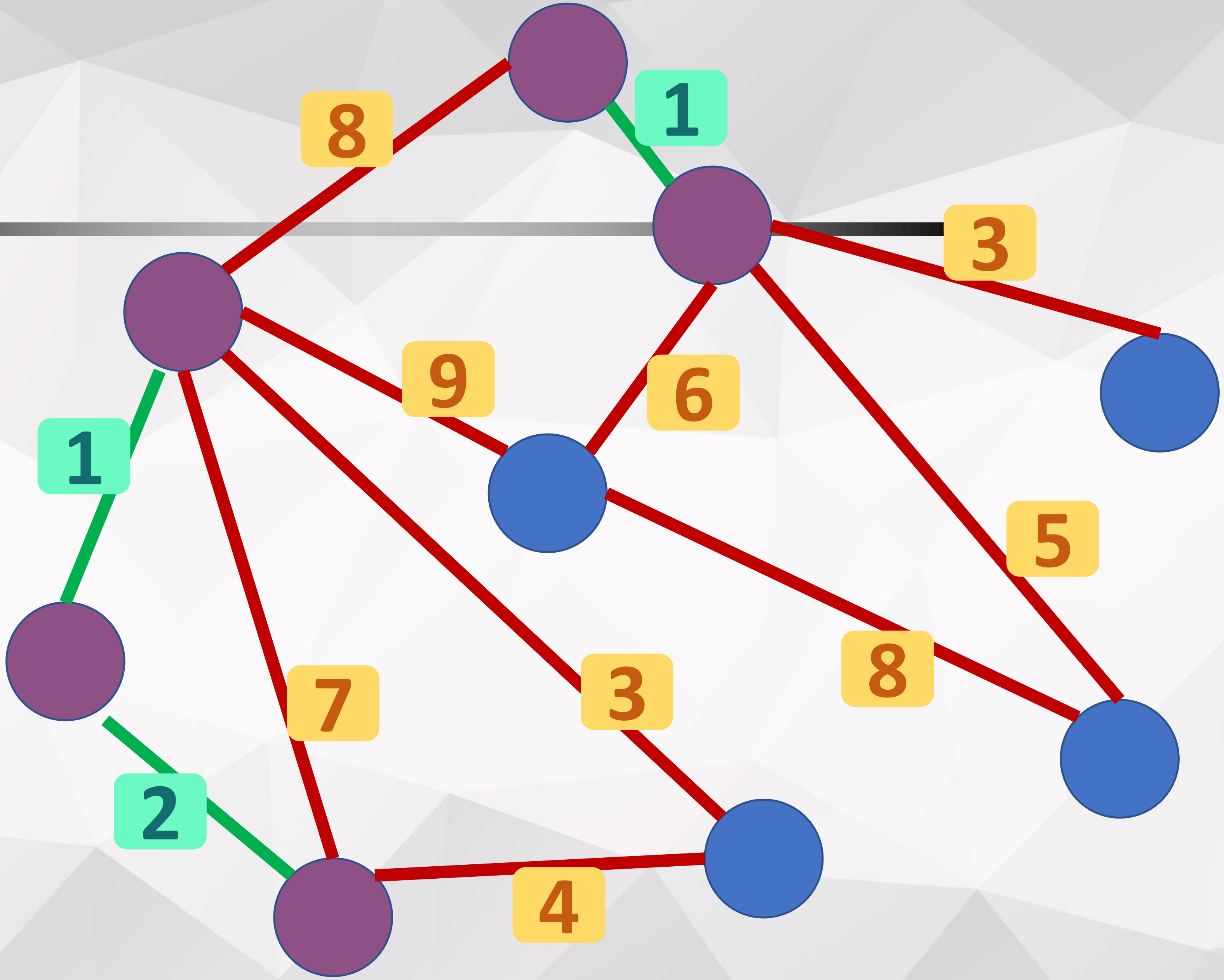


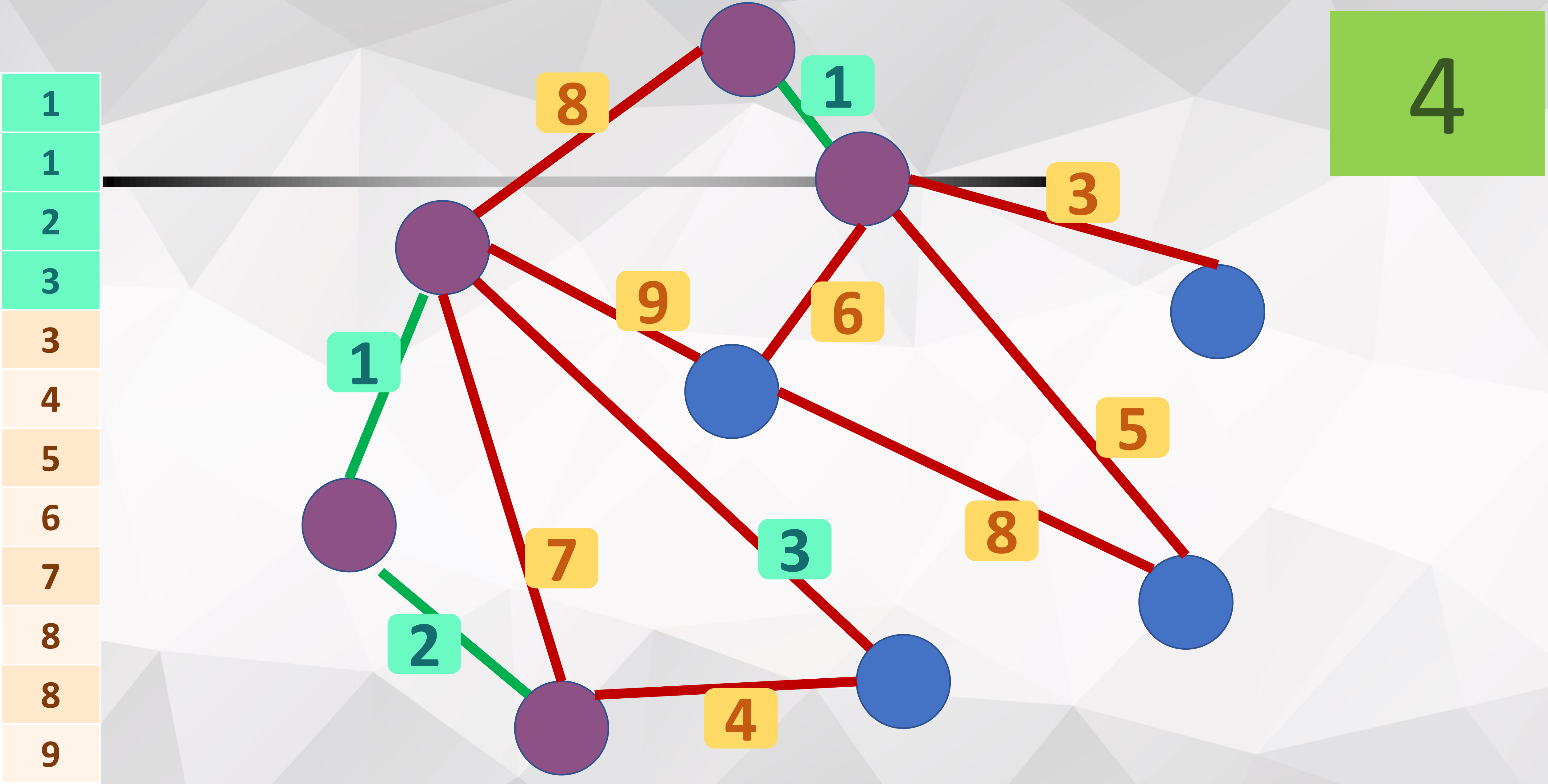
2

相連

4

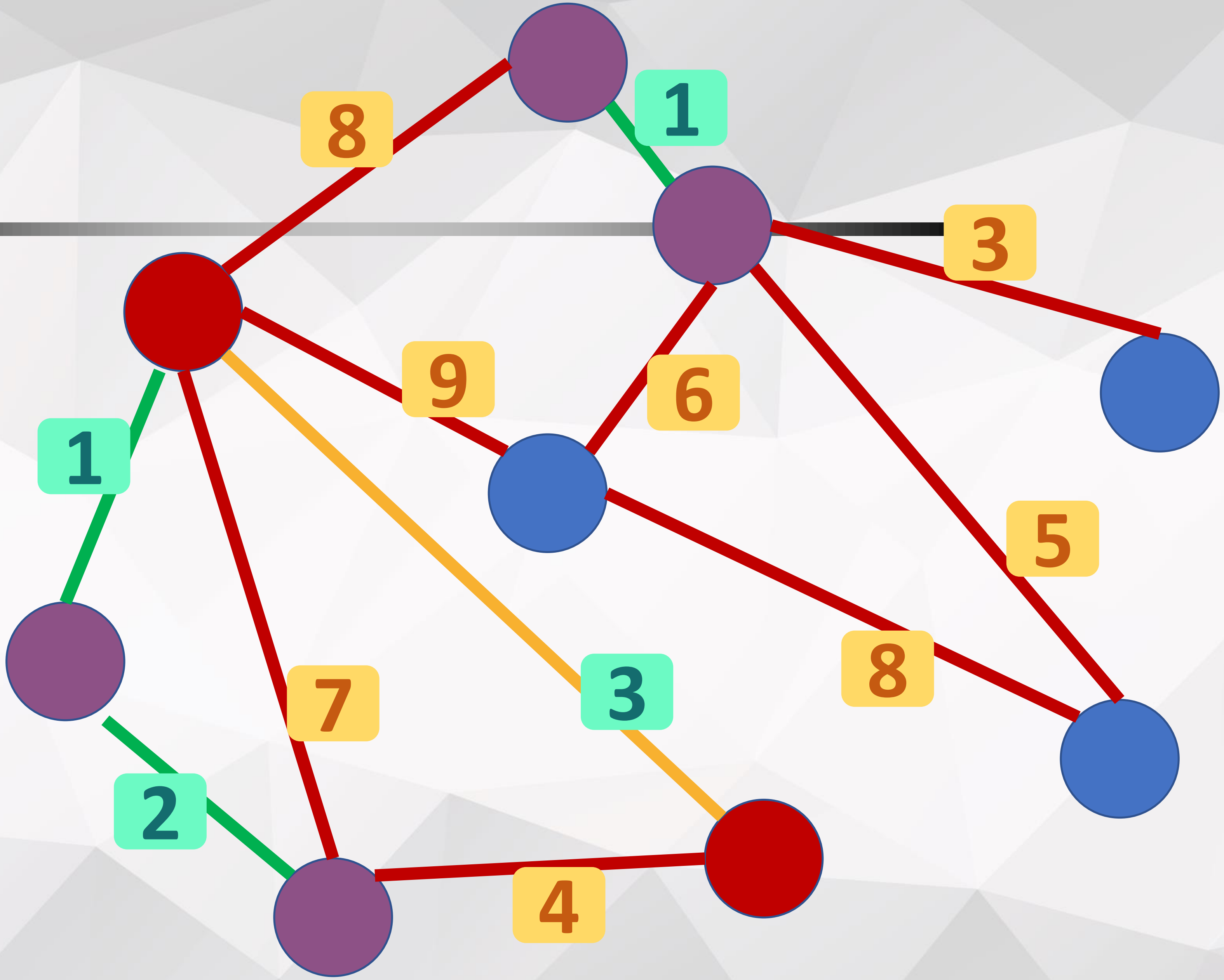
- 1
- 1
- 2
- 3
- 3
- 4
- 5
- 6
- 7
- 8
- 8
- 9




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

檢查

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



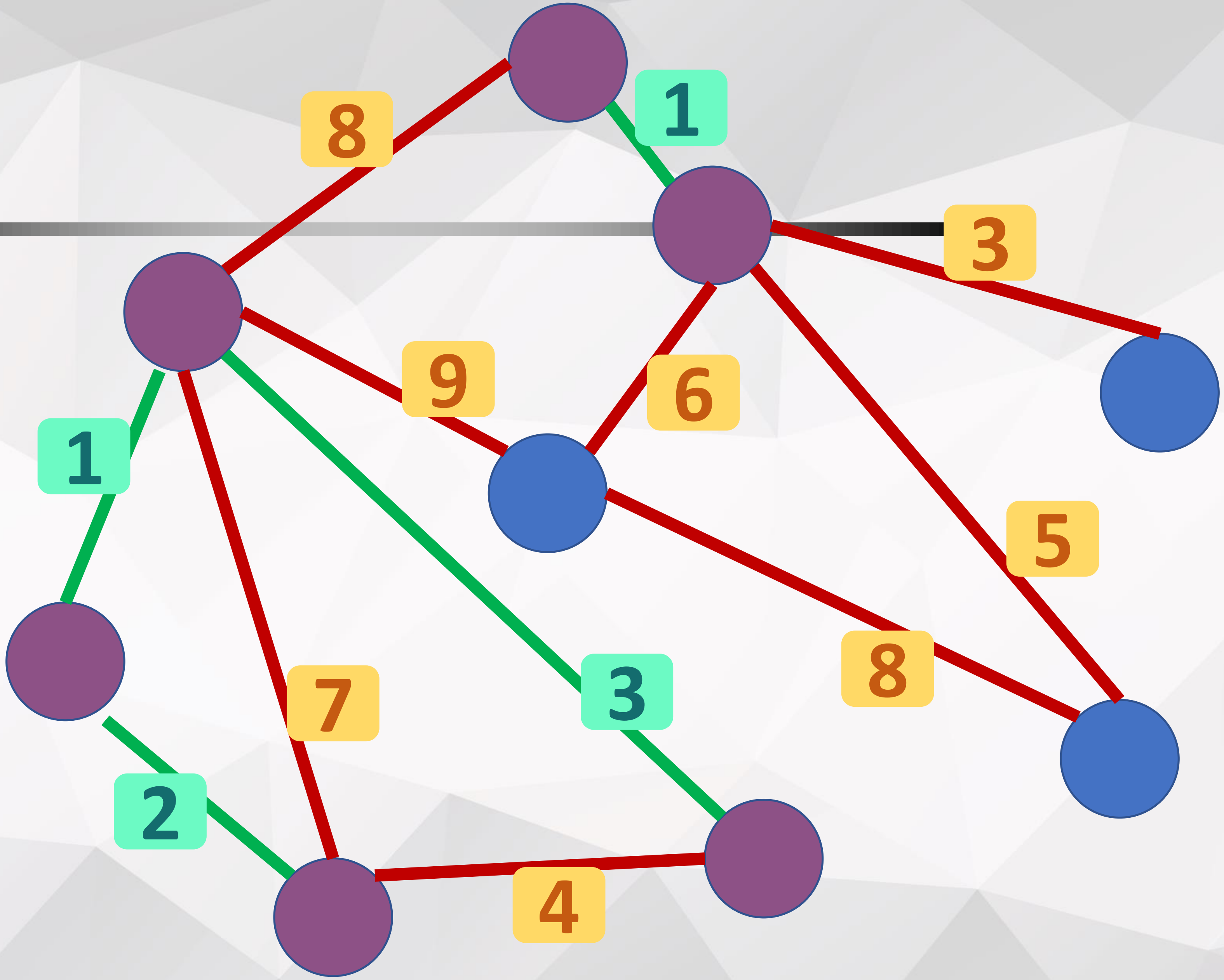
4

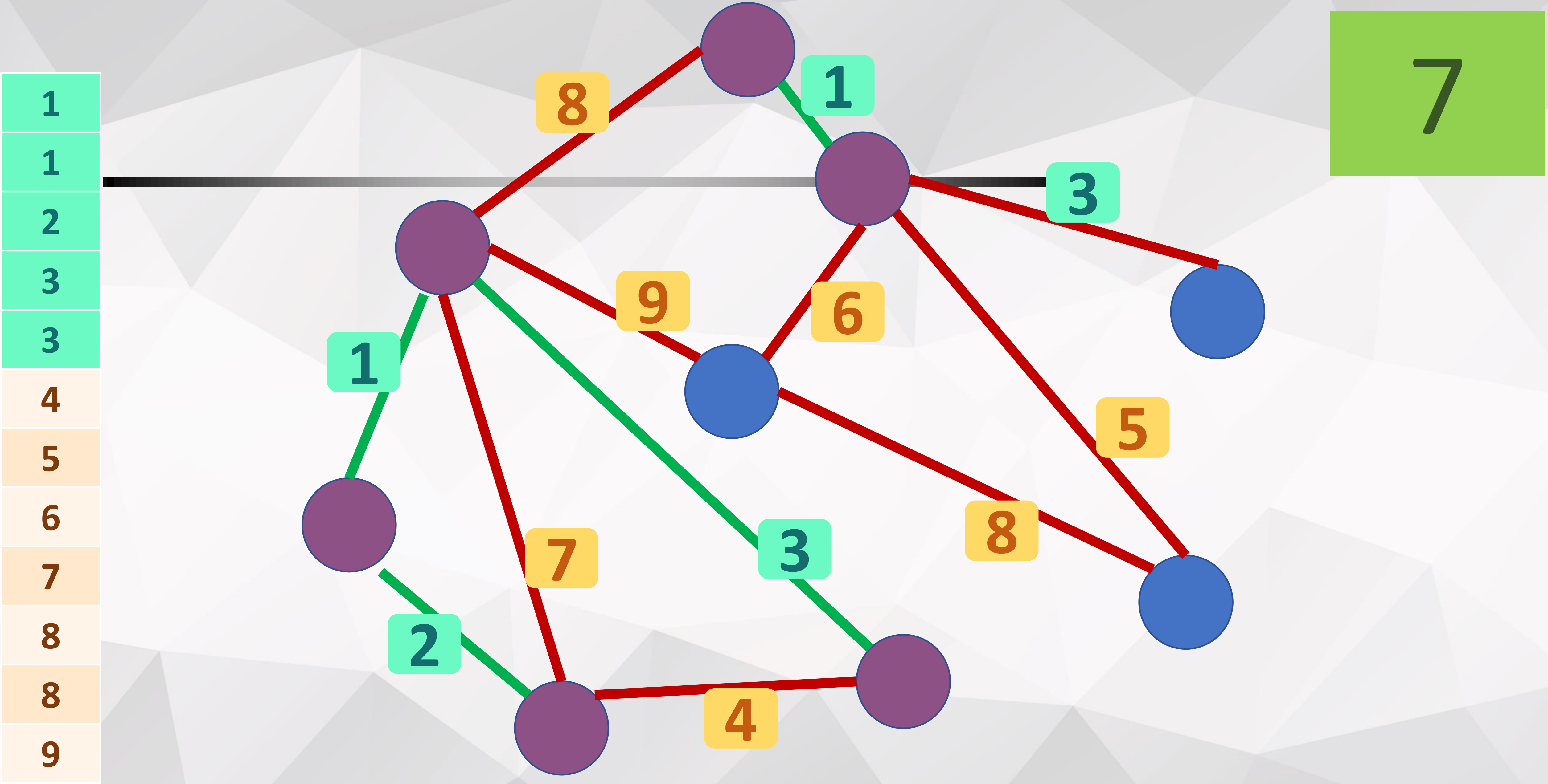
Competitive Programming

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

相連

7

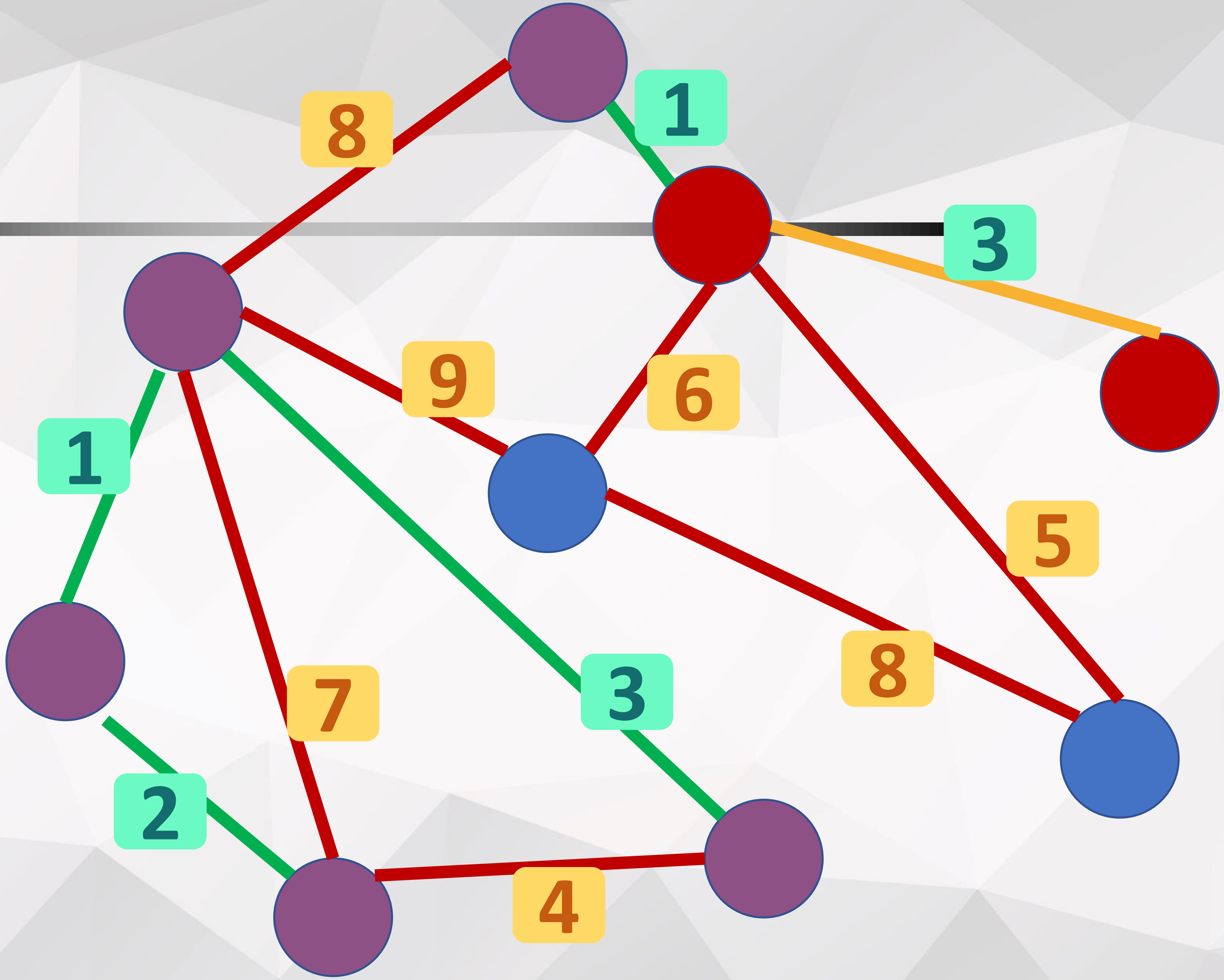




Competitive Programming

檢查

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

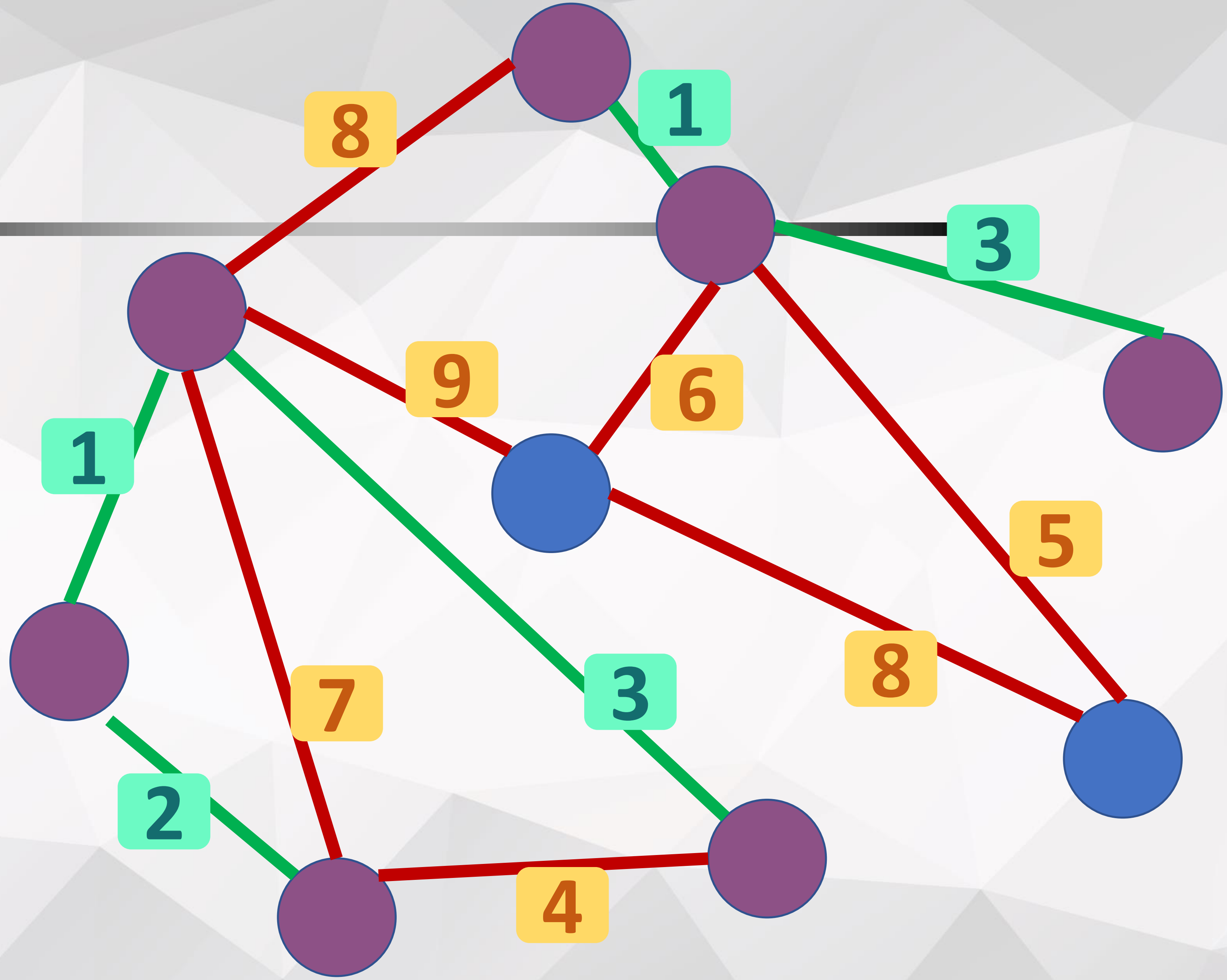


7

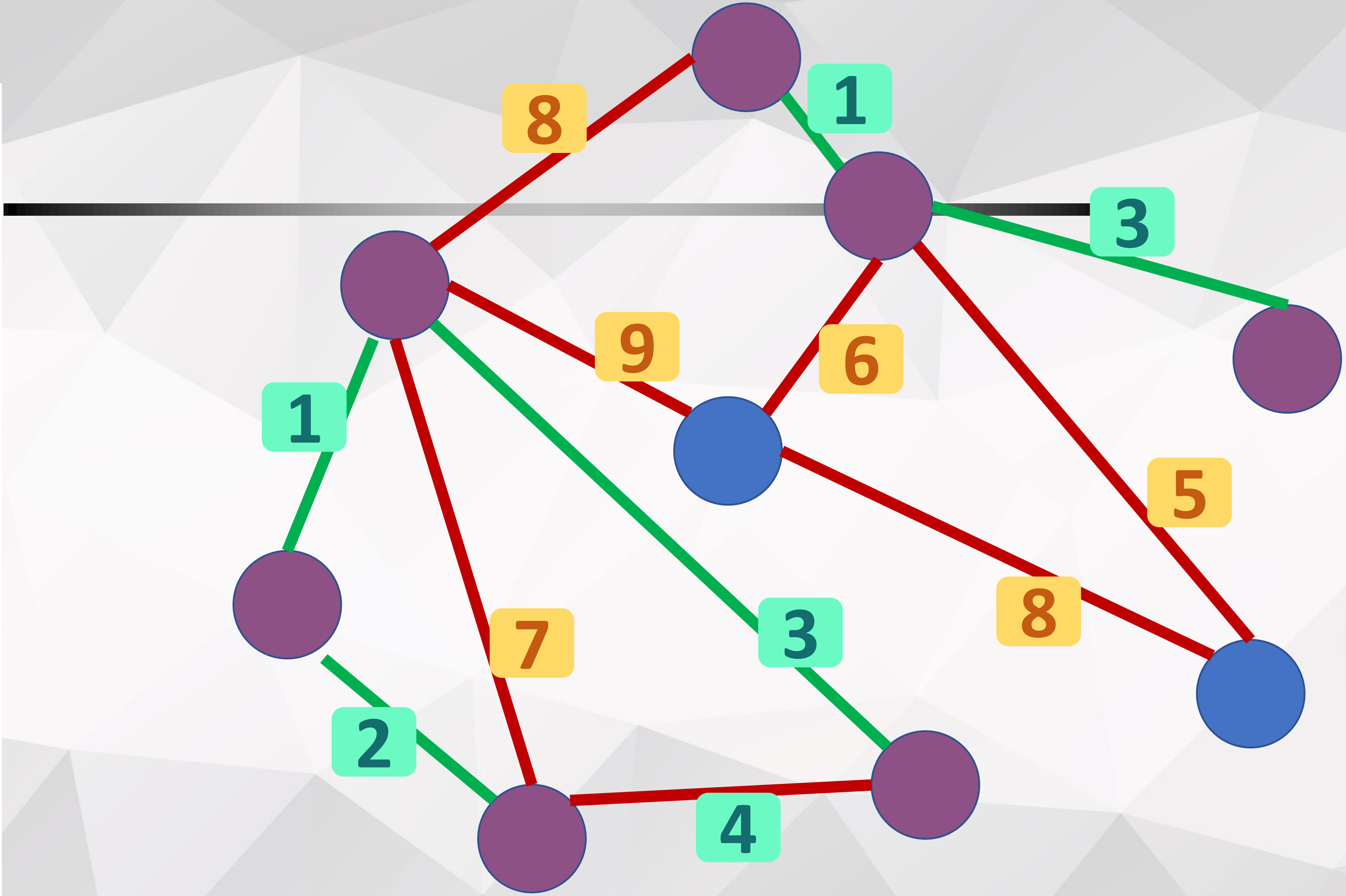
相連

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

10



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

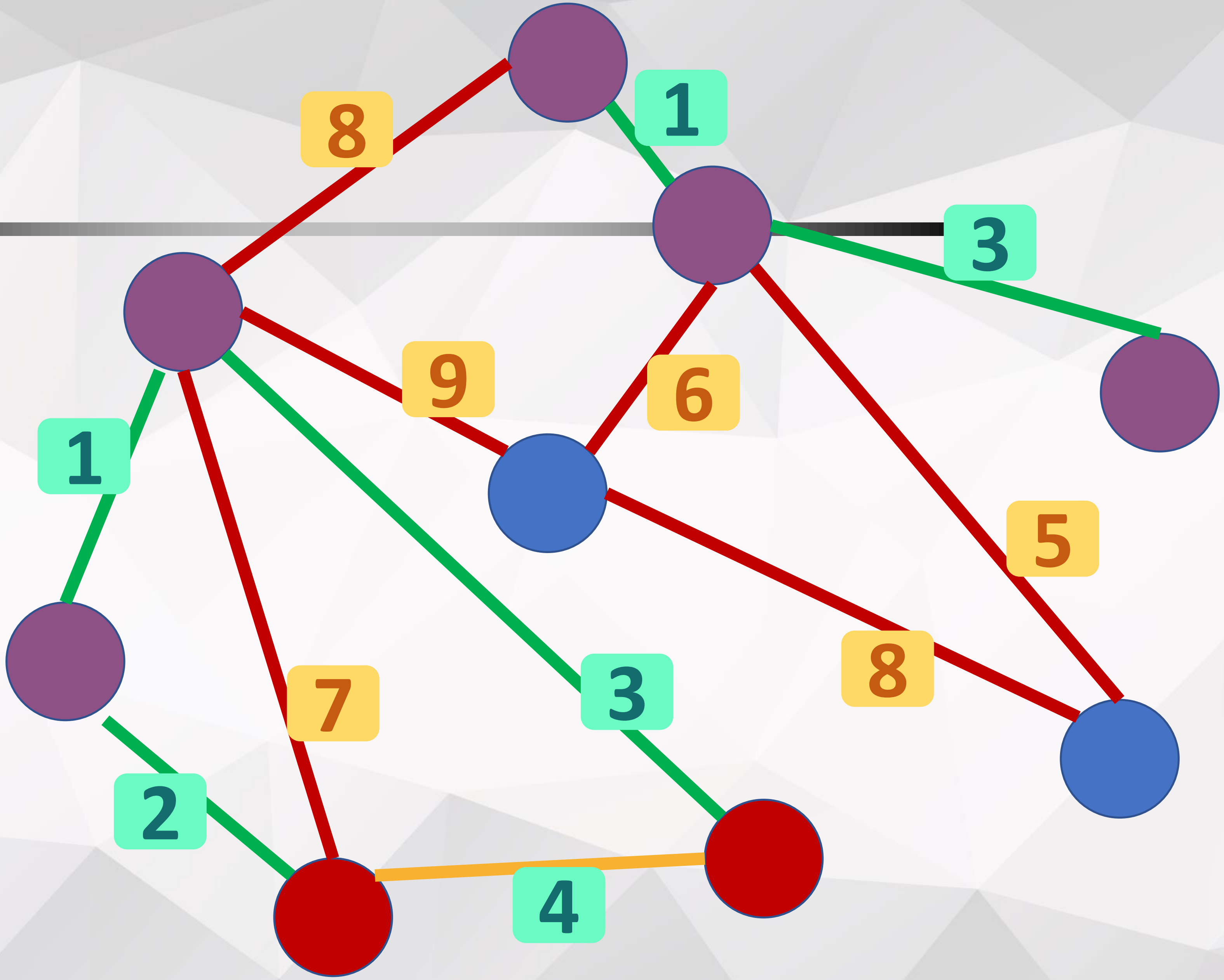


10

檢查

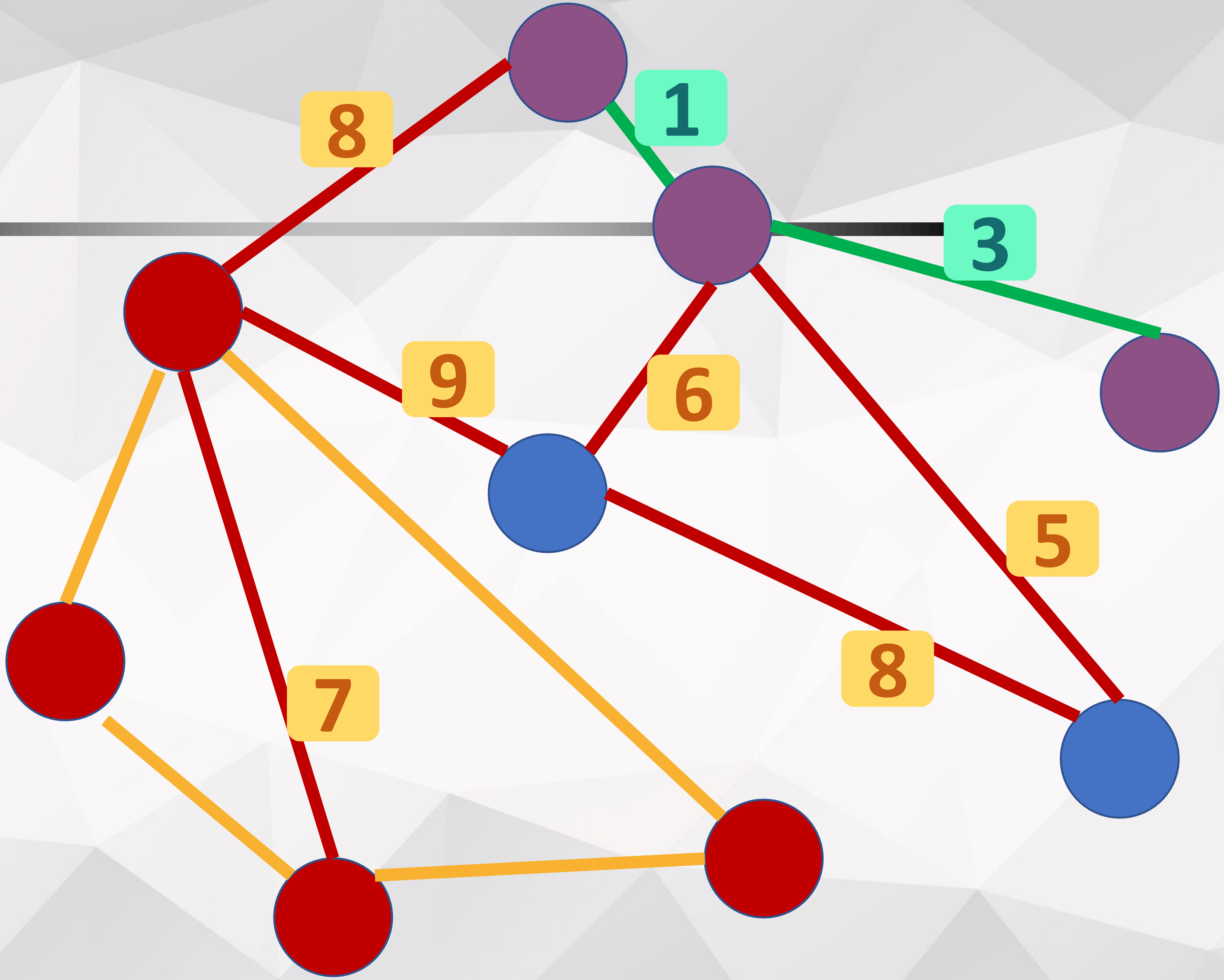
1
1
2
3
3
4
5
6
7
8
8
9

10

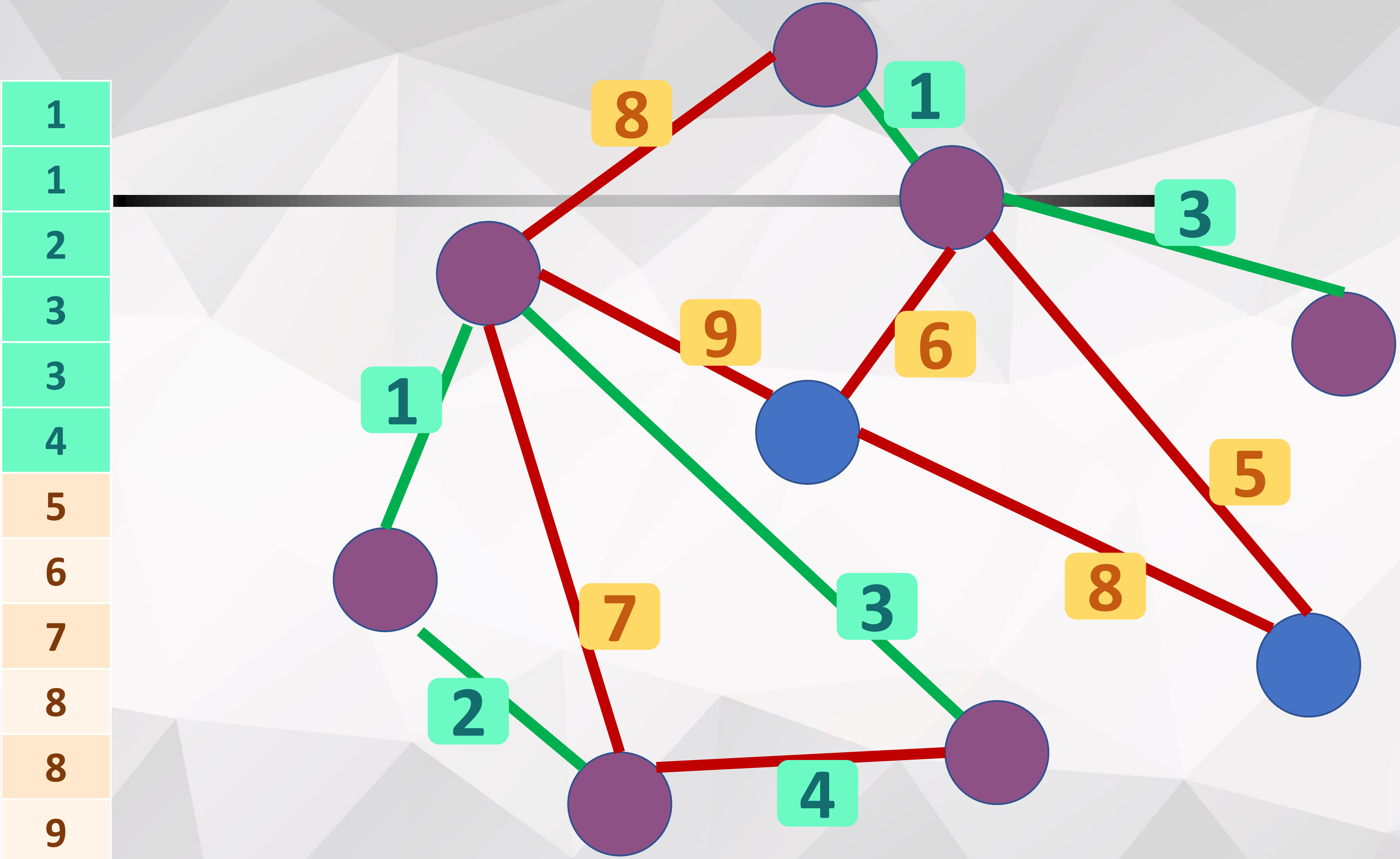


1
1
2
3
3
4
5
6
7
8
8
9

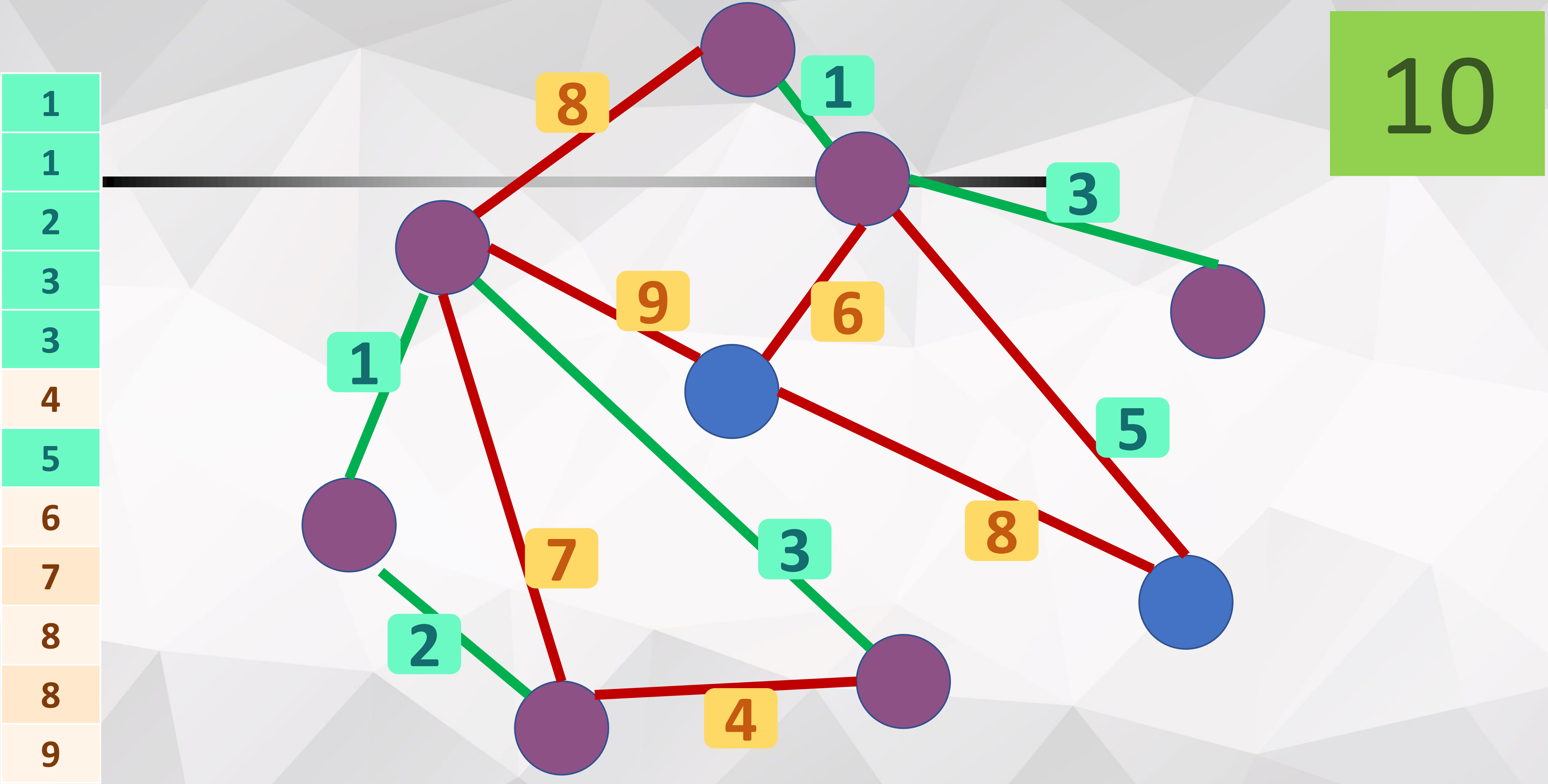
環



(⊙д⊙)



10

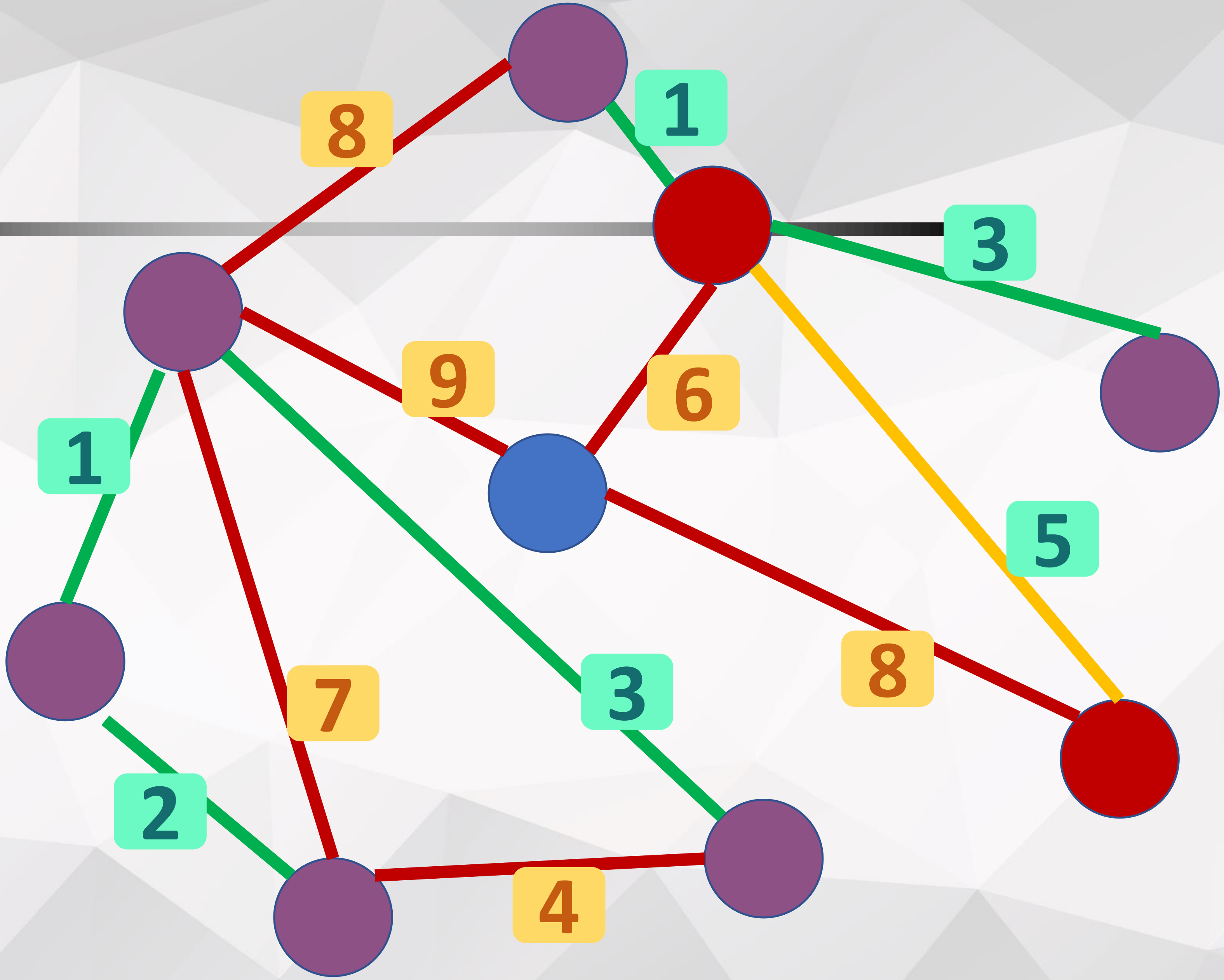


Competitive Programming

檢查

10

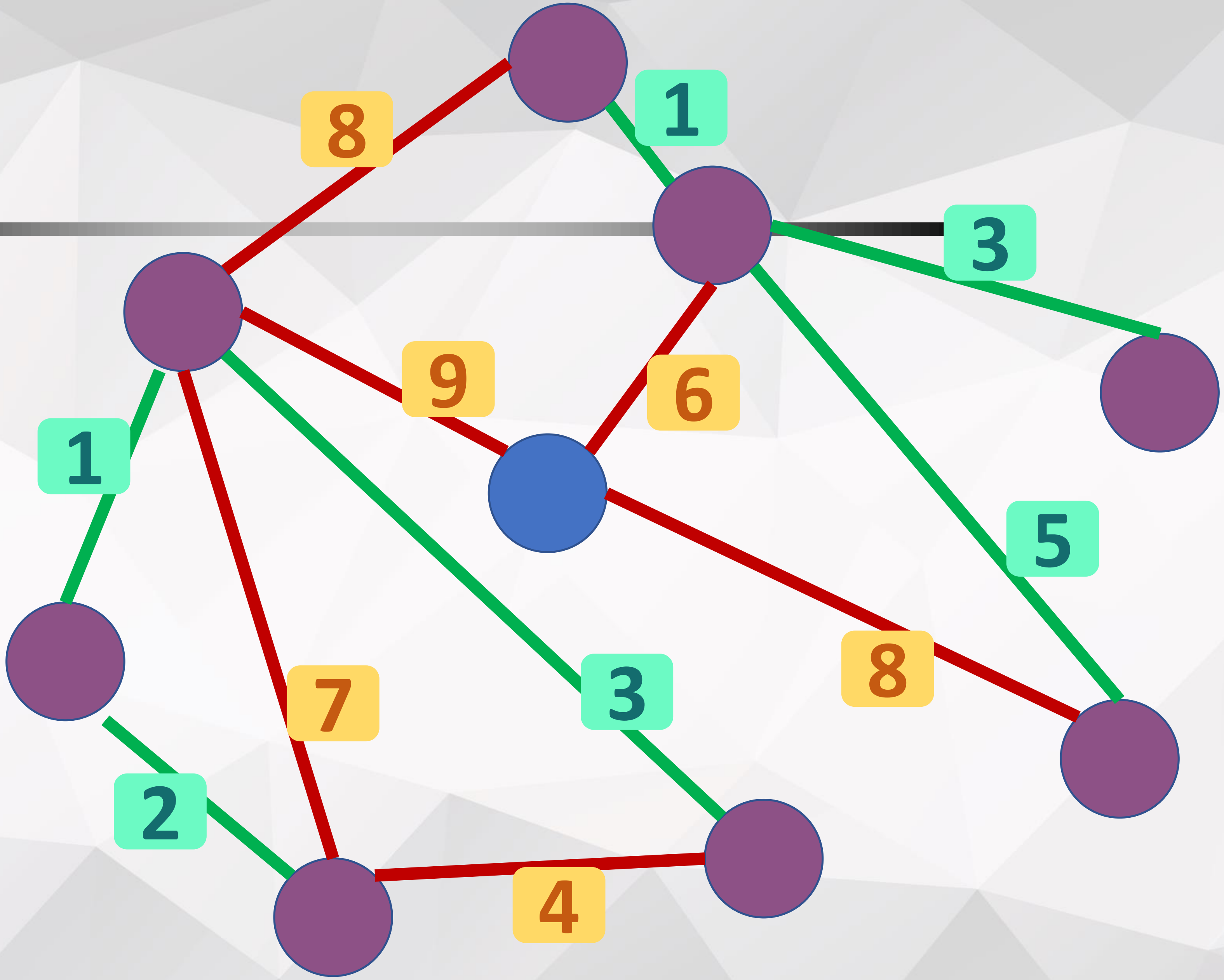
1
1
2
3
3
4
5
6
7
8
8
9

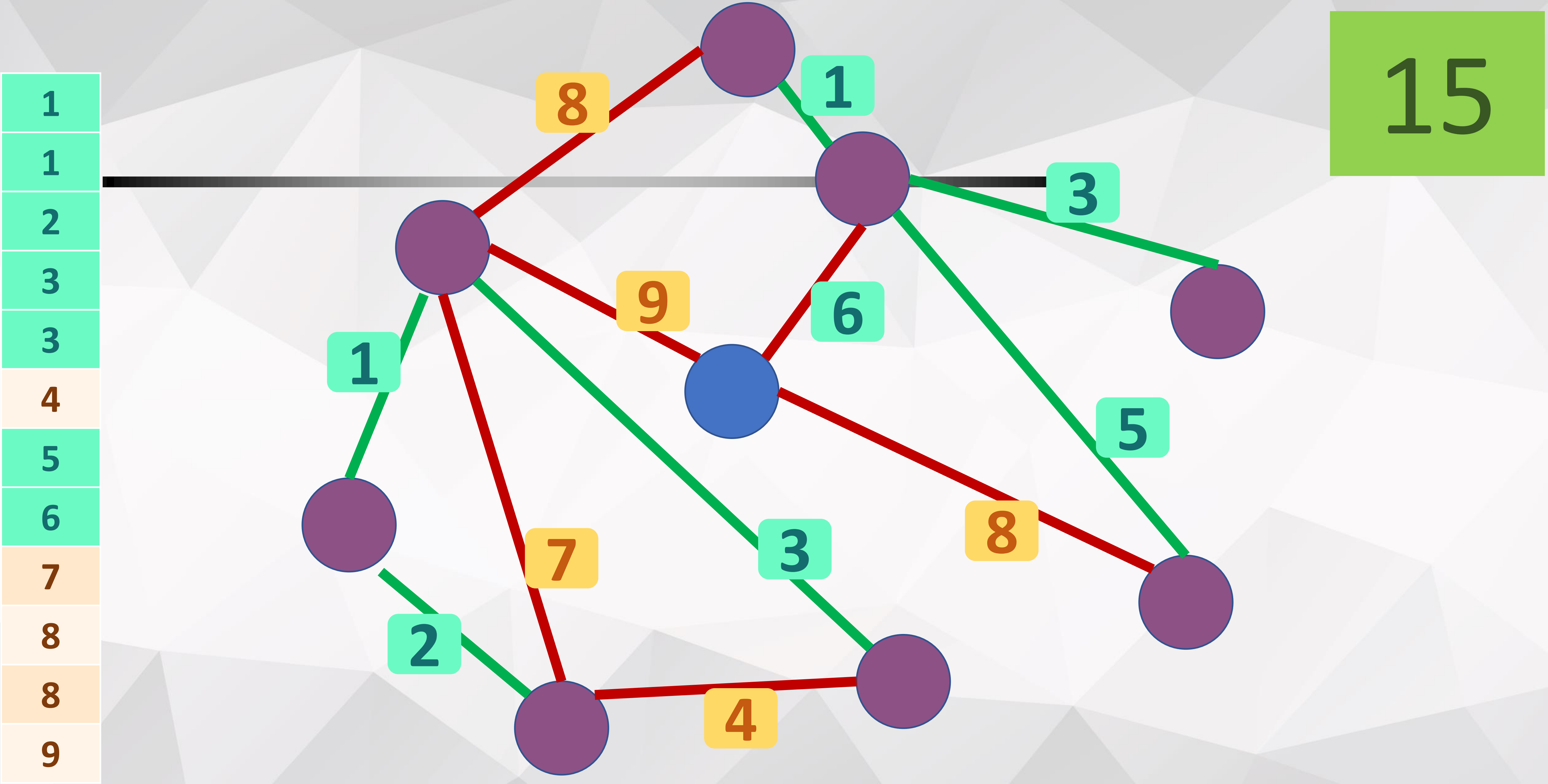


相連

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

15

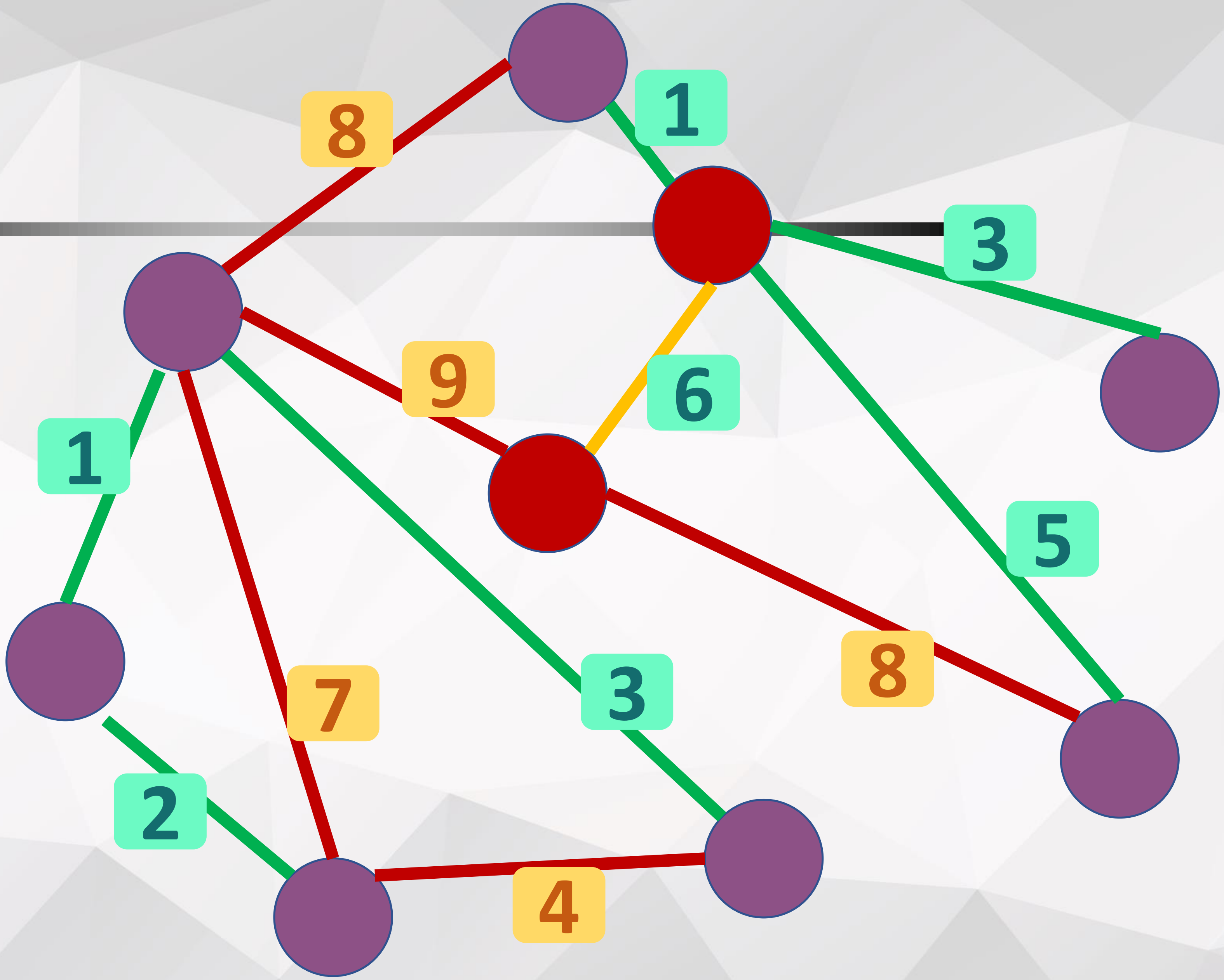



1
1
2
3
3
4
5
6
7
8
8
9
15
8
1
3
9
6
1
5
7
3
8
2
4

檢查

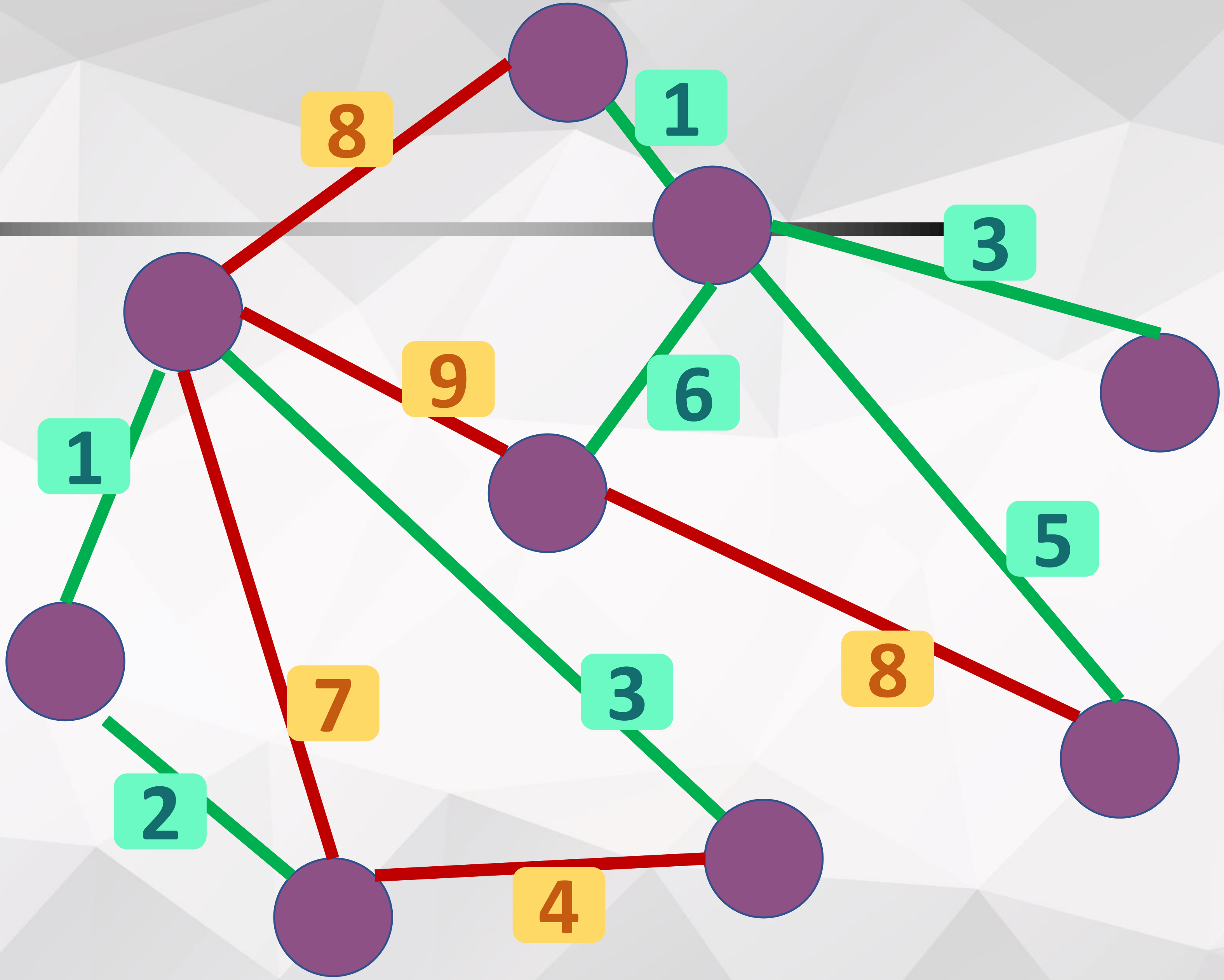
15

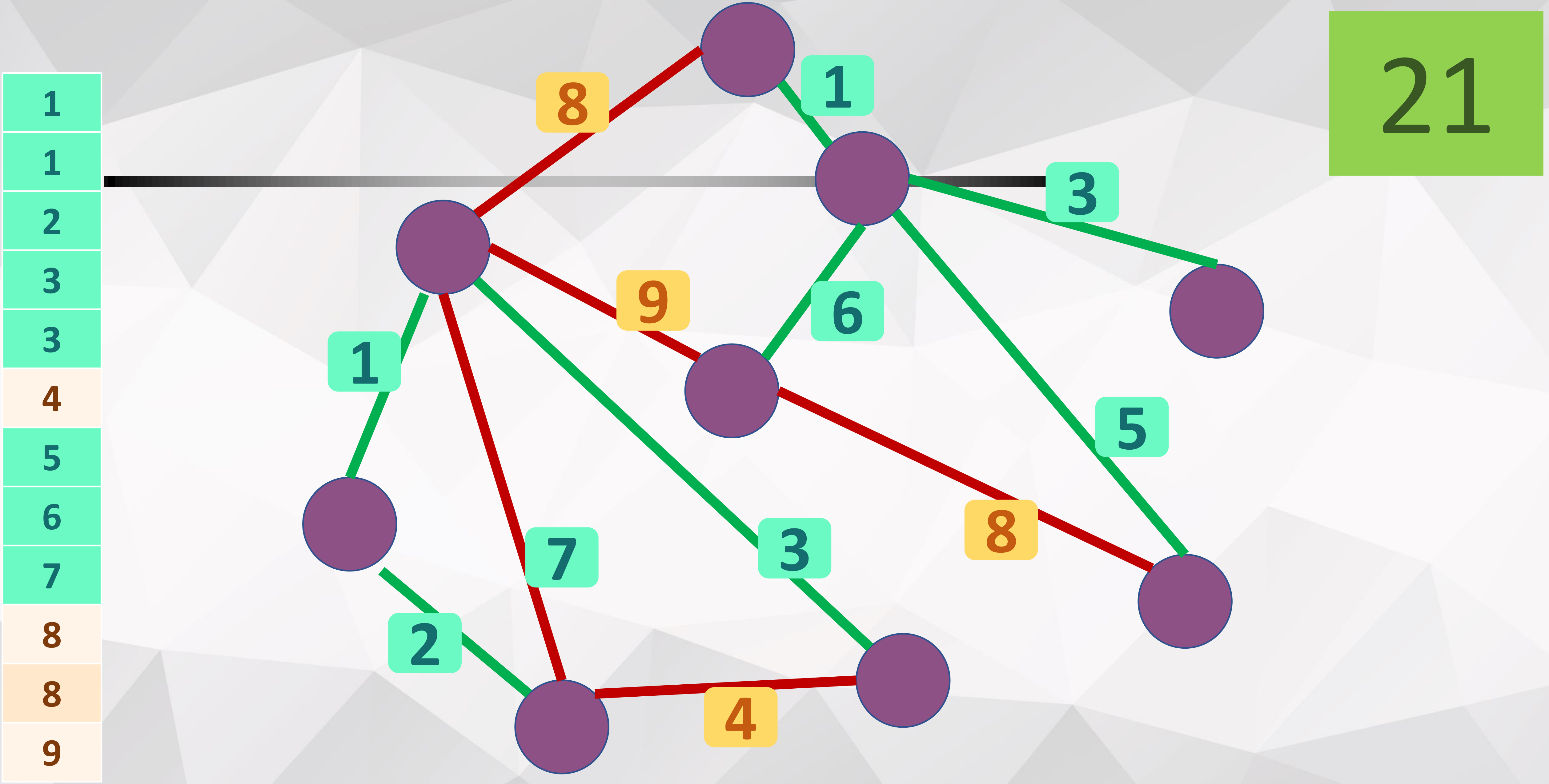
1
1
2
3
3
4
5
6
7
8
8
9



相連

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

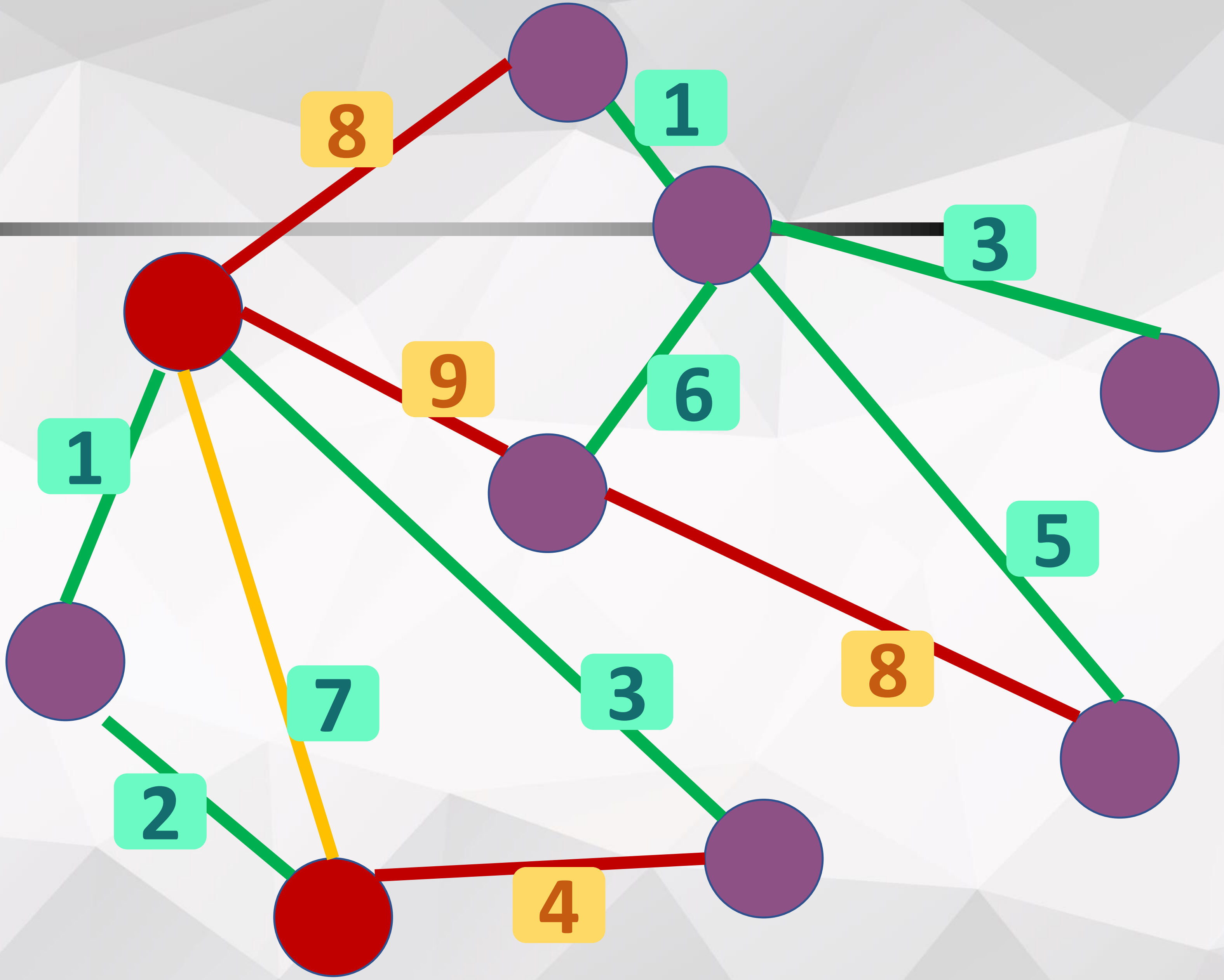




1
1
2
3
3
4
5
6
7
8
8
9
21
8
1
3
9
6
1
5
7
3
8
2
4

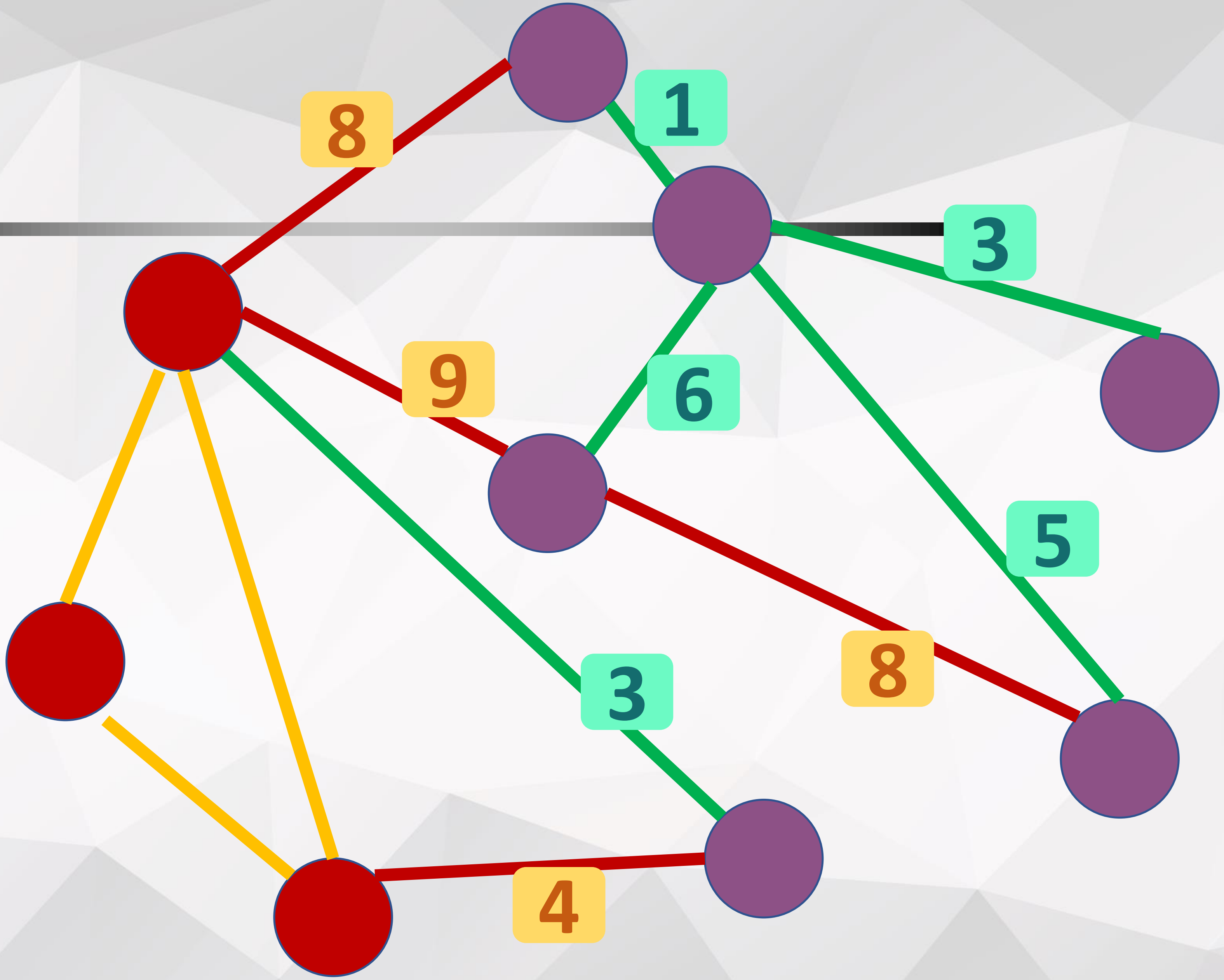
檢查

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

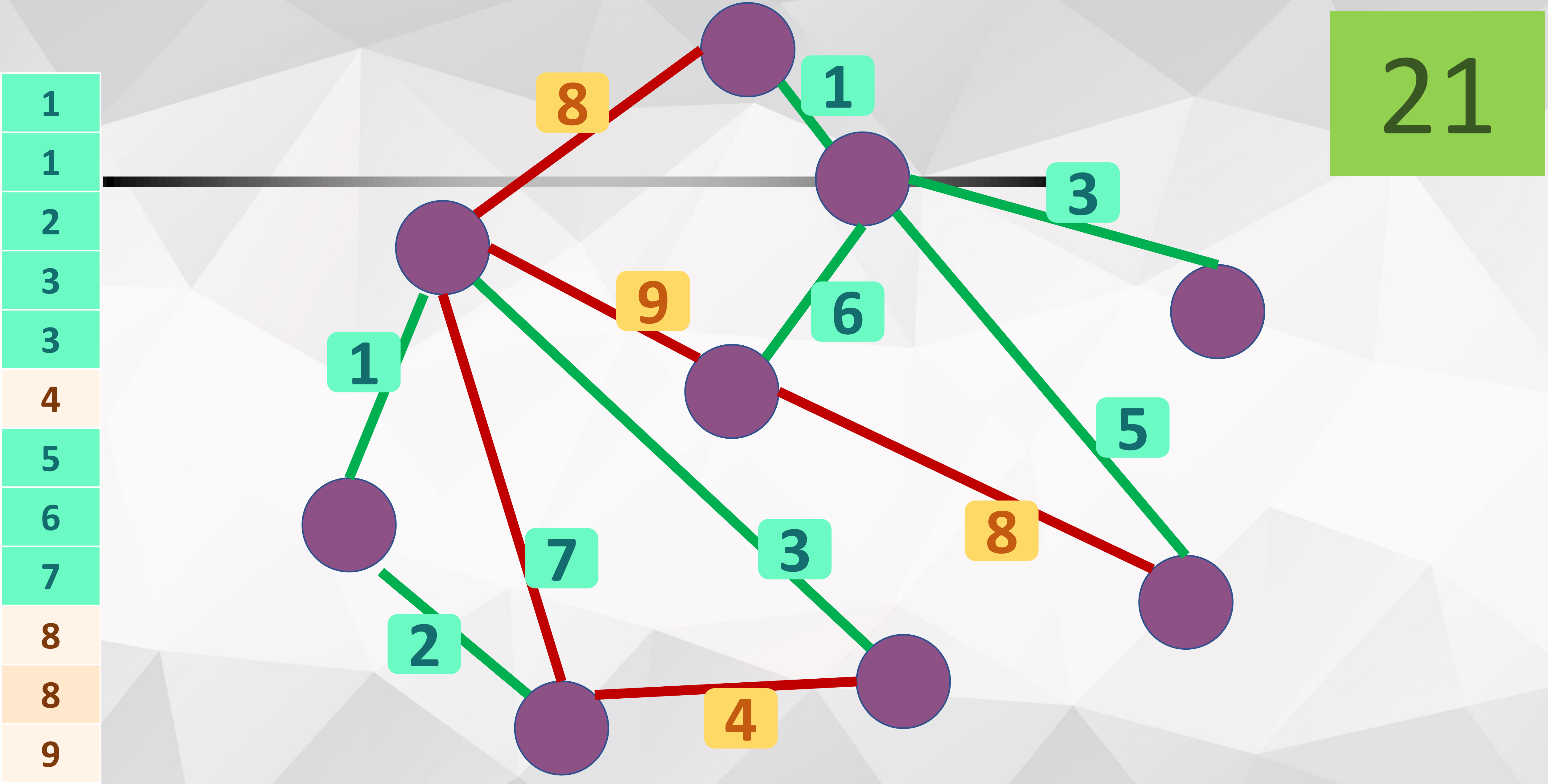


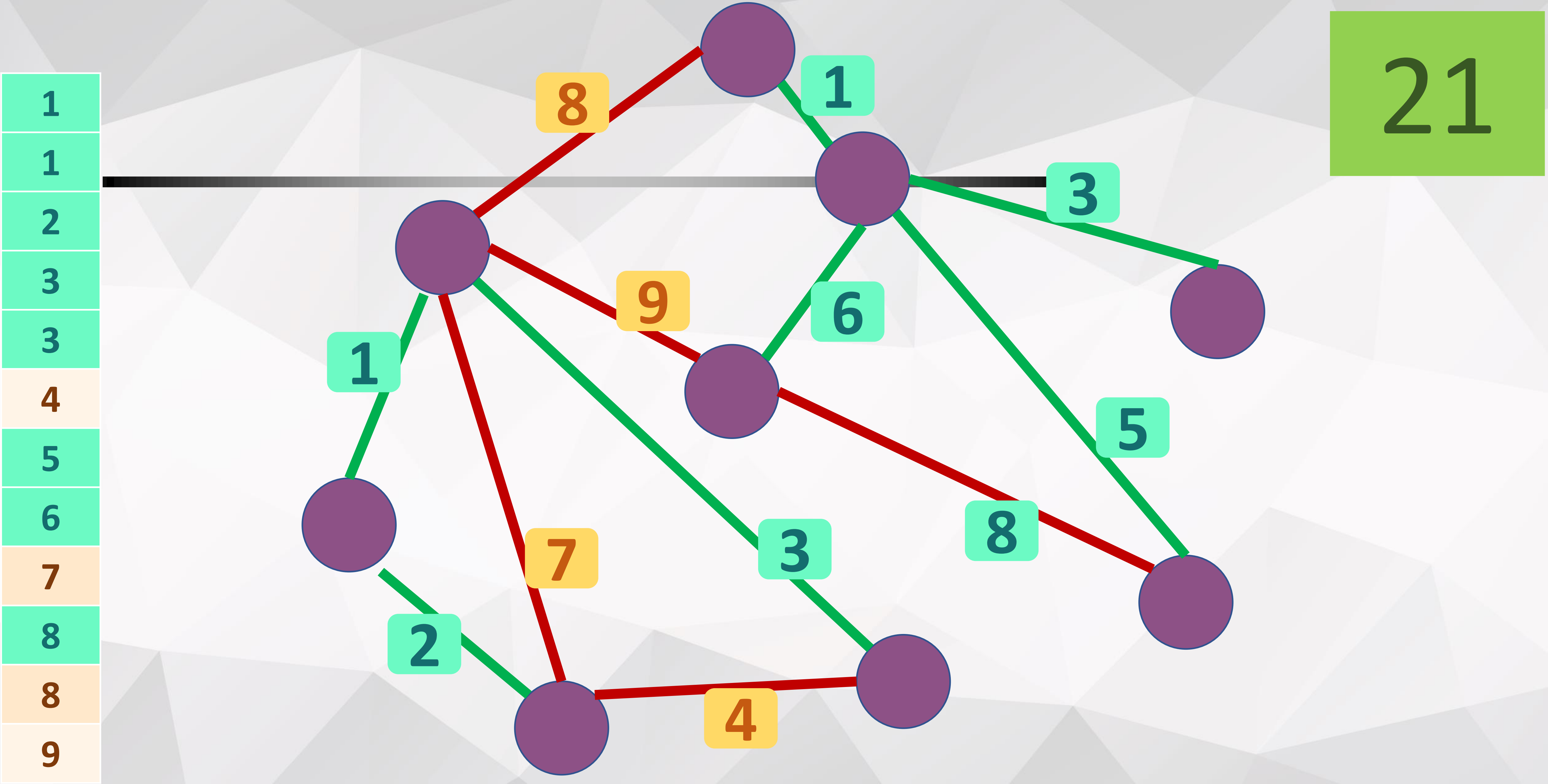
環

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



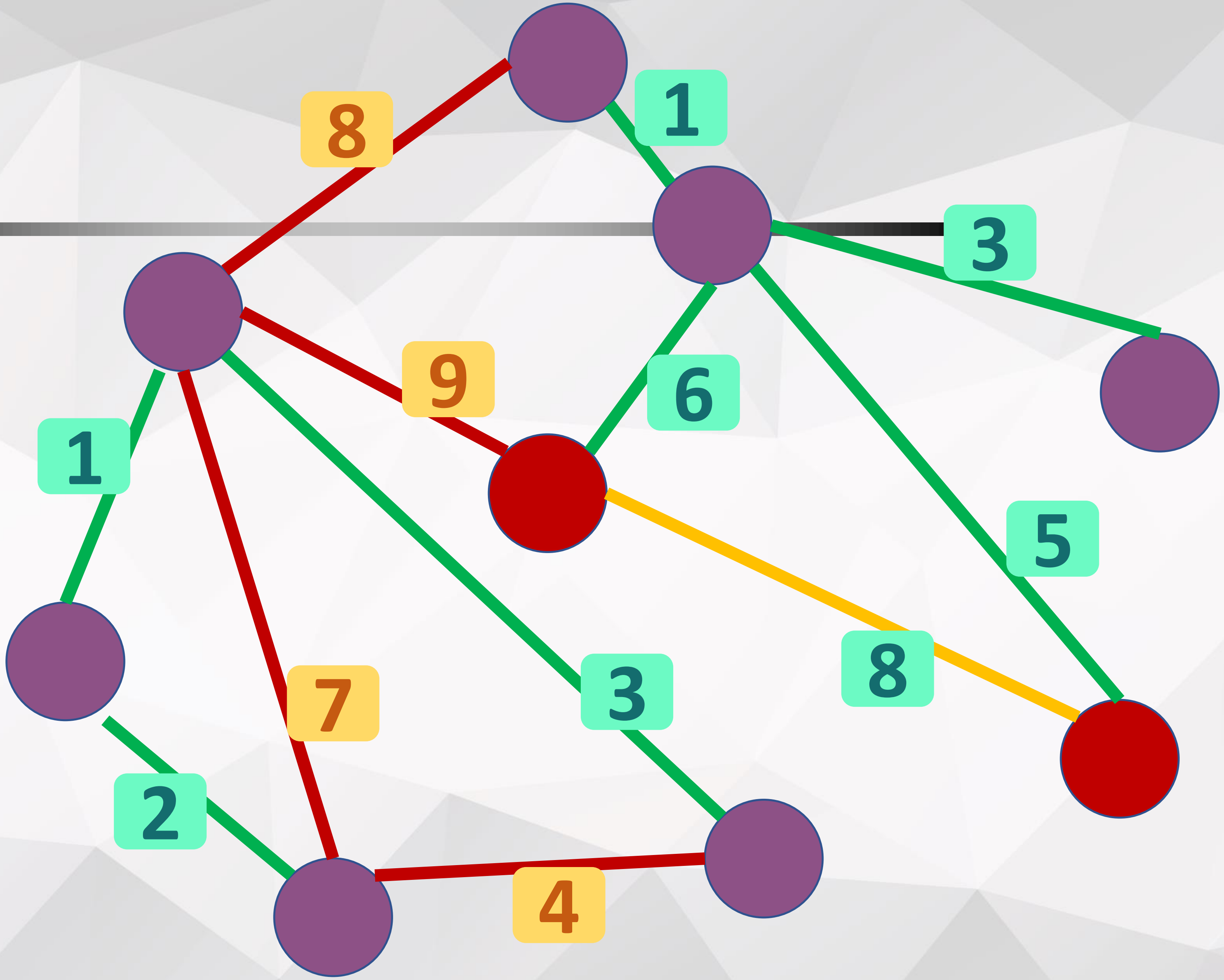
(⊙д⊙)


1
1
2
3
3
4
5
6
7
8
8
9
21
8
1
3
9
6
1
5
7
3
8
2
4


1
1
2
3
3
4
5
6
7
8
8
9
21
8
1
3
9
6
1
5
7
3
8
2
4

檢查

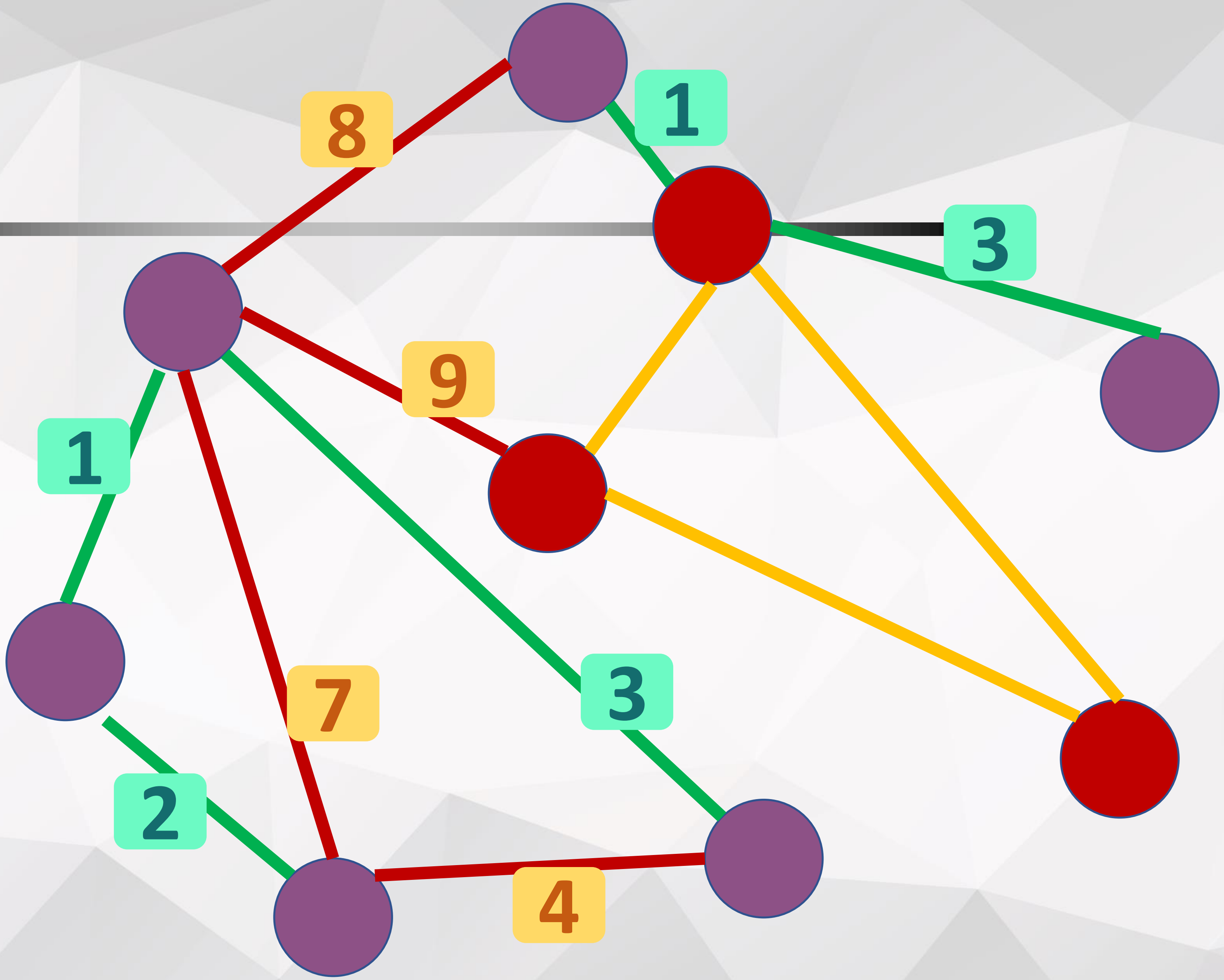
1
1
2
3
3
4
5
6
7
8
8
9



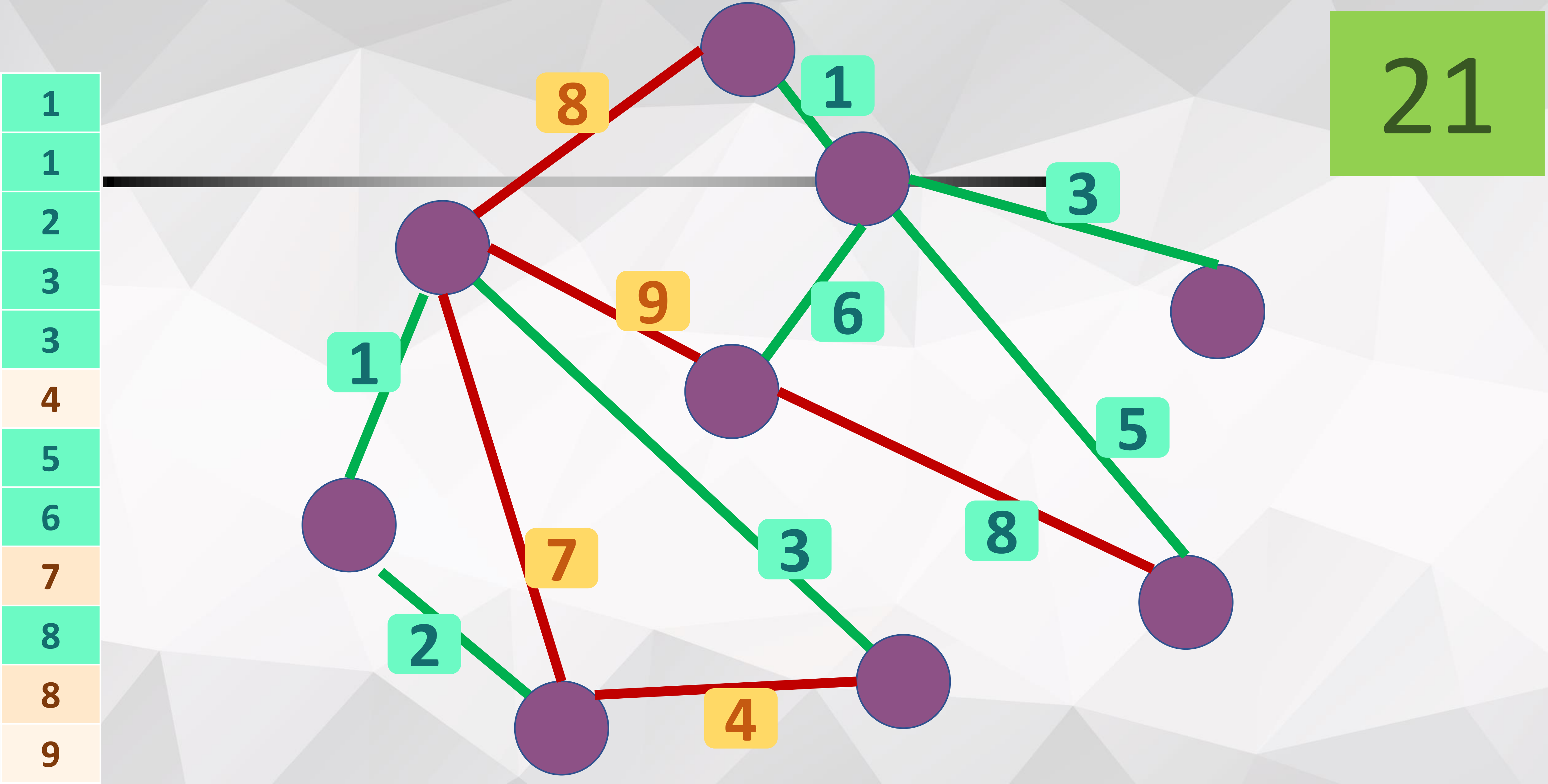
21

環

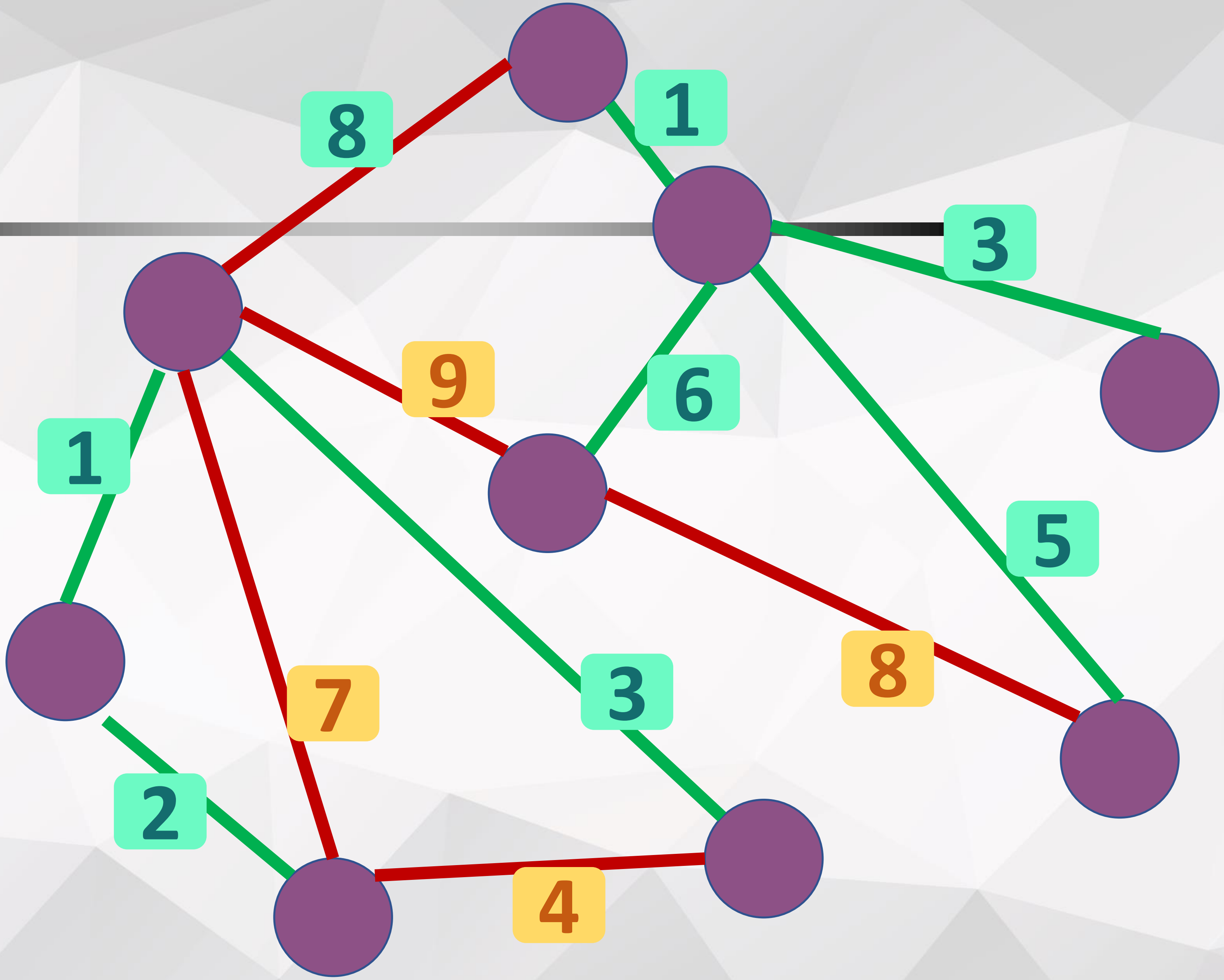
1
1
2
3
3
4
5
6
7
8
8
9



(⊙д⊙)


1
1
2
3
3
4
5
6
7
8
8
9
21
8
1
3
9
6
1
5
7
3
8
2
4

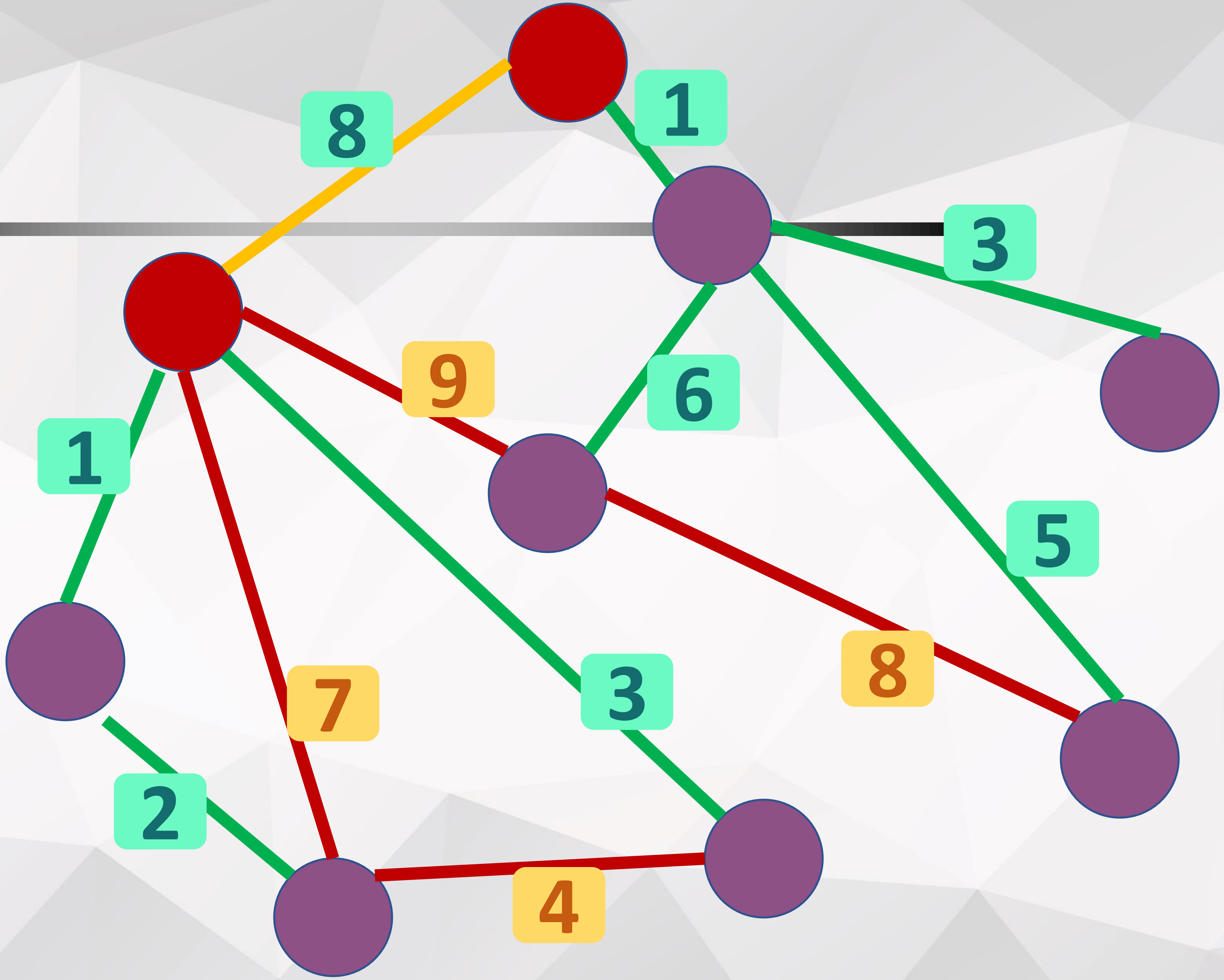
1
1
2
3
3
4
5
6
7
8
8
9



21

檢查

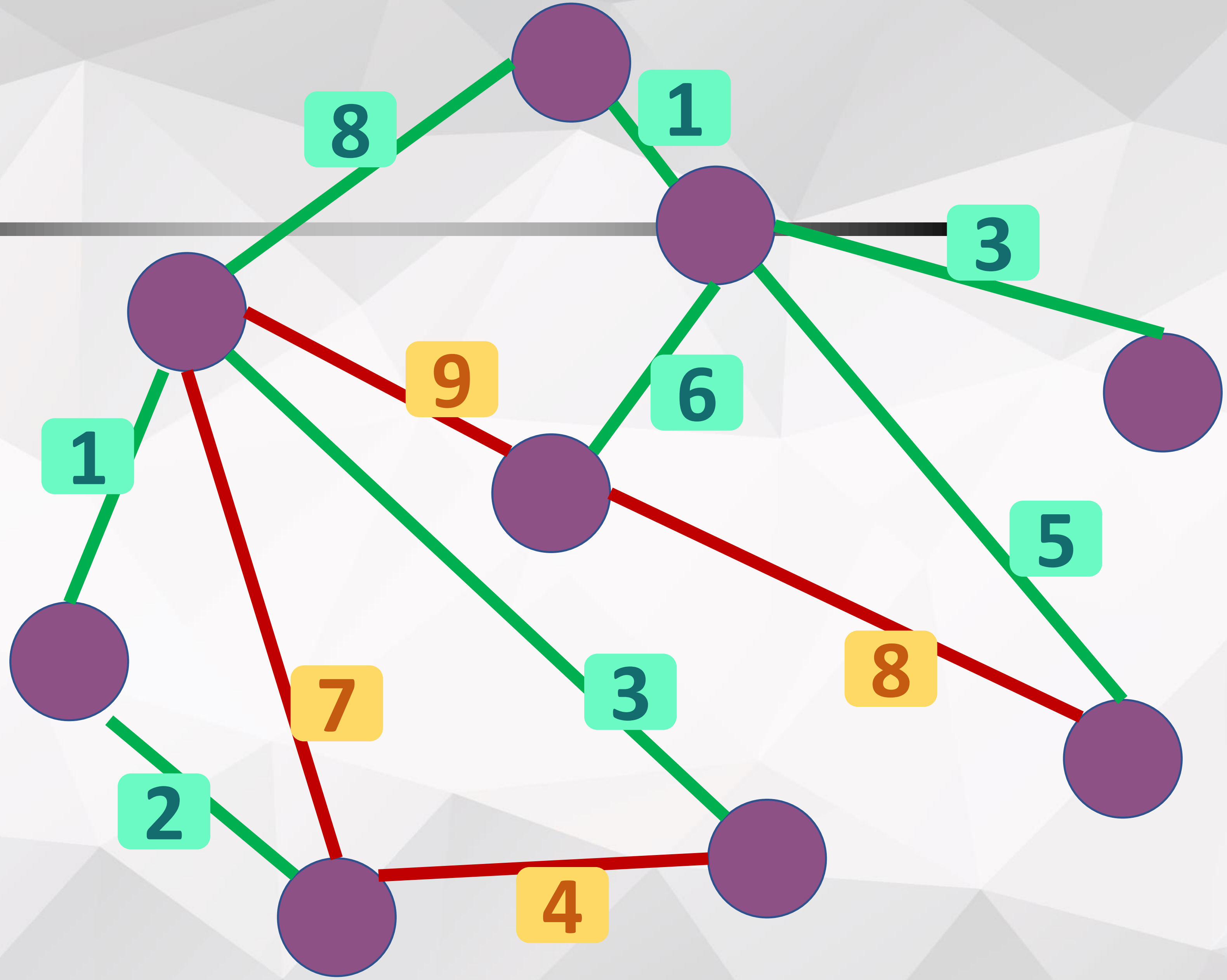
1
1
2
3
3
4
5
6
7
8
8
9

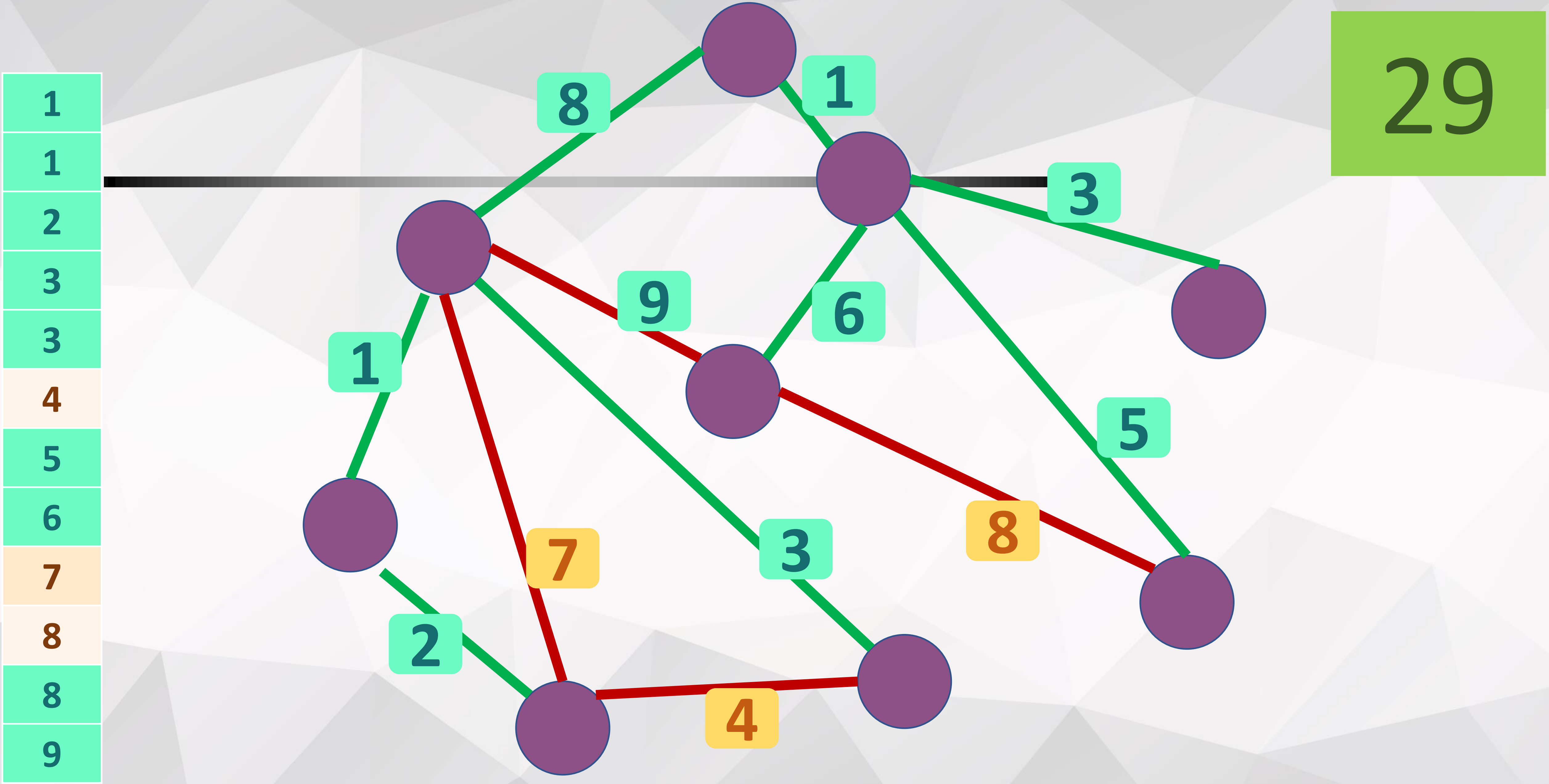


相連

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

29

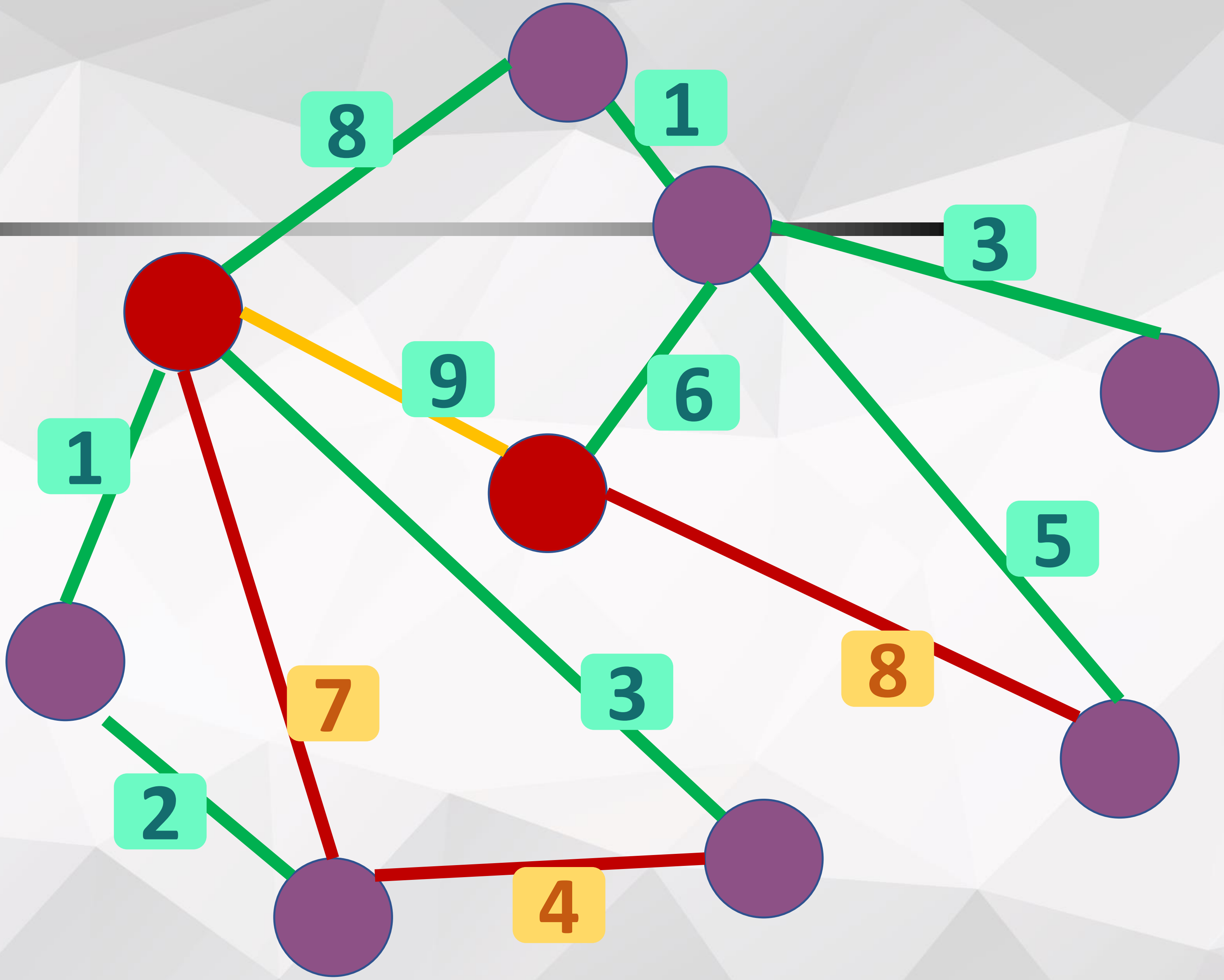



1
1
2
3
3
4
5
6
7
8
8
9
29
8
1
3
9
6
1
5
7
3
8
2
4

檢查

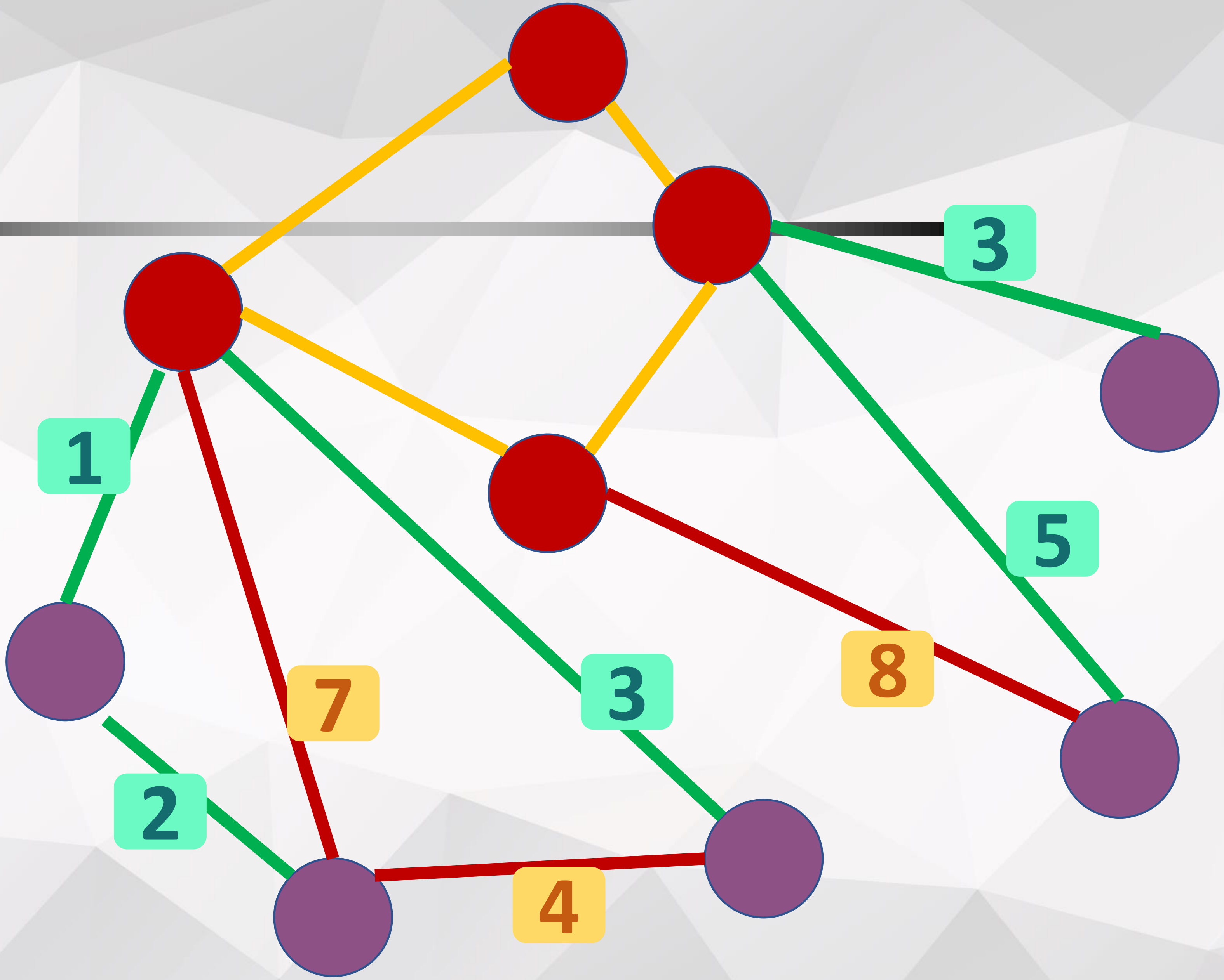
1
1
2
3
3
4
5
6
7
8
8
9

29

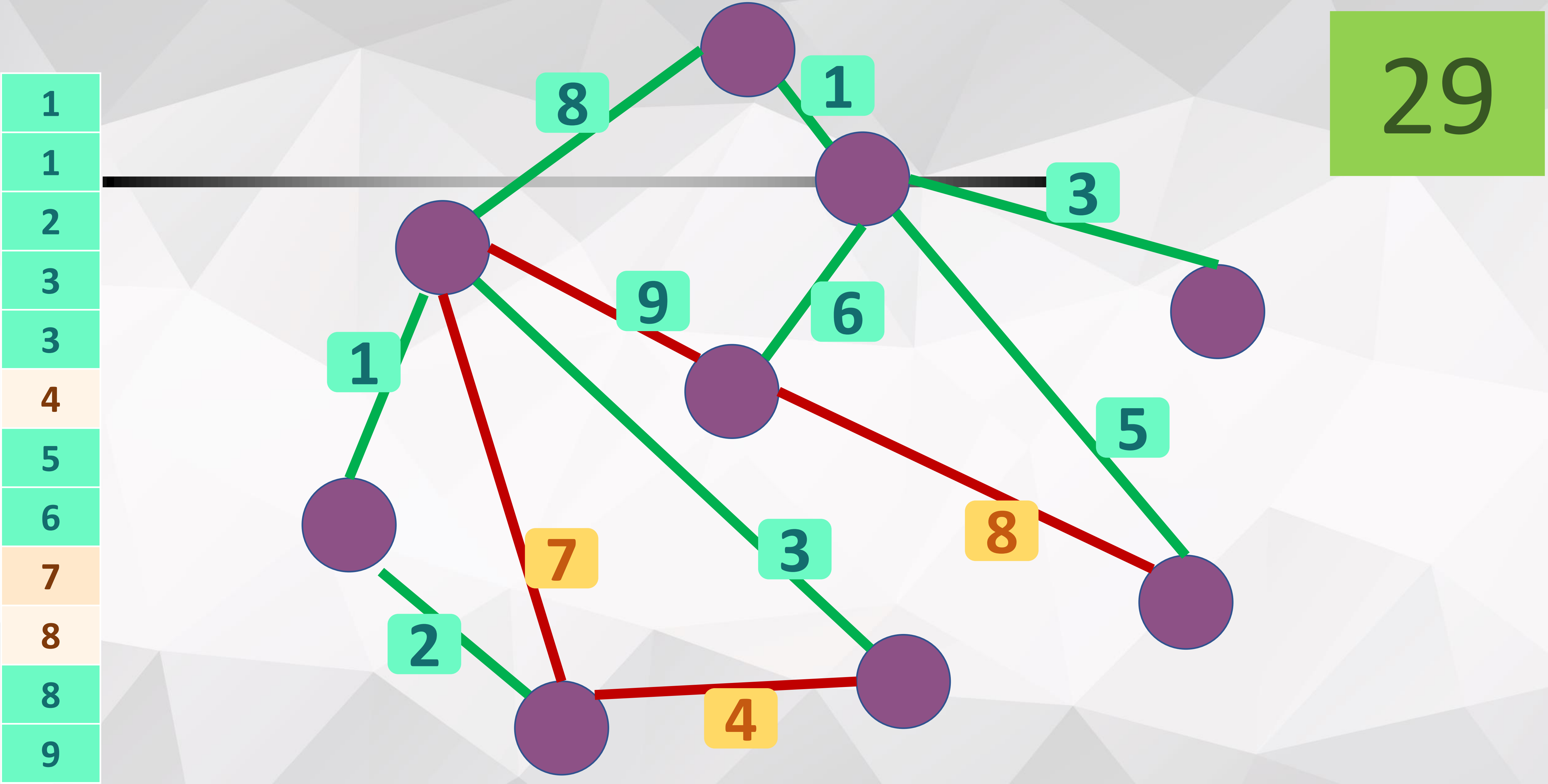


環

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



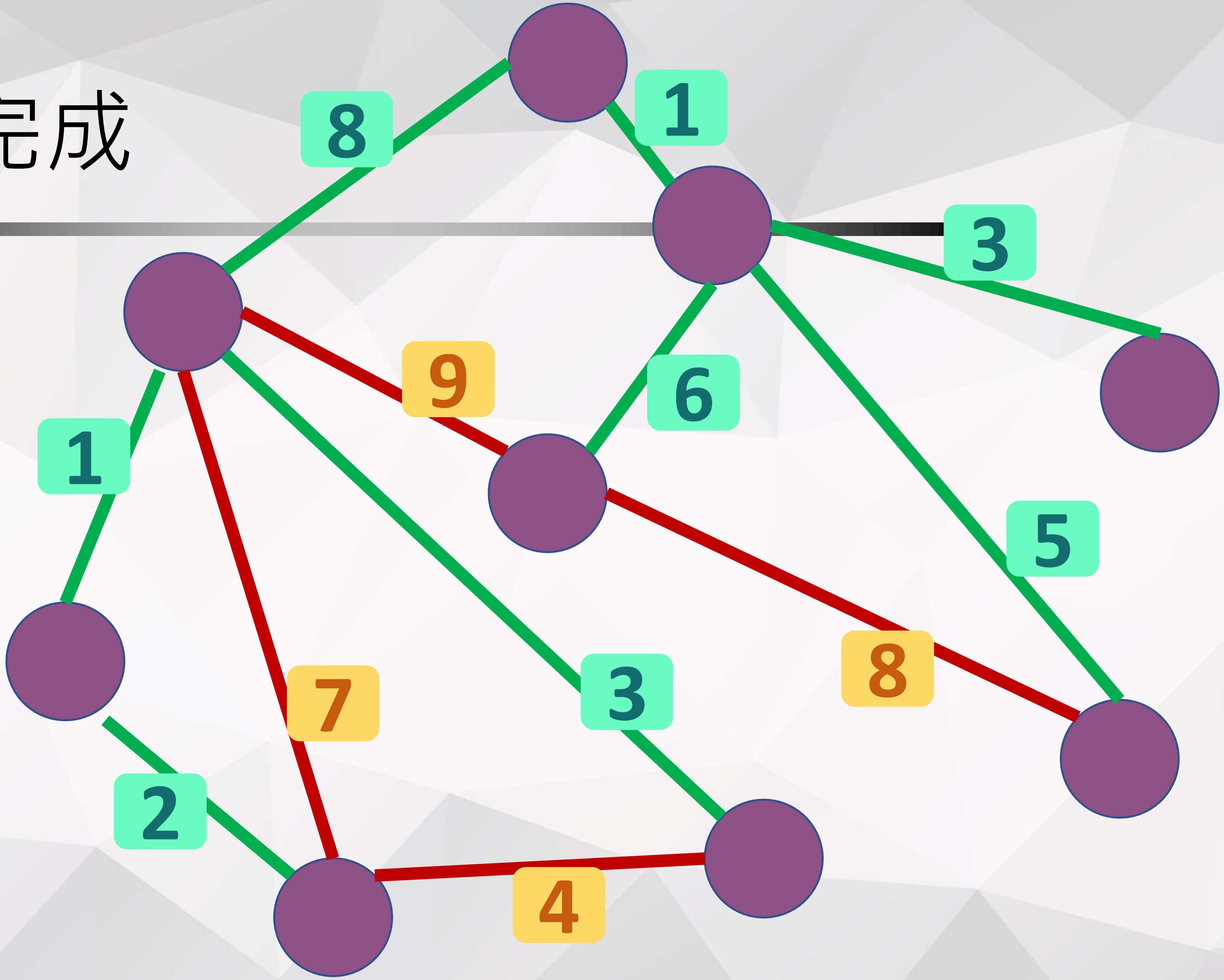
(⊙д⊙)


1
1
2
3
3
4
5
6
7
8
8
9
29
8
1
3
9
6
1
5
7
3
8
2
4

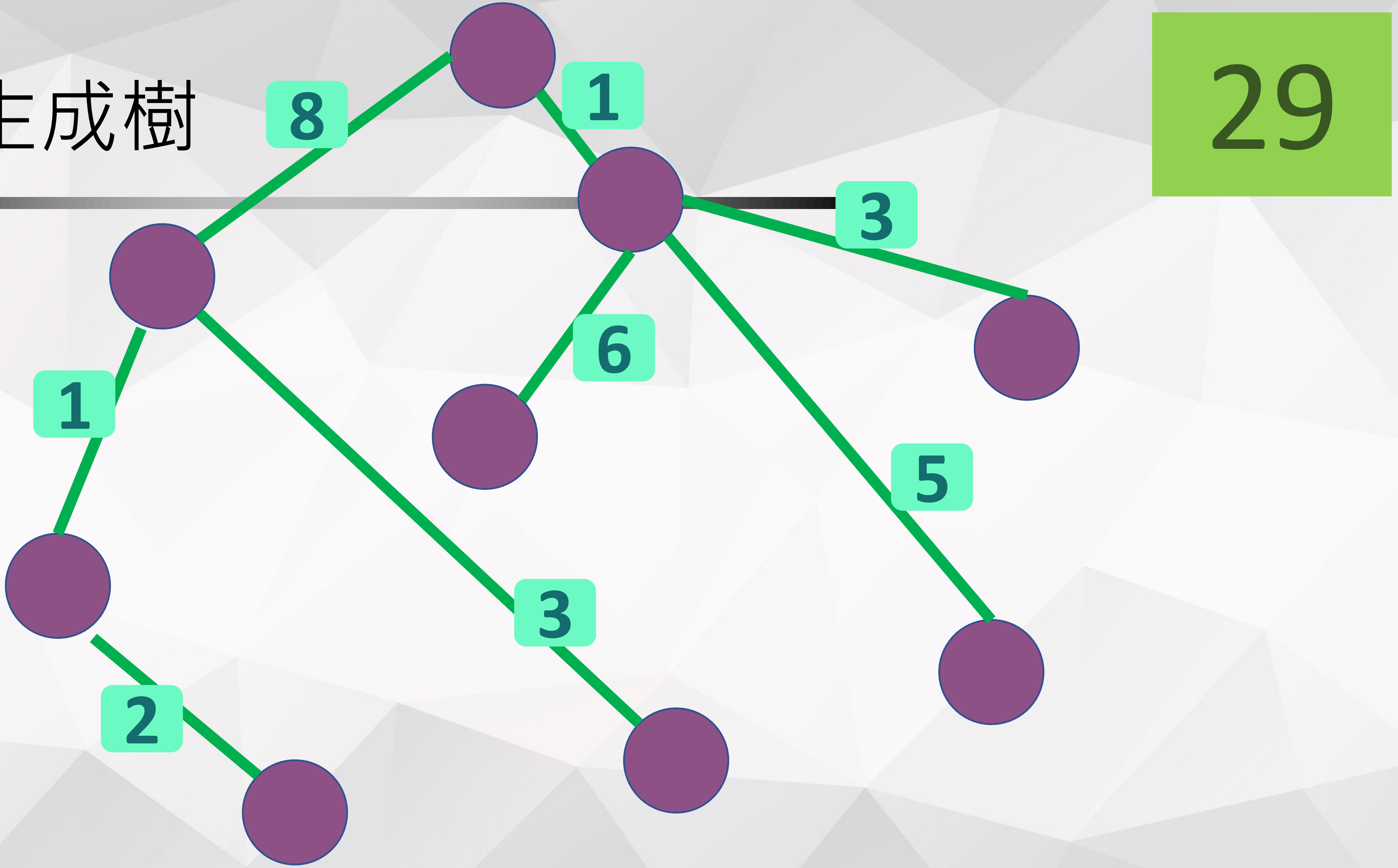
遍歷完成

29

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



最小生成樹



Kruskal 實作

若使用 DFS 判斷兩點是否屬於同個連通塊

則最終複雜度為 $O(|E|^2)$

- 枚舉每個邊 $O(|E|)$
- DFS 將連通塊上的邊都拜訪 $O(|E|)$

Kruskal 實作

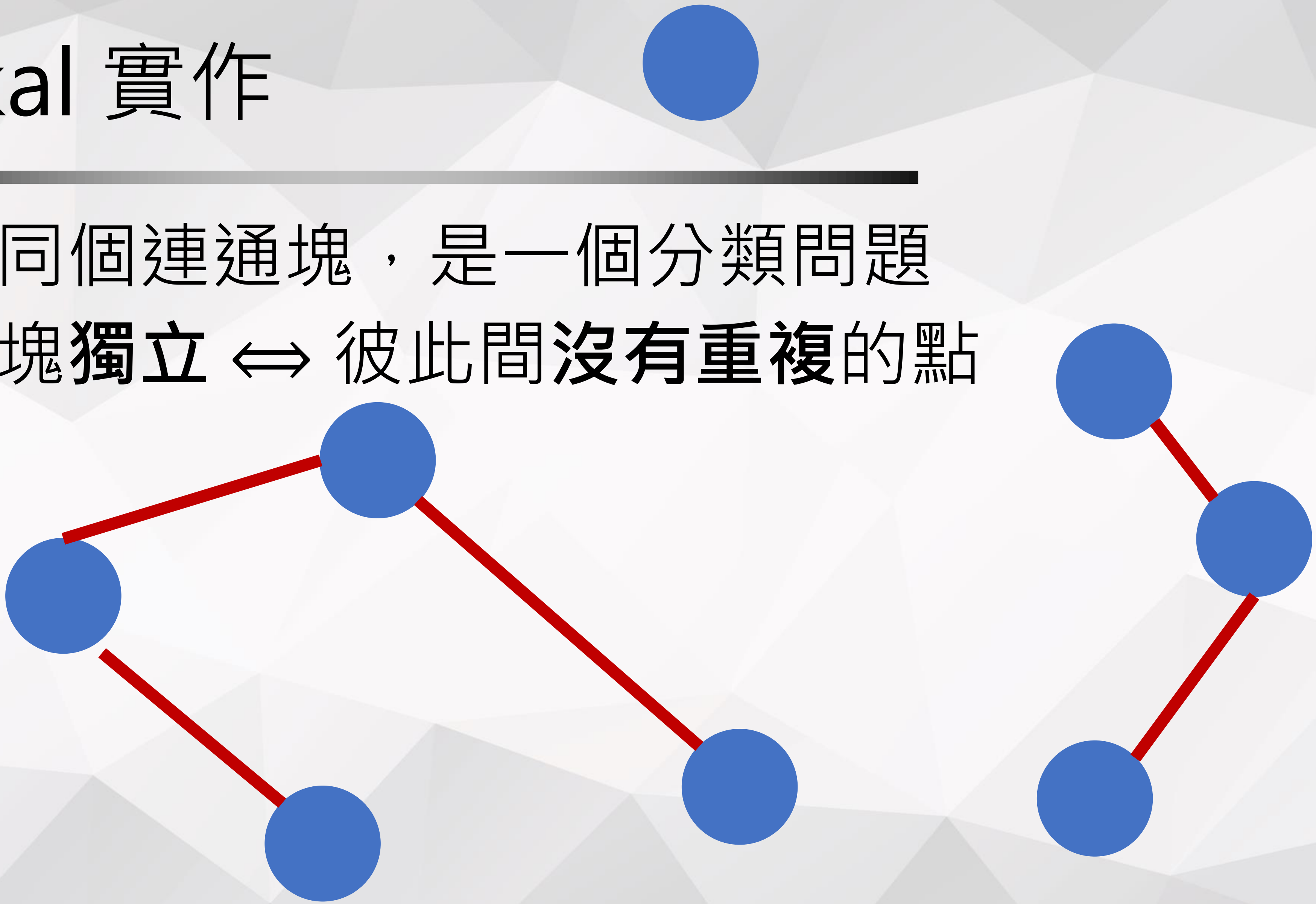
是否為同個連通塊，是一個分類問題

兩連通塊獨立 ⇔ 彼此間沒有重複的點

Kruskal 實作

是否為同個連通塊，是一個分類問題

兩連通塊**獨立** ⇔ 彼此間**沒有重複**的點



Kruskal 實作

是否為同個連通塊，是一個分類問題
兩連通塊獨立 ⟺ 彼此間沒有重複的點

也就是說，連通塊們是個 Disjoint Sets
可以用 Union-Find Forest 改善複雜度
第四週教材中有 Union-Find Forest 的介紹

Kruskal 實作

```
bool cmp(const edge &A, const edge &B)
  { return A.w < B.w; }
```

Kruskal 實作

```
vector<edge> E; // 邊集合
   :
   .
sort(E.begin(), E.end(), cmp);

for (edge e: E) {
   int a = Find(e.u), b = Find(e.v);
   if (a != b) {
      Union(e.u, e.v);
      cost += E.w;
      MST.emplace_back(u, v, w);
   }
}
```

Kruskal 實作

用 Union-Find Forest 改善複雜度

複雜度為 $O(|E|\log_2|E| + |E|\cdot \alpha)$

- α 為 Union-Find Forest 的時間成本

Questions?

練習

- UVa OJ 10369 Arctic Network
- AIZU 1280 Slim Span

最小生成樹

- Kruskal 演算法
- Prim 演算法

Prim 演算法

很類似的

兩個重要前提

- 樹是**無環**的**連通**圖

- 若圖只有點無任何邊，那每點都是彼此獨立**連通塊**

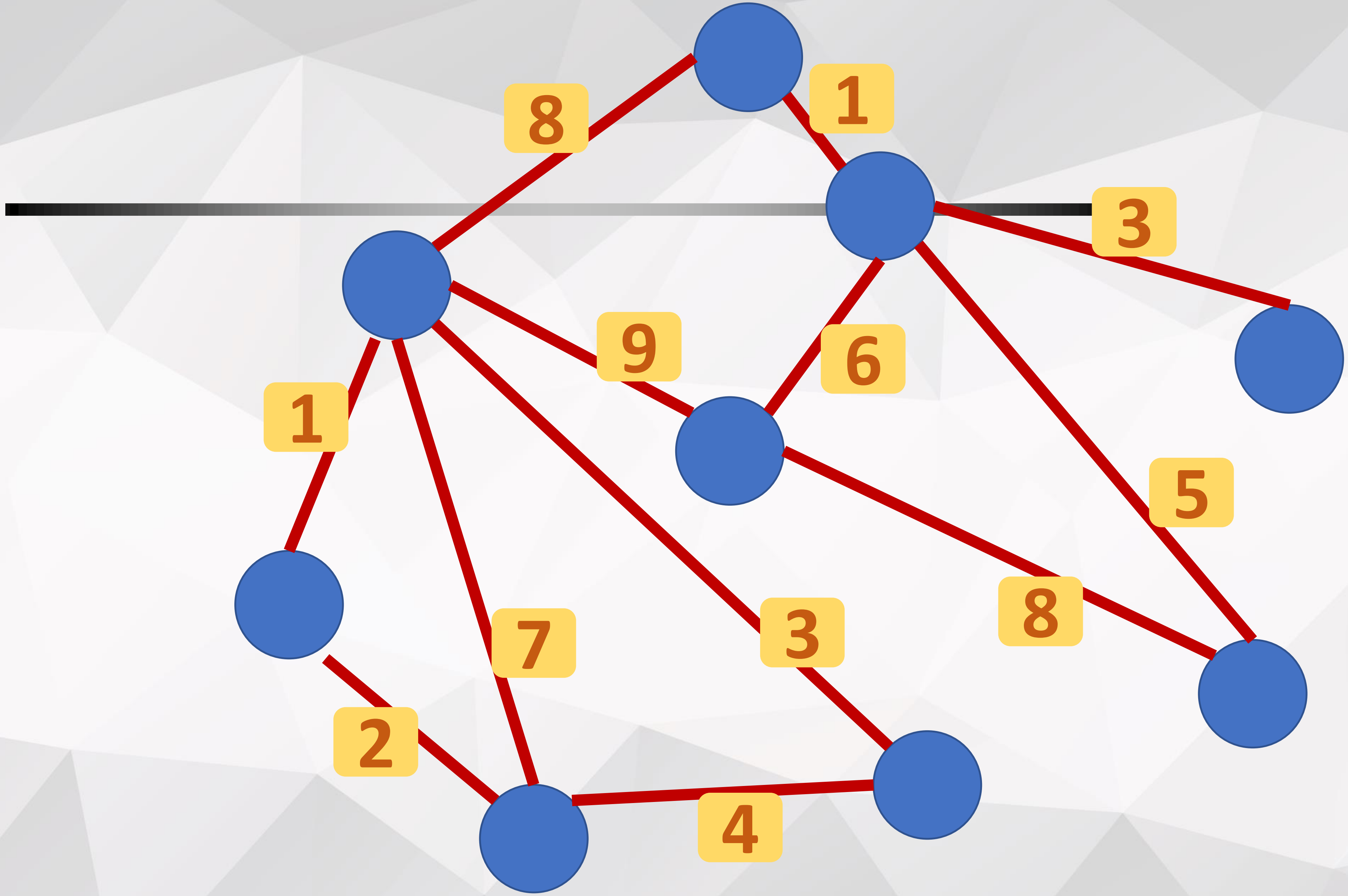
Prim 演算法

Prim 維護一個**未完成的生成樹**

Prim 演算法

Prim 維護一個未完成的生成樹

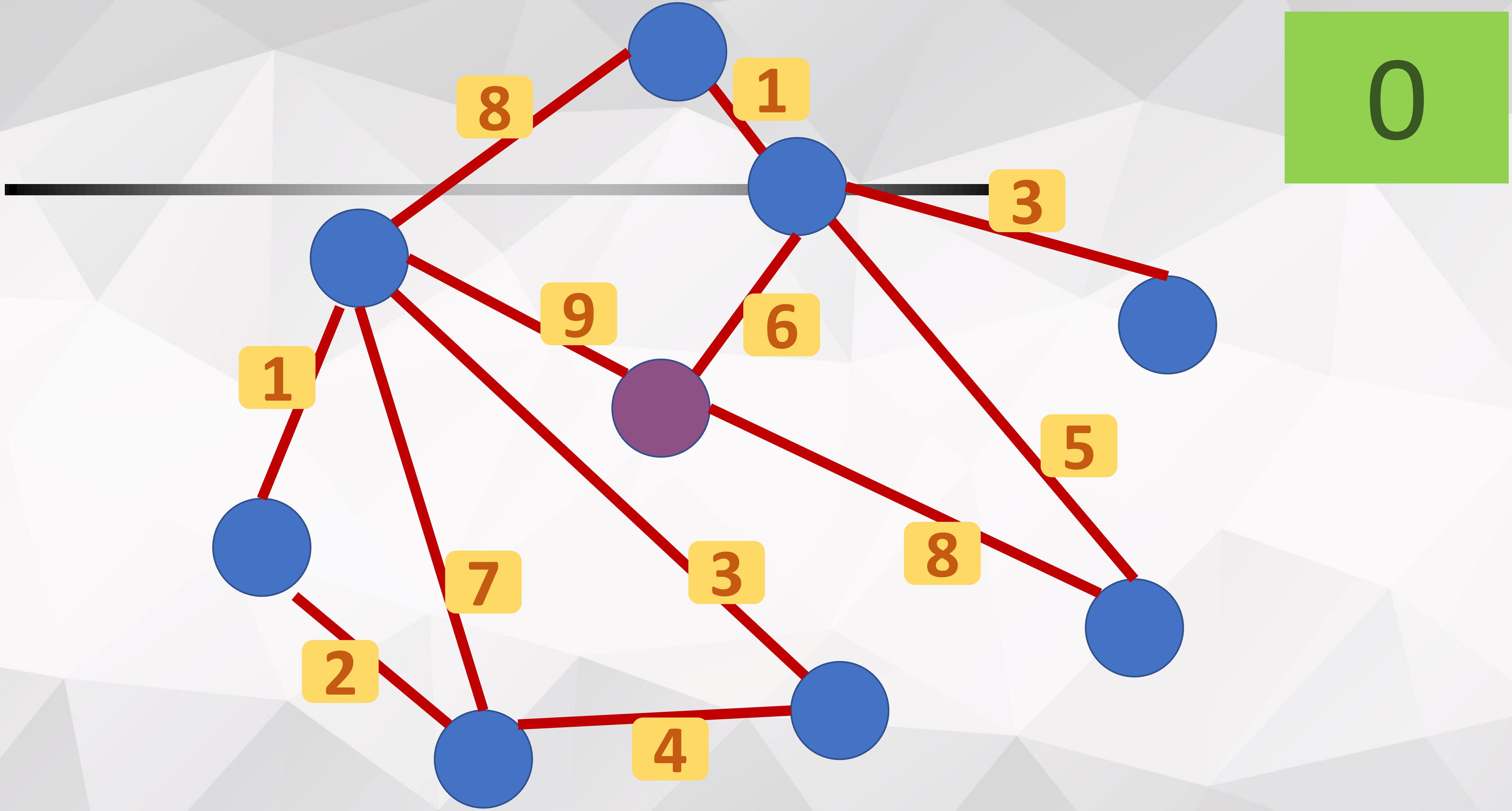
每次將樹**周遭**有**最小權重**的邊接到樹上，
使樹最終成長至最小生成樹



0

Prim 演算法

先隨便的挑任意點



0
8
1
3
9
6
1
5
8
7
3
2
4

0

Prim 演算法

先隨便的挑任意點

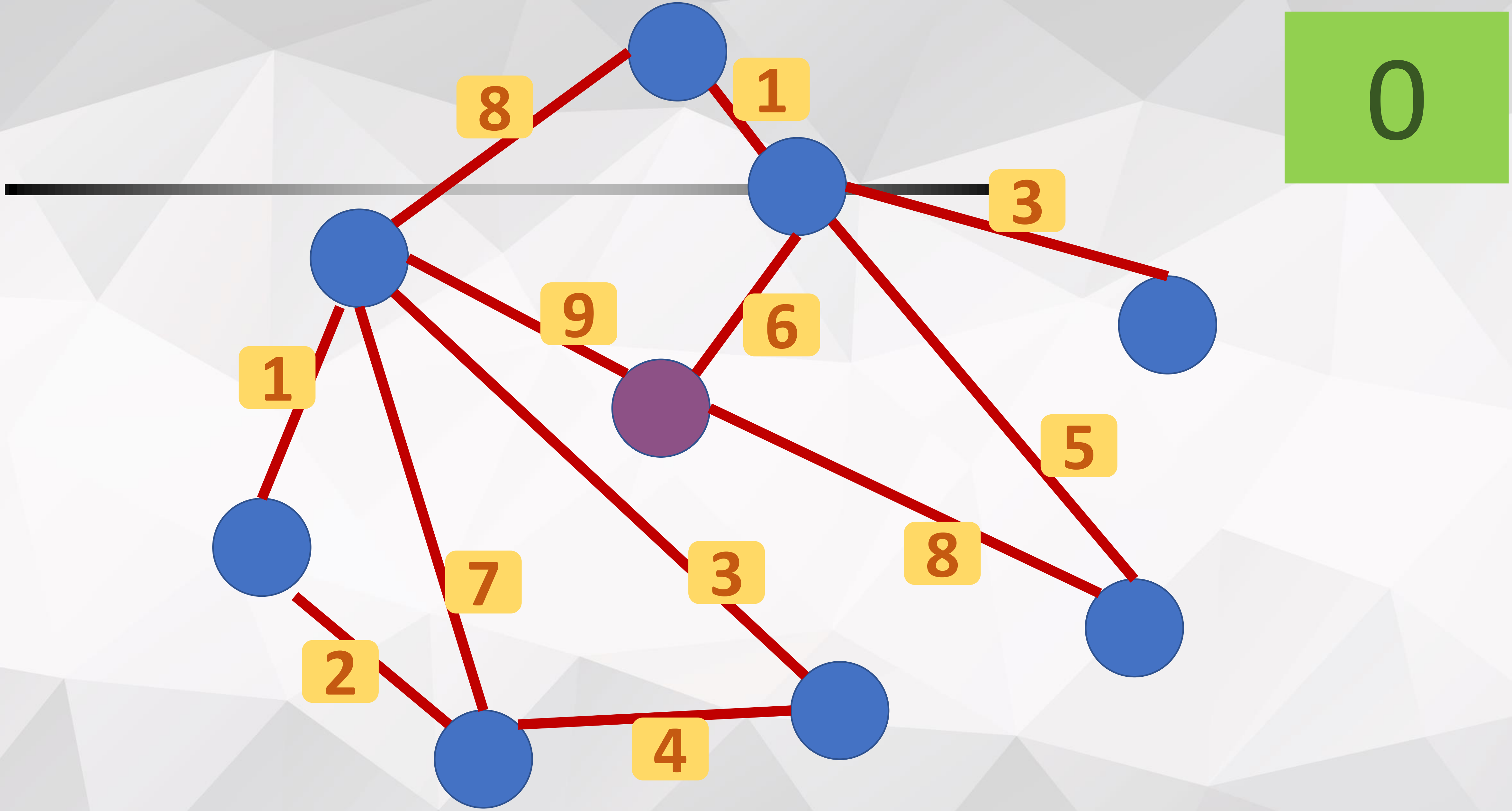
使它為初始的**未完成的生成樹**

Prim 演算法

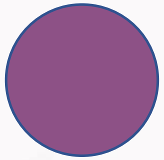
先隨便的挑任意點

使它為初始的**未完成的生成樹**，稱它為 MST

明顯的，它是個無環連通圖


0
8
1
3
9
6
1
5
7
3
8
2
4

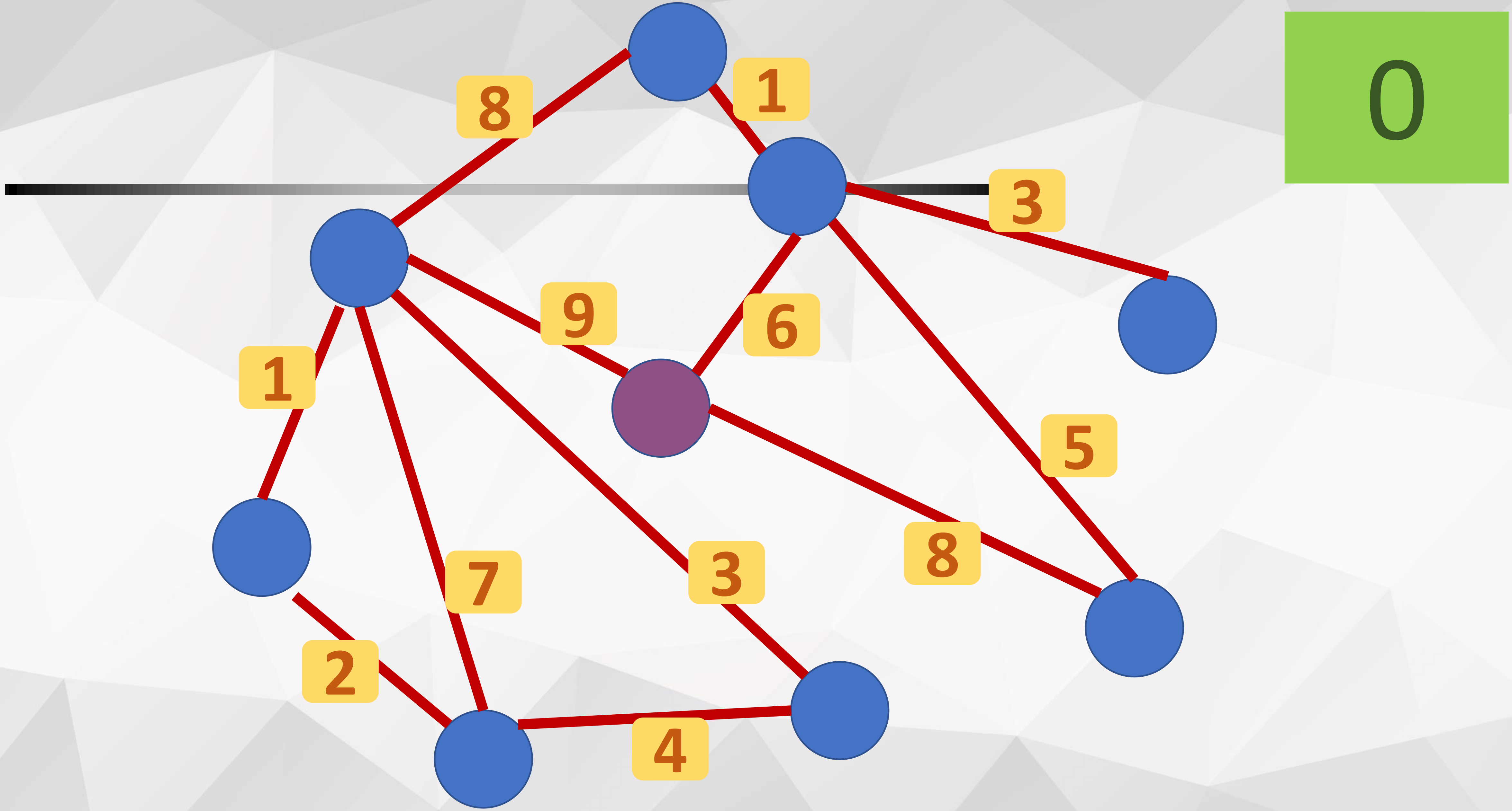
MST



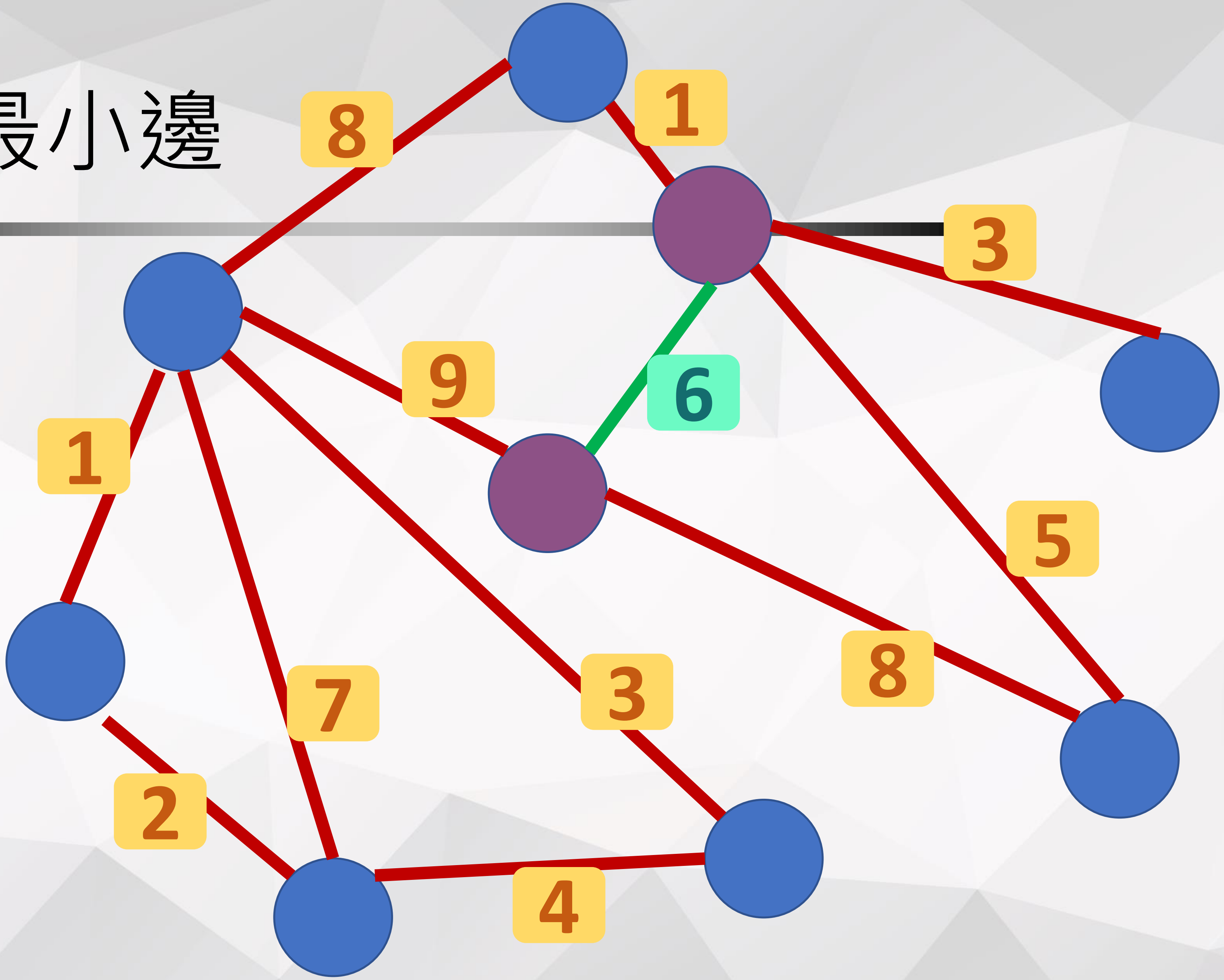
Prim 演算法

直覺的，每次將 MST 周遭權重最小的邊接上去

那麼最終產生的生成樹為最小生成樹

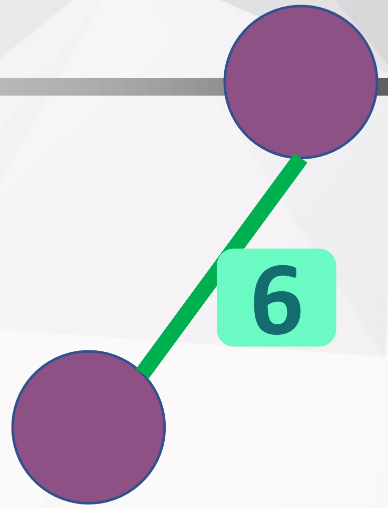

0
8
1
3
9
6
1
5
8
7
3
2
4

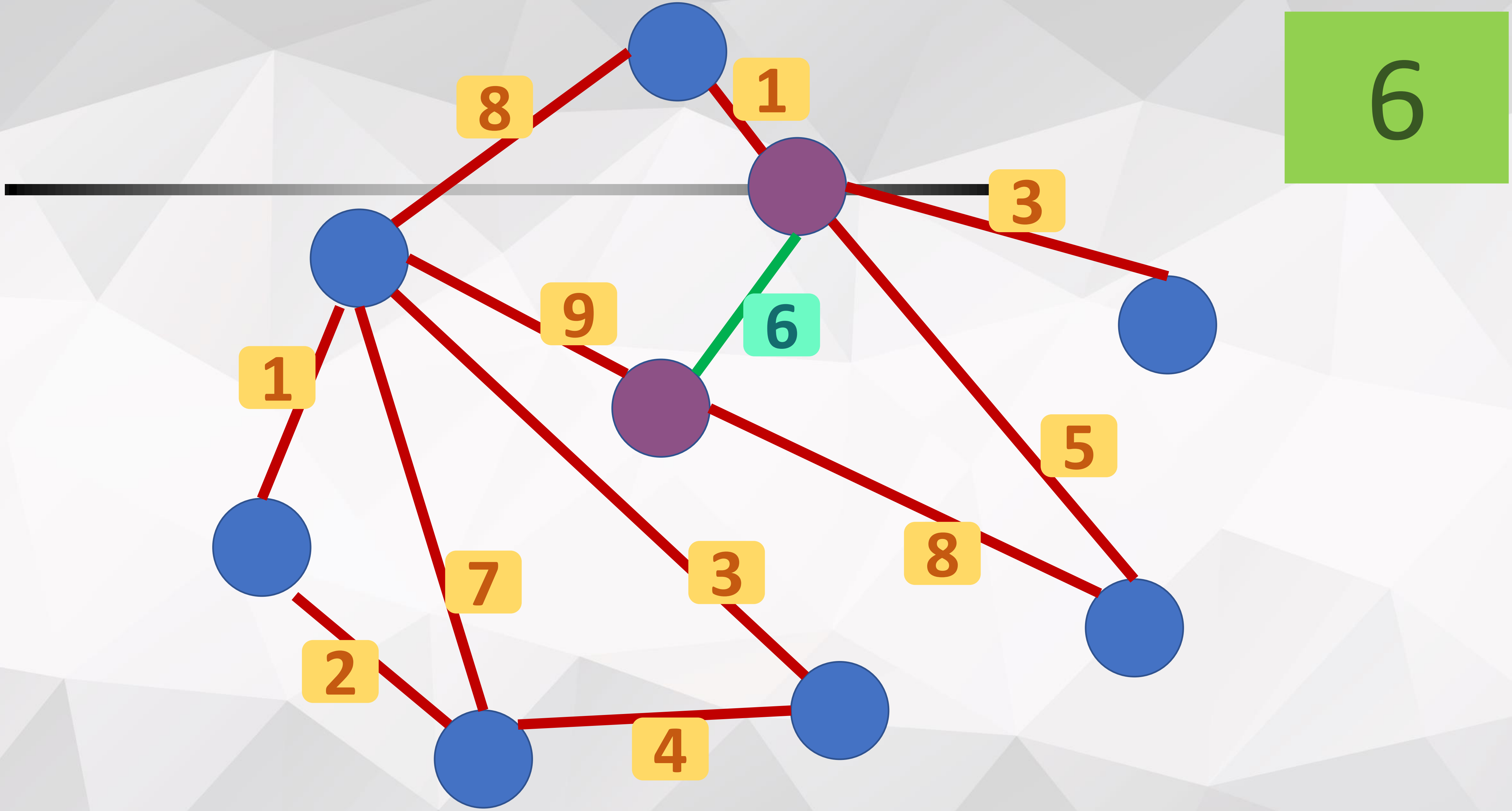
周圍最小邊



6

MST

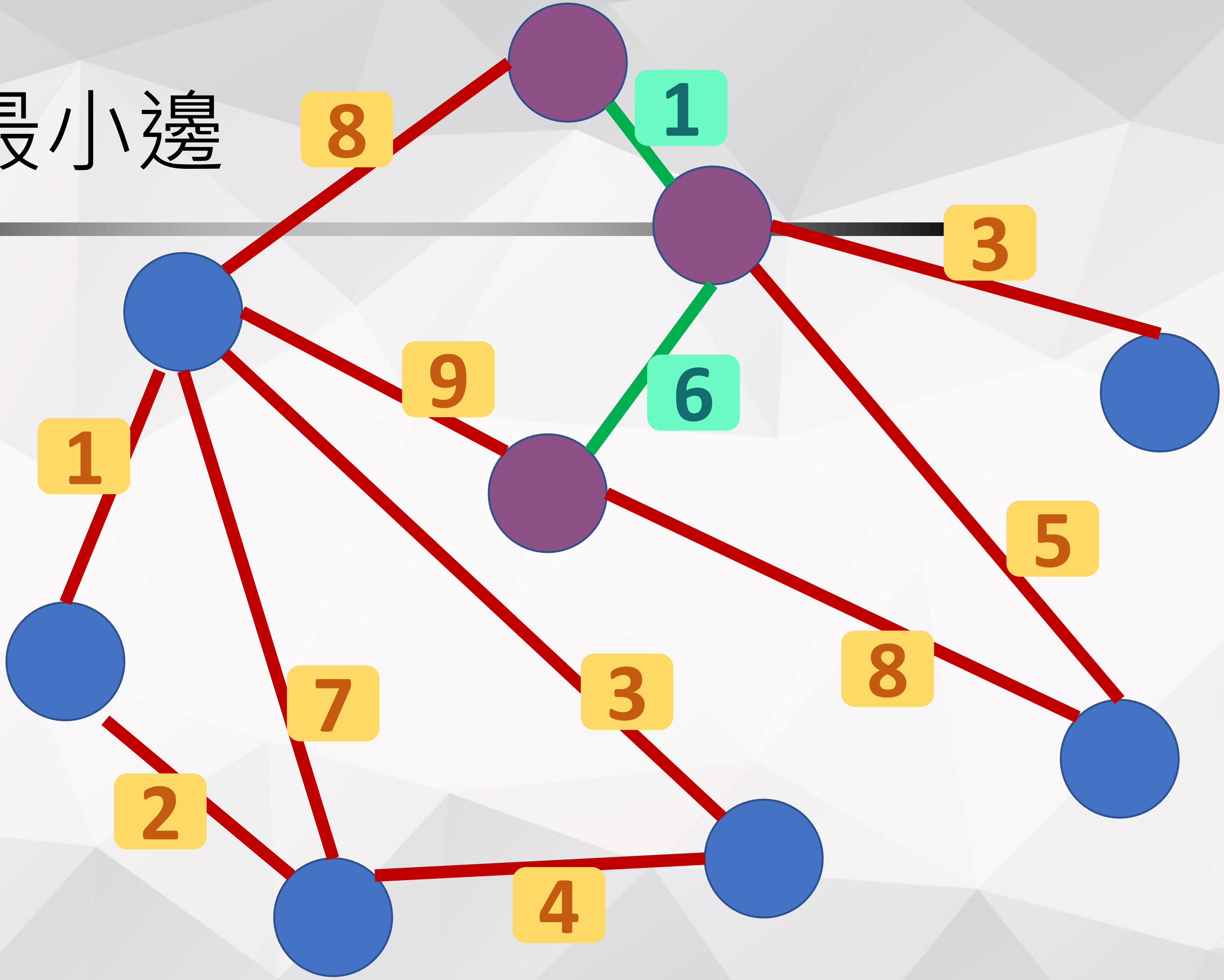




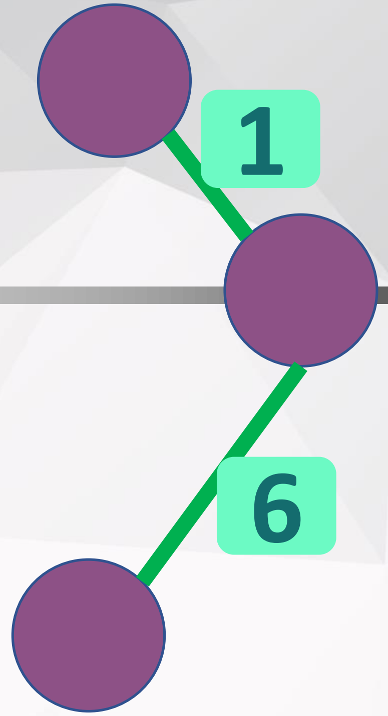
Competitive Programming

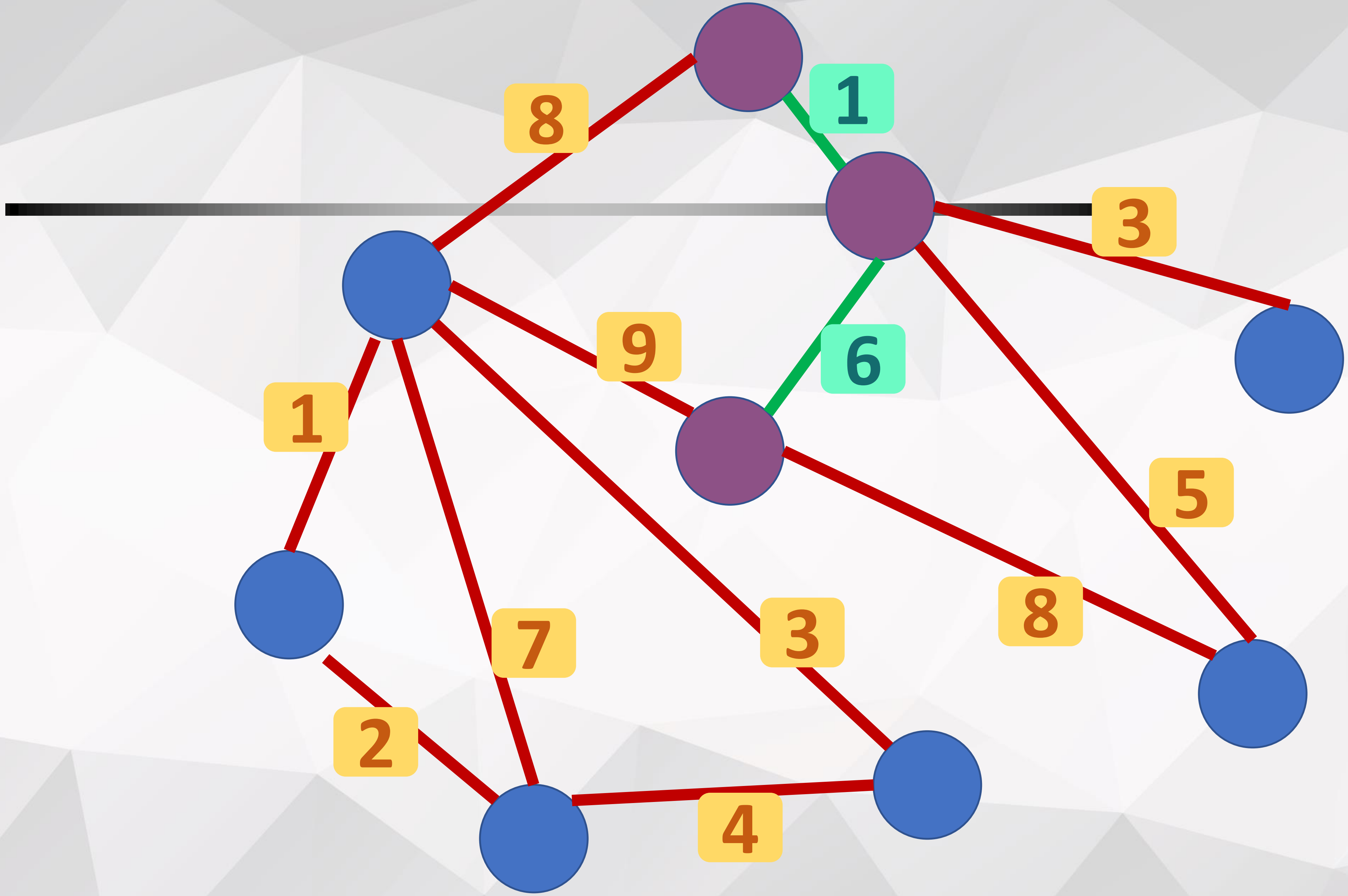
Made by 培訓團隊群 CCNS

周圍最小邊



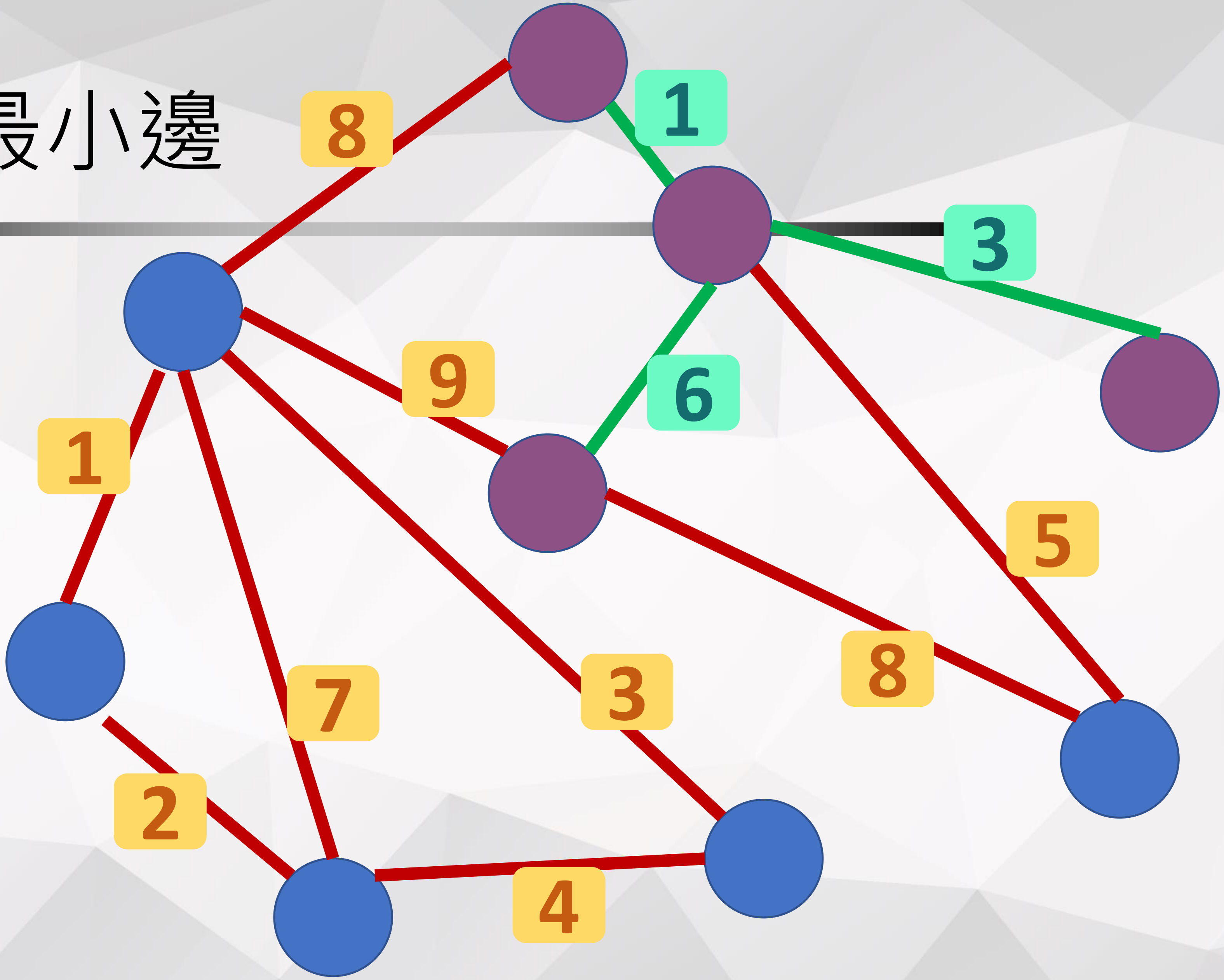
MST





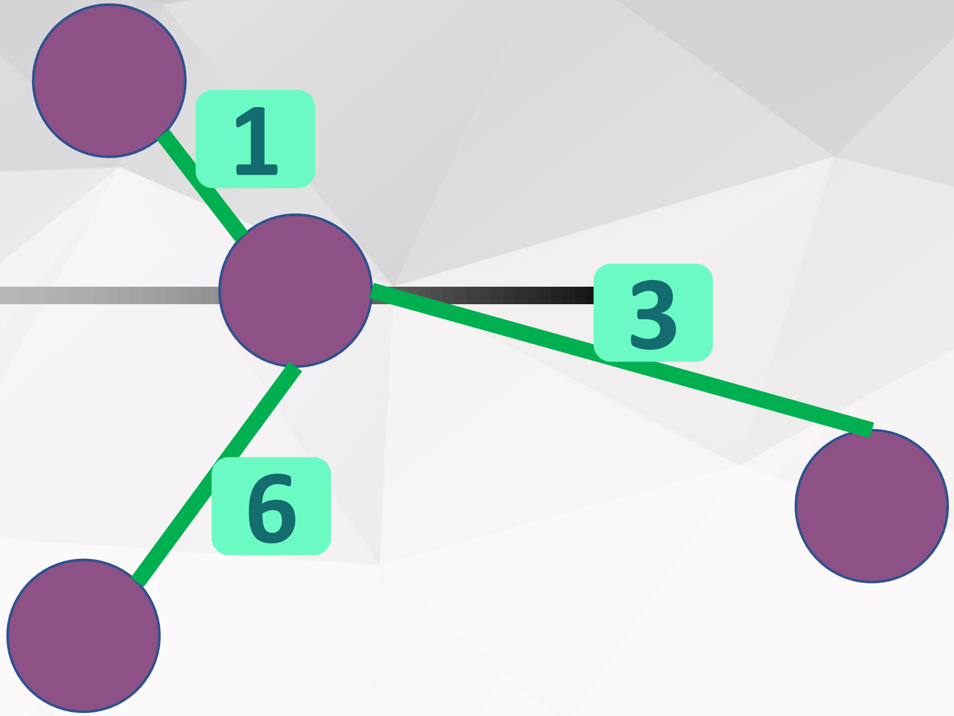
7

周圍最小邊



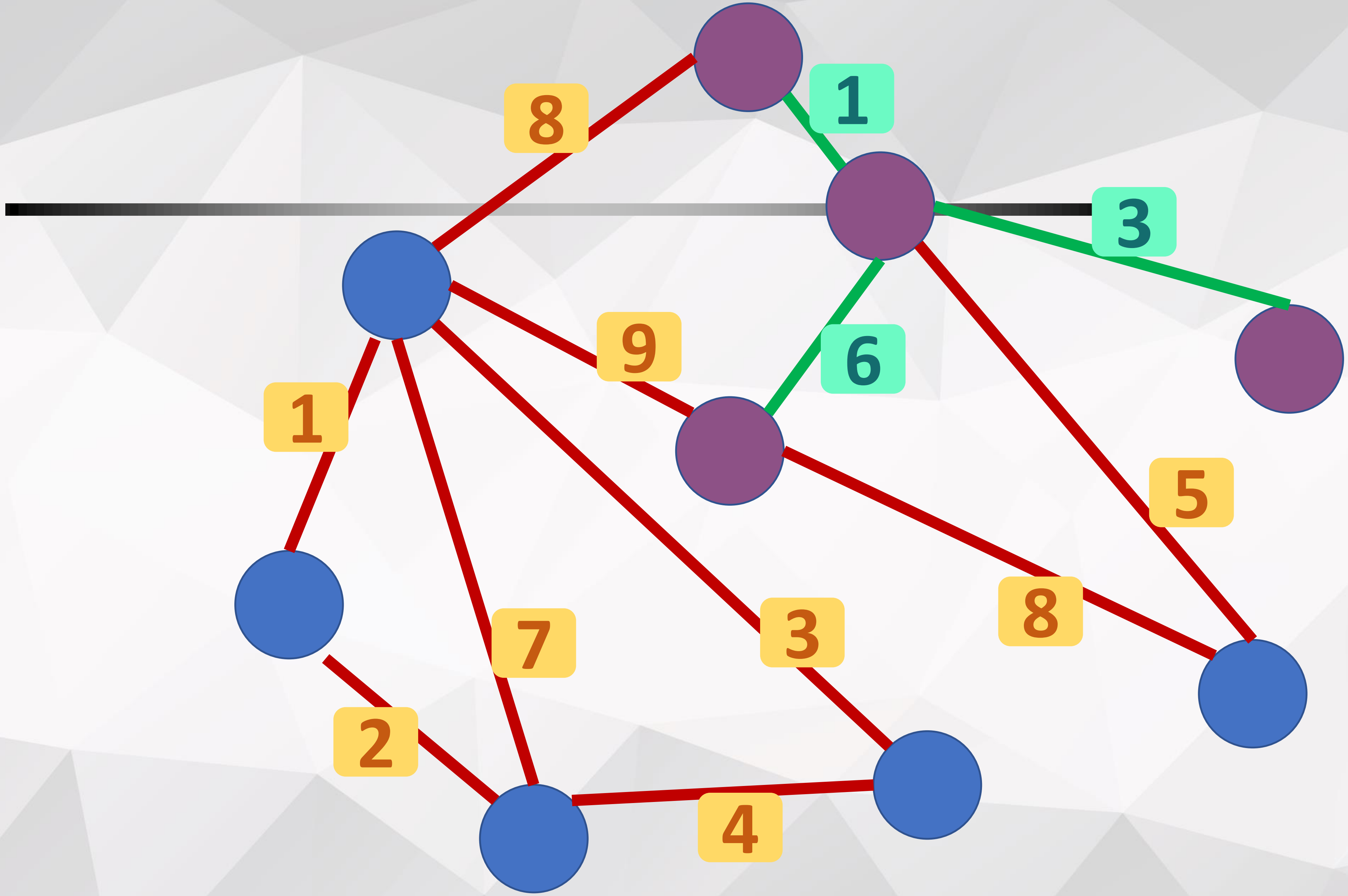
10

MST



10

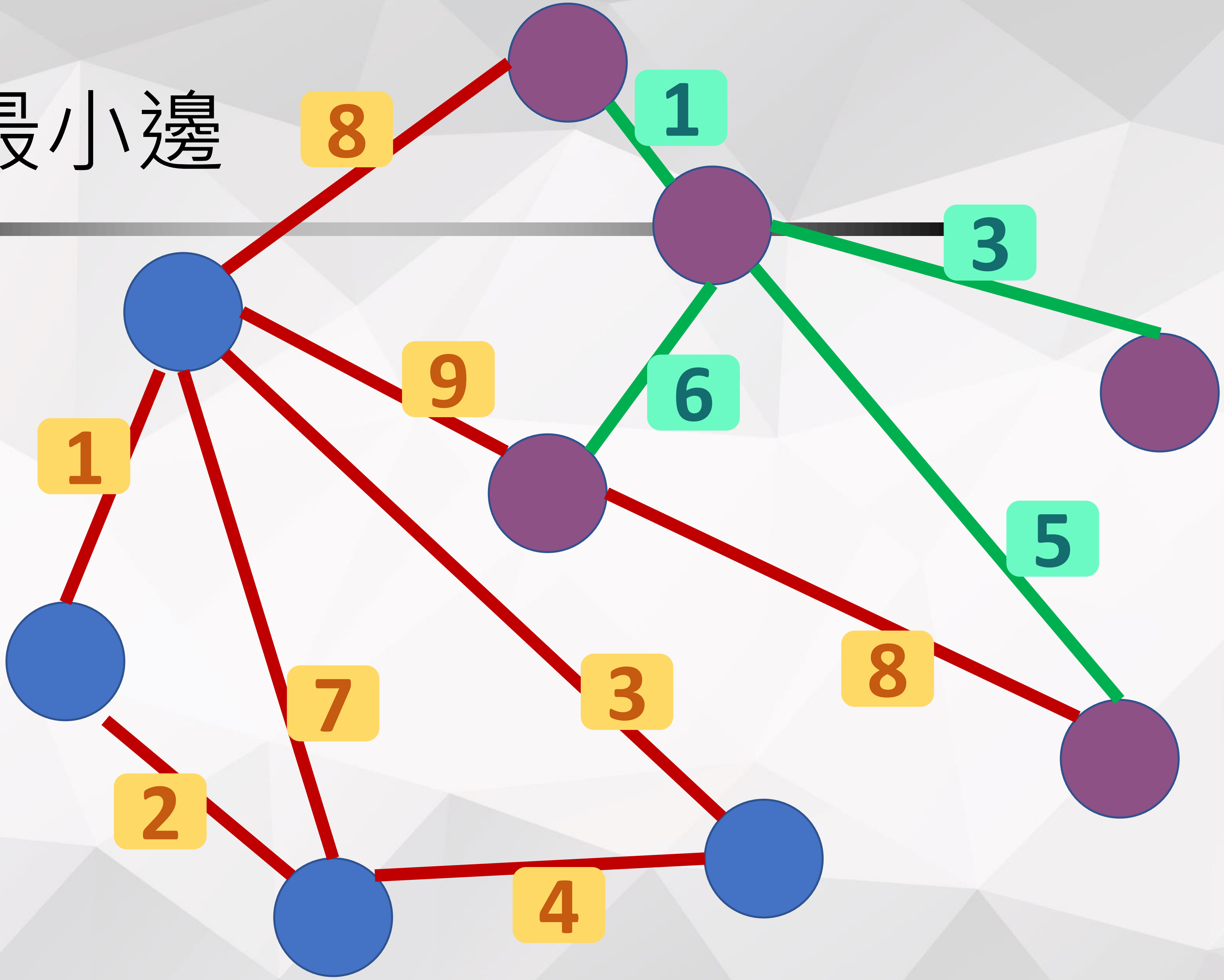
Competitive Programming



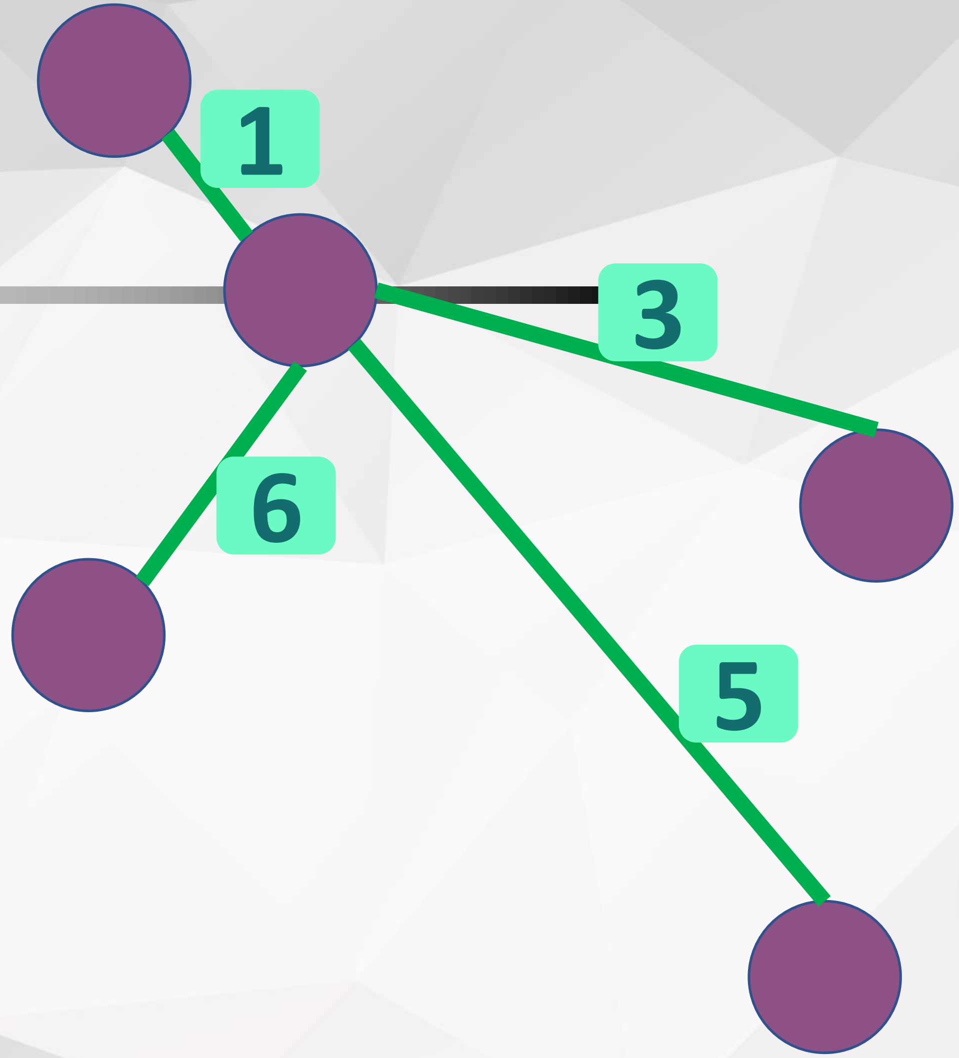
10

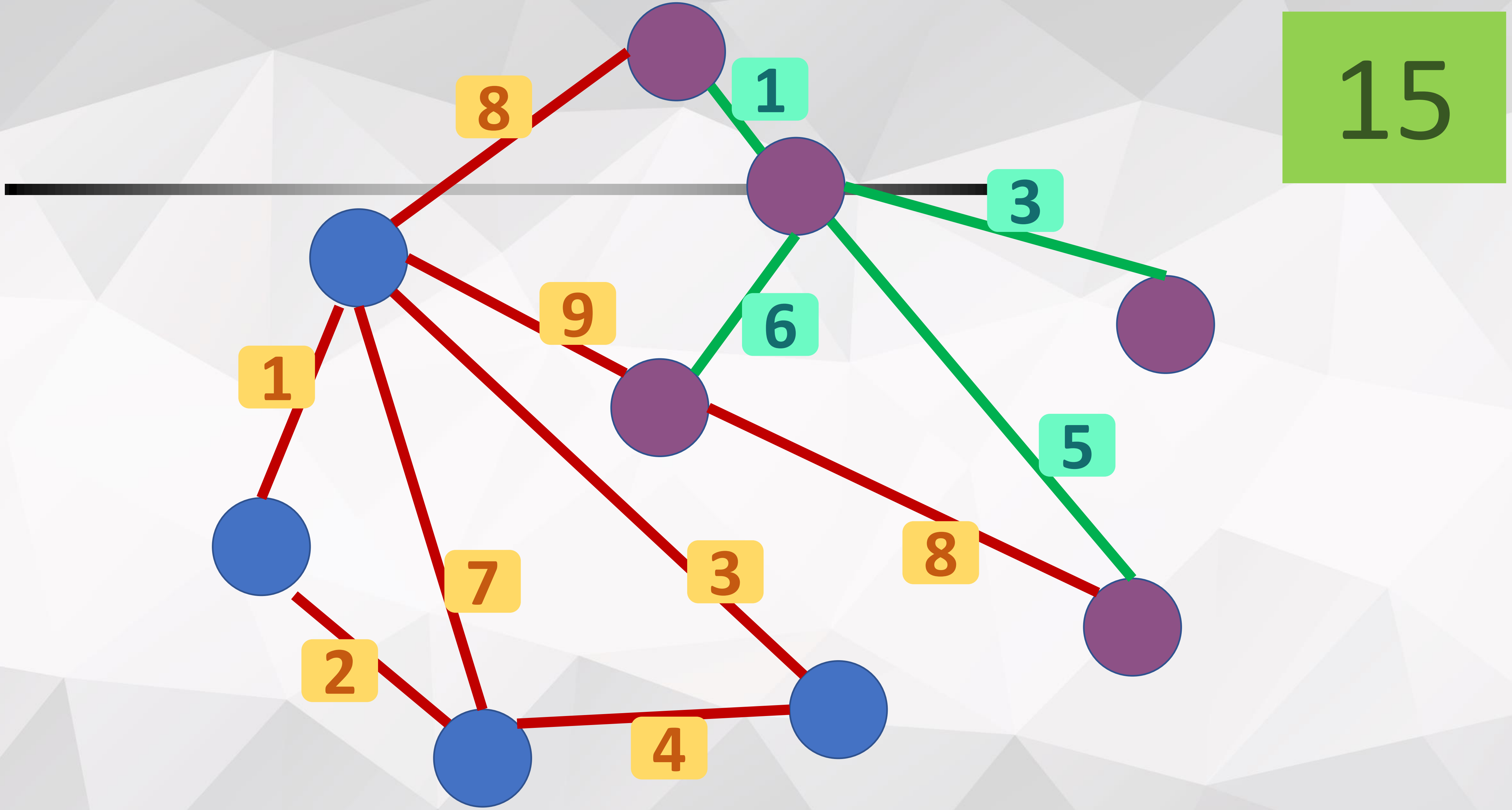
周圍最小邊

15

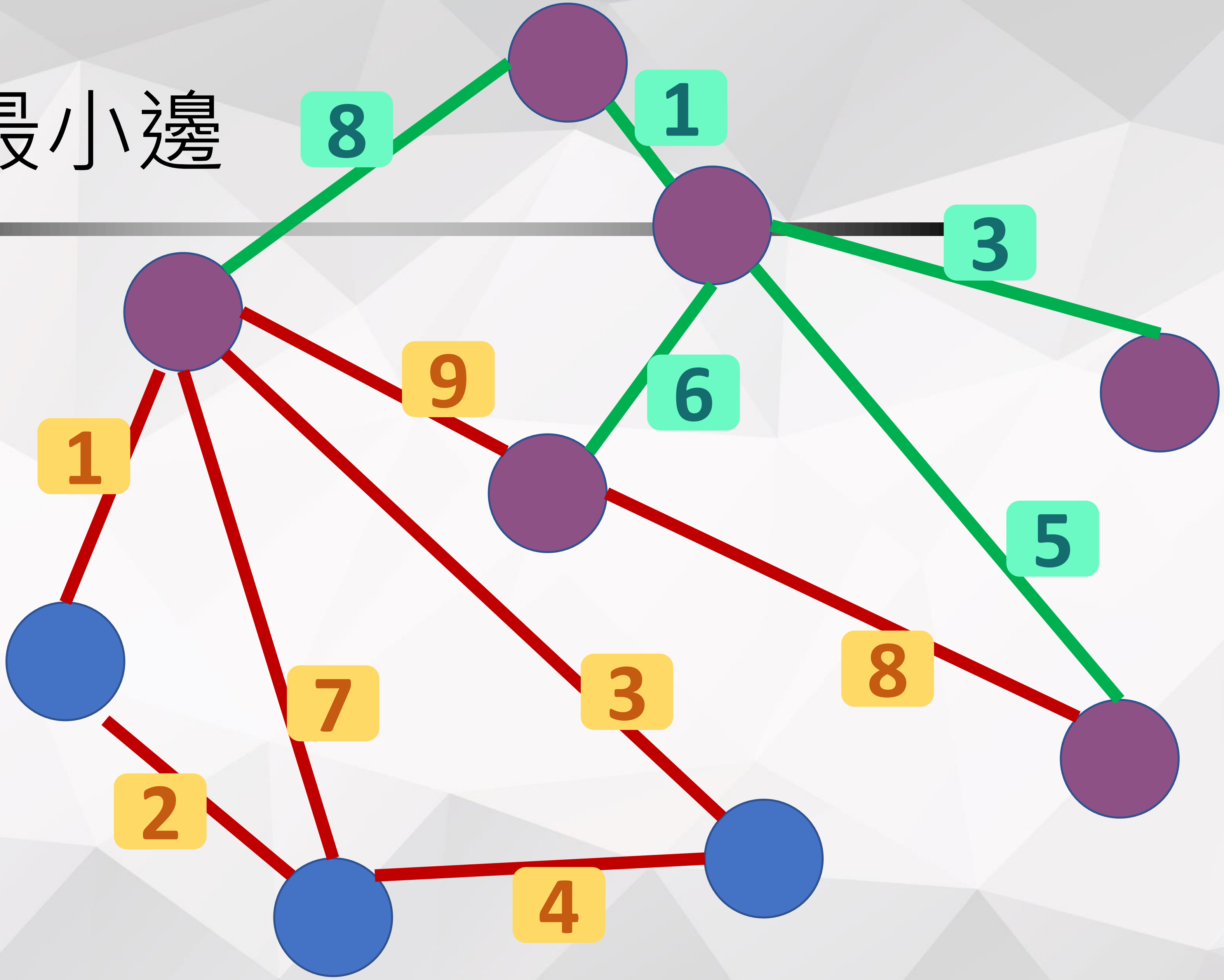


MST



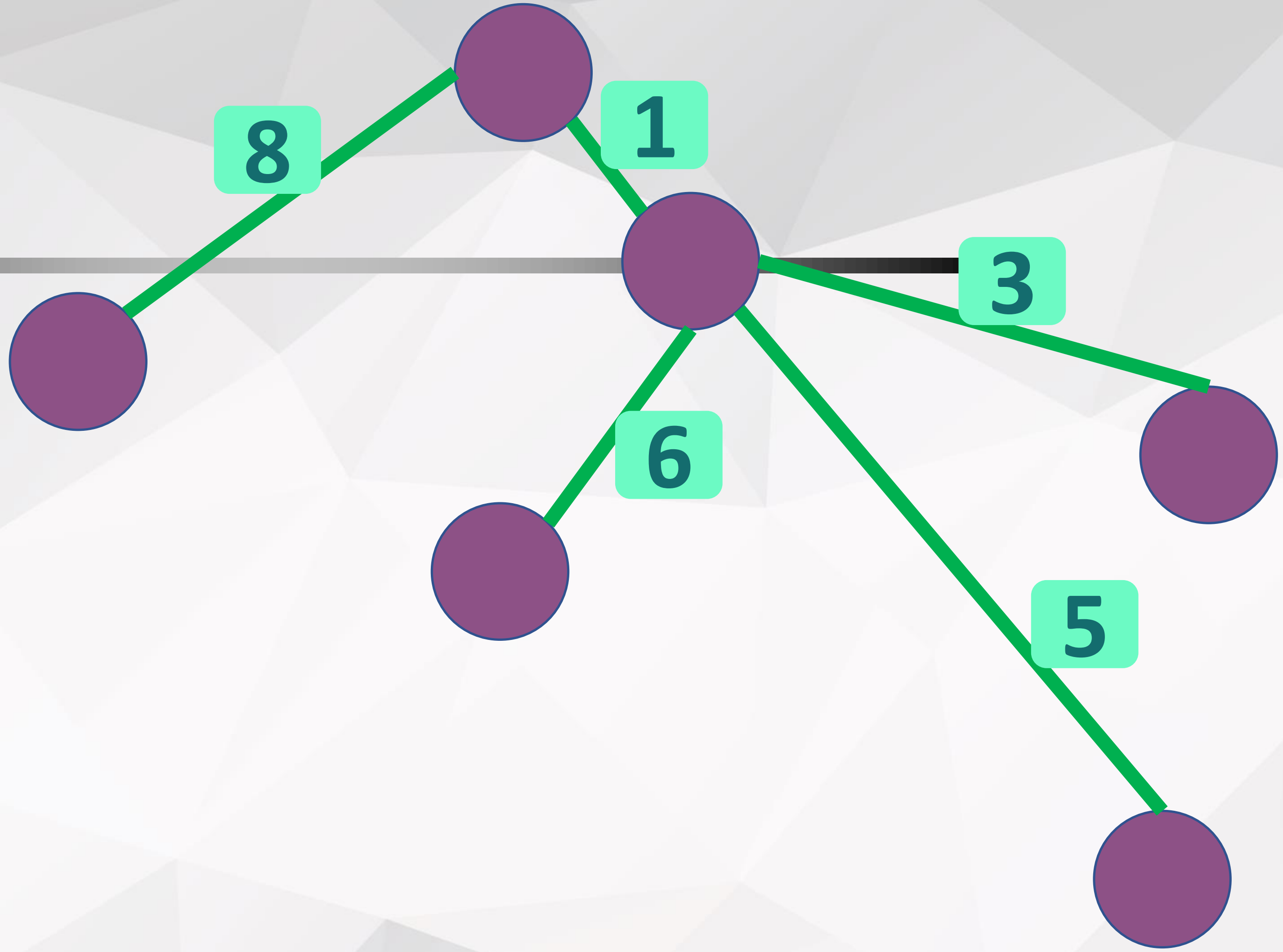

15
8
1
3
9
6
1
5
7
3
8
2
4

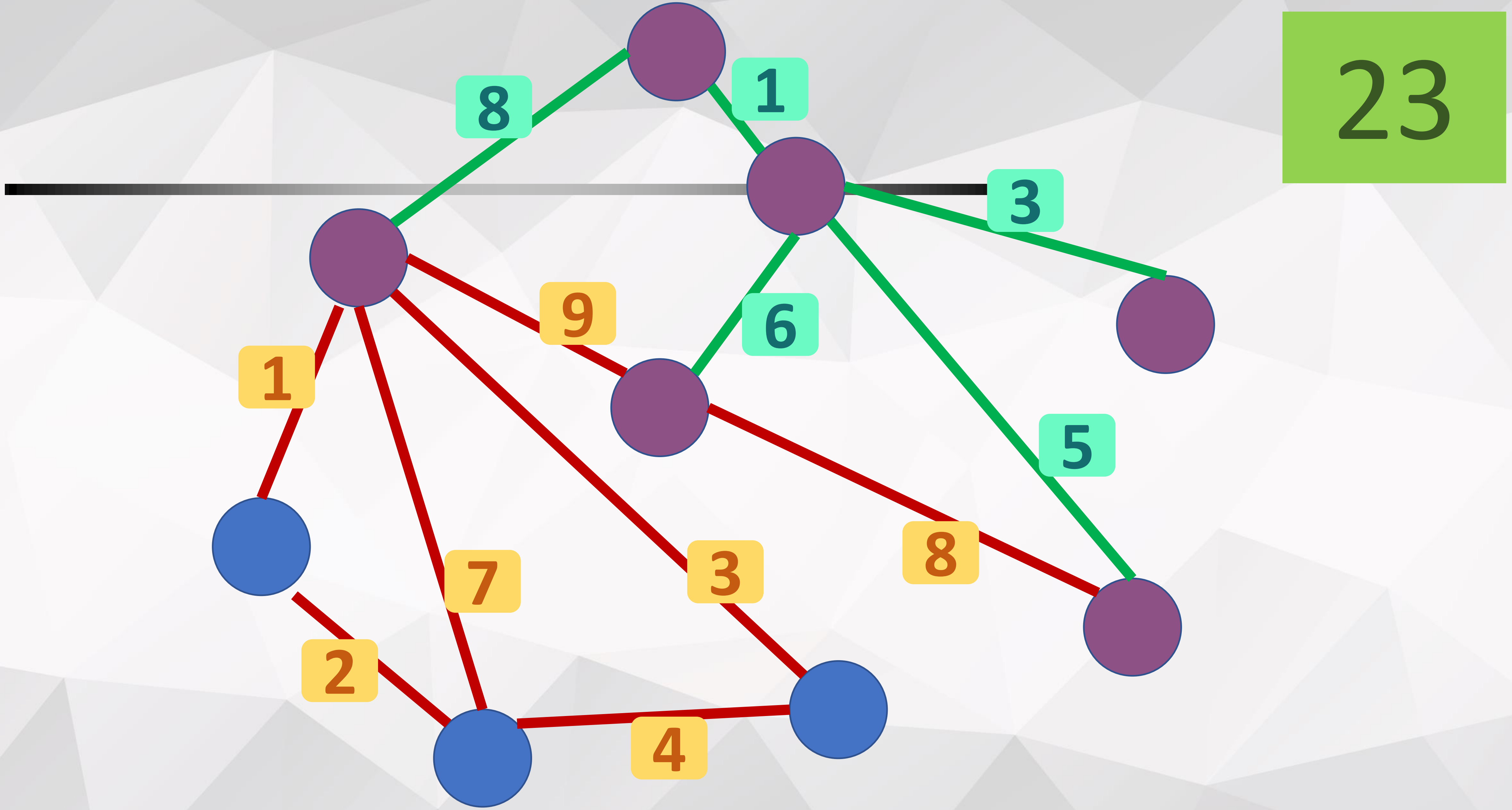
周圍最小邊



23

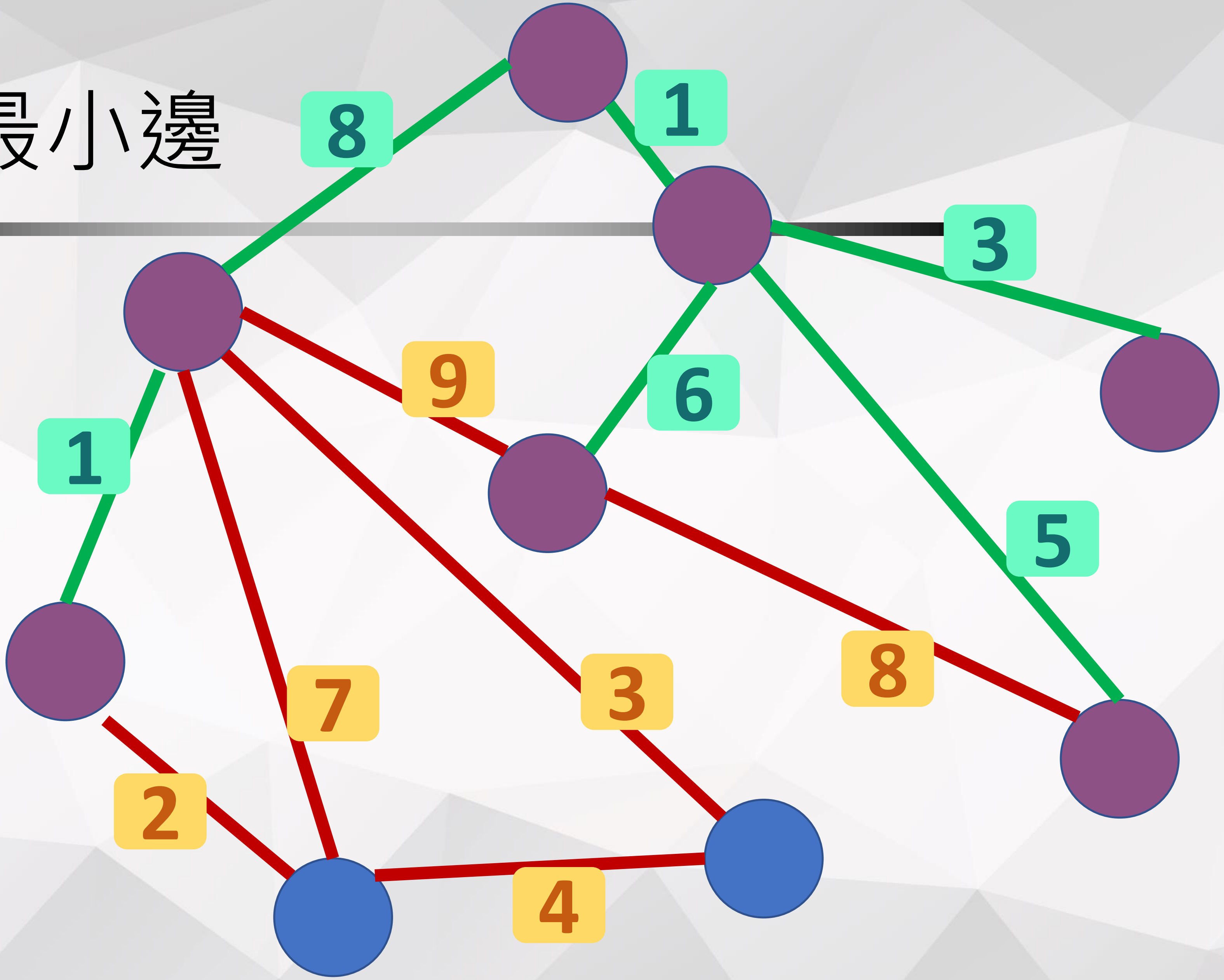
MST



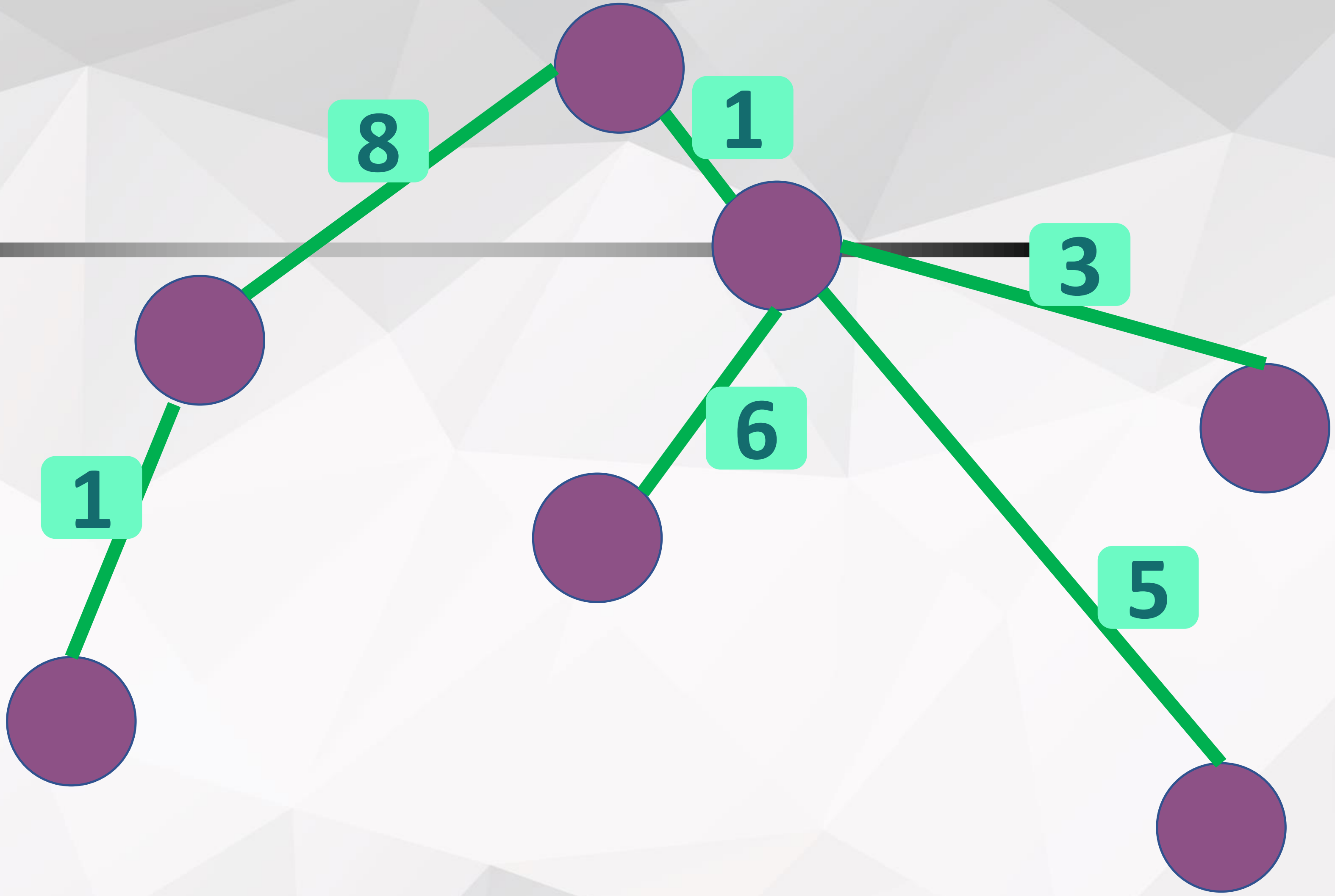

23
8
1
3
9
6
1
5
7
3
8
2
4

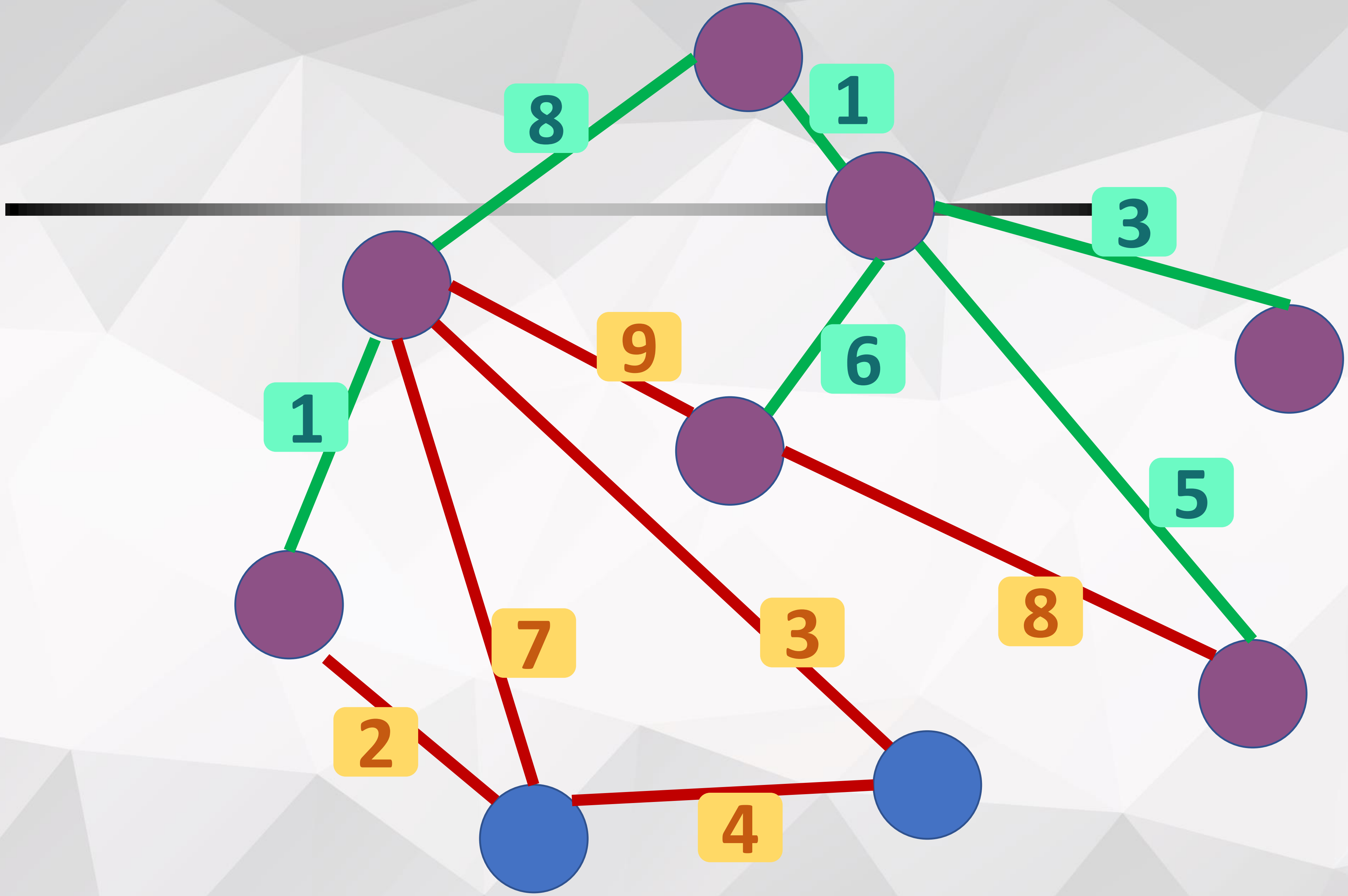
周圍最小邊

24



MST

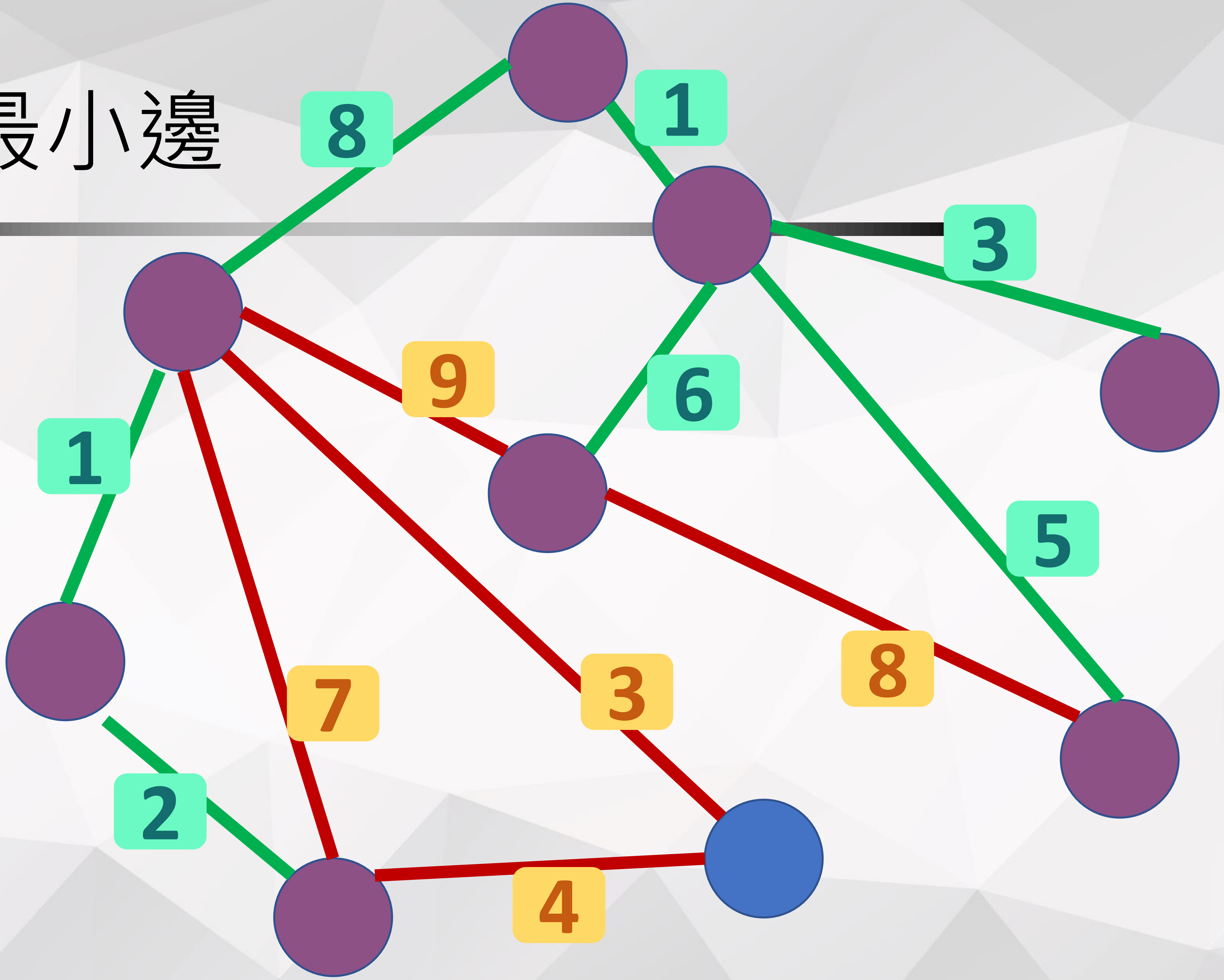




24

周圍最小邊

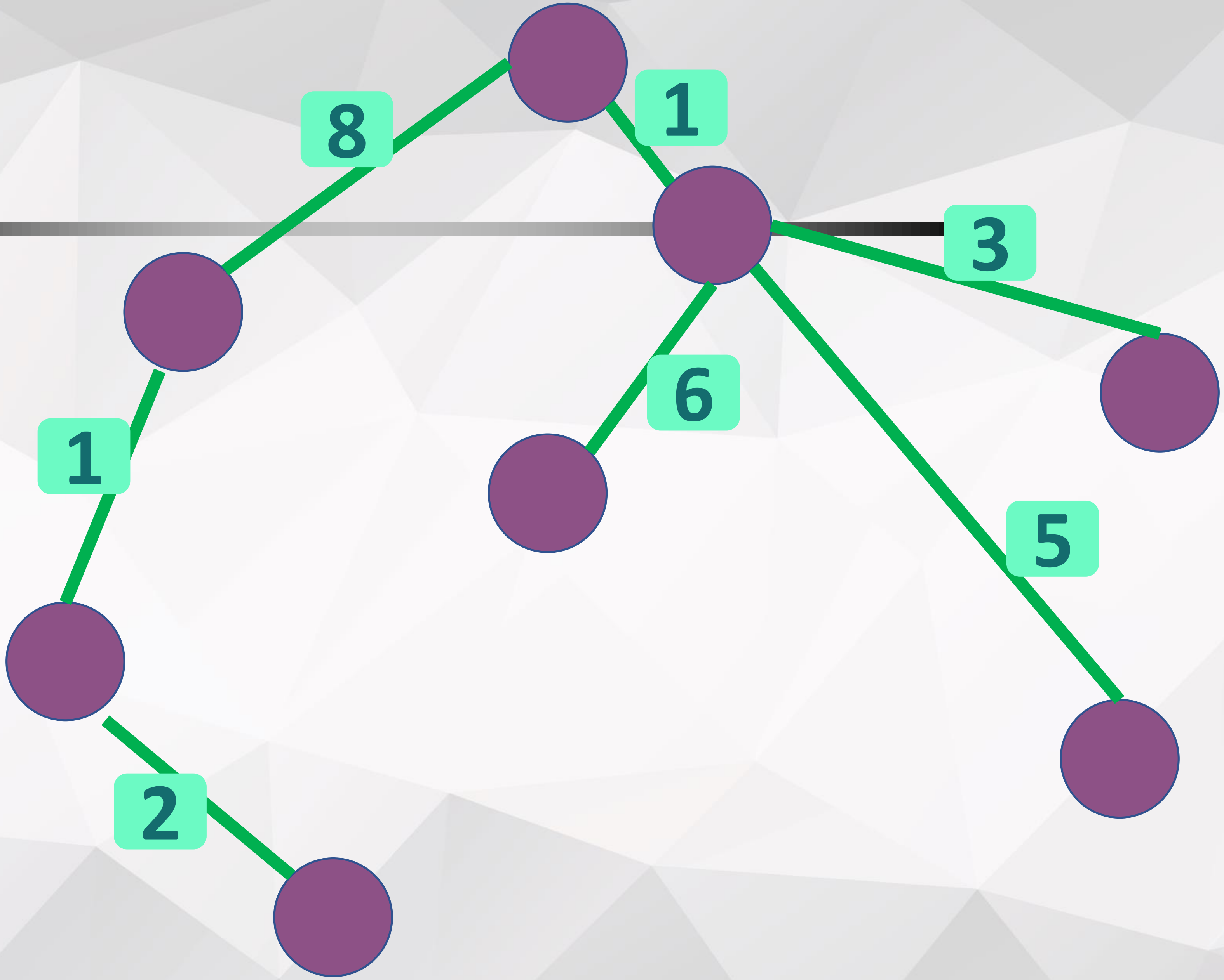
26



Competitive Programming

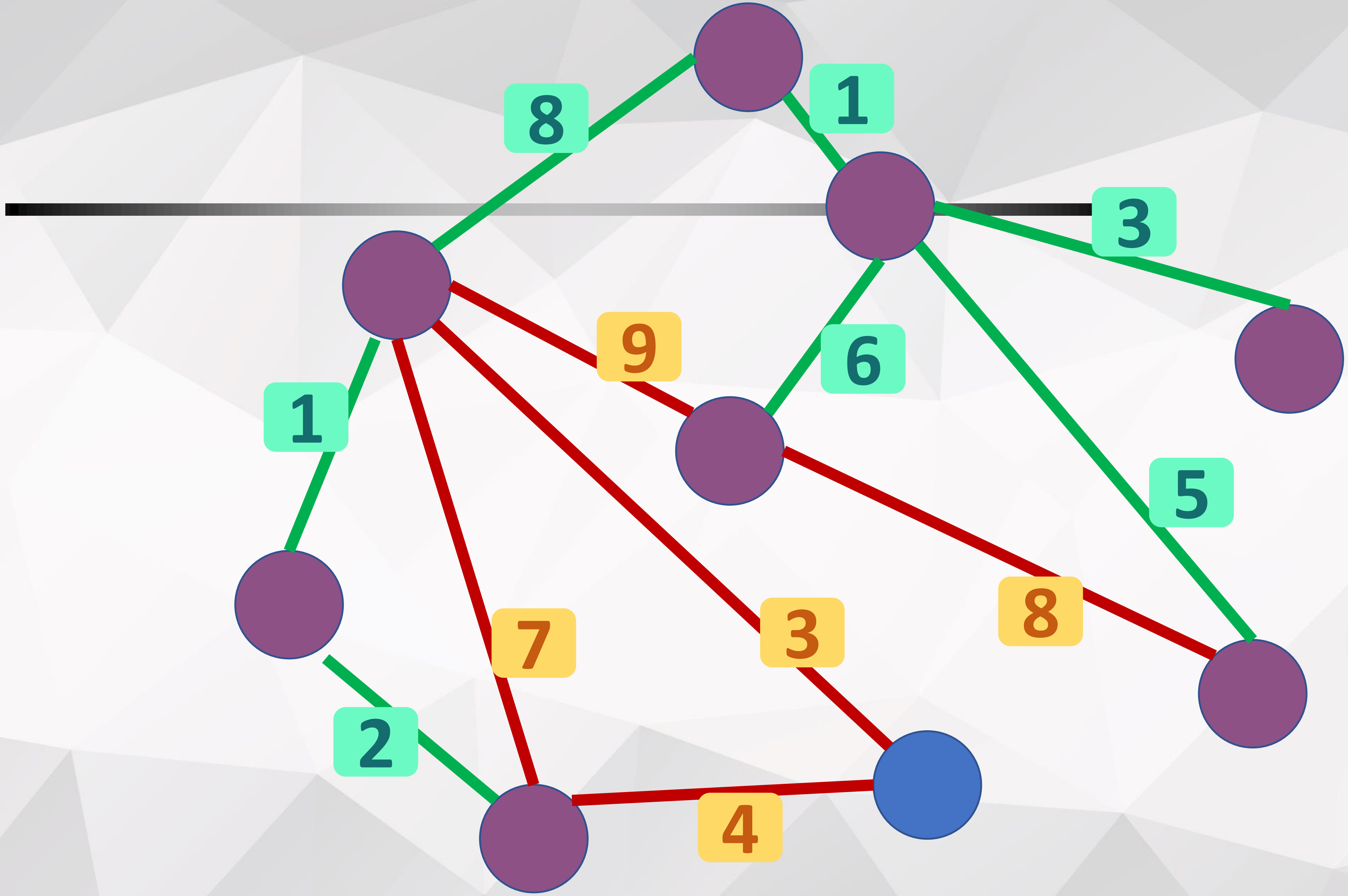
Made by 培訓團隊群 CCNS

MST



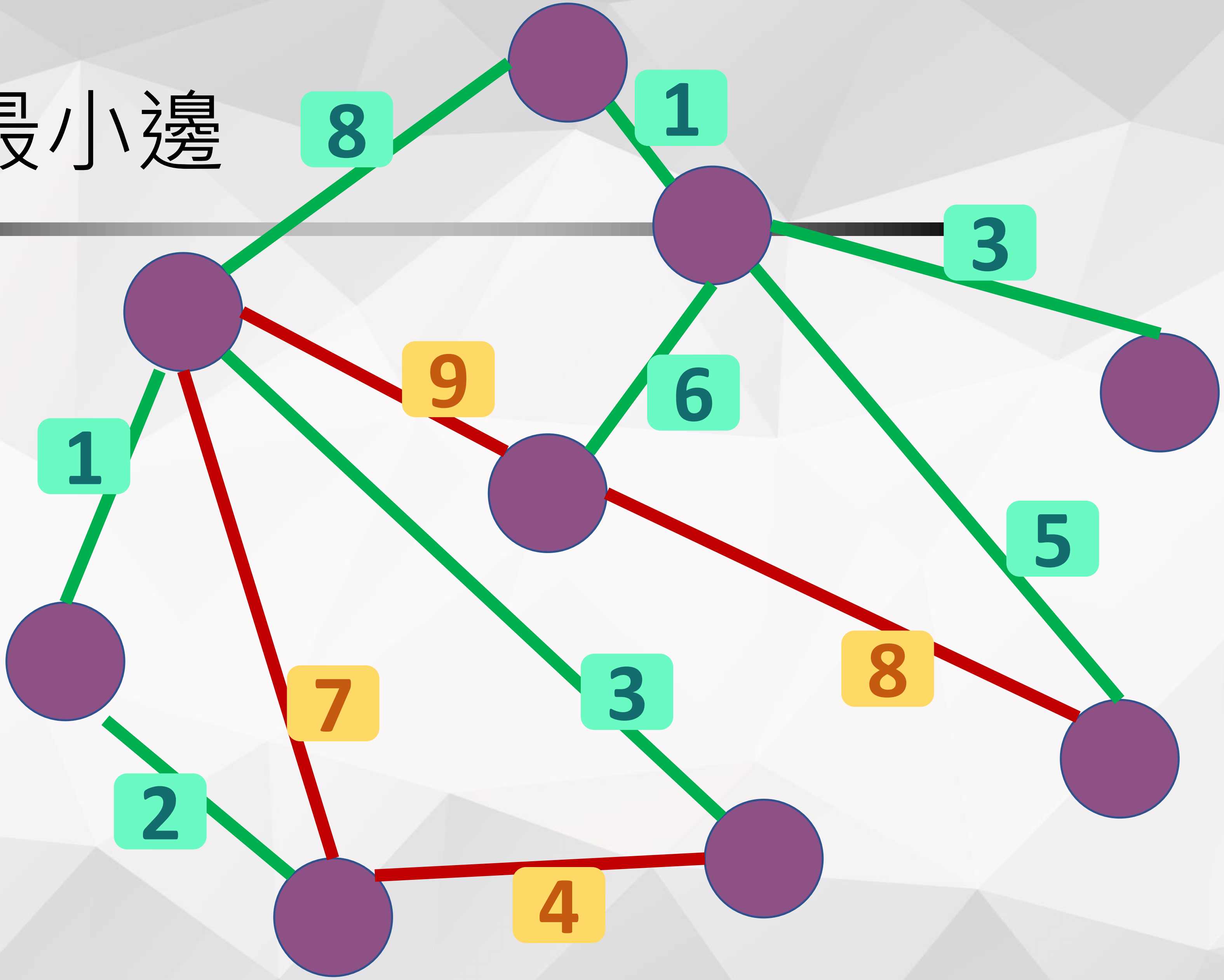
26

Competitive Programming

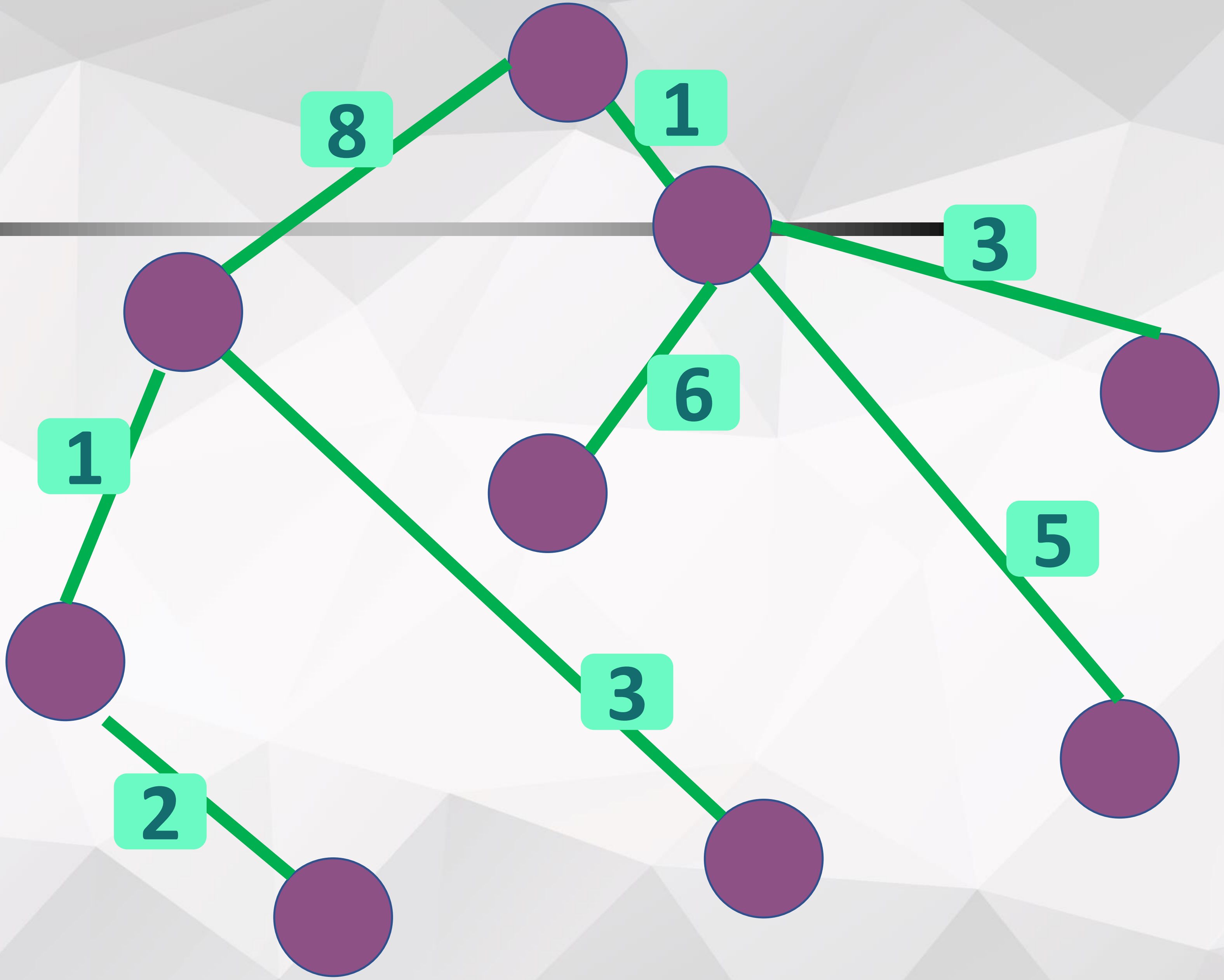


26

周圍最小邊



MST



Prim 實作

```
struct node {
   int id; // 點的編號
   int w; // 連結到此點的權重（邊權重）
};
```

Prim 實作

```
vector<node> E[maxv]; // maxv 為最大節點數
    :
    .

/* 假設輸入完邊的資訊了 */
```

Prim 實作

```
/* 每次挑最小權重的邊 */
priority_queue<edge> Q;



/* 初始的生成樹（只有一個點）*/
Q.push({1, 1, 0});
```

Prim 實作

```
while(!Q.empty()) {
   edge e = Q.top(); Q.pop();
   int u = e.v;

   if(!vis[u]) { // 避免出現環
      MST.push_back(e);
      cost += e.w;

      for(auto v: E[u])
         if(!vis[v.id]) Q.push({u, v.id, v.w});
   }

   vis[u] = true;
}
```

Prim 實作

跟 Kruskal 比較

Prim 枚舉的是**點**
Kruskal 枚舉的是**邊**

其複雜度為 $O(|E|\log_2|V|)$

Questions?

Outline

- 術語複習
  - Graph
  - Tree
- 最小生成樹
- A* 搜尋法則
- 單源最短路徑
- 全點對最短路徑

Single-Source Shortest Paths

SSSP

- 給定 **源**點/起點(Source)

- 問每條路徑的最小成本
  - 源點到各點的最小總成本

SSSP

- Breadth-First Search
- Relaxation
- Dijkstra's algorithm
- Bellman-Ford's algorithm

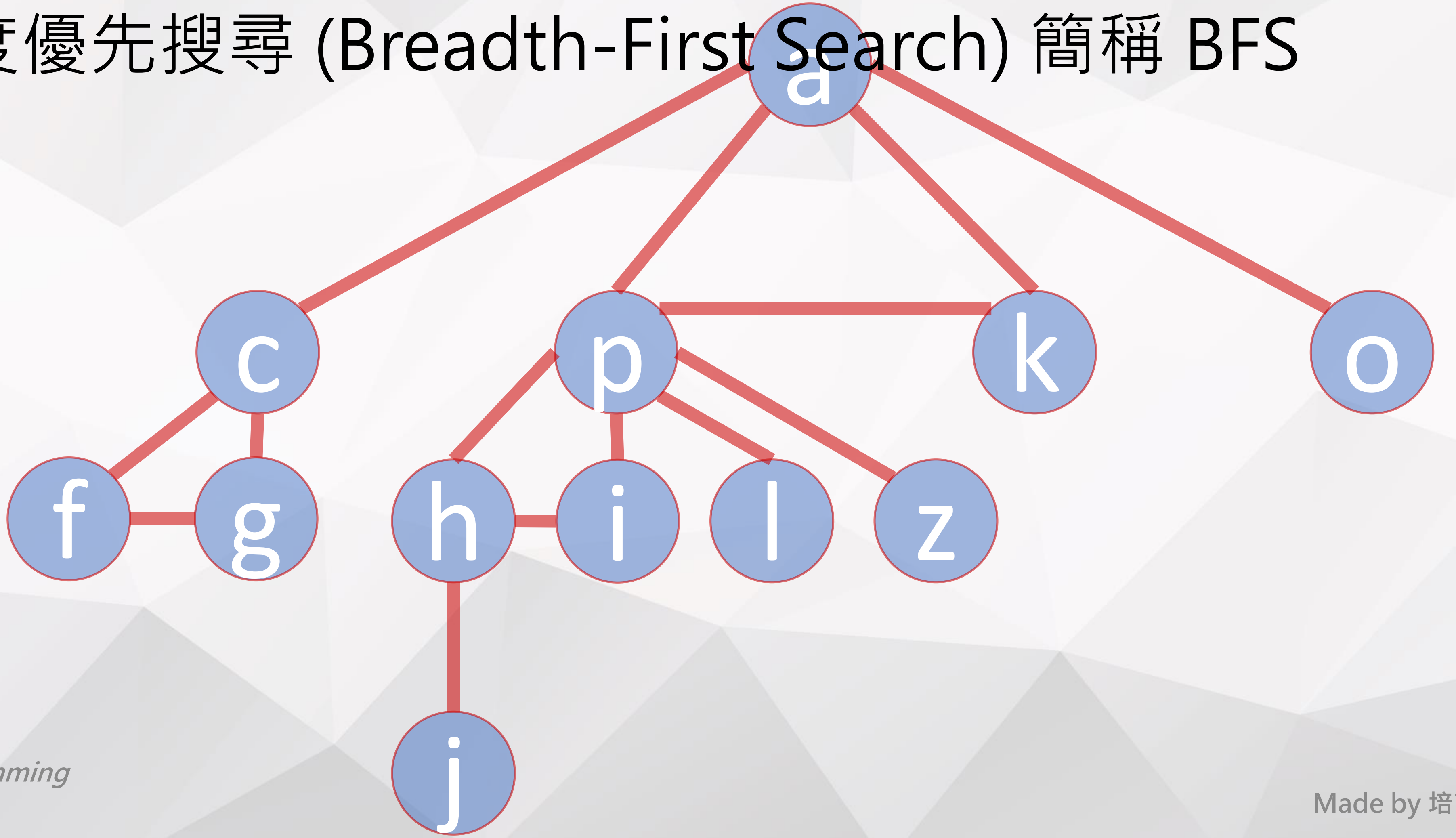
SSSP

- Breadth-First Search
- Relaxation
- Dijkstra's algorithm
- Bellman-Ford's algorithm

廣度優先搜尋

BFS

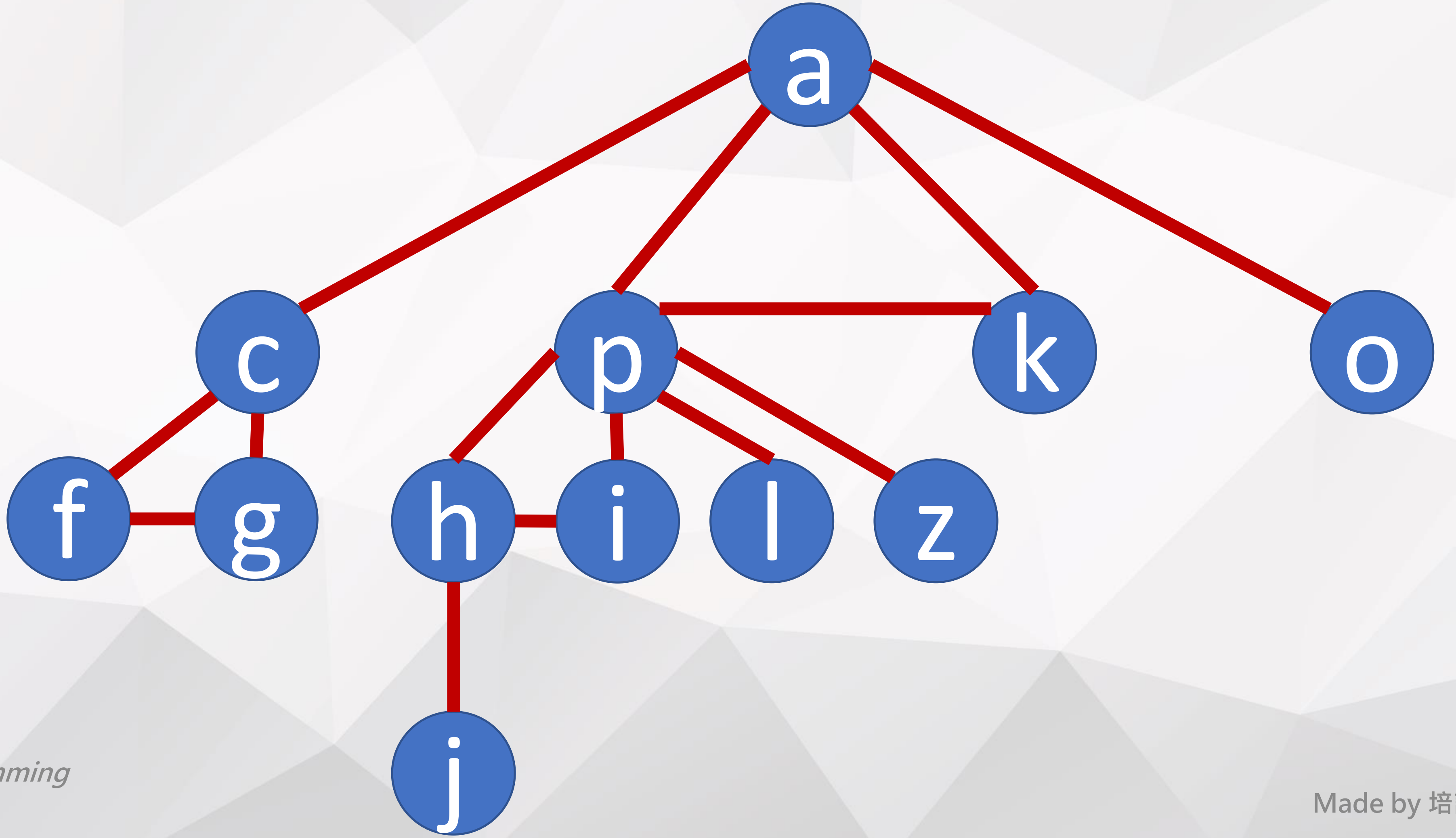
廣度優先搜尋 (Breadth-First Search) 簡稱 BFS



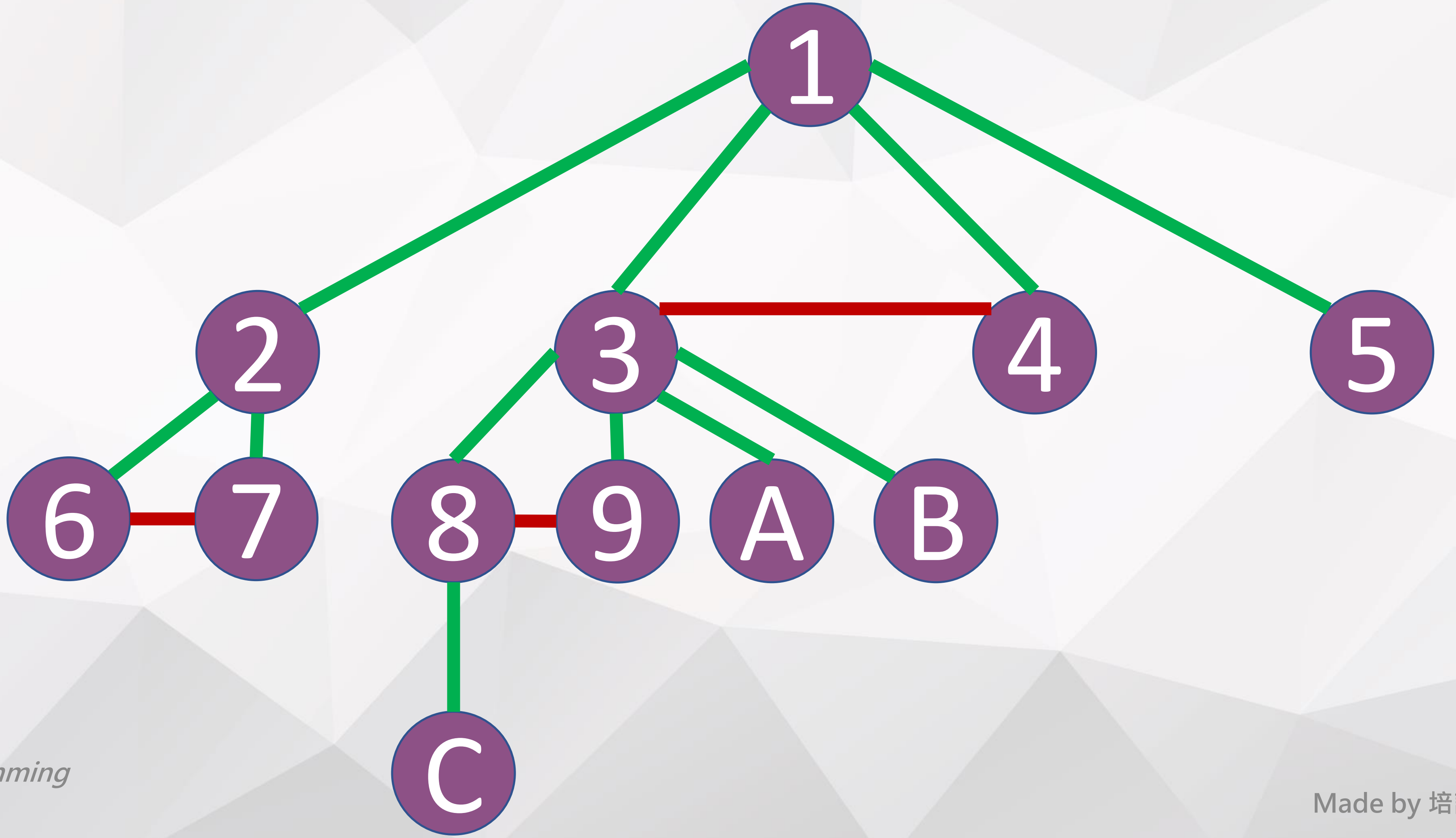
BFS 程式碼

```cpp
queue<int> Q;
Q.push(source);
vis[source] = true;

while (!Q.empty()) {
  int u = Q.front(); Q.pop();
  for (auto v: E[u]) {
    if (vis[v]) continue;
    vis[v] = true;
    Q.push(v);
  }
}
```

BFS 的**點**遍歷順序

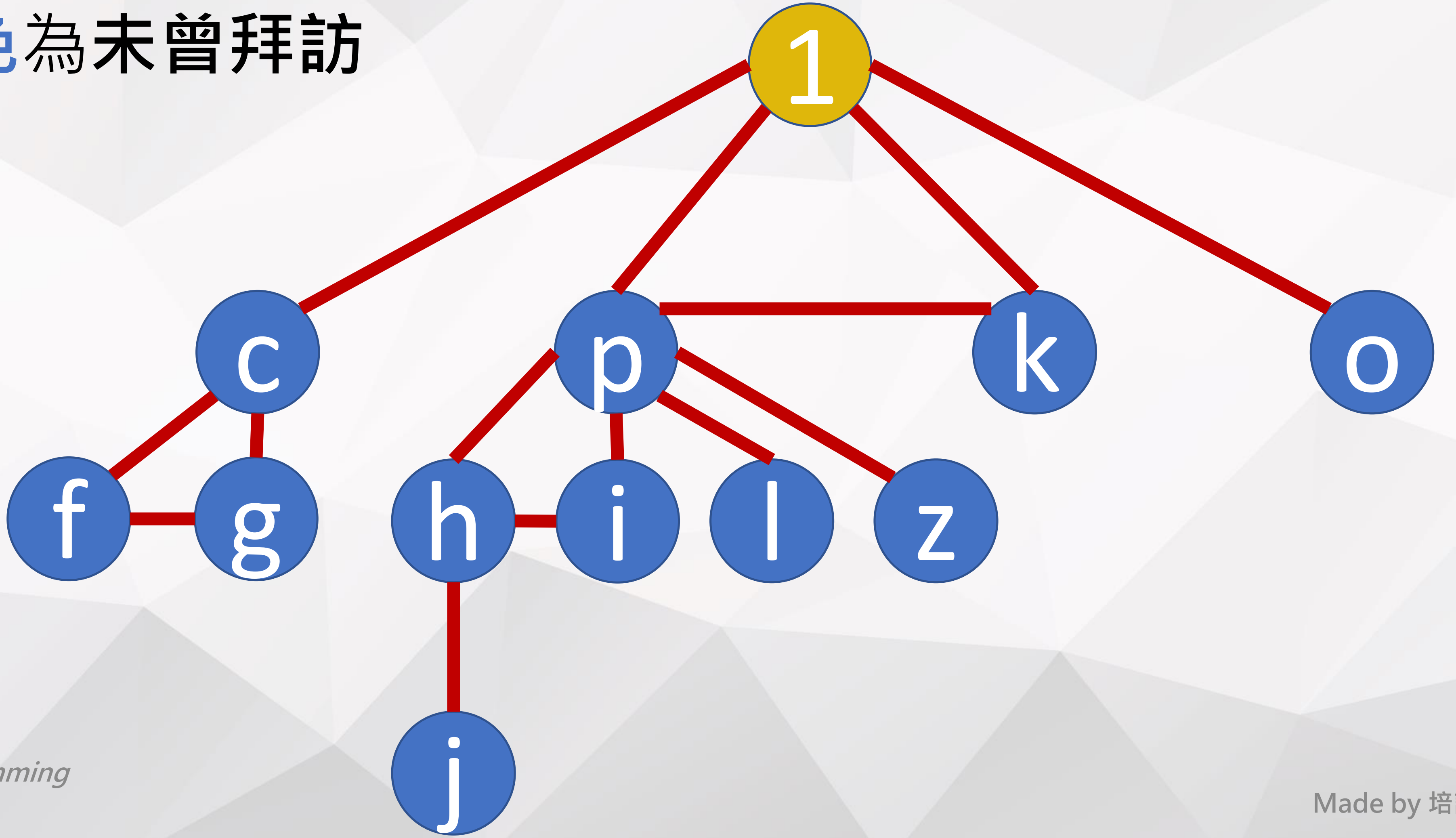


BFS 的**點**遍歷順序



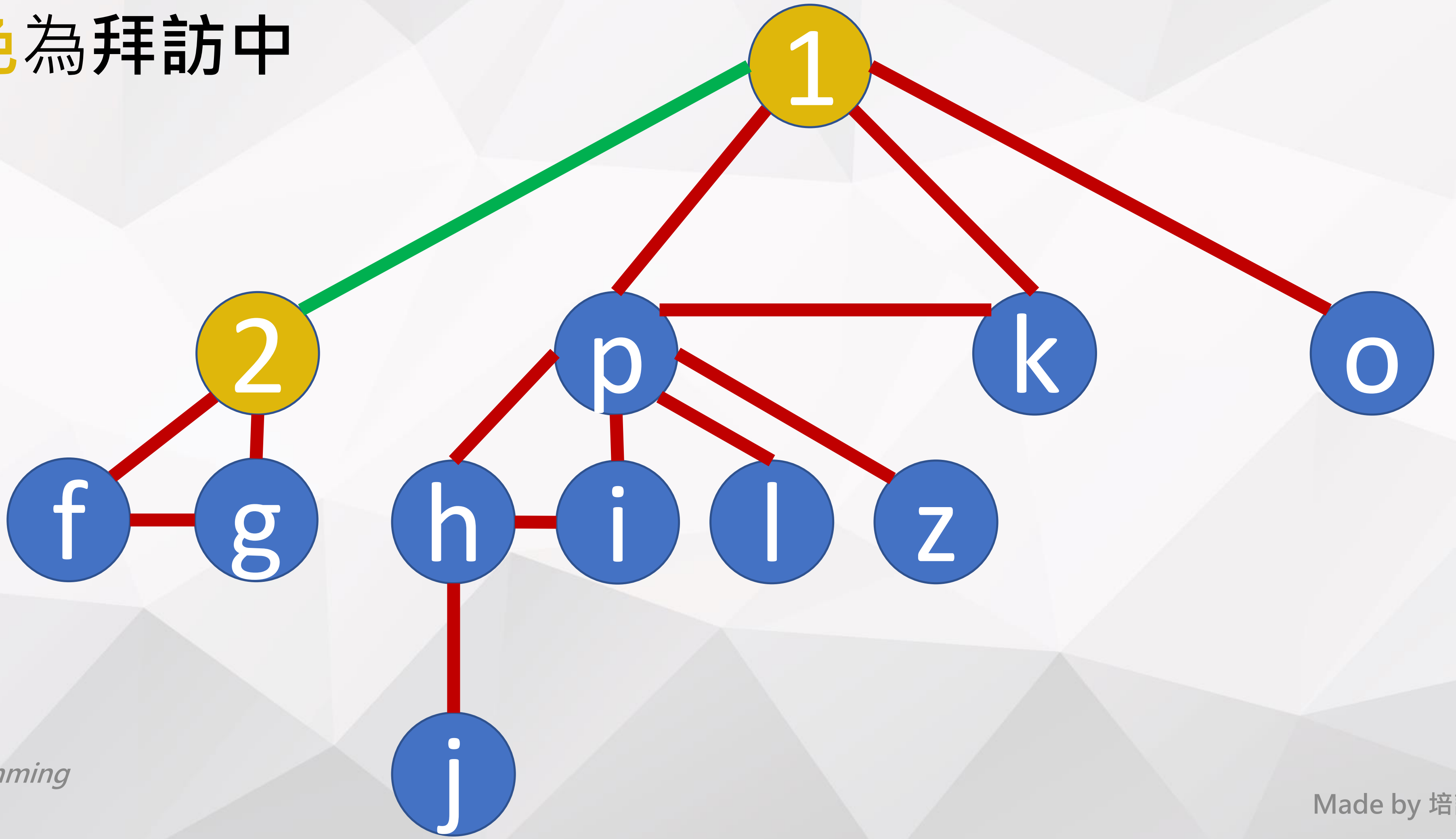
第一個拜訪的為**根**

藍色為**未曾拜訪**

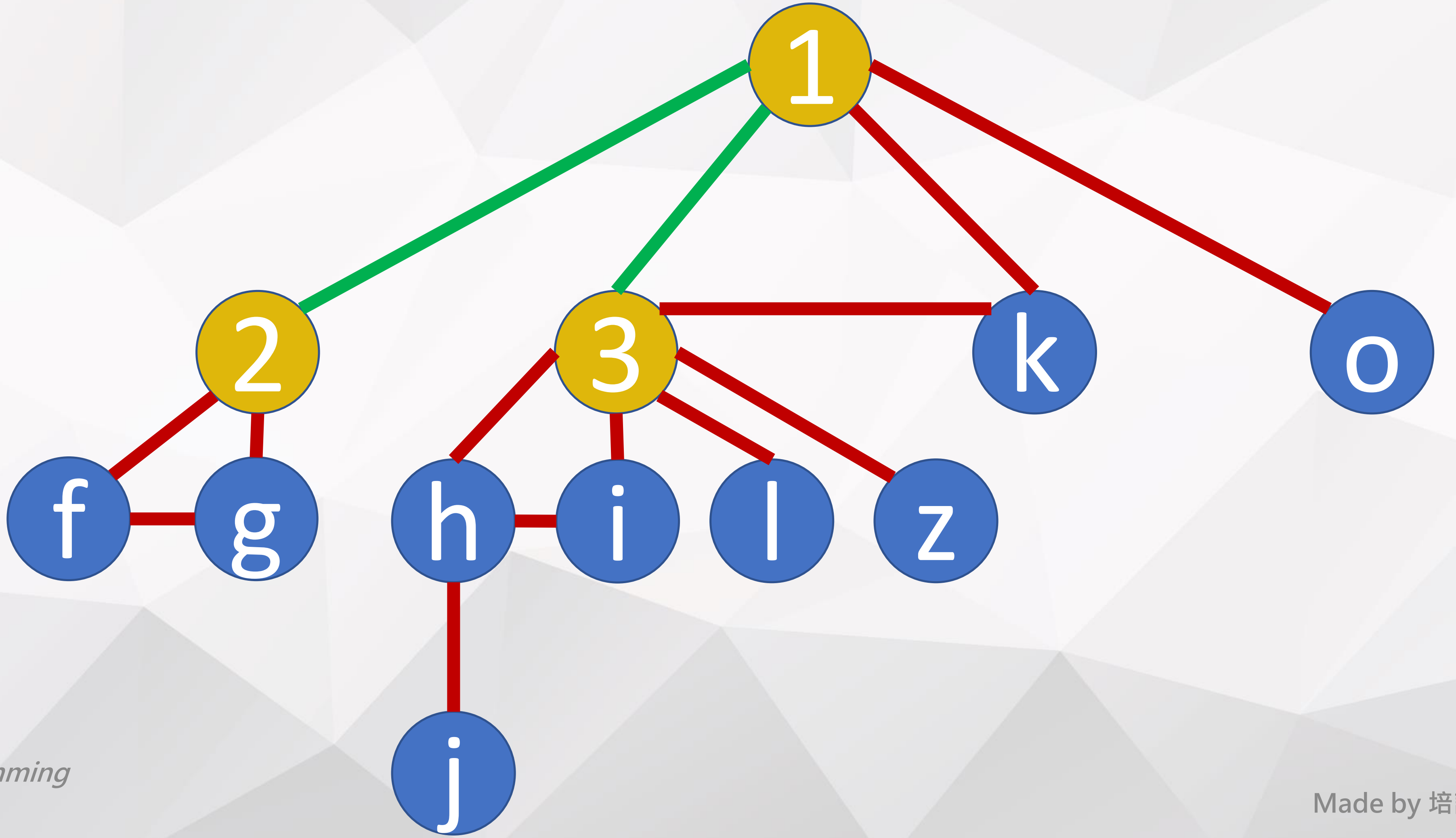


拜訪所有鄰點

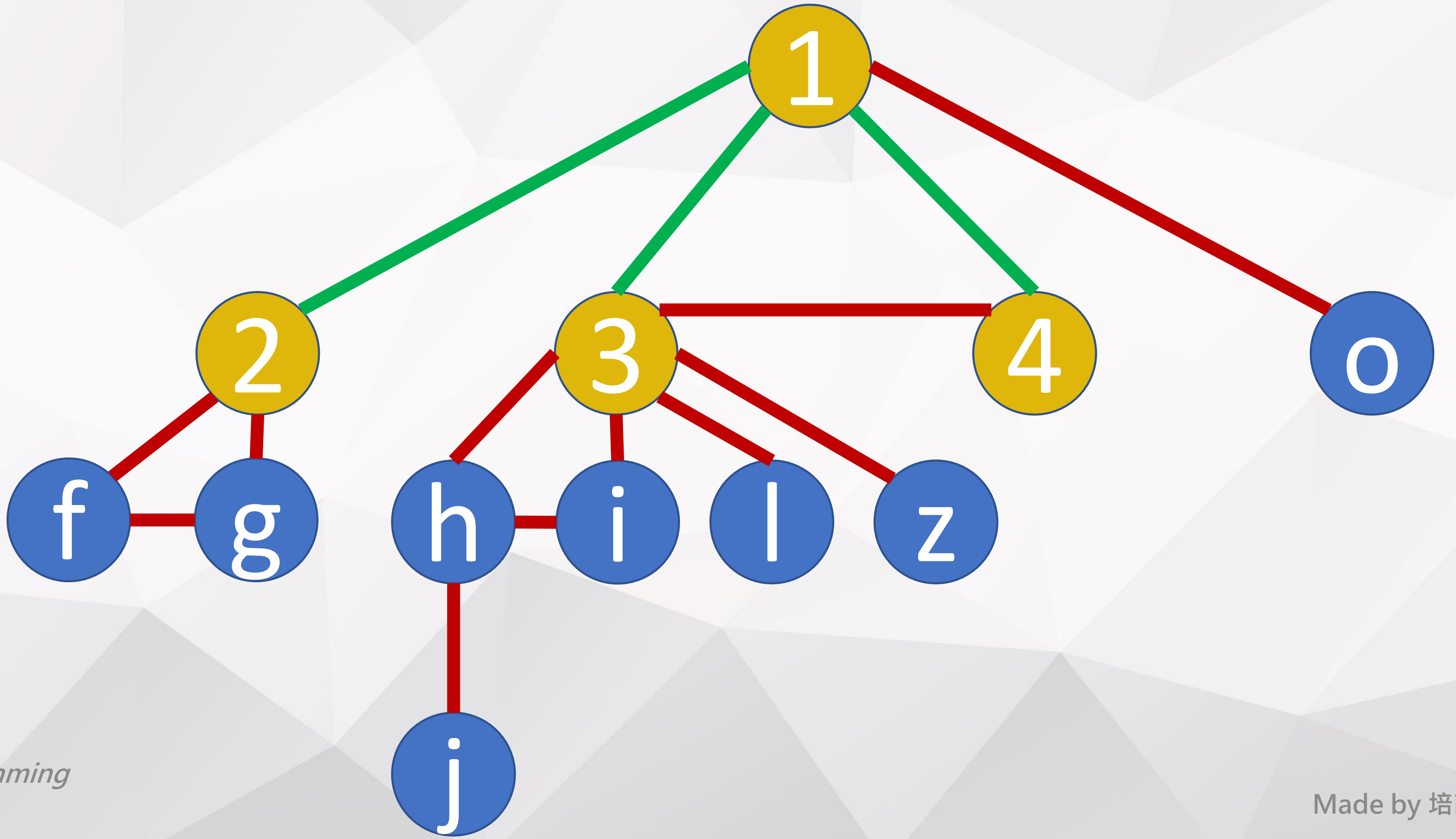
黃色為拜訪中



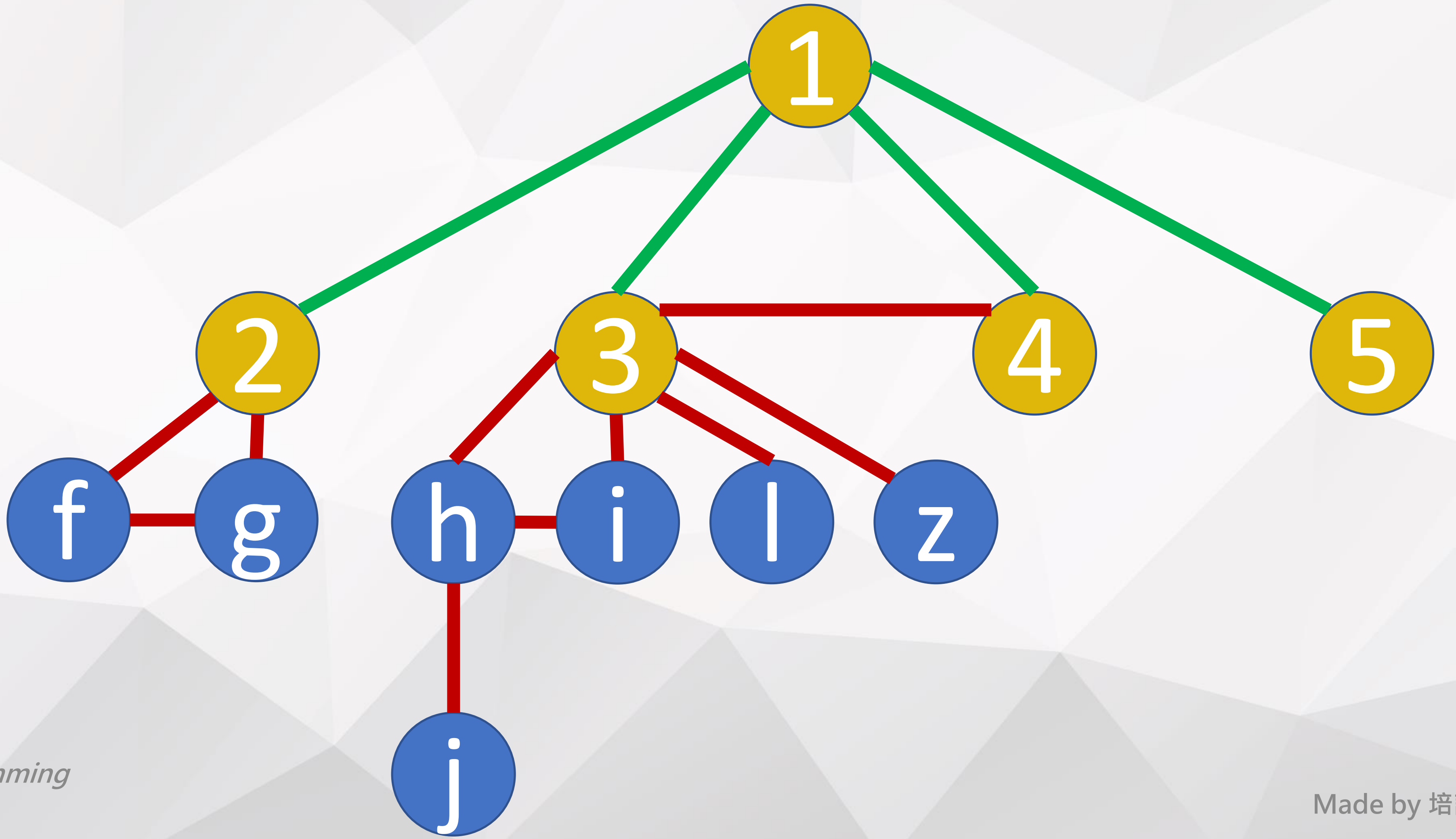
拜訪所有鄰點



拜訪所有鄰點

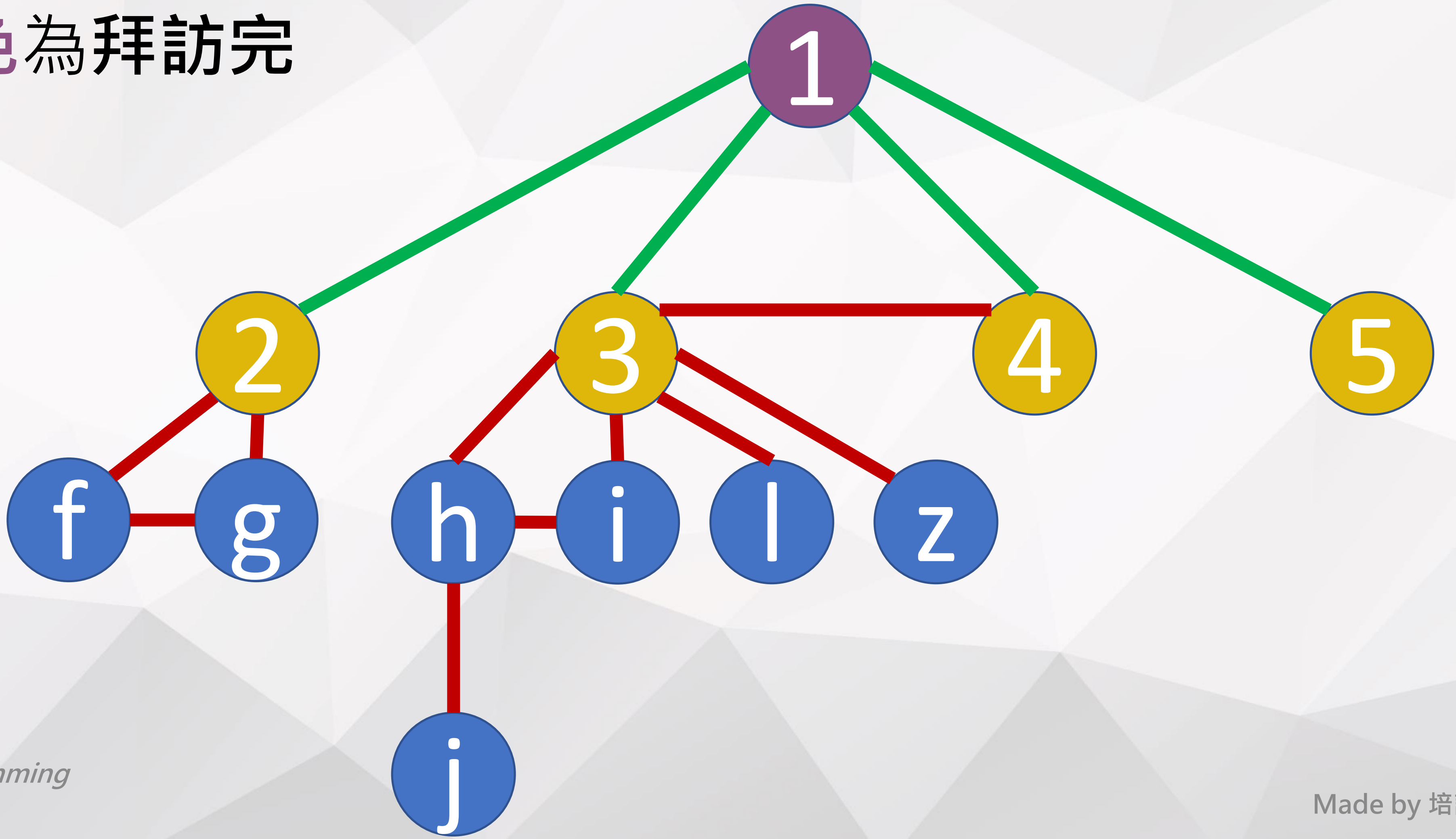


拜訪所有鄰點

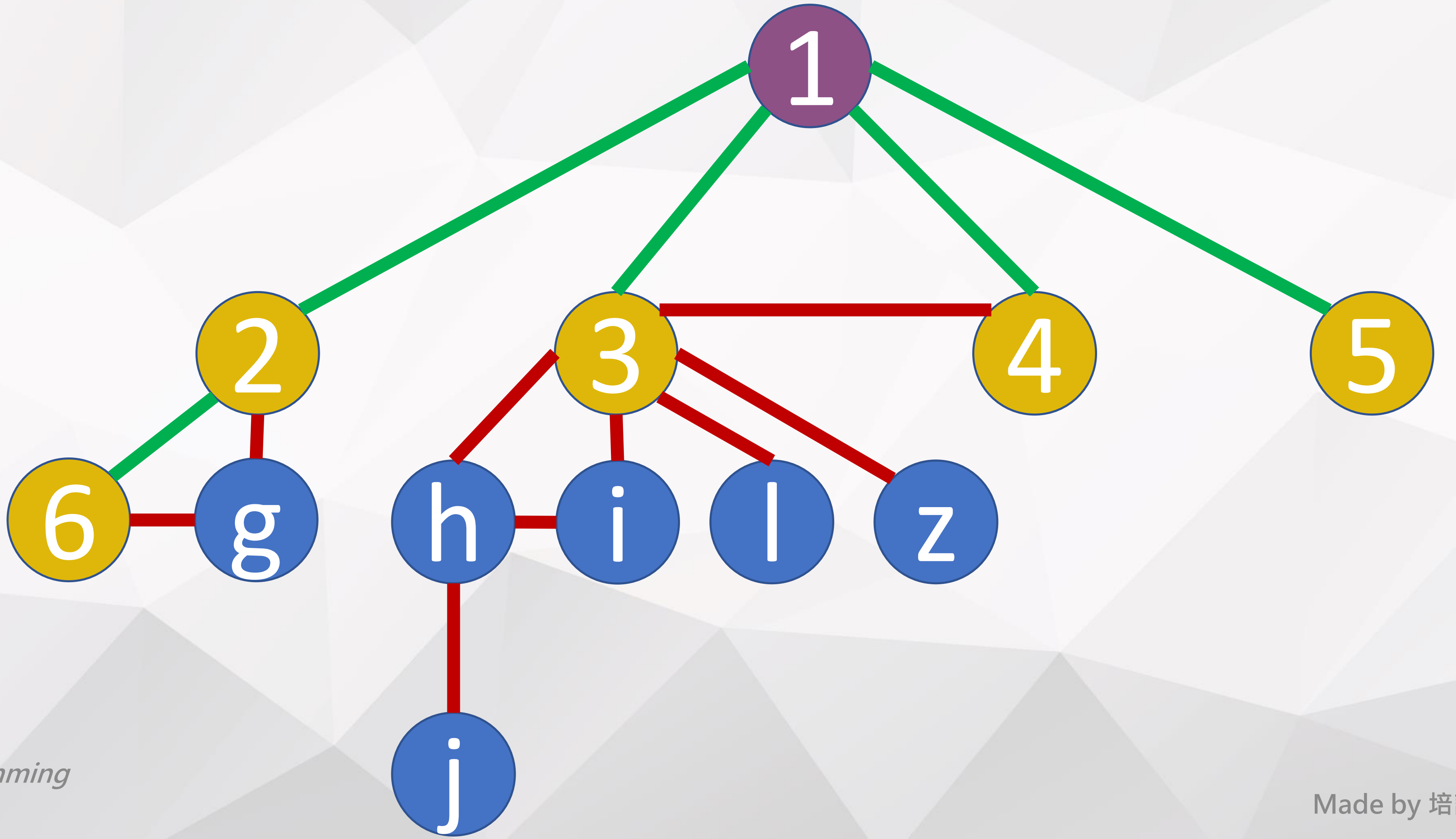


根拜訪完

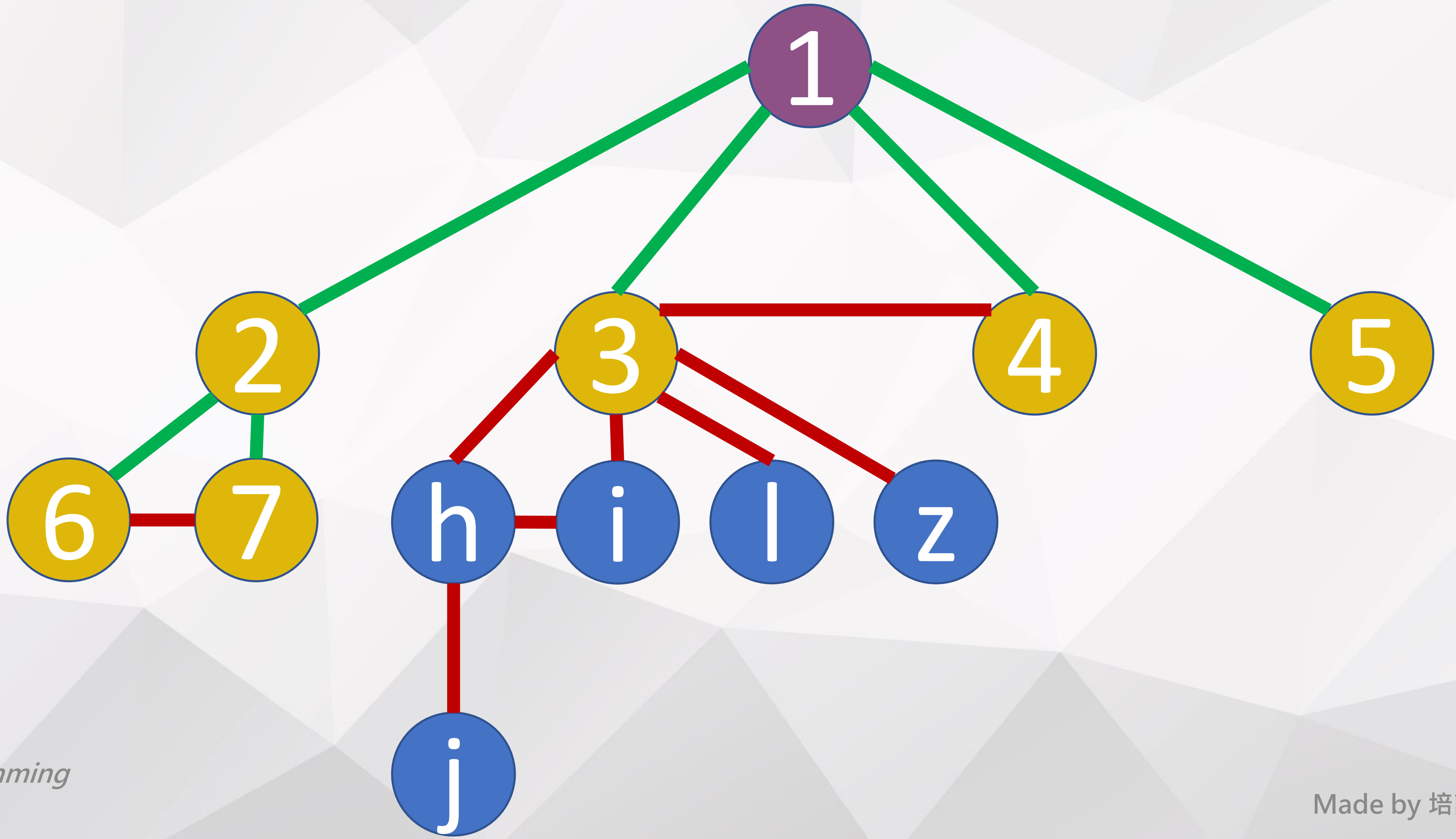
紫色為拜訪完



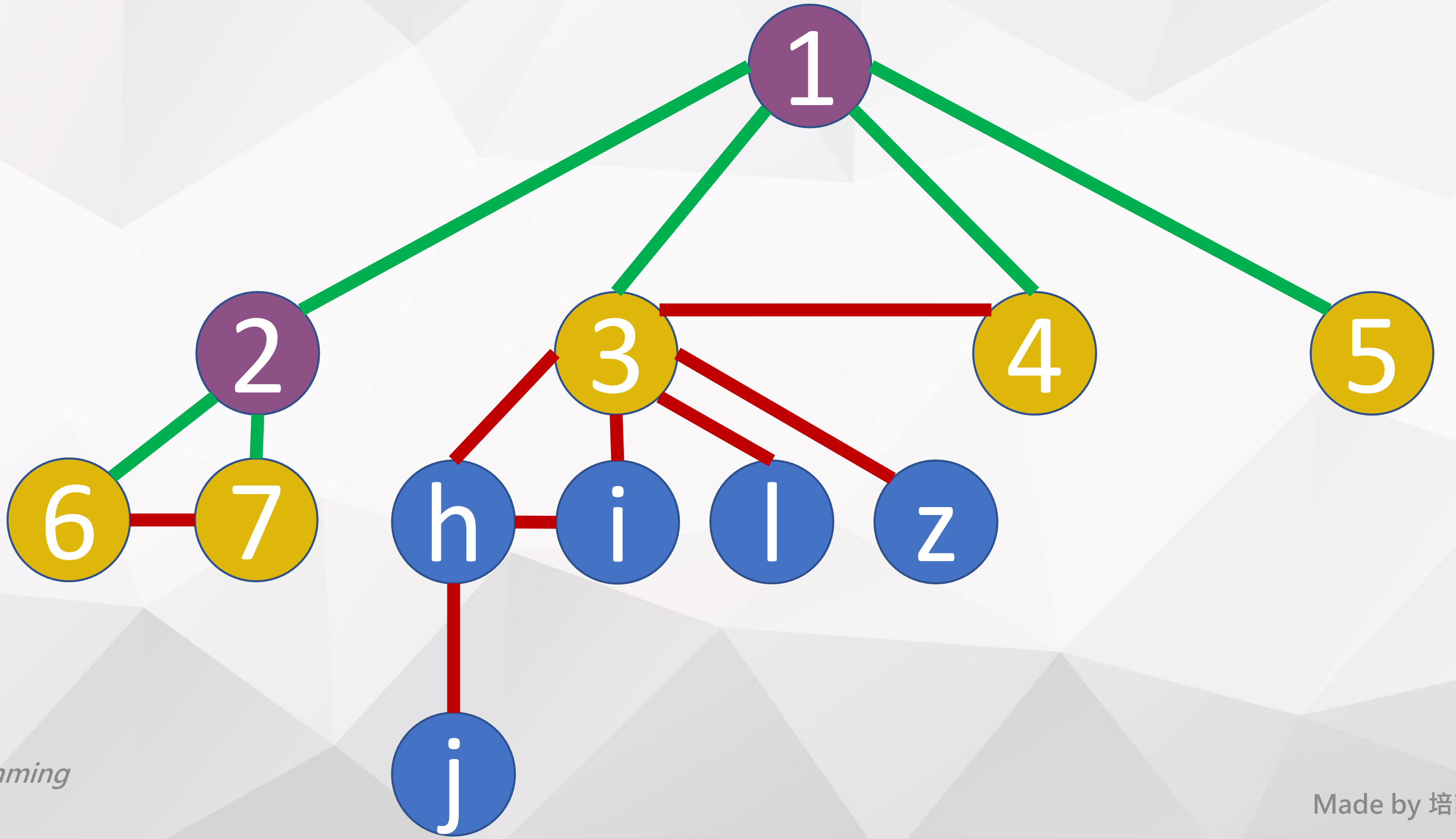
拜訪所有鄰點



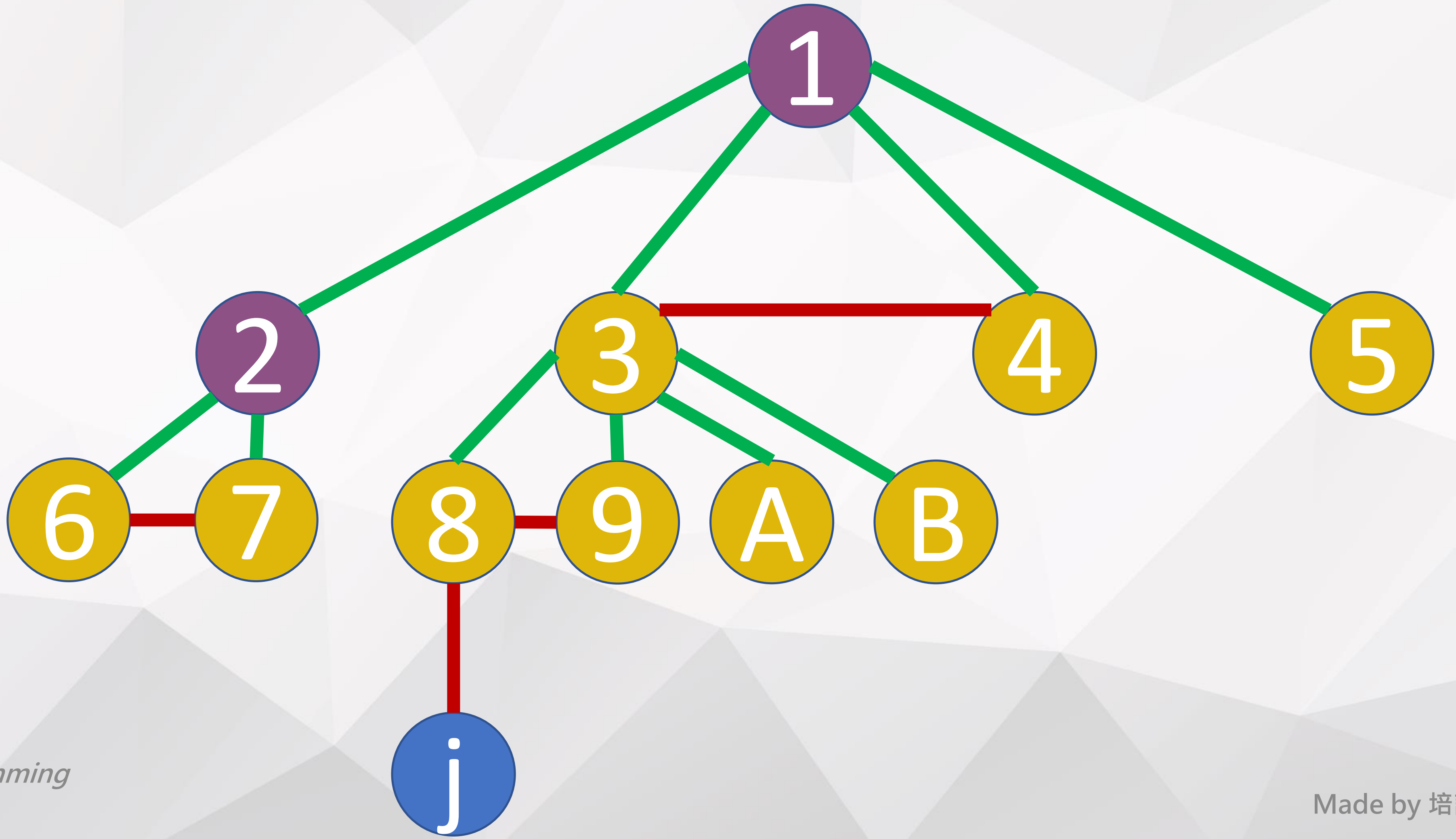
拜訪所有鄰點



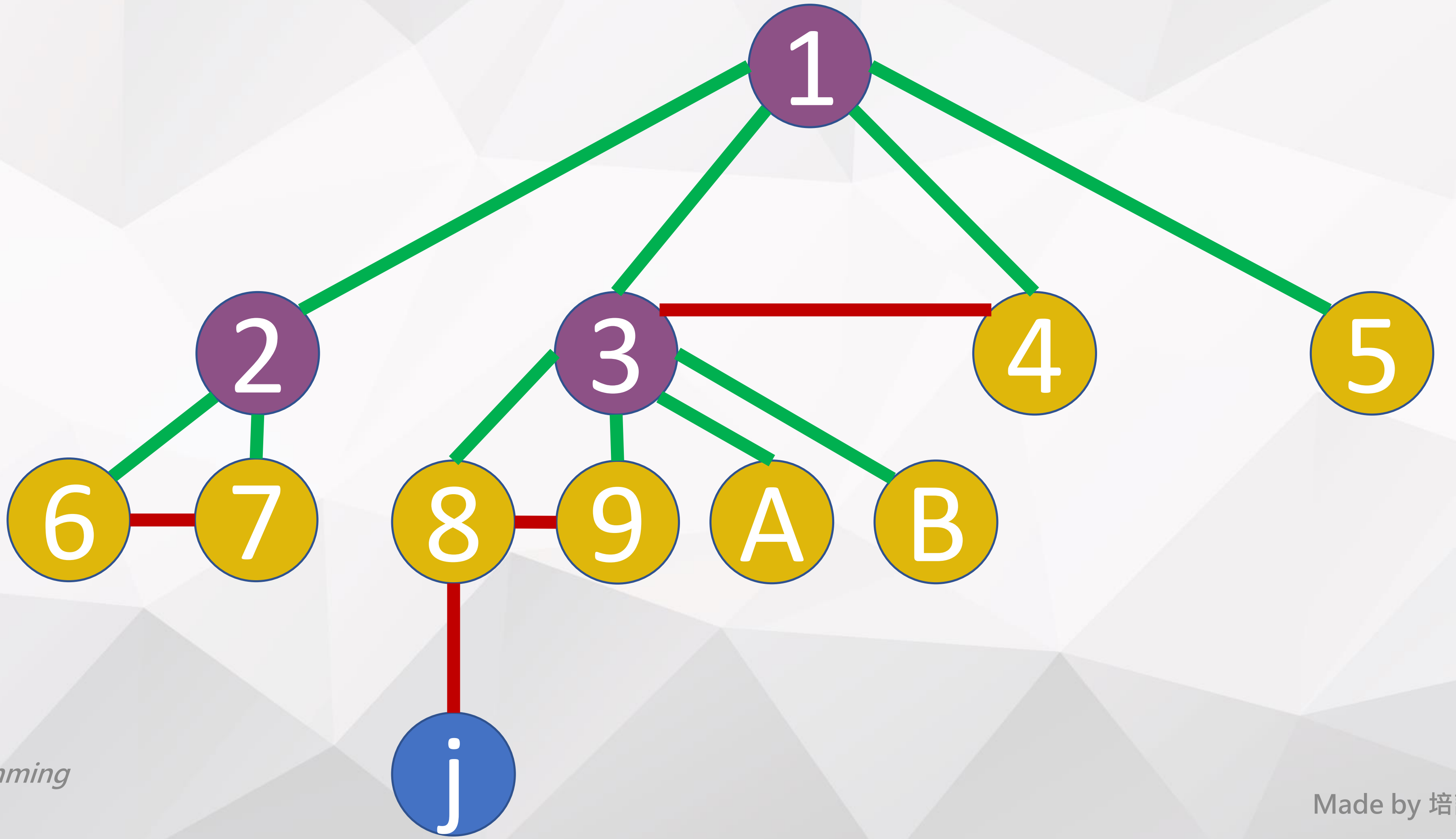
拜訪完



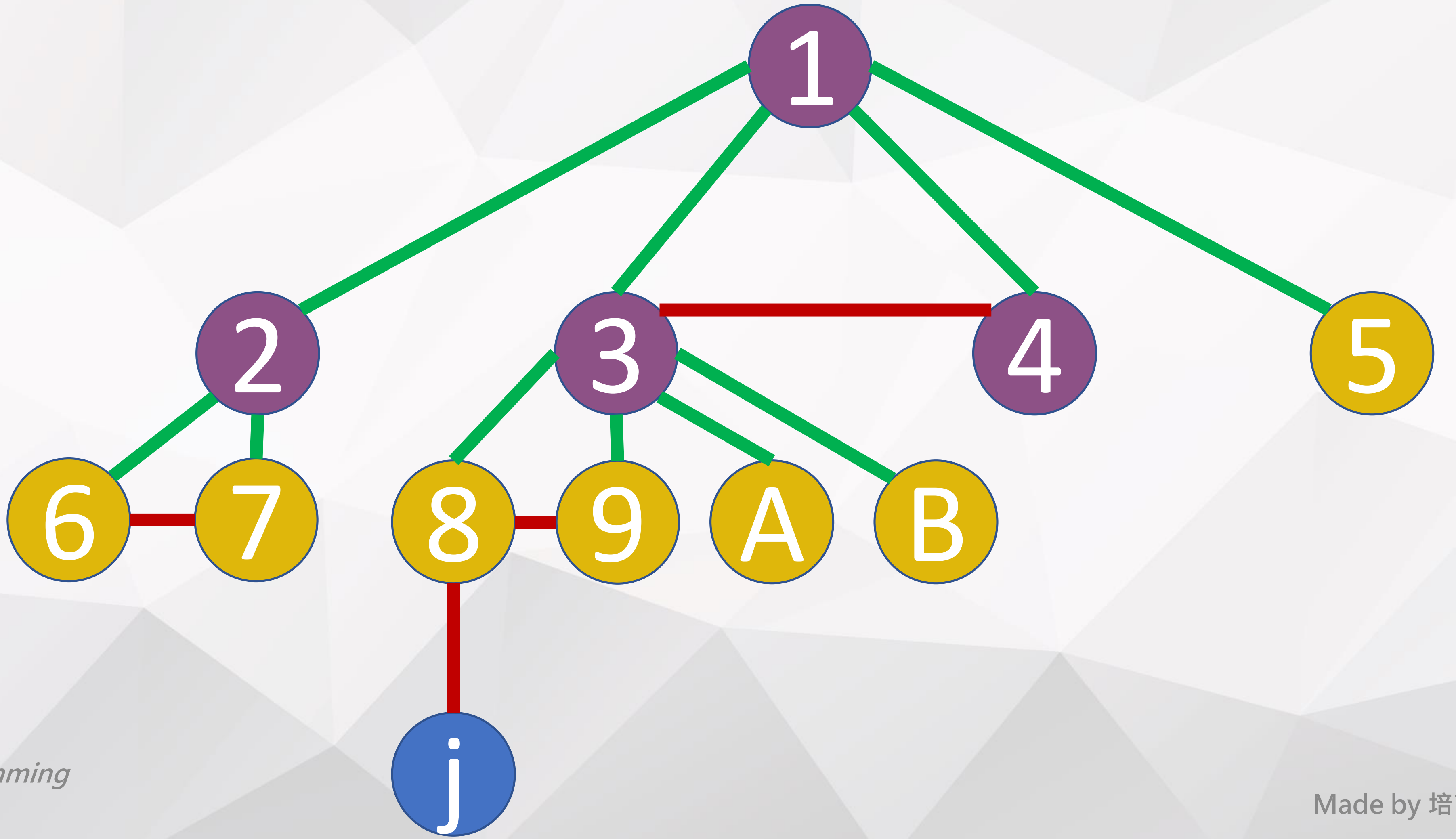
拜訪所有鄰點



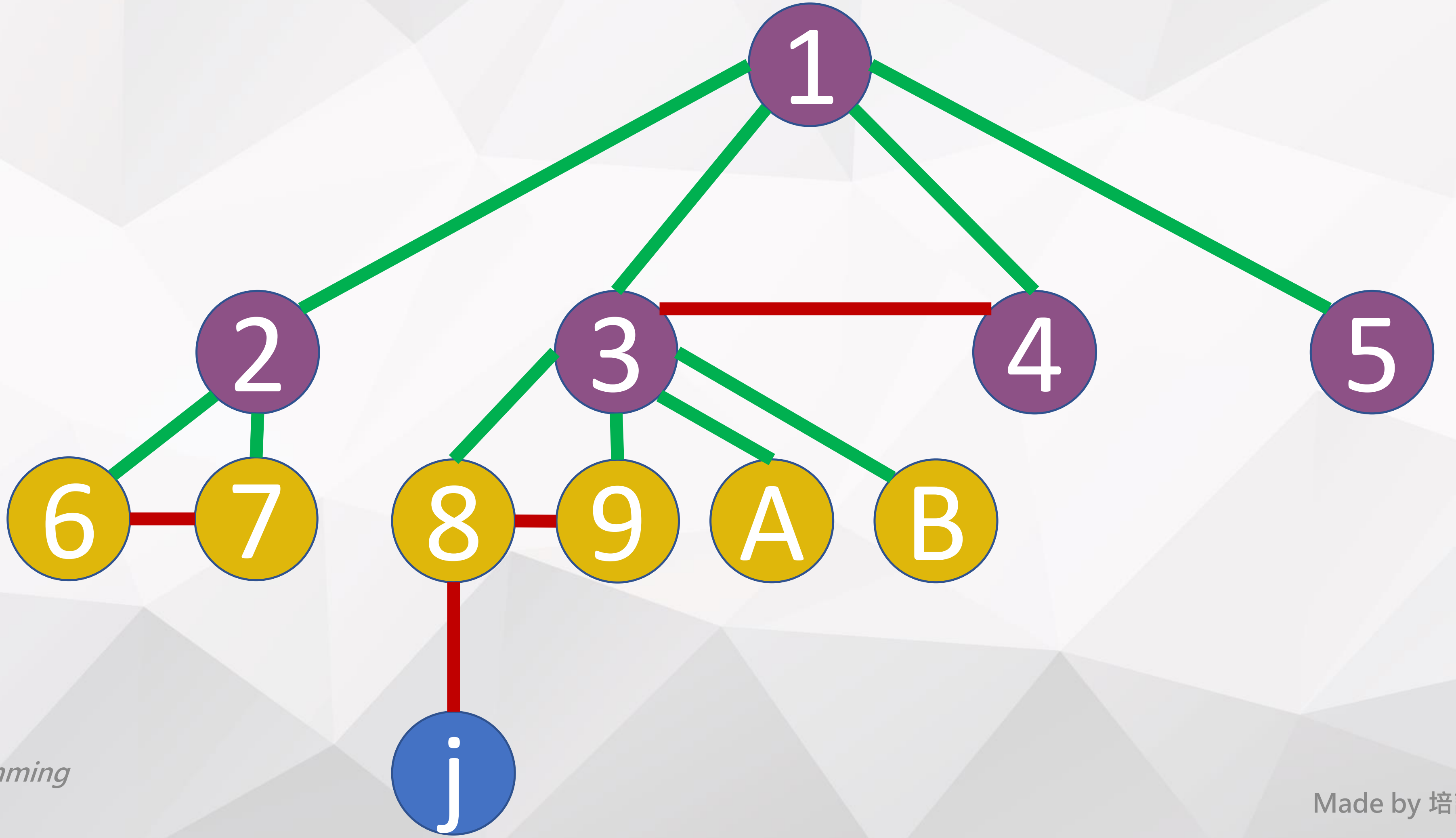
拜訪完



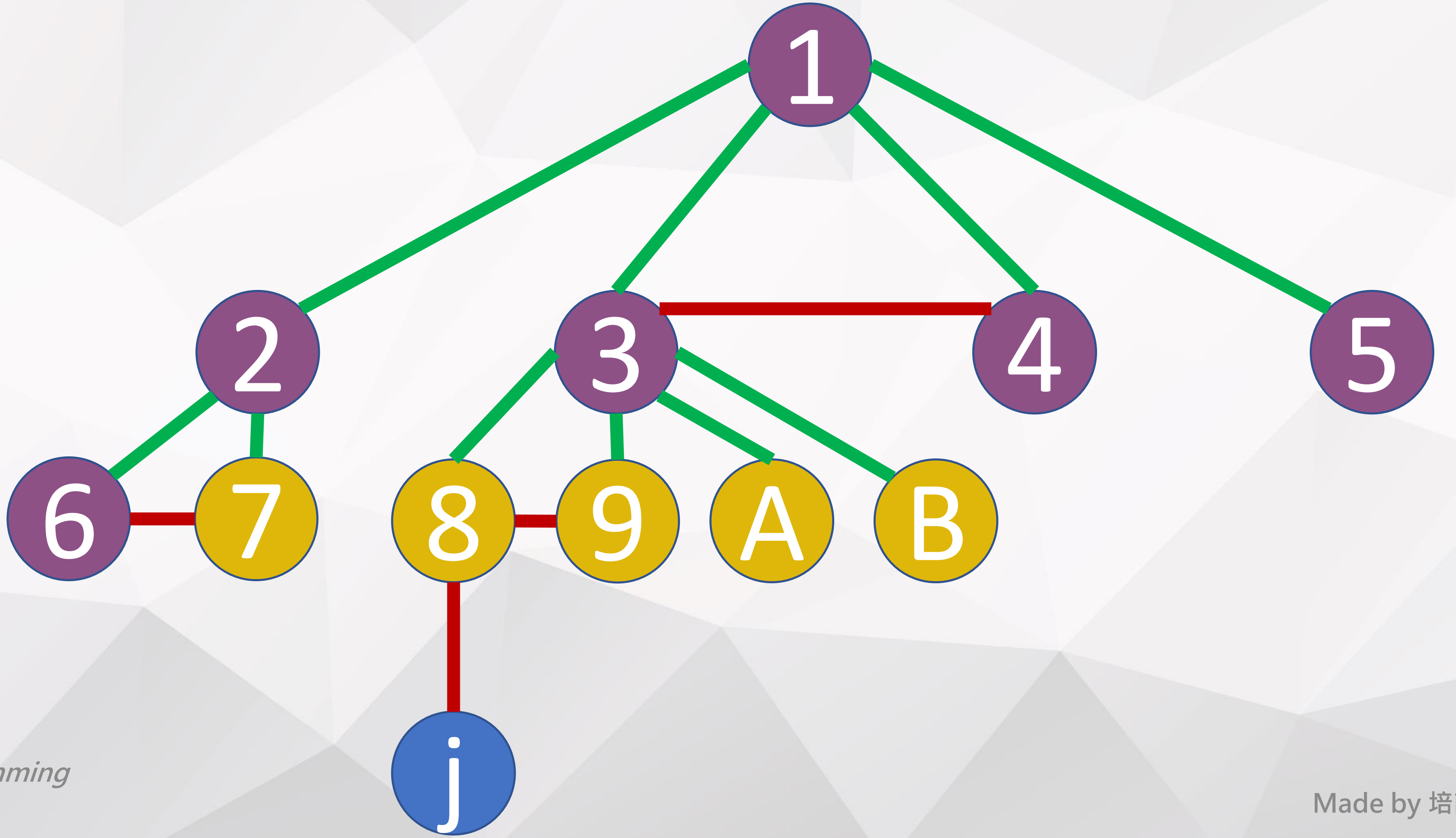
拜訪完



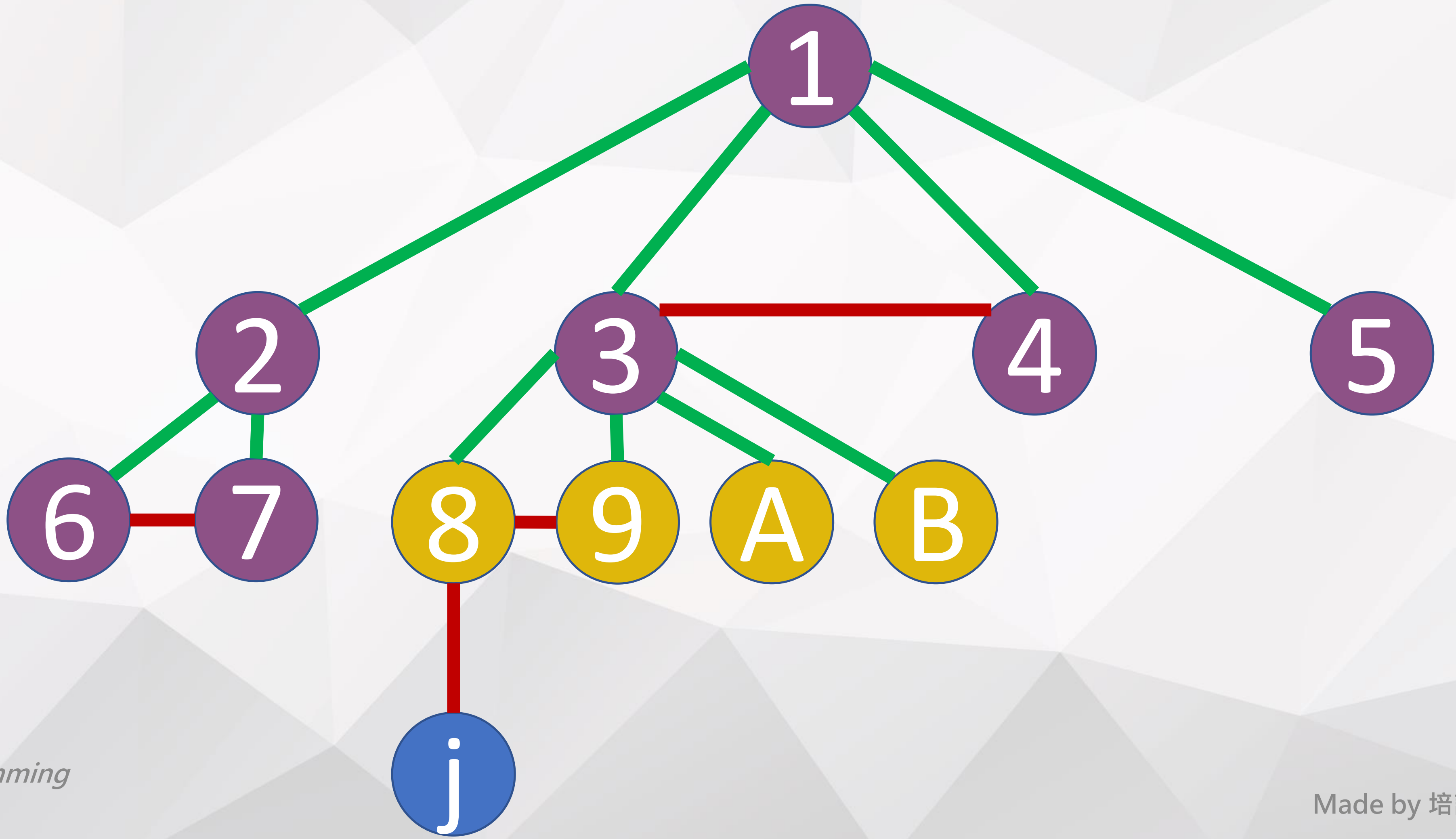
拜訪完



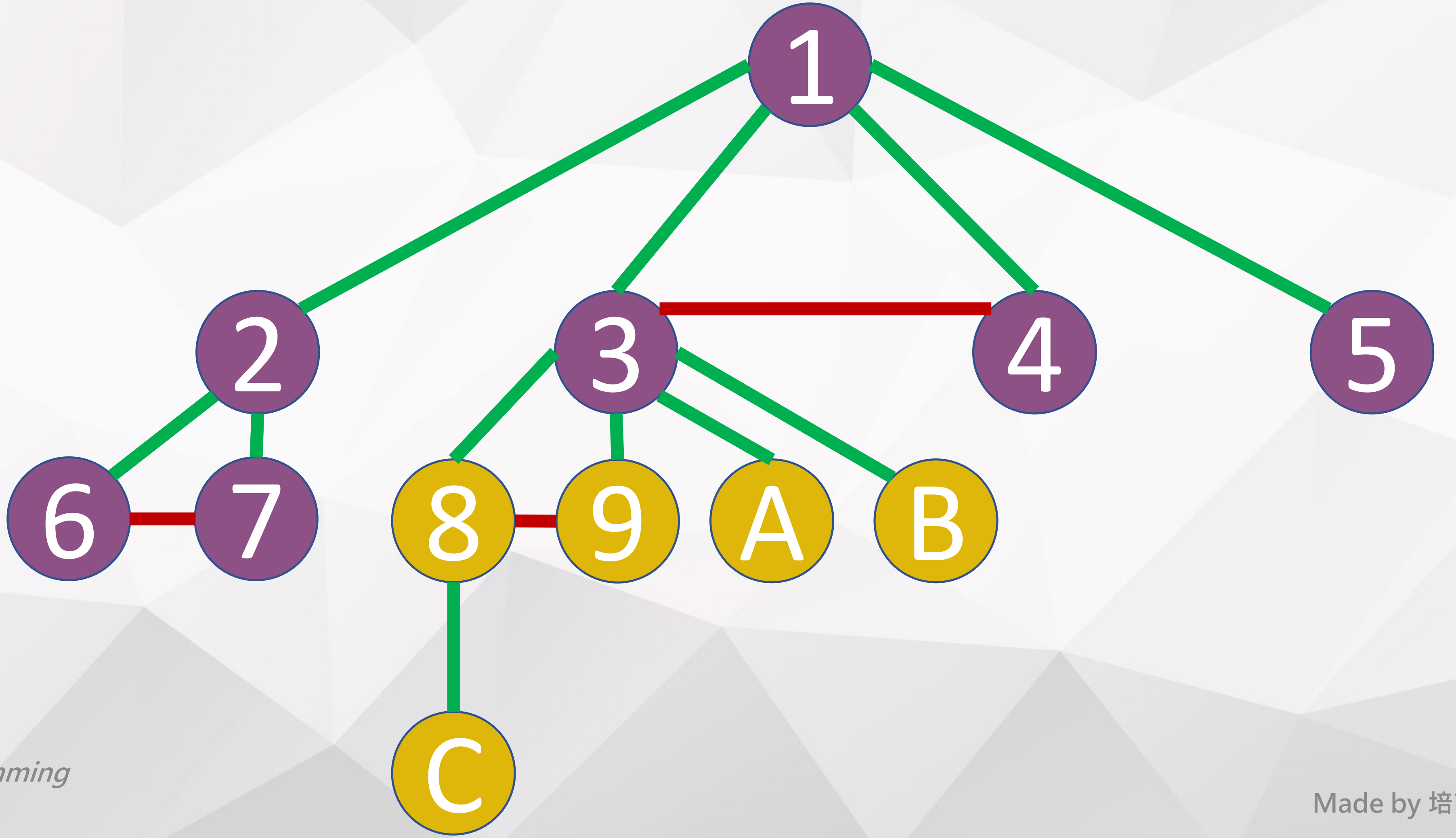
拜訪完



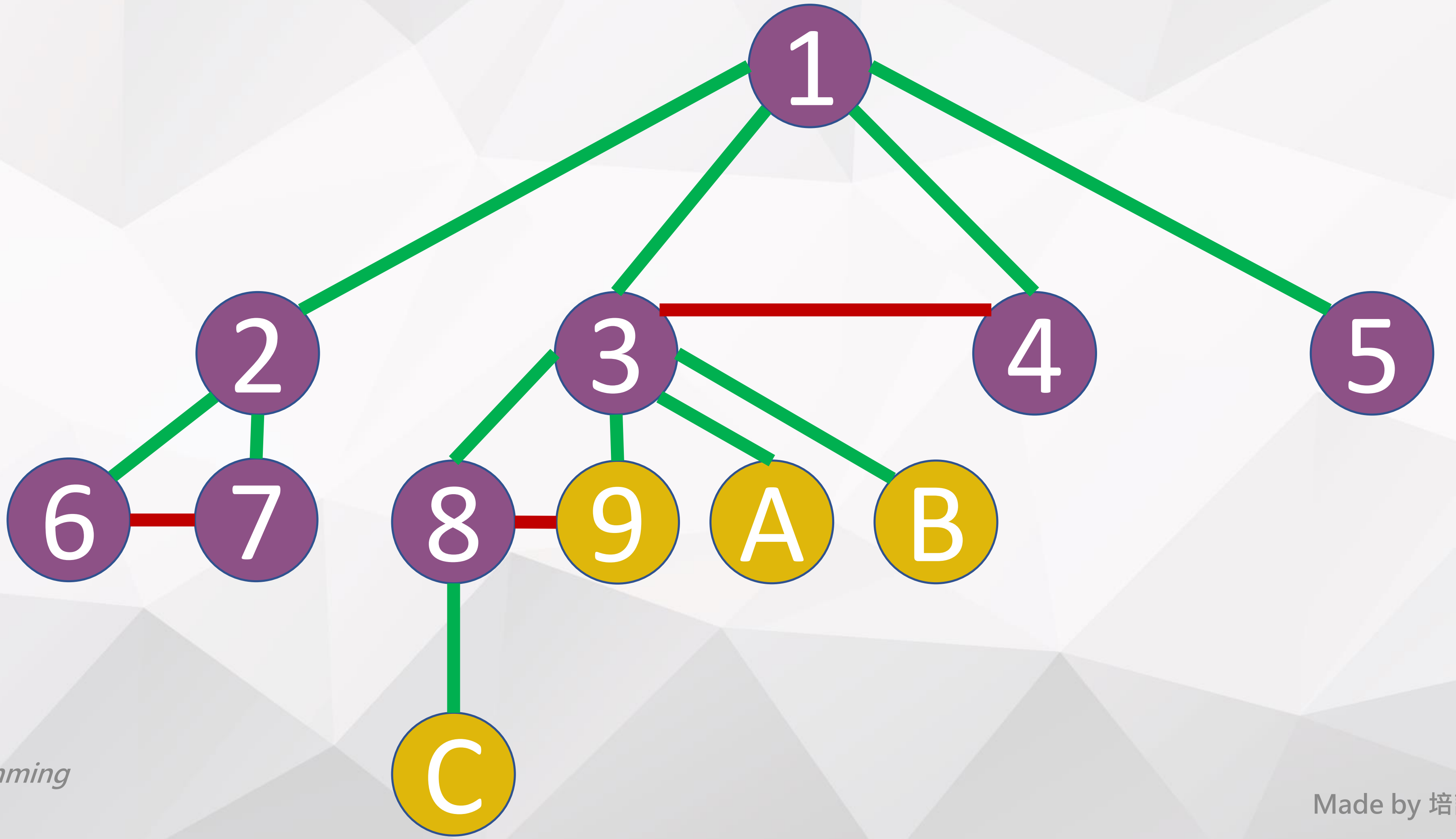
拜訪完



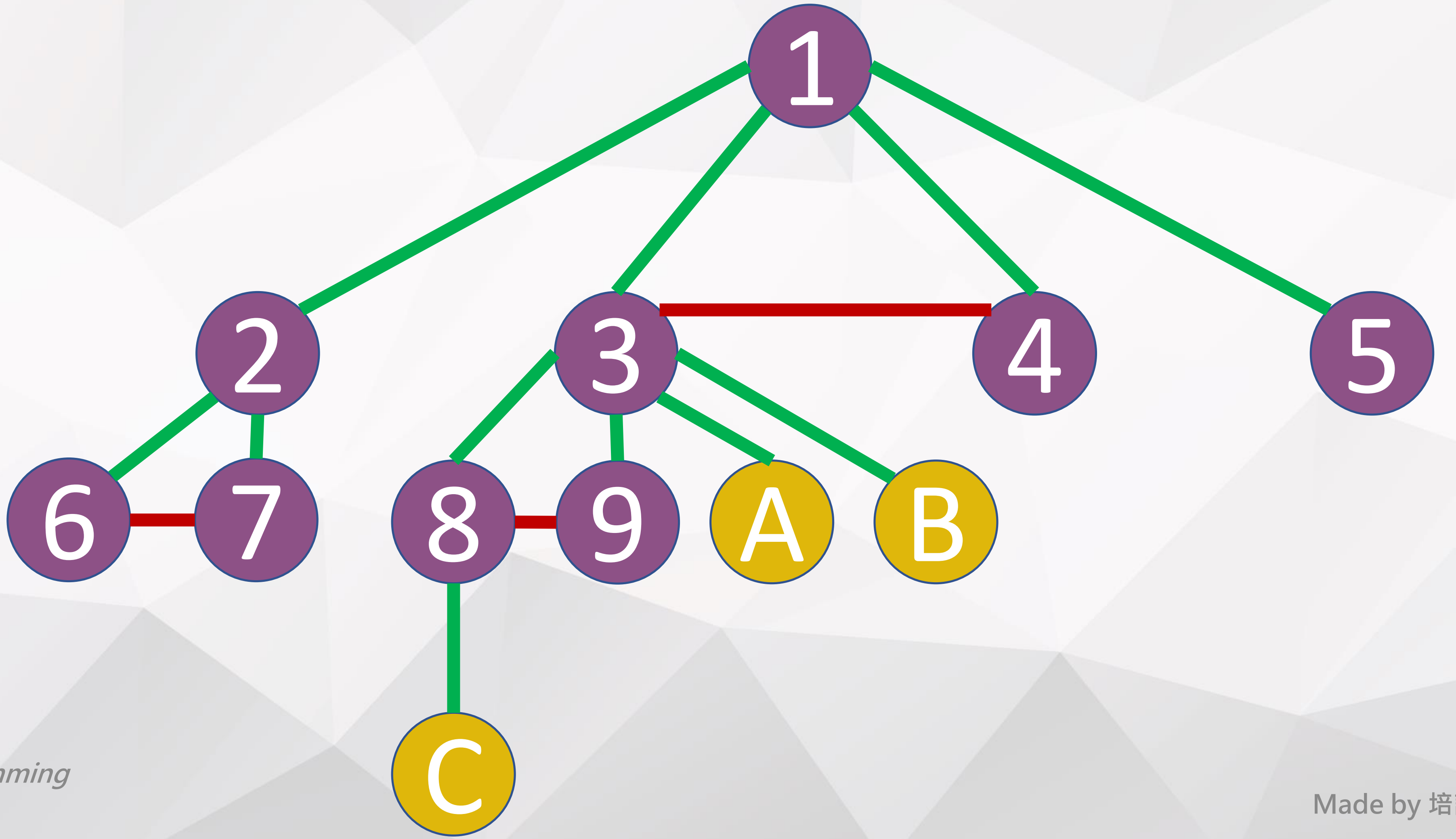
拜訪**所有**鄰點



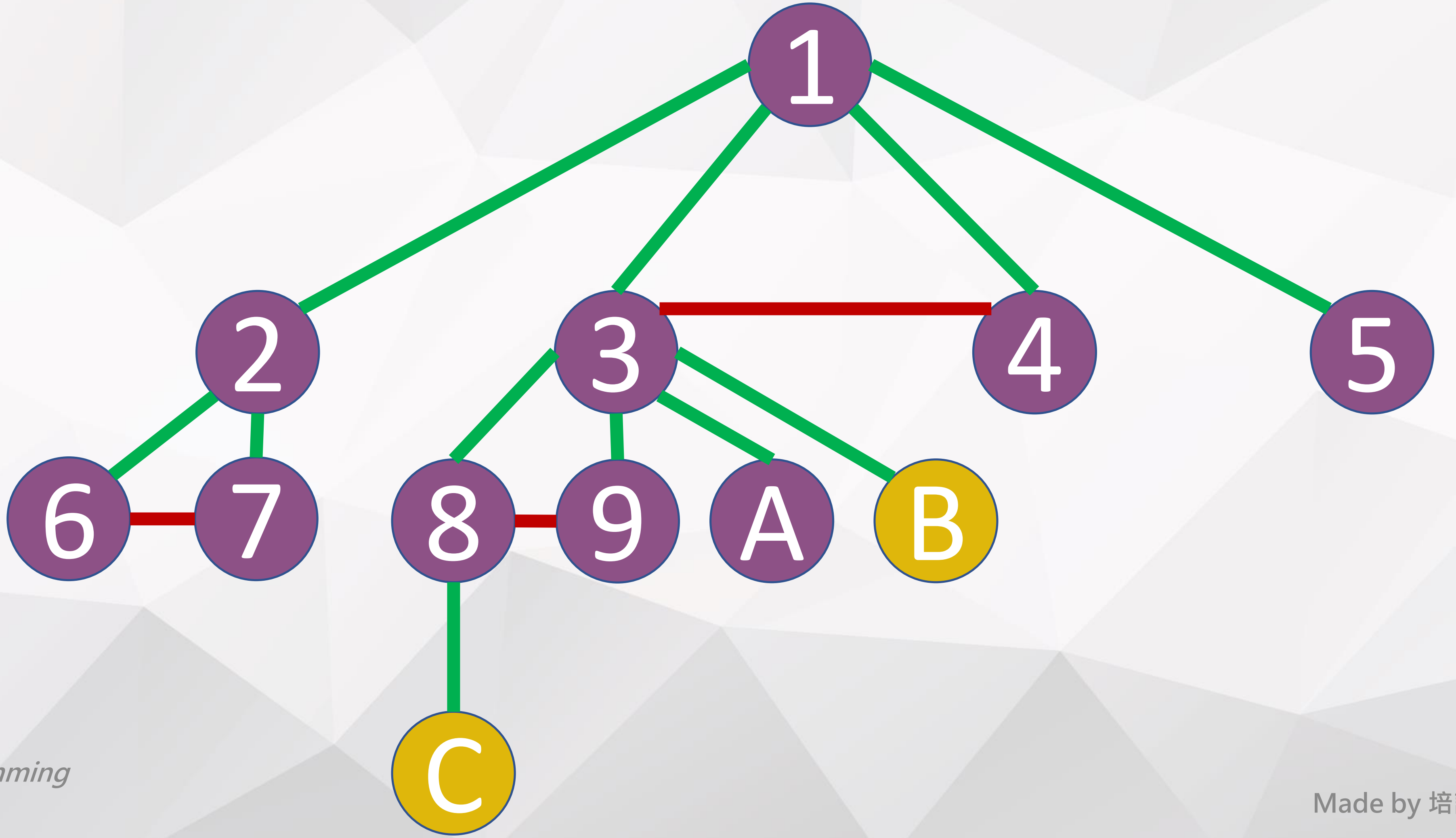
拜訪完



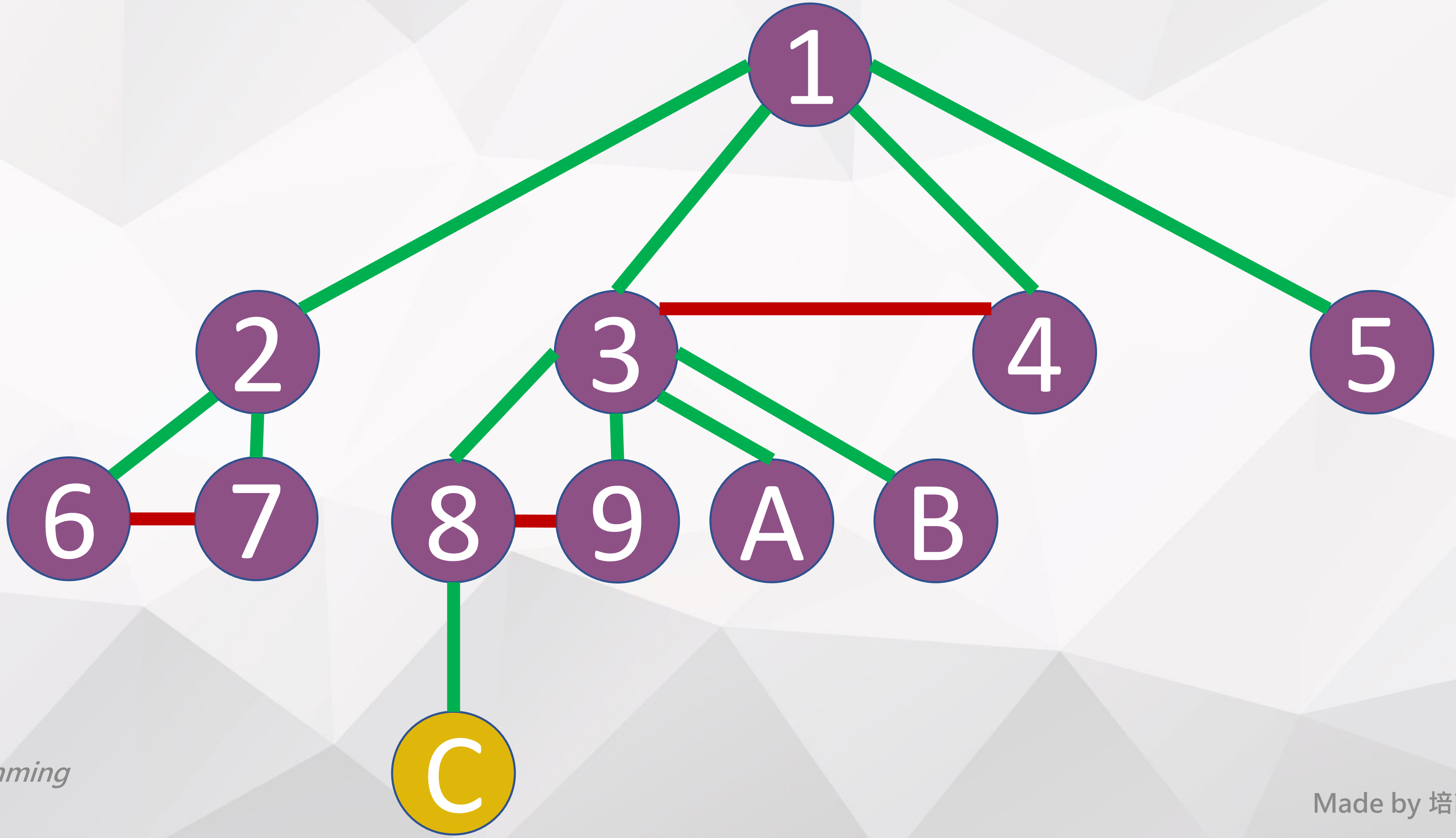
拜訪完



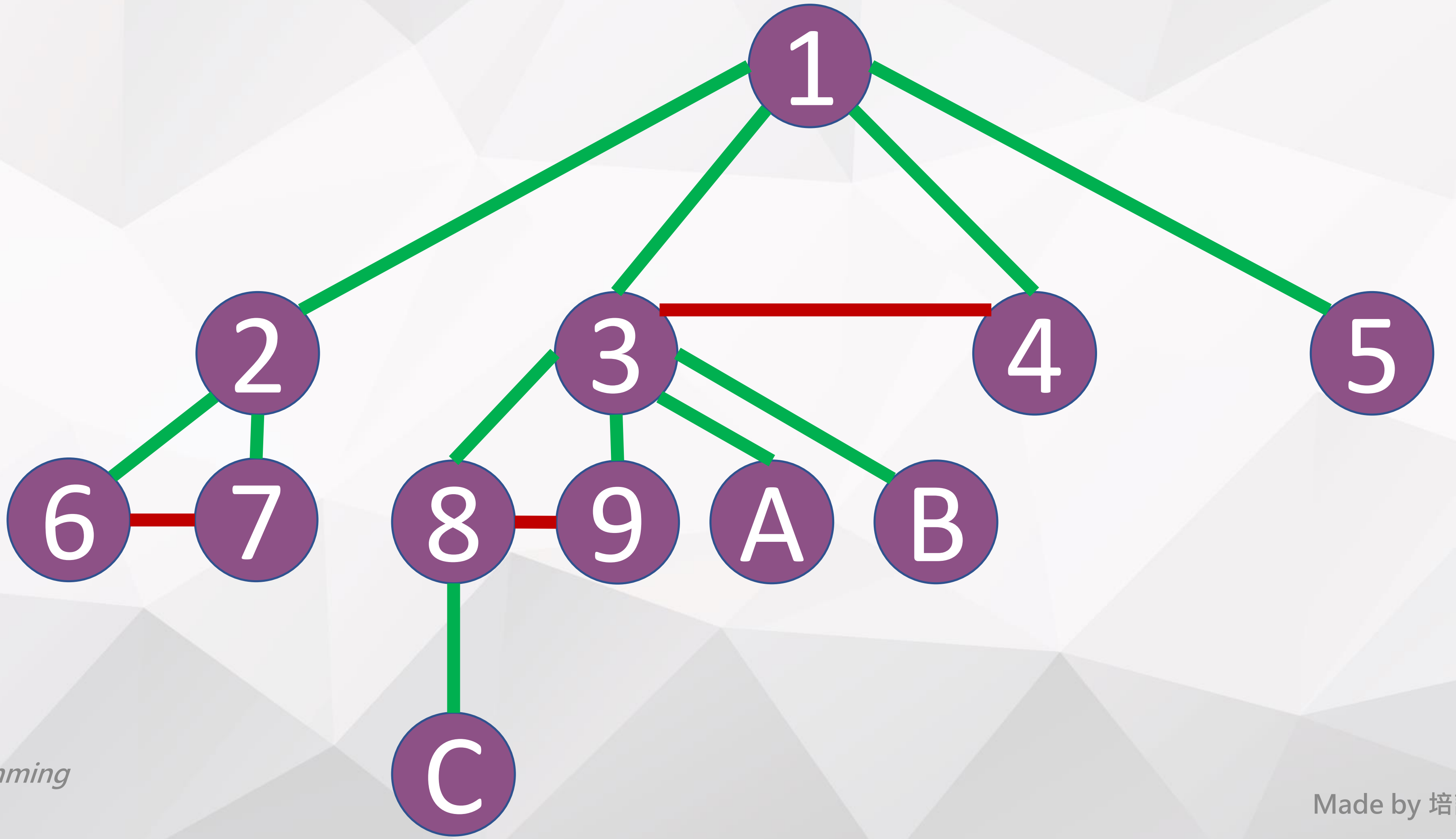
拜訪完



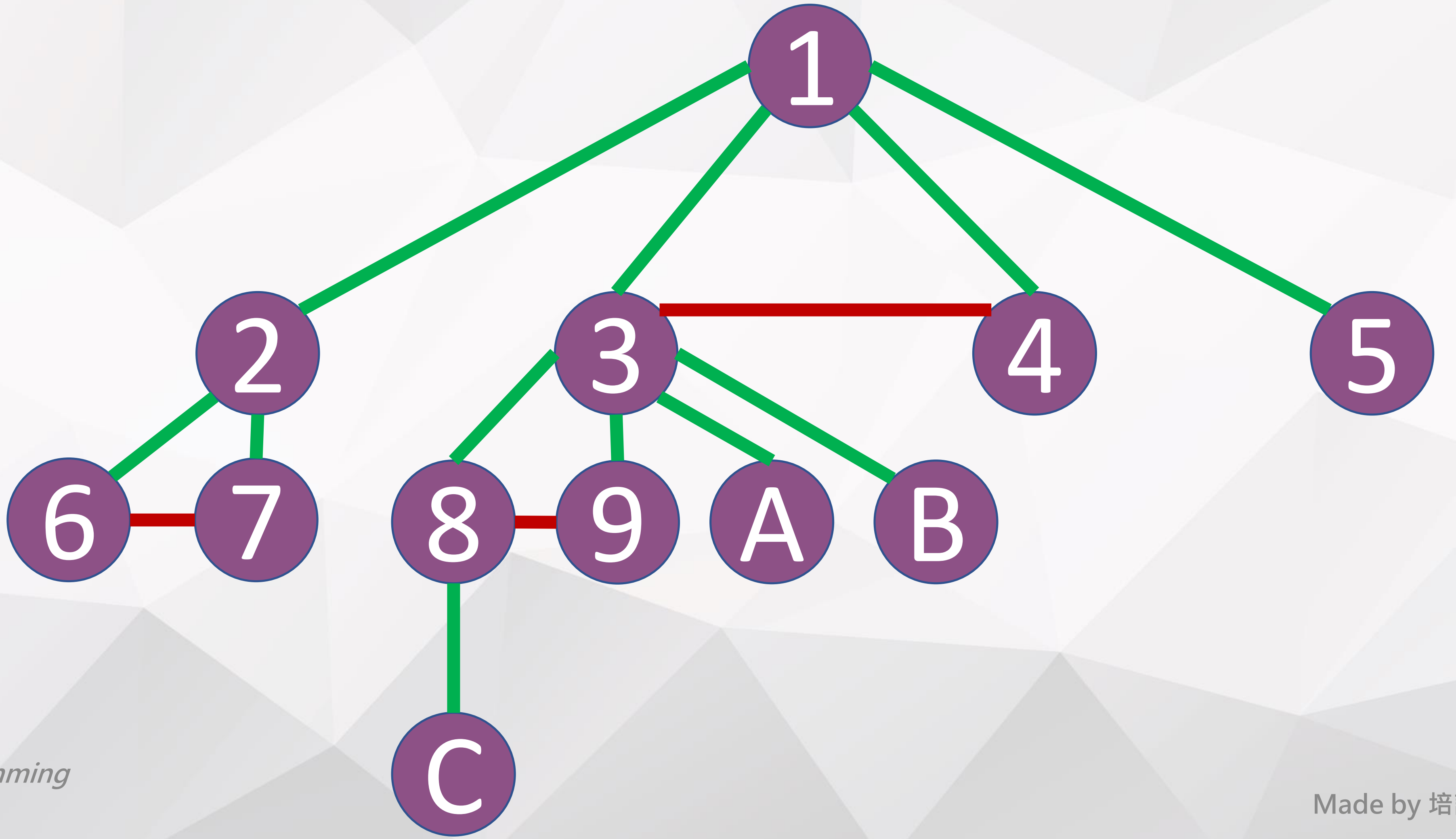
拜訪完



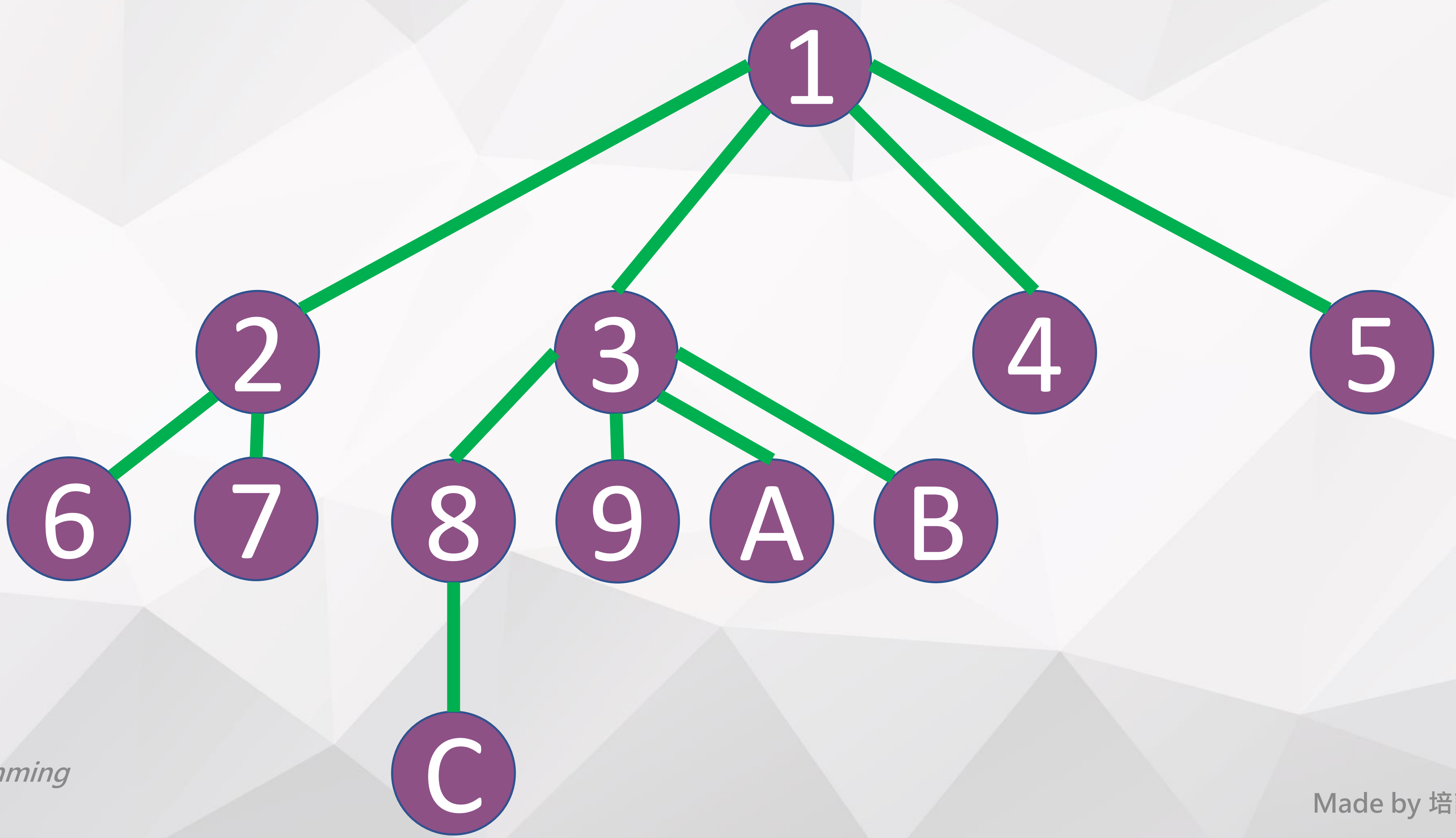
拜訪完



BFS 樹

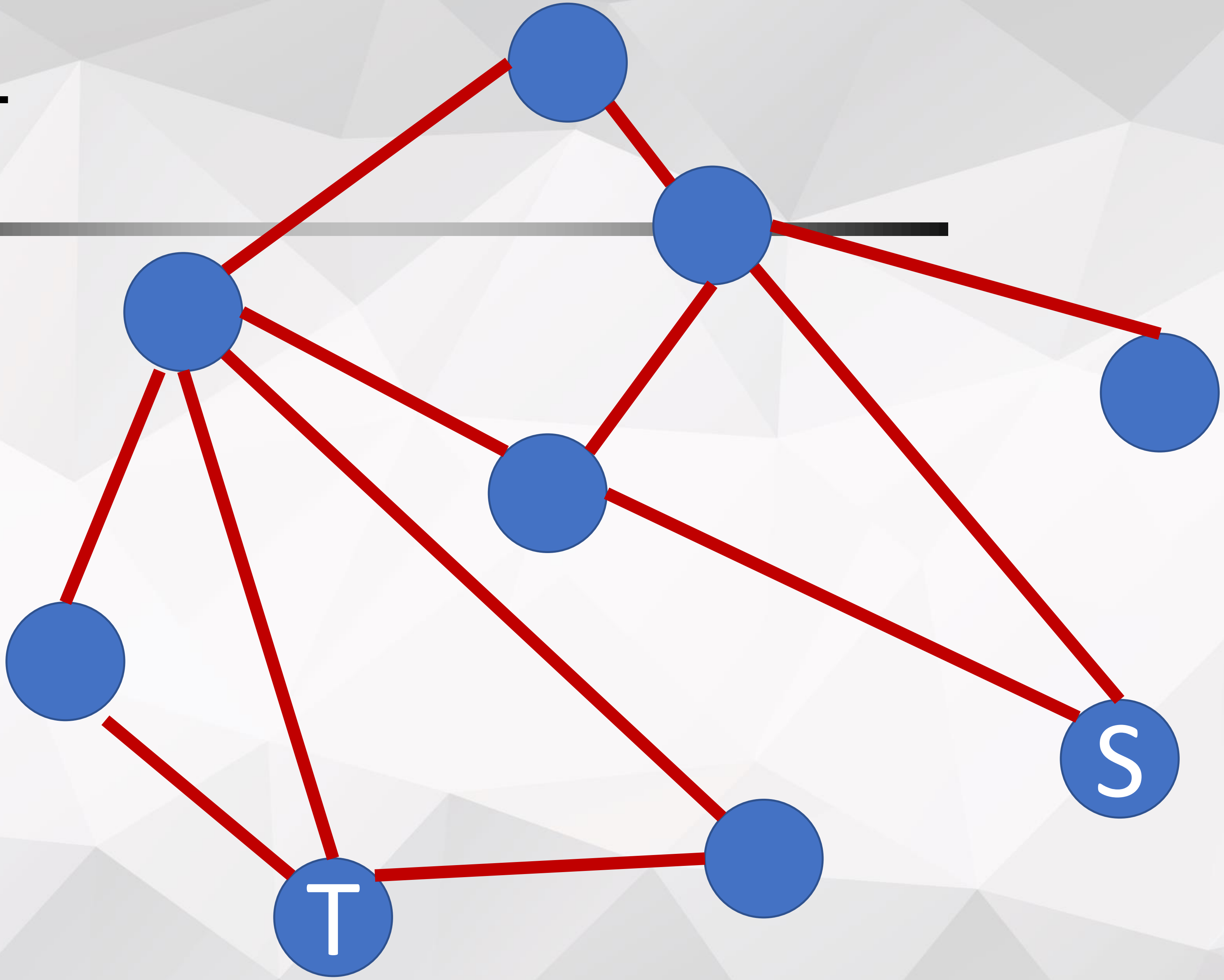


BFS 樹



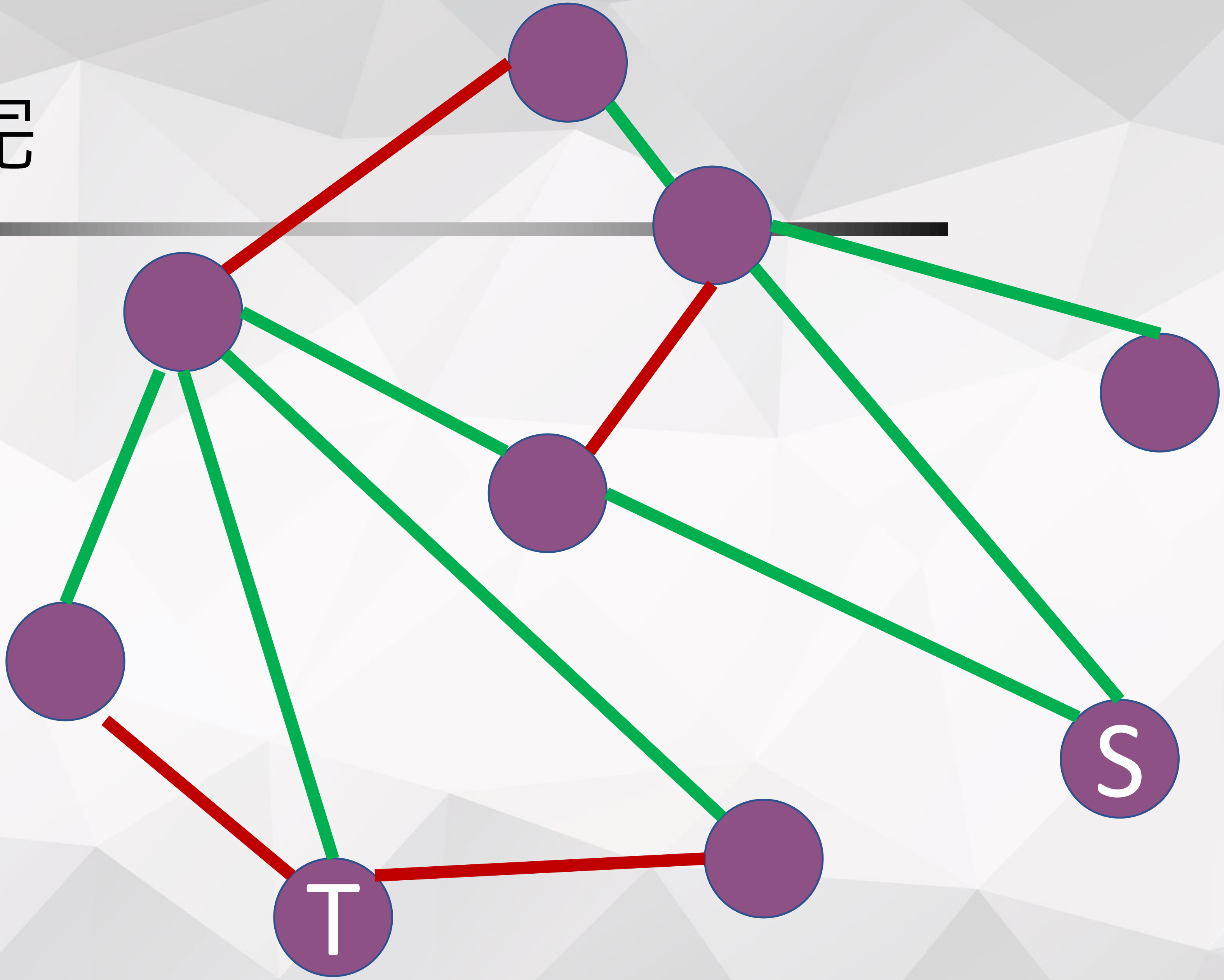
S 到 T

0



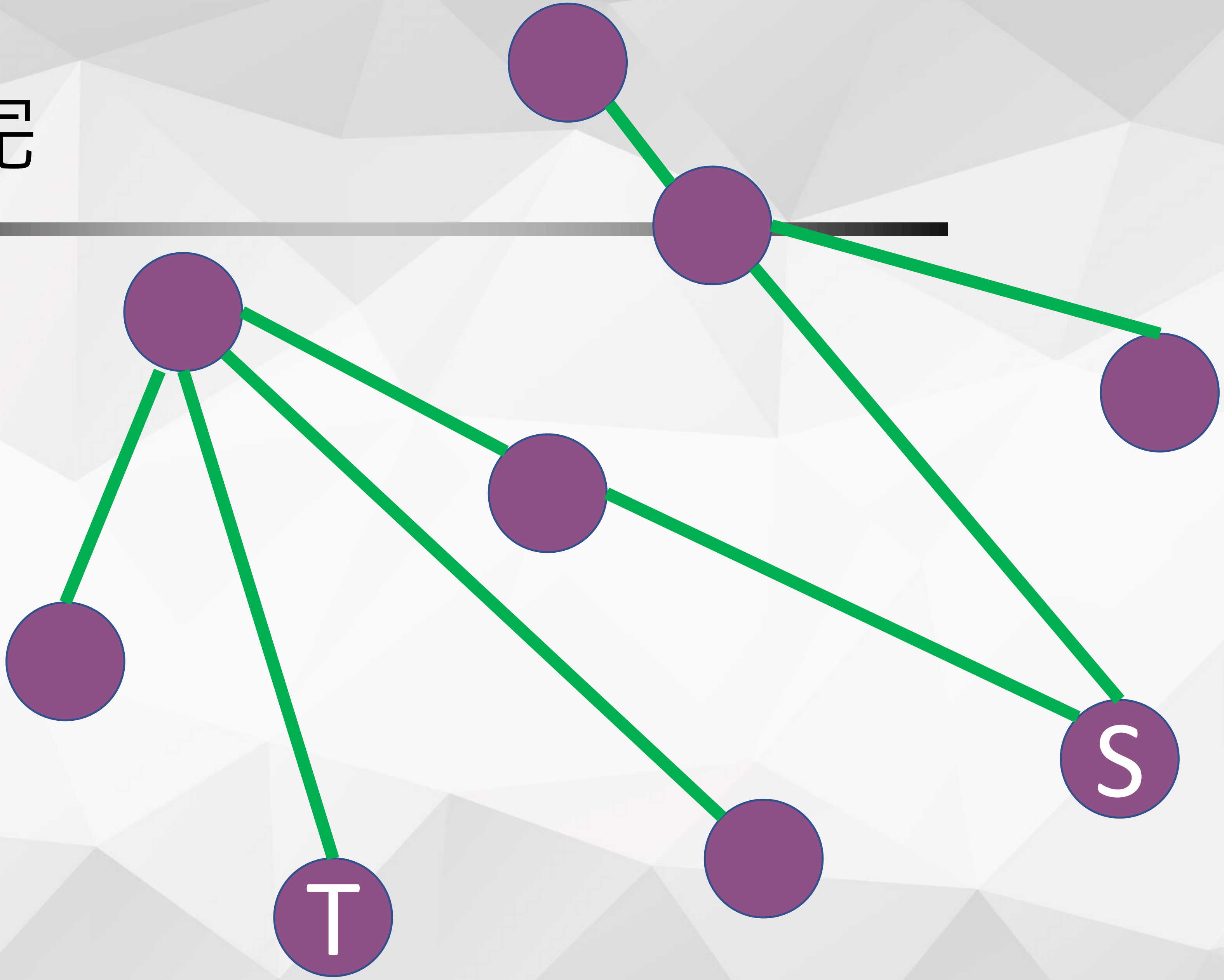
BFS 完

0



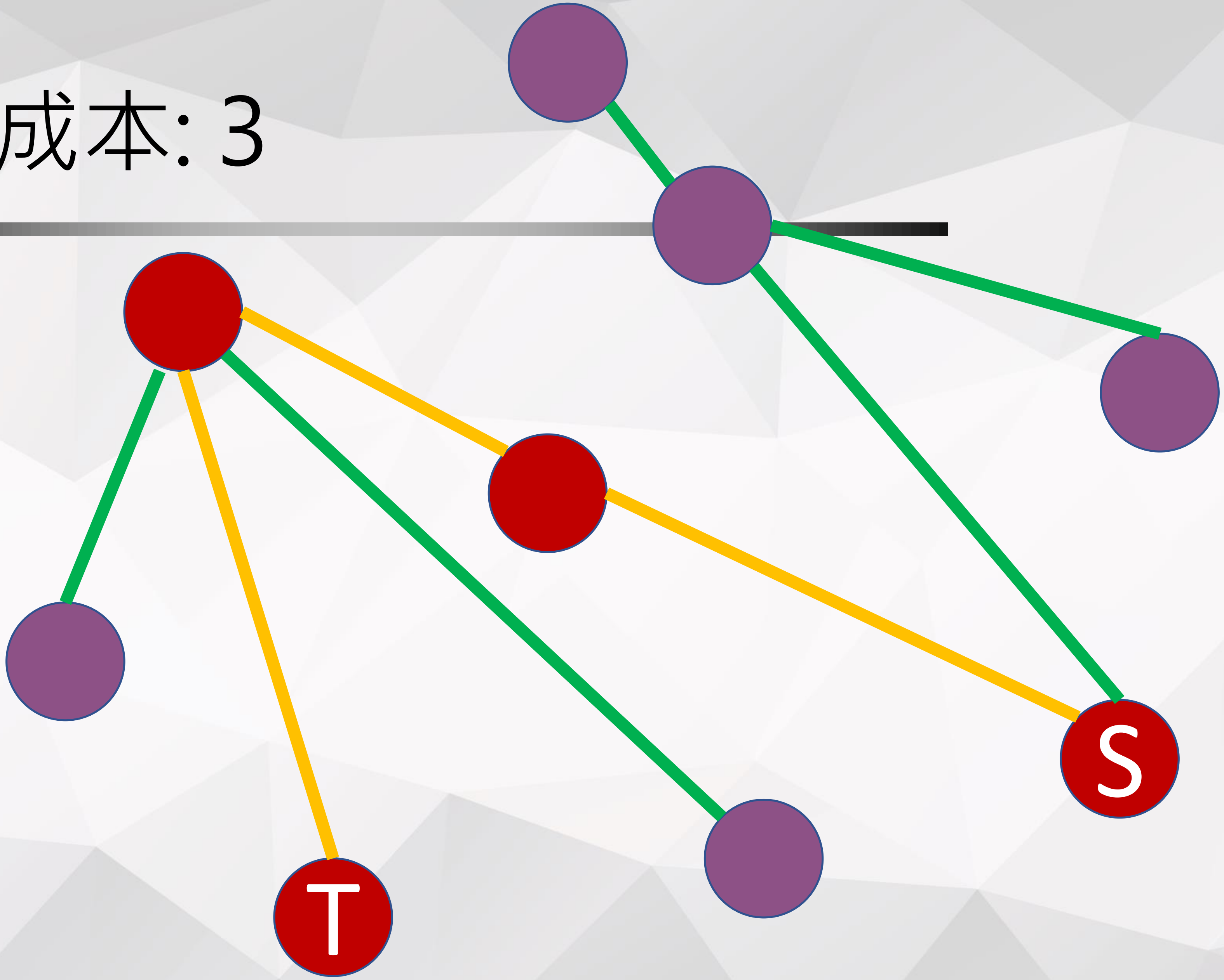
BFS 完

0

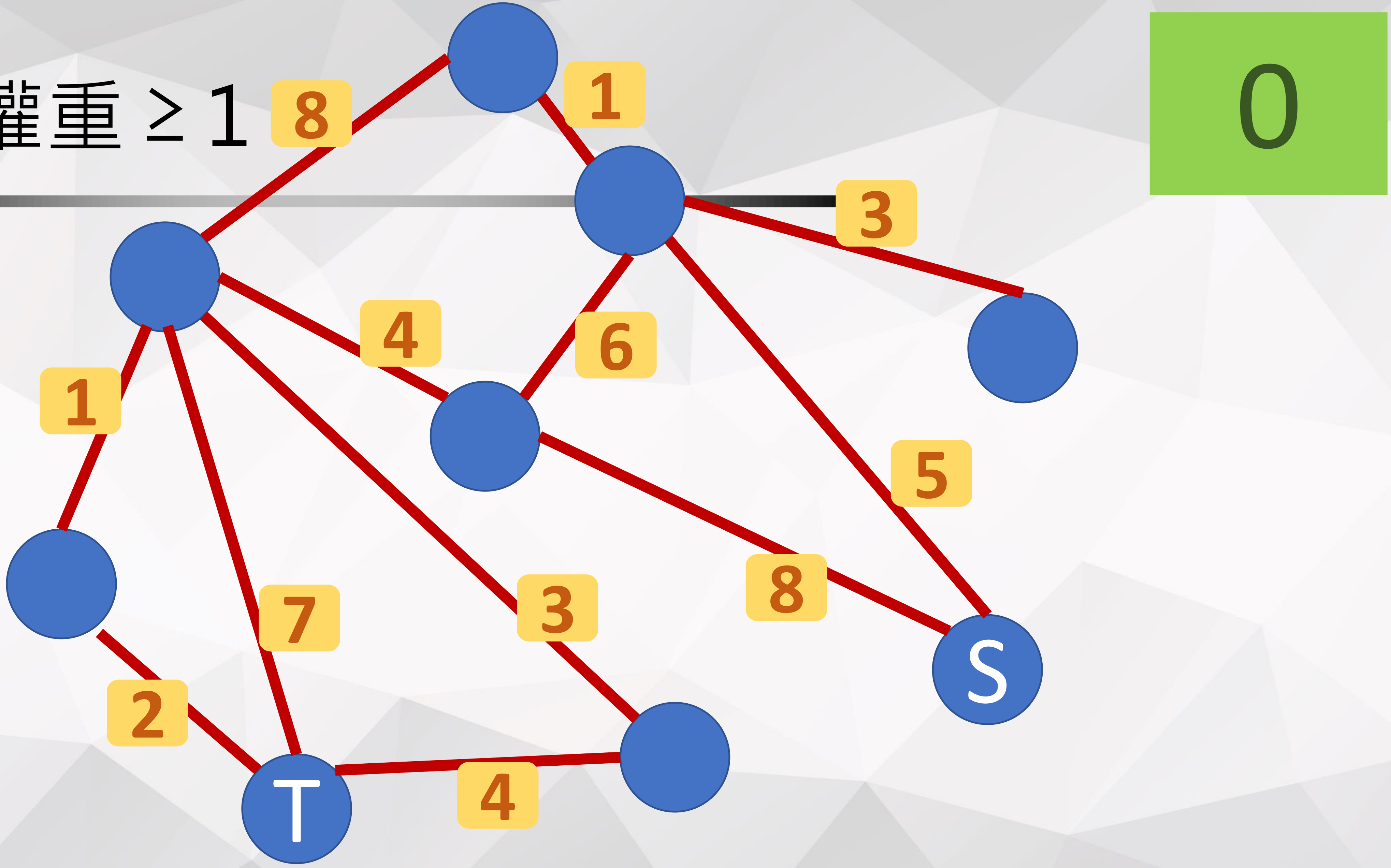


深度/成本: 3

3



若邊權重 ≥ 1



若邊權重 ≥ 1

怎麼辦?

將權重切段

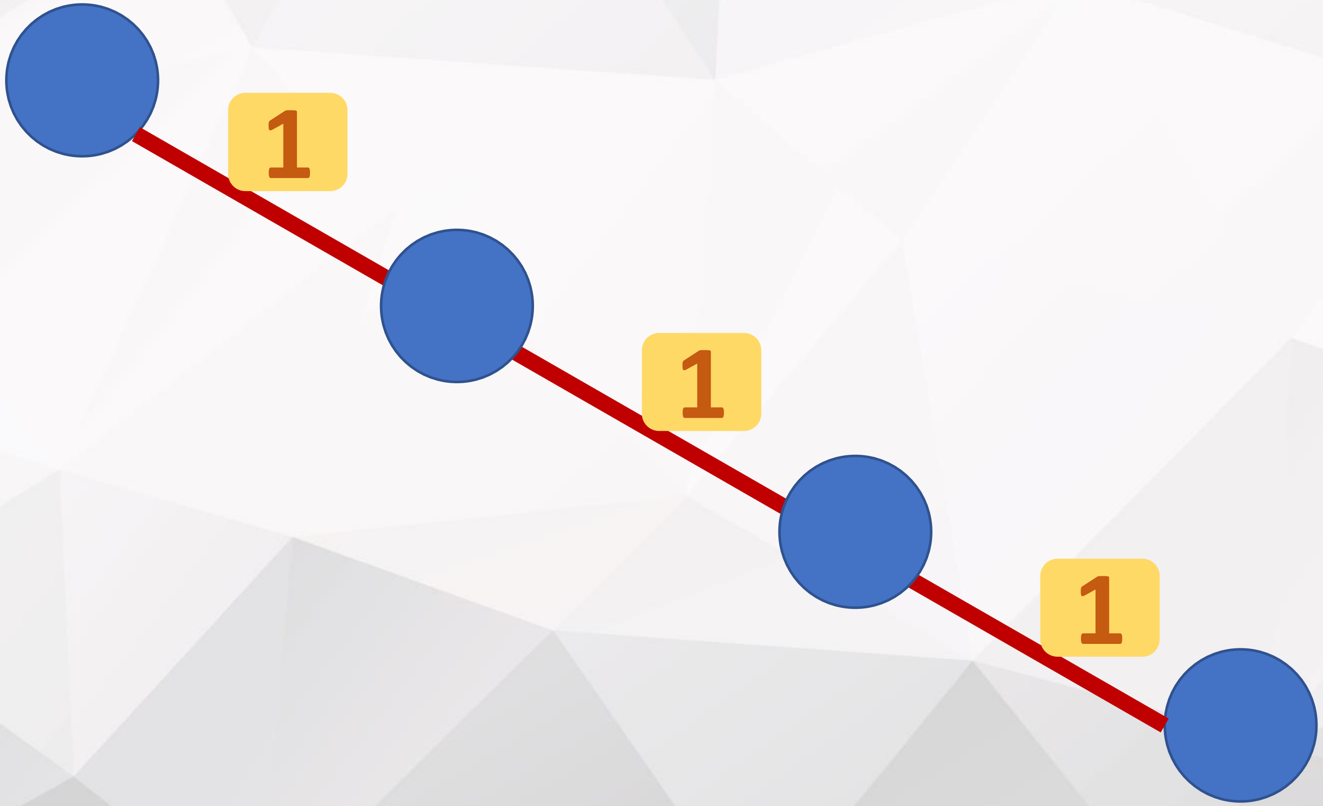
例如 x 權重，切成共 x 段的 1 權重邊

將權重切段

例如 3 權重

將權重切段

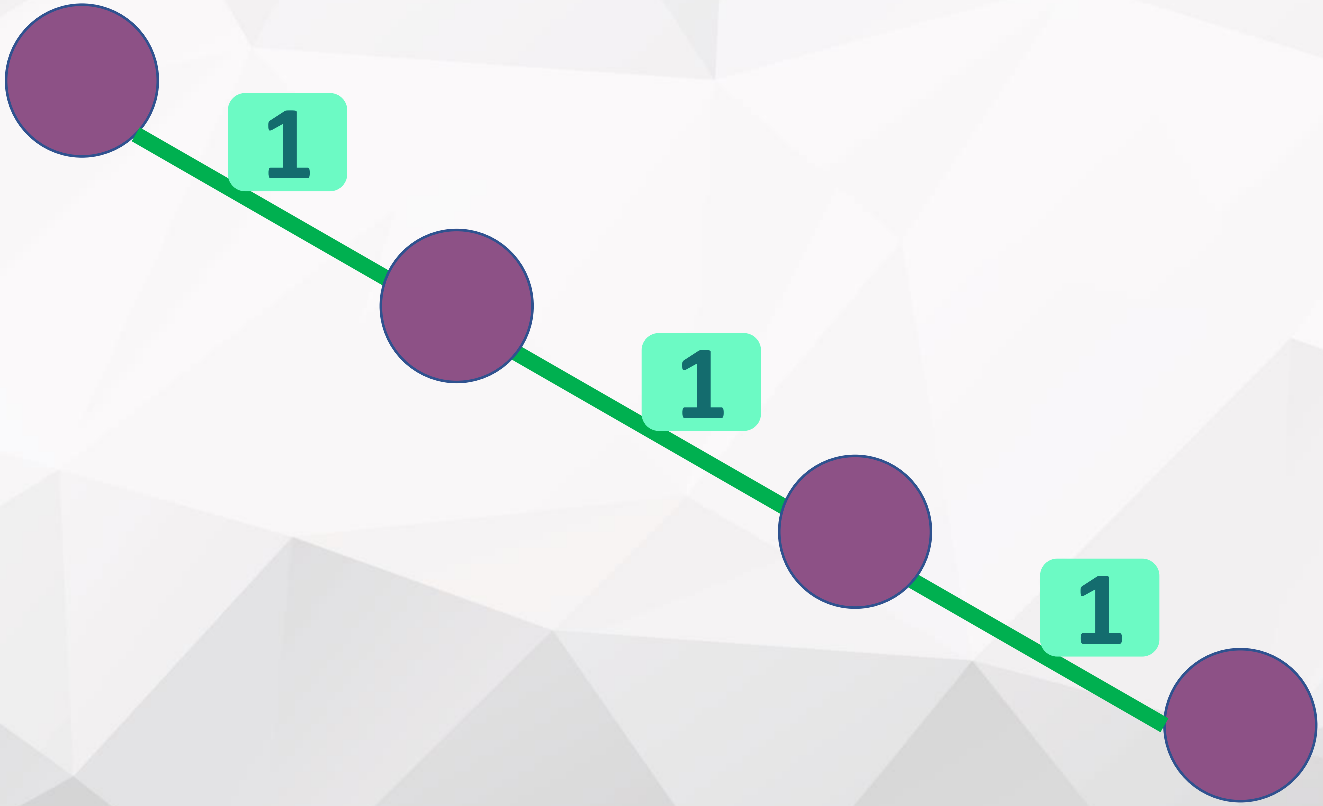
例如 3 權重，切成共 3 段的 1 權重邊



將權重切段

例如 3 權重，切成共 3 段的 1 權重邊

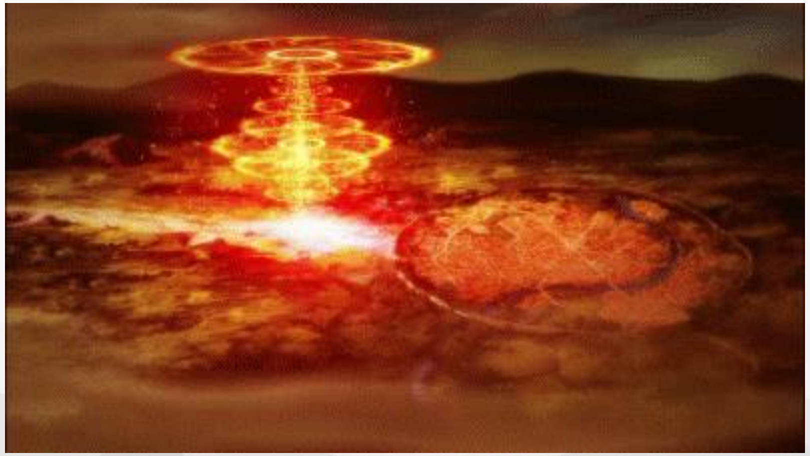
就能 BFS 了!



將權重切段

例如 3 權重，切成共 3 段的 1 權重邊就能 BFS 了!

複雜度肯定會爆炸!



Questions?

單源最短路徑

- Relaxation
- Dijkstra's algorithm
- Bellman-Ford's algorithm

單源最短路徑

- Relaxation
- Dijkstra's algorithm
- Bellman-Ford's algorithm

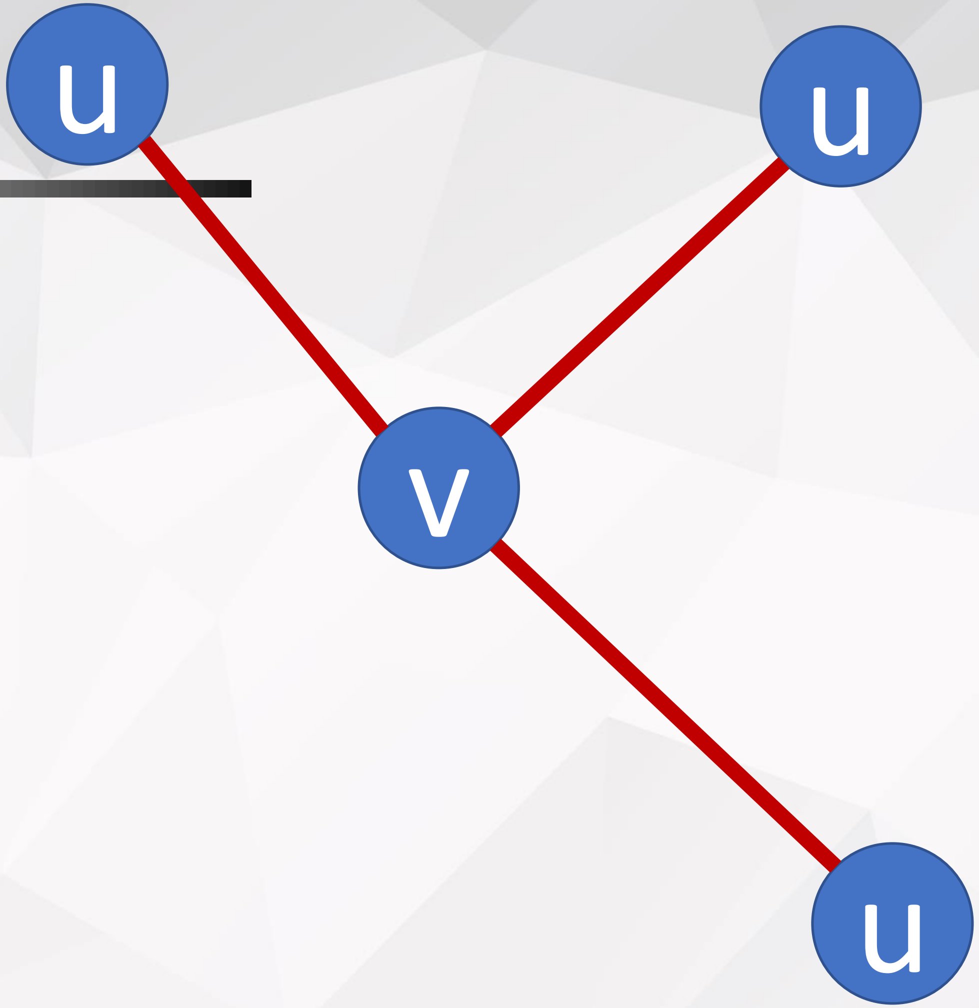
Relaxation

想像自己是圖中的某點 v

Relaxation

想像自己是圖中的某點 v

身旁有一些鄰點 u

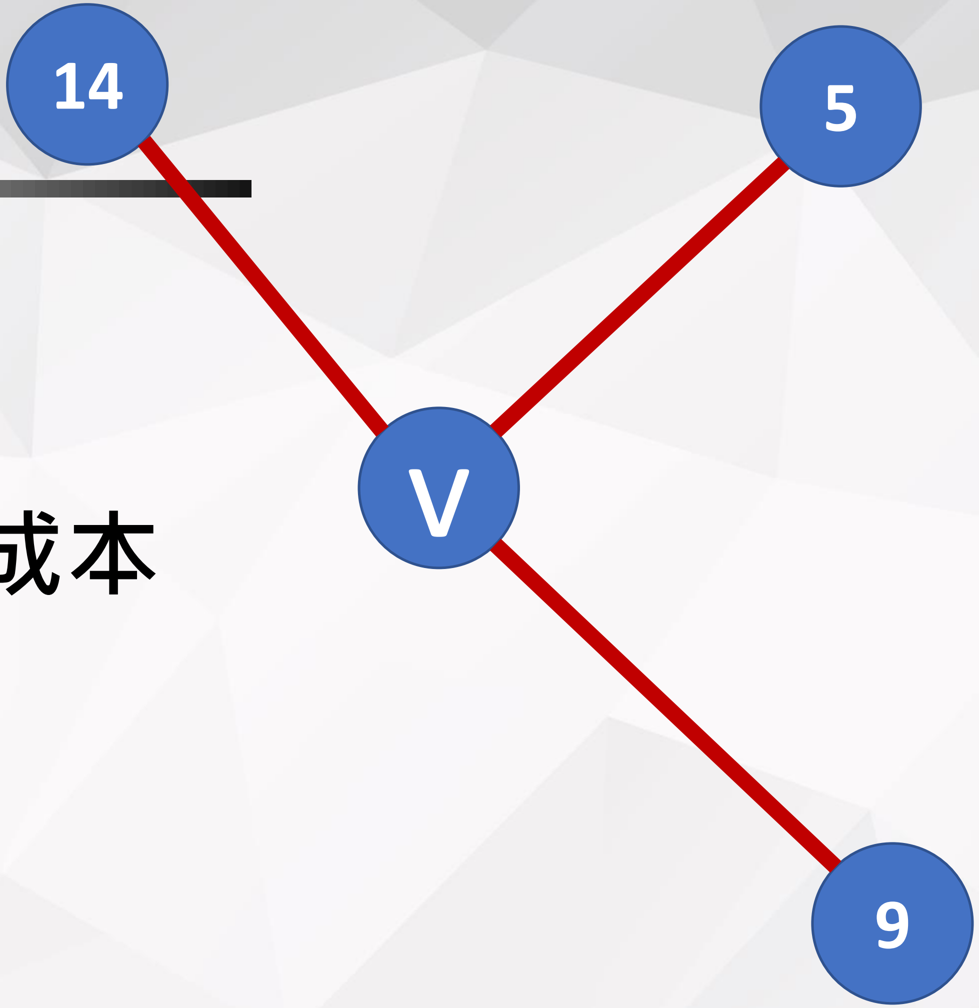


Relaxation

想像自己是圖中的某點 v

身旁有一些鄰點 u

u 知道源點到他們那裡的**最小成本**



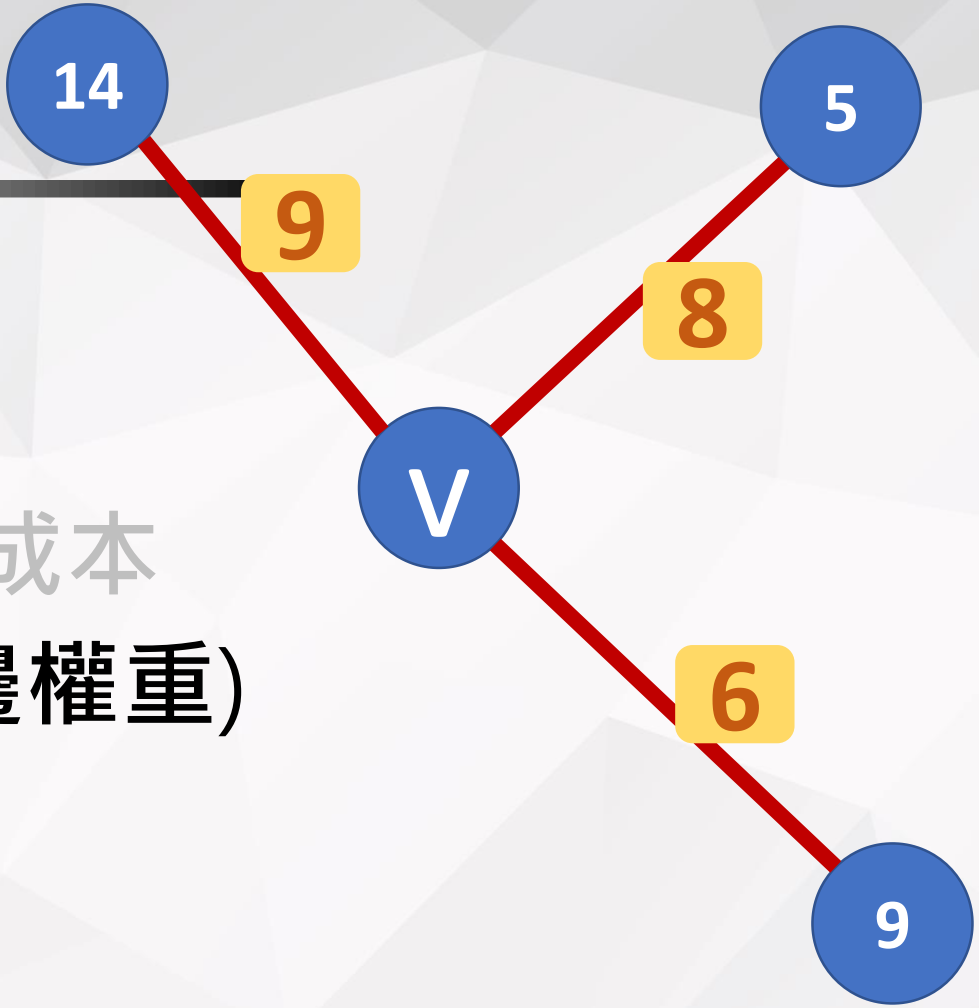
Relaxation

想像自己是圖中的某點 v

身旁有一些鄰點 u

u 知道源點到他們那裡的**最小成本**

v 知道 u 到自己那裡的成本 (**邊權重**)



Relaxation

想像自己是圖中的某點 v

身旁有一些鄰點 u

u 知道源點到他們那裡的**最小成本**

v 知道 u 到自己那裡的成本 (**邊權重**)

v 能計算各點到 v 的成本

Relaxation

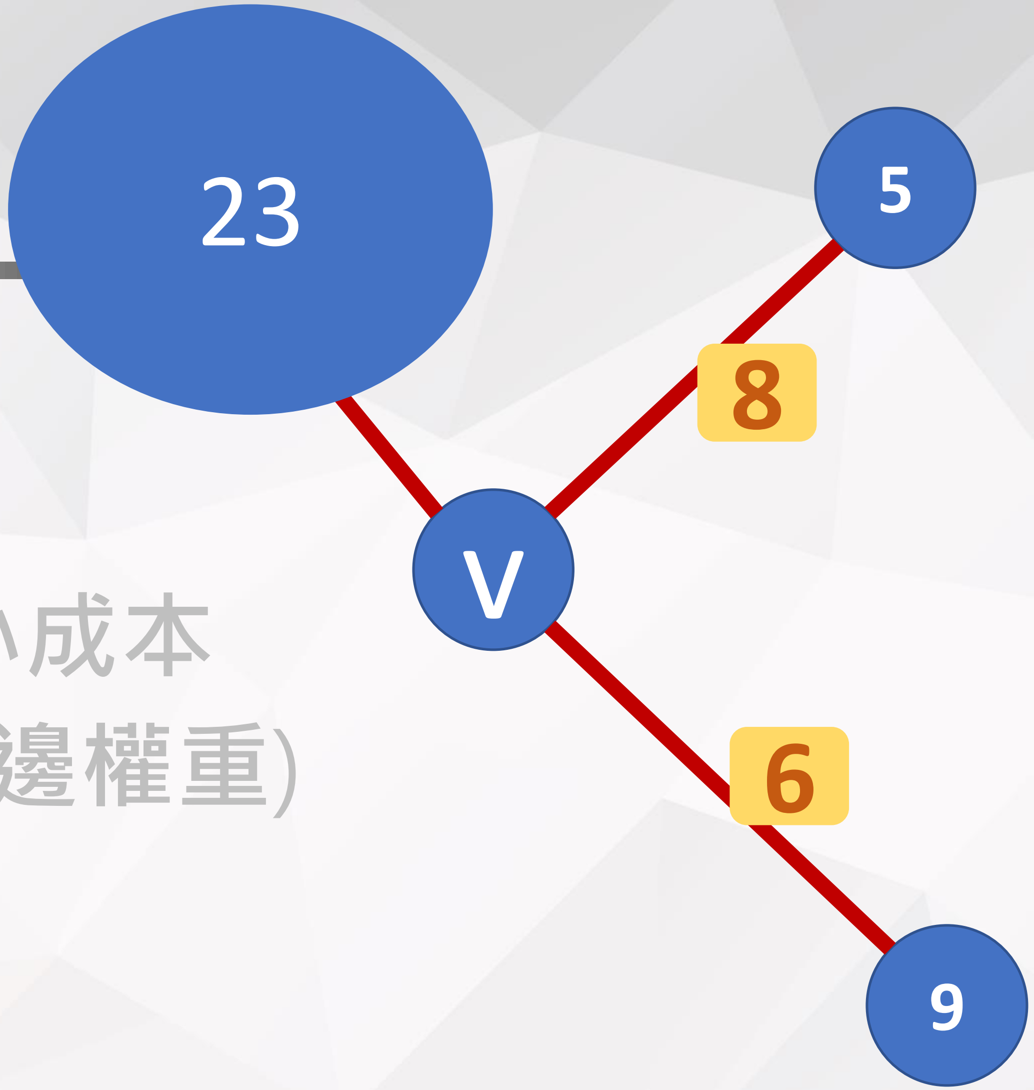
想像自己是圖中的某點 v

身旁有一些鄰點 u

u 知道源點到他們那裡的**最小成本**

v 知道 u 到自己那裡的成本 (**邊權重**)

v 能計算各點到 v 的成本



Relaxation

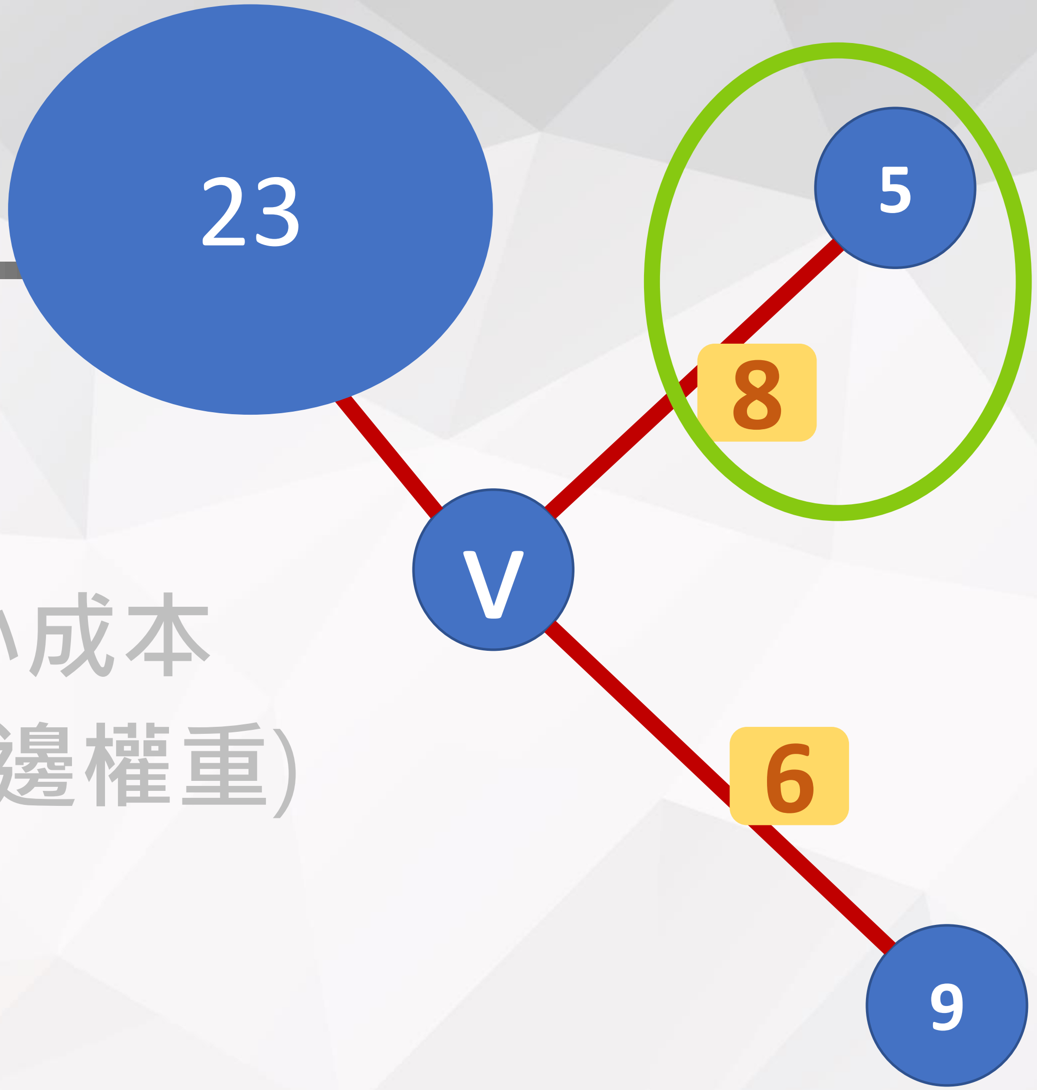
想像自己是圖中的某點 v

身旁有一些鄰點 u

u 知道源點到他們那裡的**最小成本**

v 知道 u 到自己那裡的成本 (**邊權重**)

v 能計算各點到 v 的成本



Relaxation

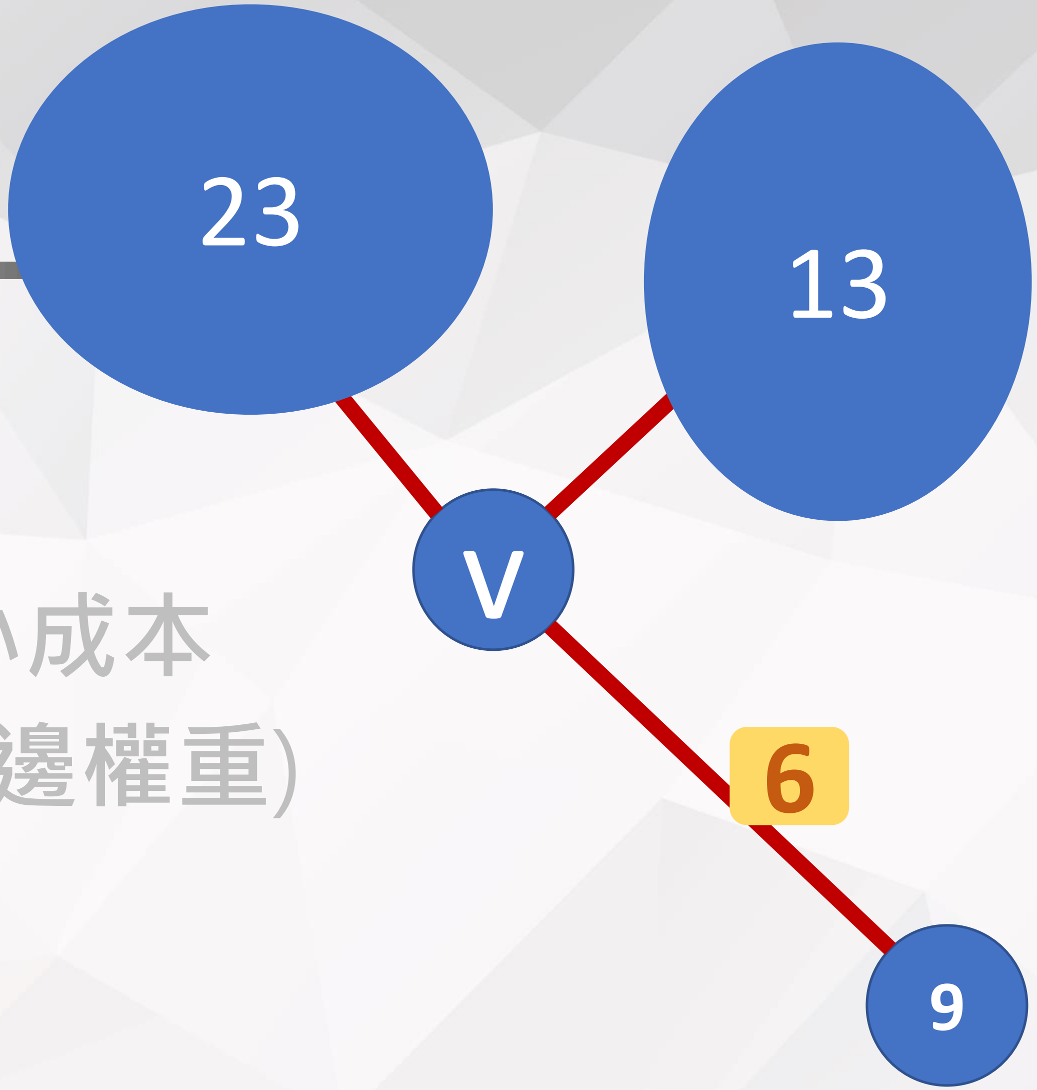
想像自己是圖中的某點 v

身旁有一些鄰點 u

u 知道源點到他們那裡的**最小成本**

v 知道 u 到自己那裡的成本 (**邊權重**)

v 能計算各點到 v 的成本



Relaxation

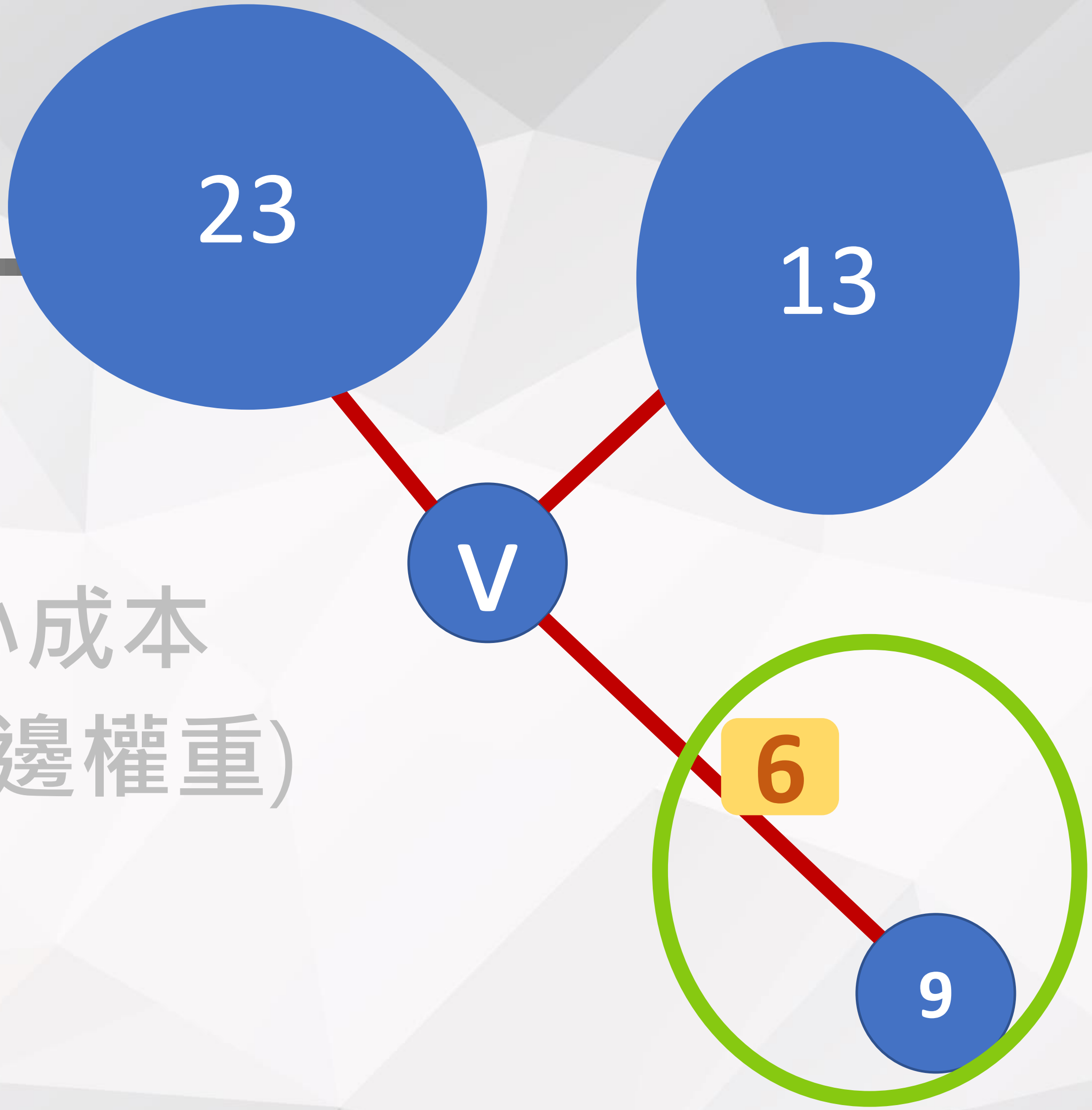
想像自己是圖中的某點 v

身旁有一些鄰點 u

u 知道源點到他們那裡的**最小成本**

v 知道 u 到自己那裡的成本 (**邊權重**)

v 能計算各點到 v 的成本



Relaxation

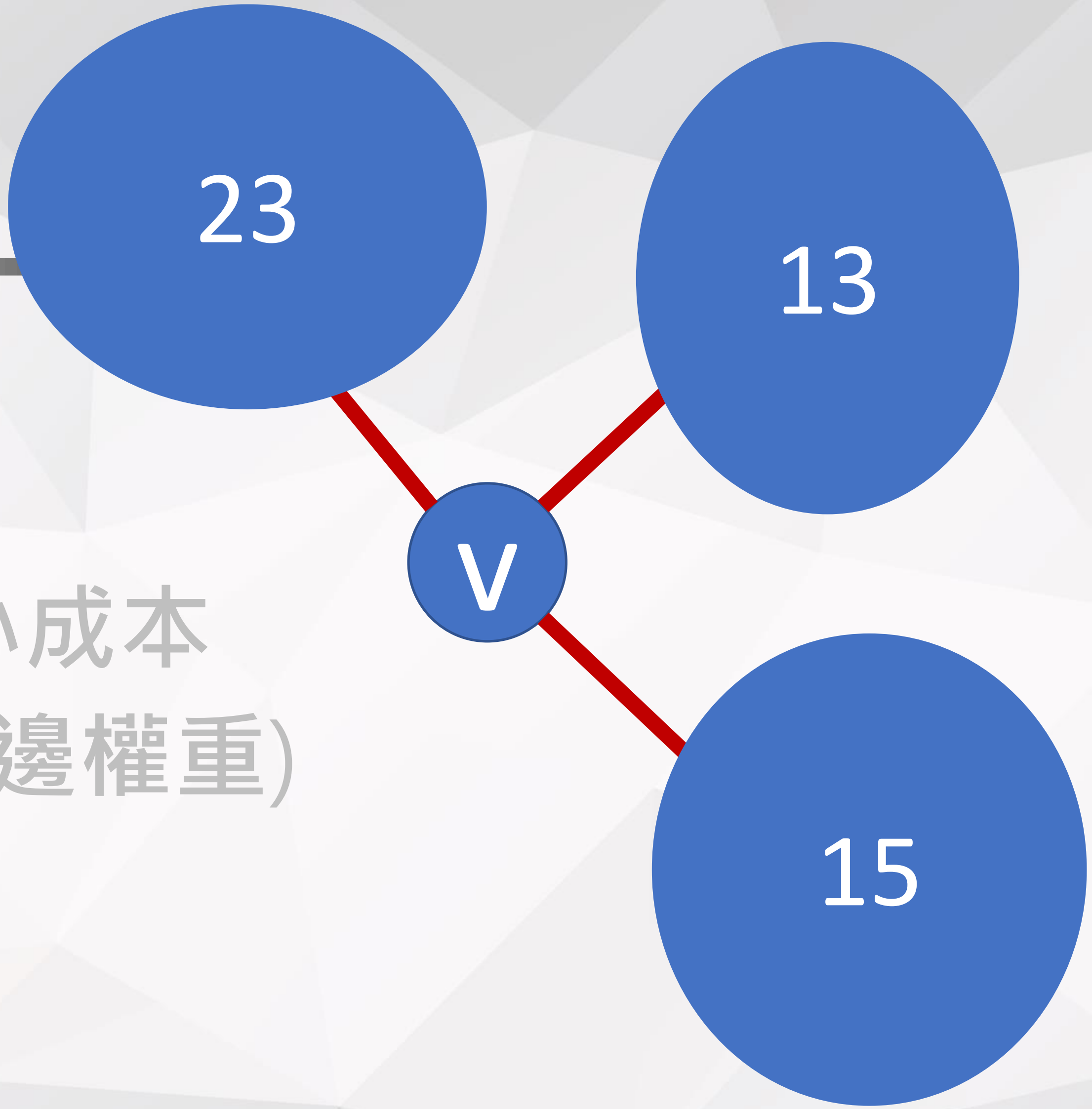
想像自己是圖中的某點 v

身旁有一些鄰點 u

u 知道源點到他們那裡的**最小成本**

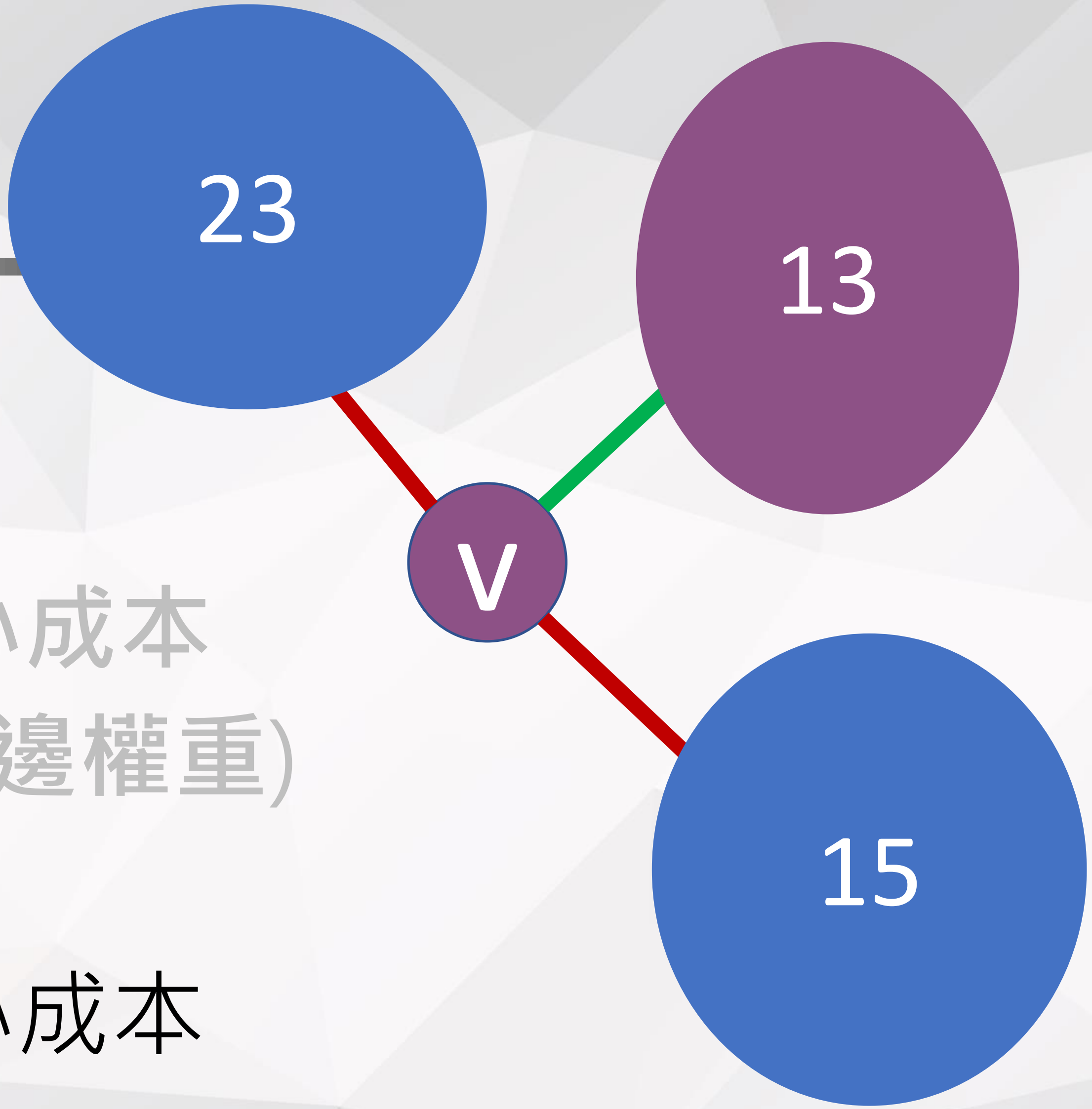
v 知道 u 到自己那裡的成本 (**邊權重**)

v 能計算各點到 v 的成本



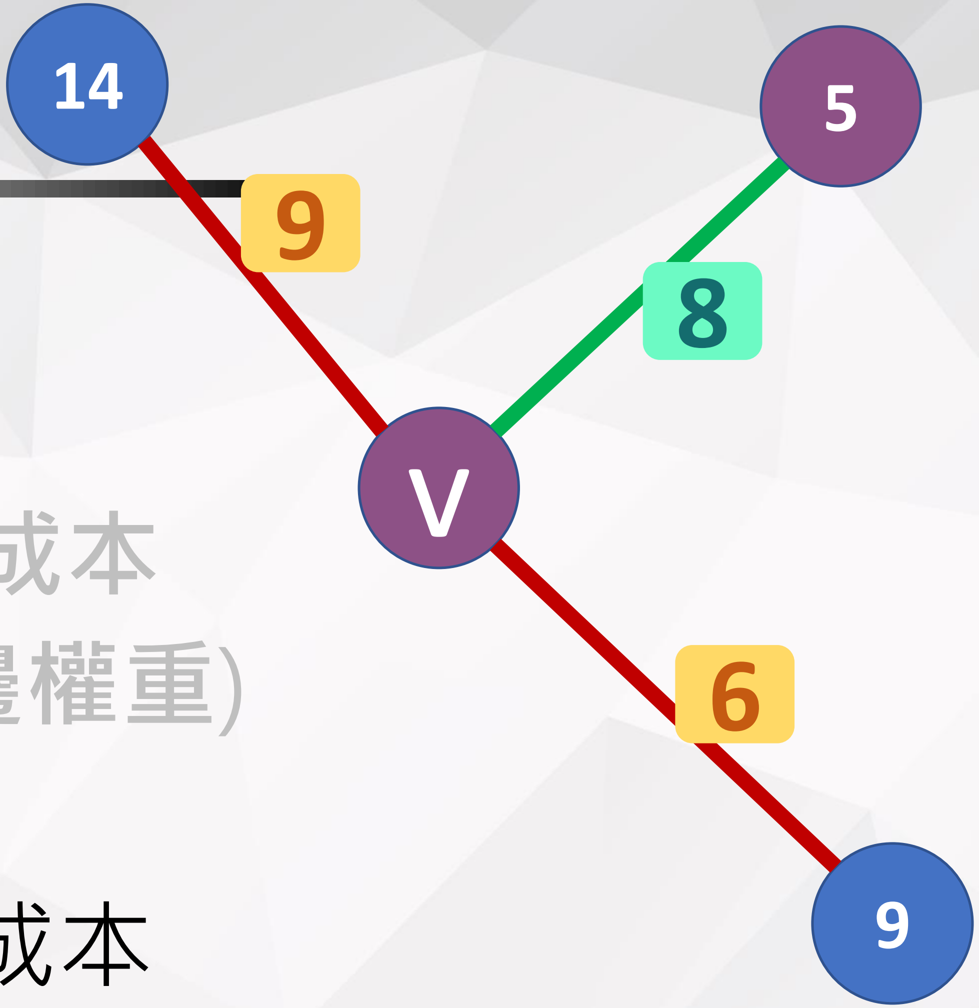
Relaxation

想像自己是圖中的某點 v
身旁有一些鄰點 u
u 知道源點到他們那裡的**最小成本**
v 知道 u 到自己那裡的成本 (**邊權重**)
v 能計算各點到 v 的成本
v 就能得知，源點到 v 的最小成本



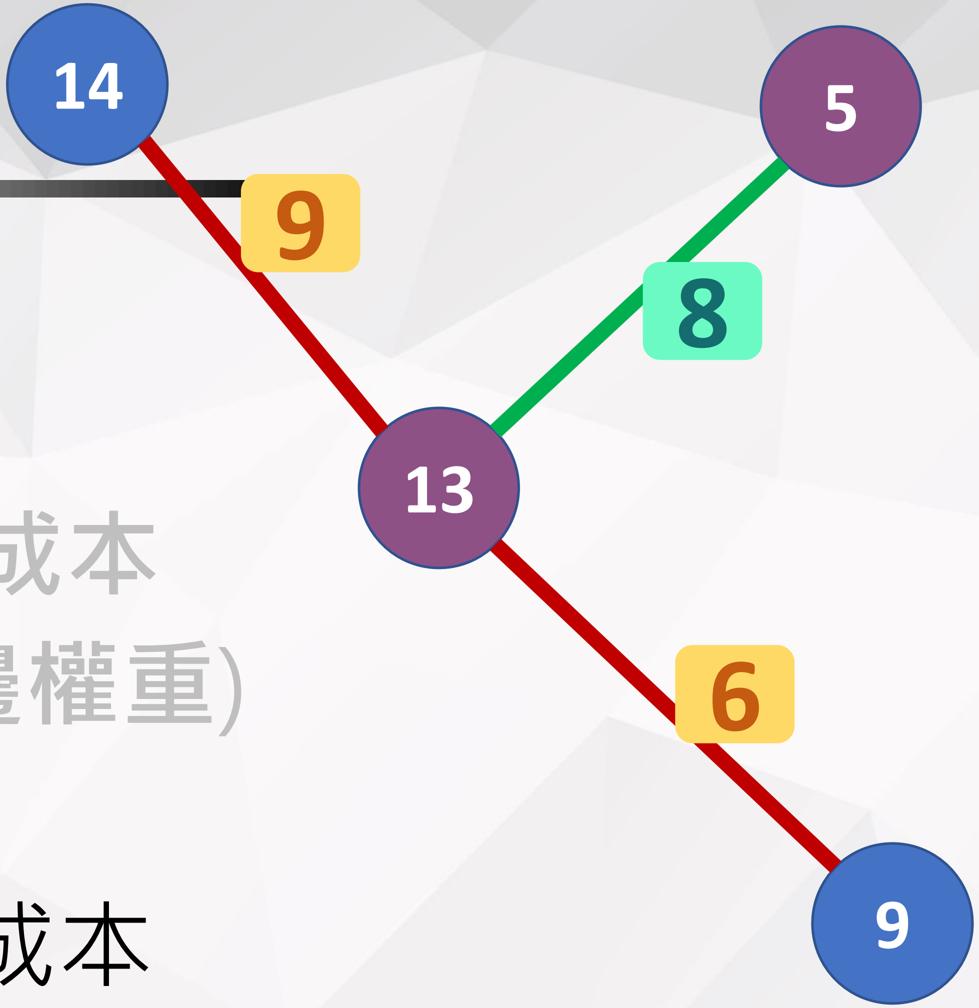
Relaxation

想像自己是圖中的某點 v
身旁有一些鄰點 u
u 知道源點到他們那裡的**最小成本**
v 知道 u 到自己那裡的成本 (**邊權重**)
v 能計算各點到 v 的成本
v 就能得知，源點到 v 的最小成本



Relaxation

想像自己是圖中的某點 v
身旁有一些鄰點 u
u 知道源點到他們那裡的**最小成本**
v 知道 u 到自己那裡的成本 (**邊權重**)
v 能計算各點到 v 的成本
v 就能得知，源點到 v 的最小成本



Relaxation 實作

```
int update = cost[u] + w; // w := weight

cost[v] = min(cost[v], update);
```

Questions?

單源最短路徑

- Relaxation
- Dijkstra's algorithm
- Bellman-Ford's algorithm

Dijkstra's algorithm

Dijkstra 實作

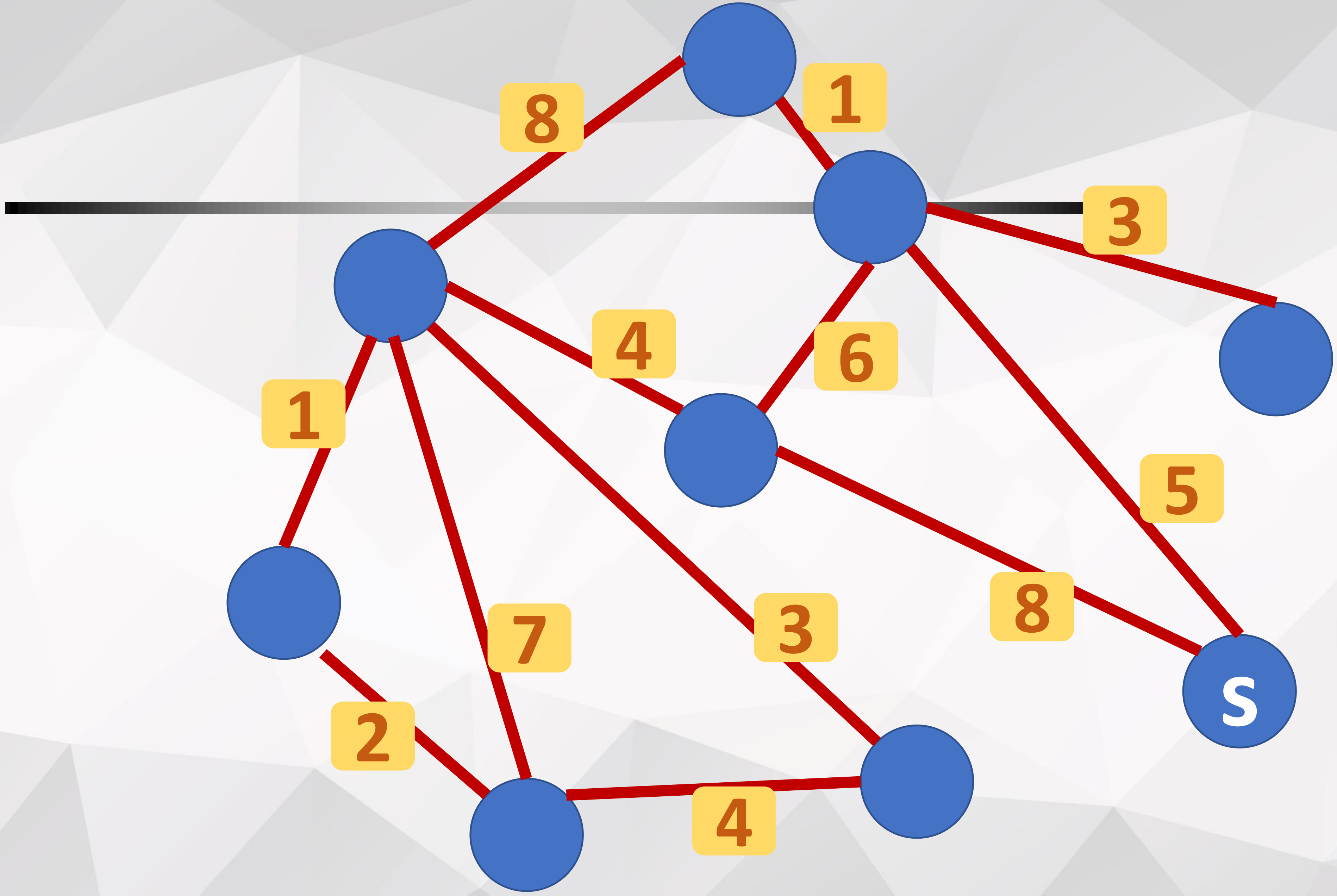
```
vector<node> E[maxn]; // 邊集合
    ⋮

/* 假設輸入完邊的資訊了 */
```

Dijkstra 實作 (**初始化**)

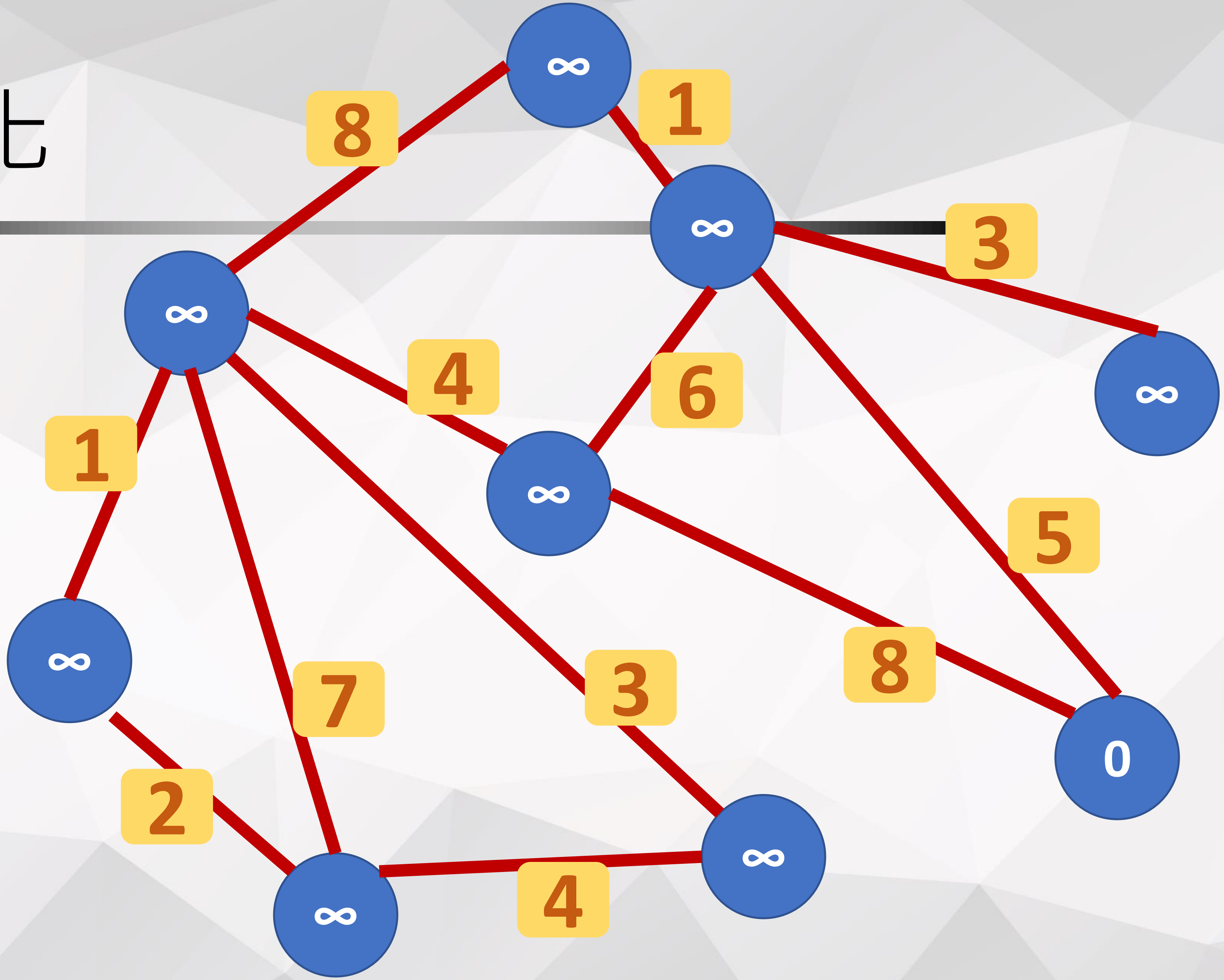
```
memset(s, 0x3f, sizeof(s)); //初始為無限大

priority_queue<node> Q;

Q.push({source, s[source] = 0});
```

Dijkstra 實作

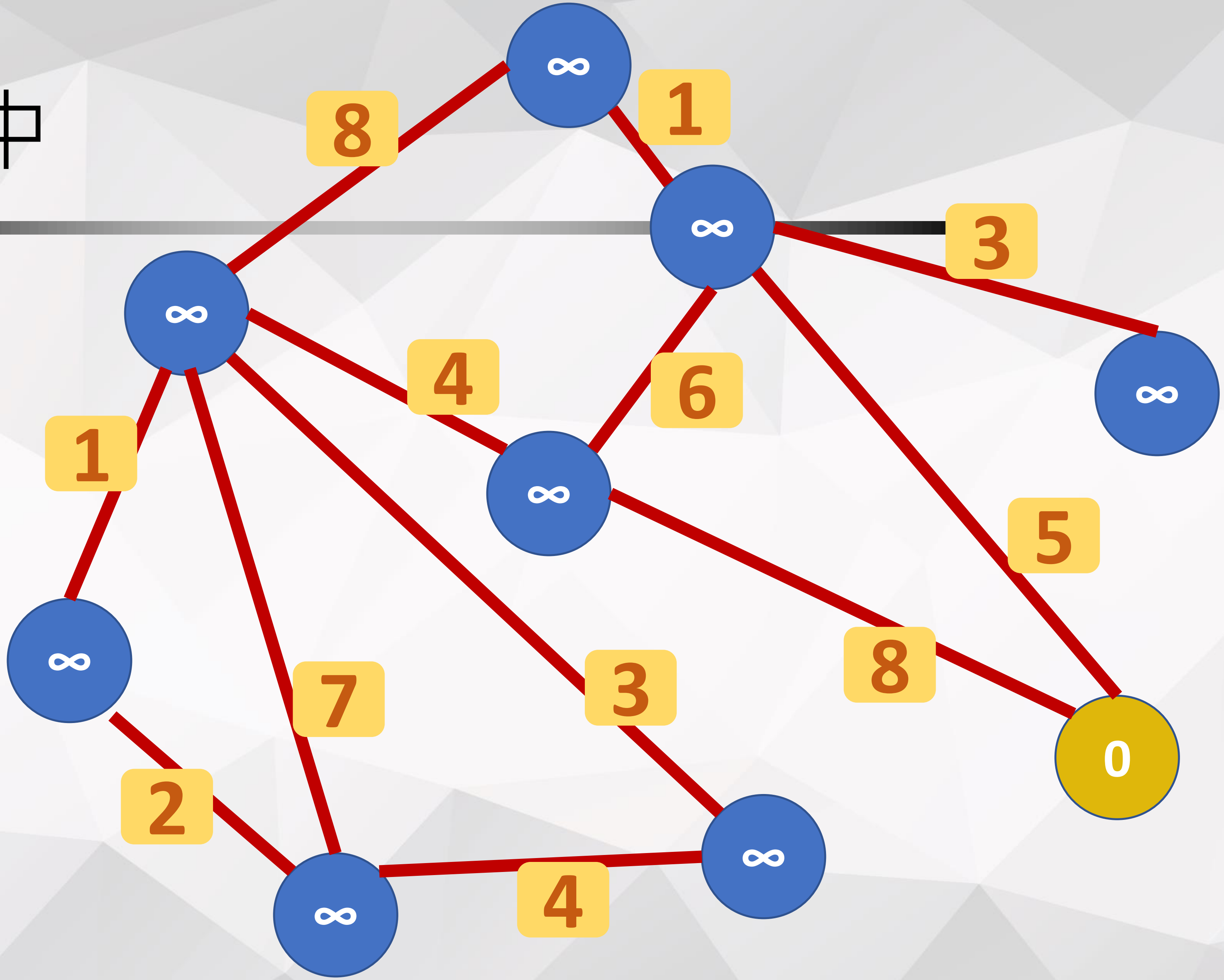
```
while (!Q.empty()) {
    node u = Q.top(); Q.pop();

    for (node v: E[u.id]) {
        int update = u.w + v.w;

        if (update < s[v.id])
            Q.push({v.id, s[v.id] = update});
    }
}
```


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

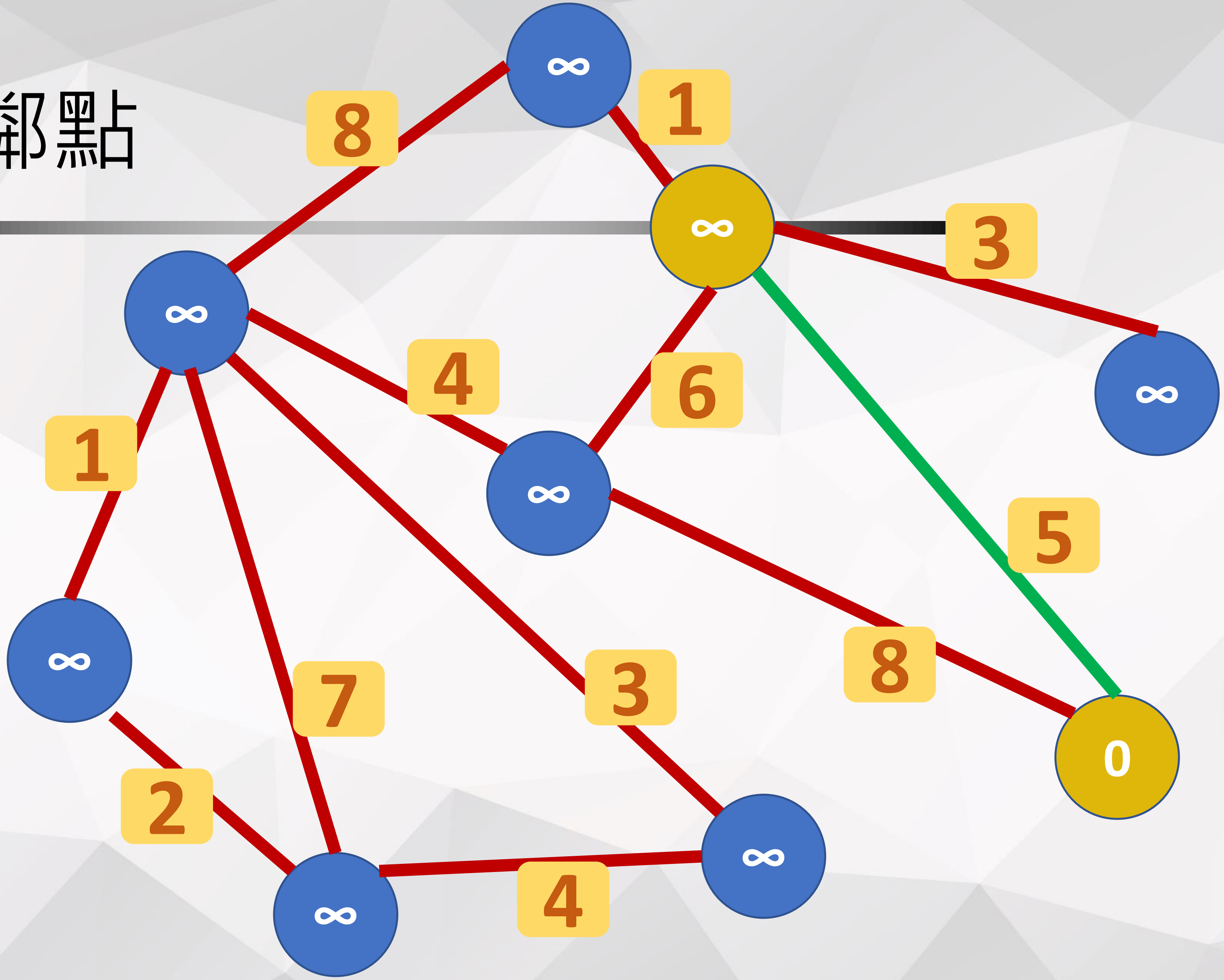
初始化



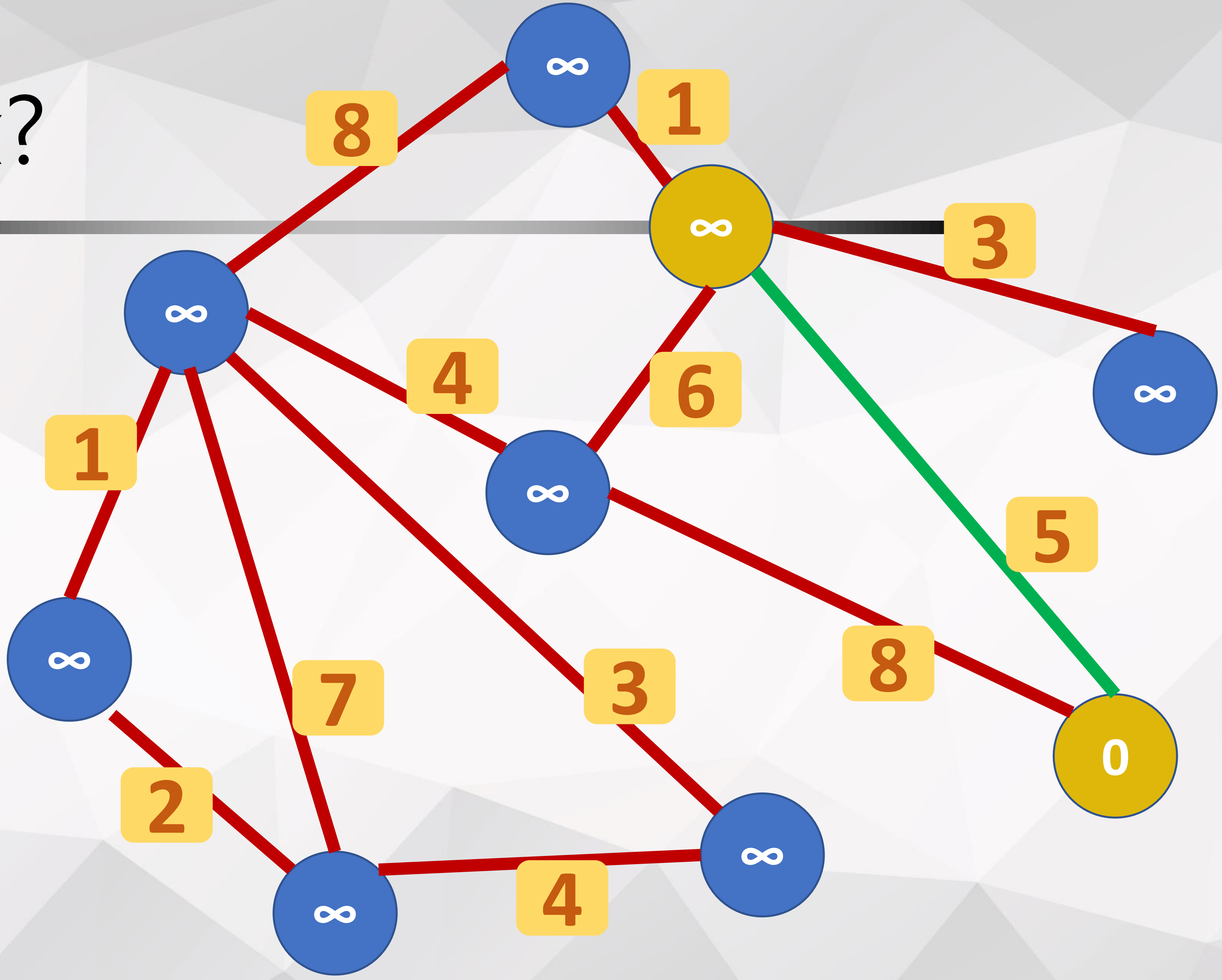
拜訪中



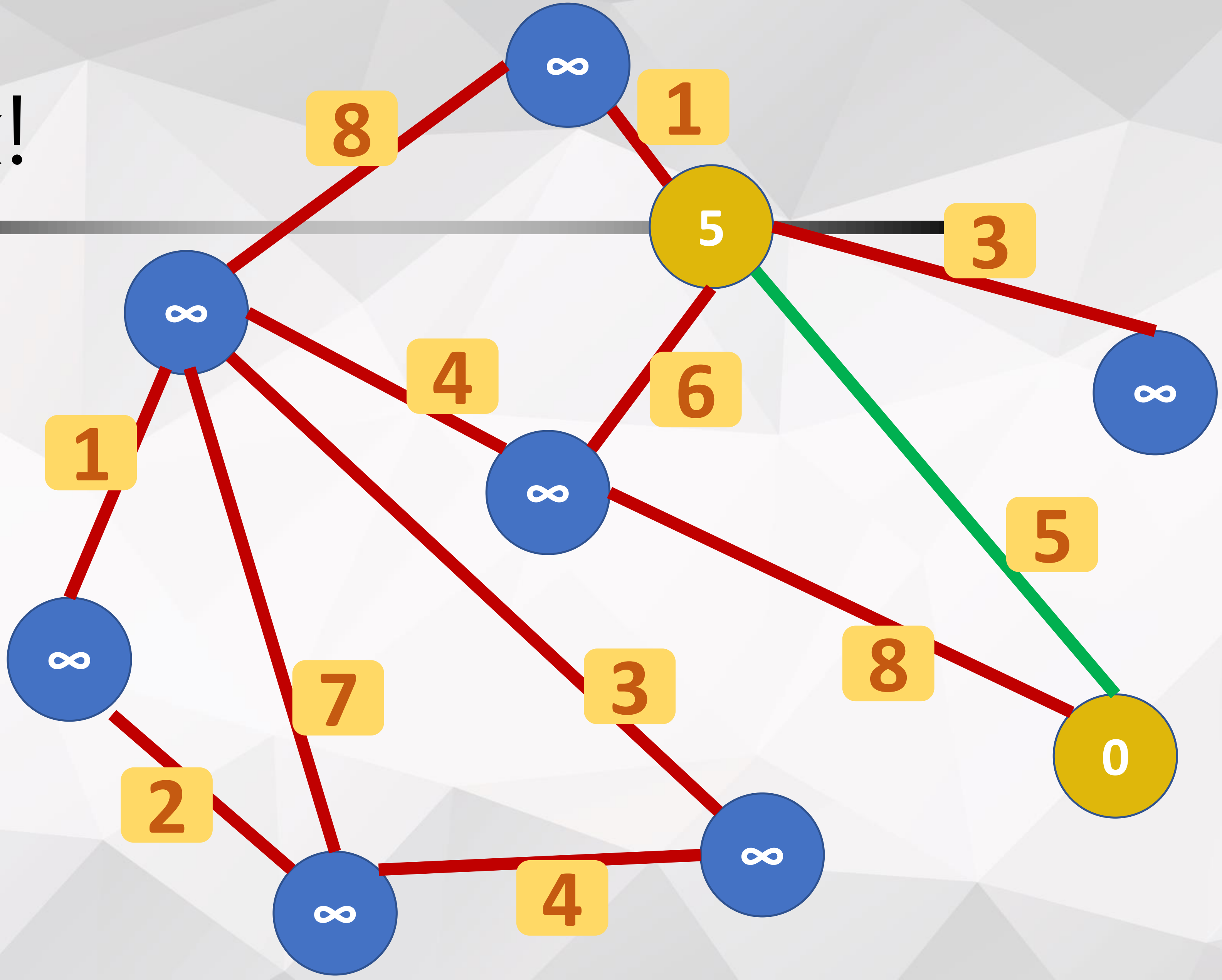
拜訪鄰點



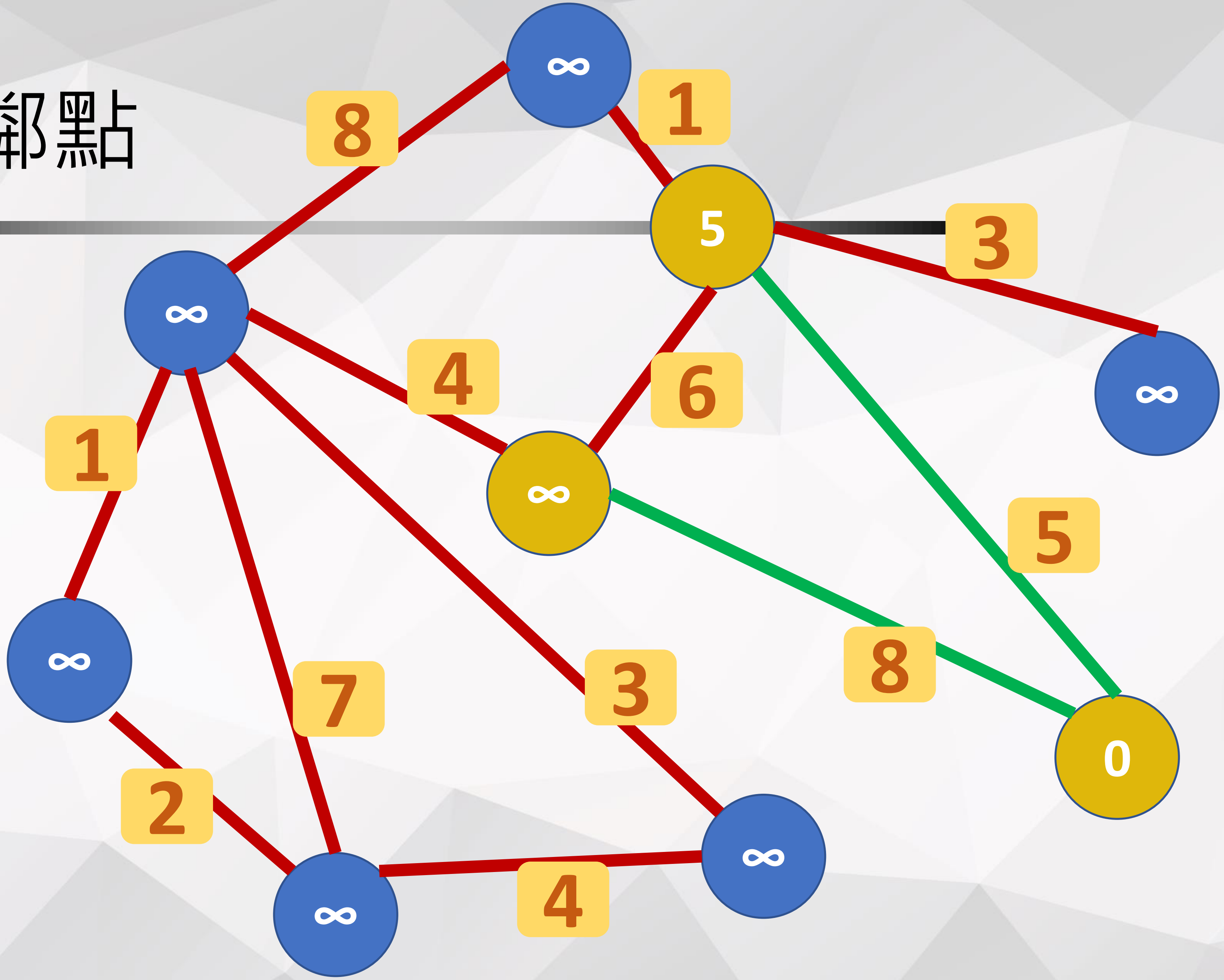
Relax?



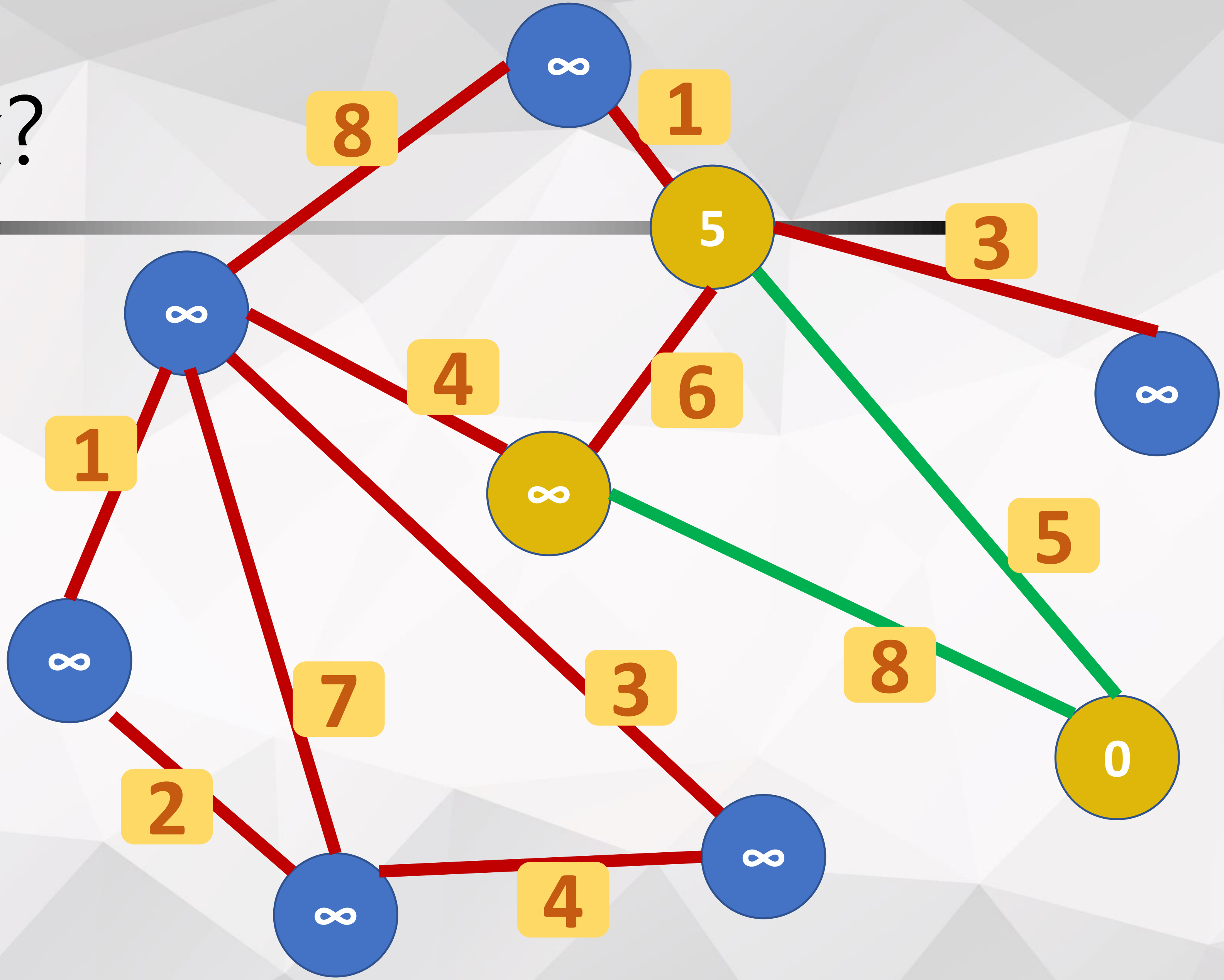
Relax!



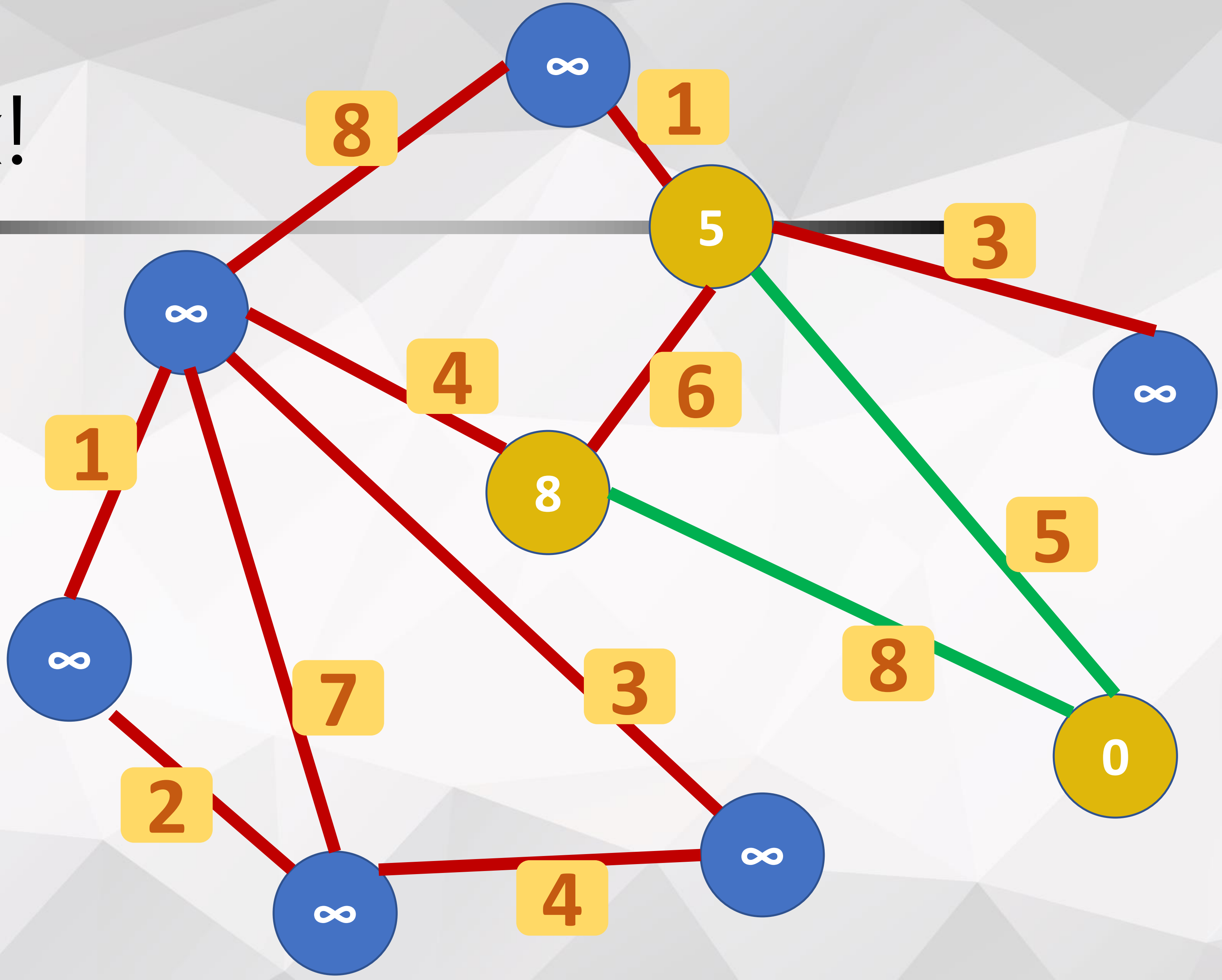
拜訪鄰點



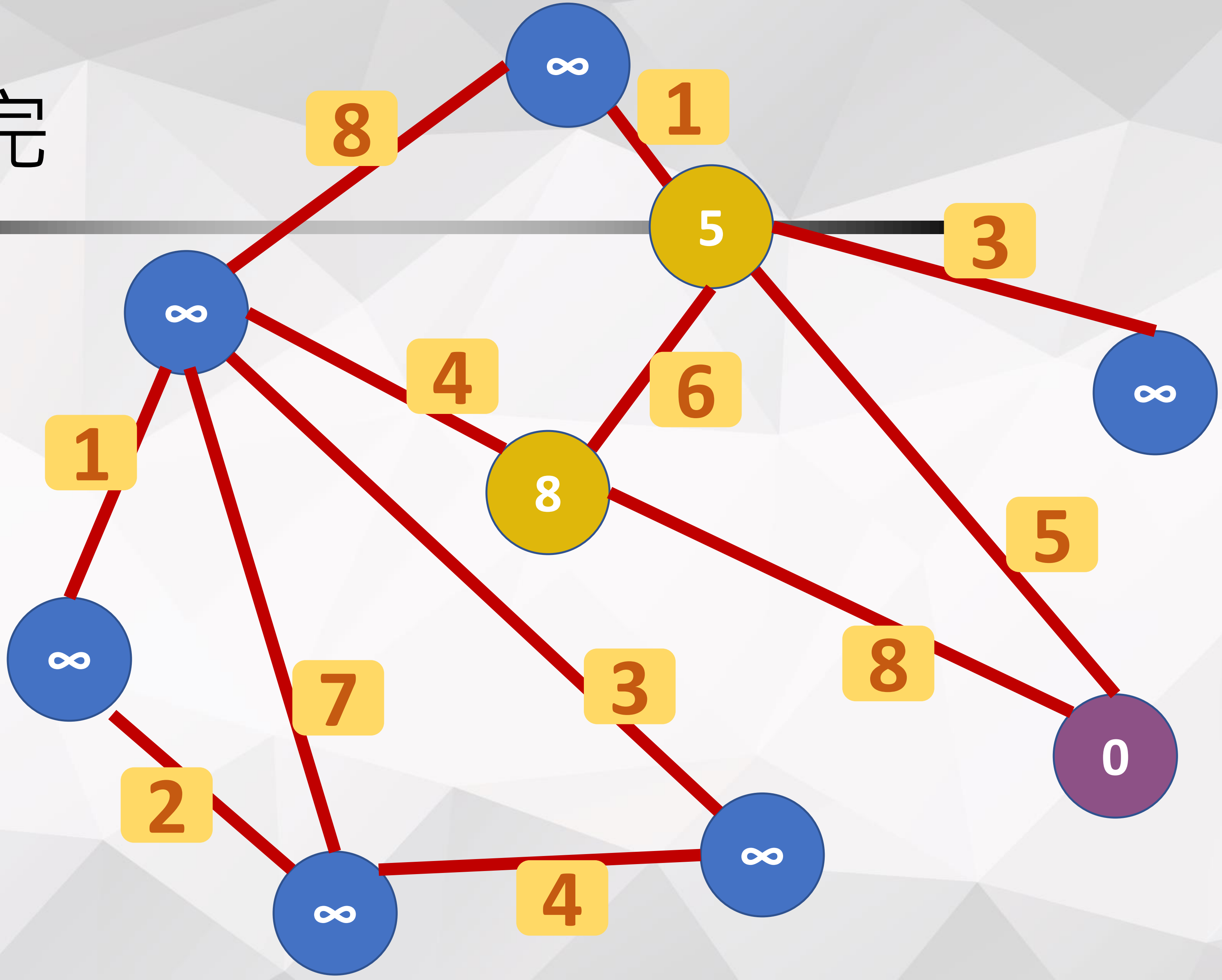
Relax?



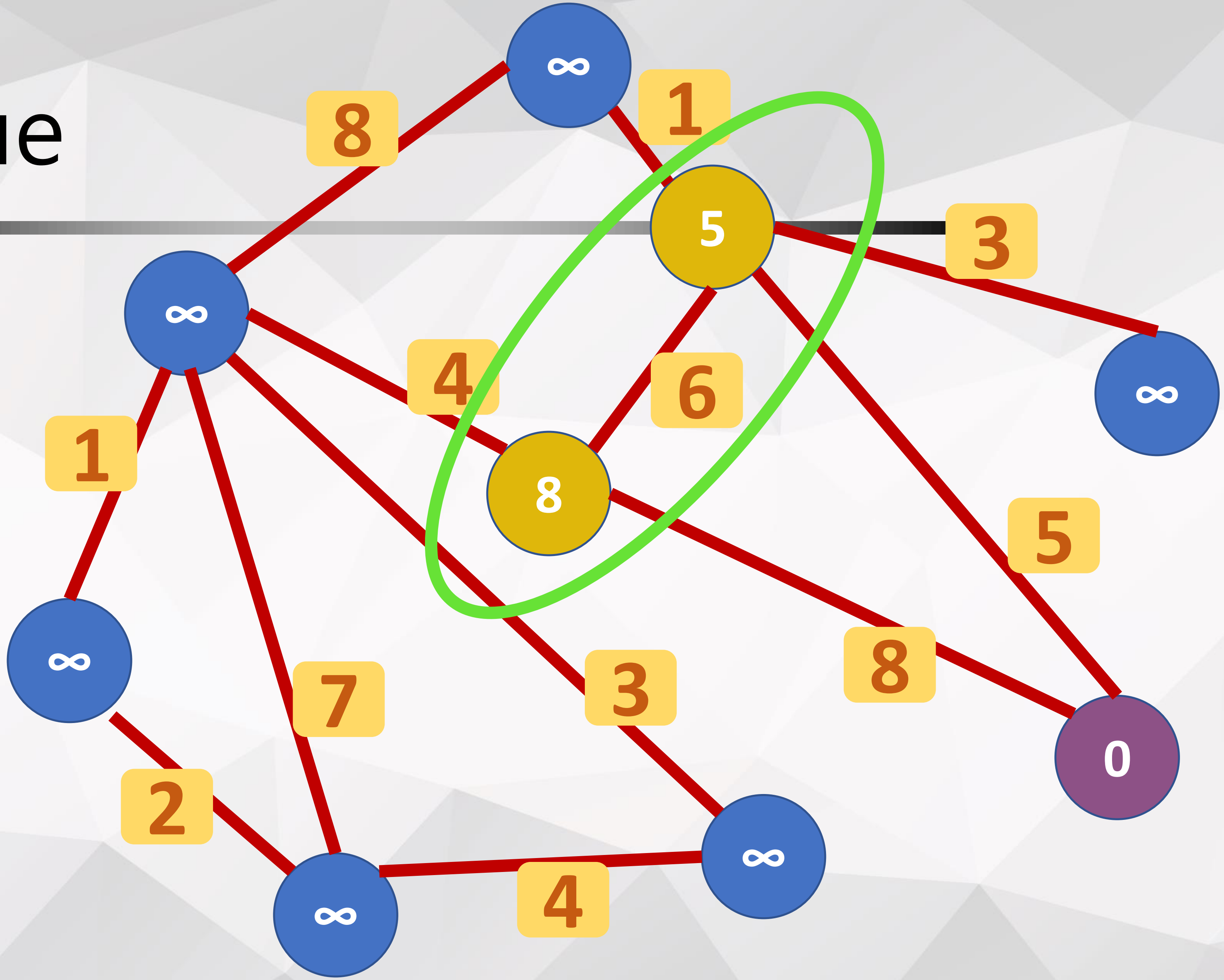
Relax!



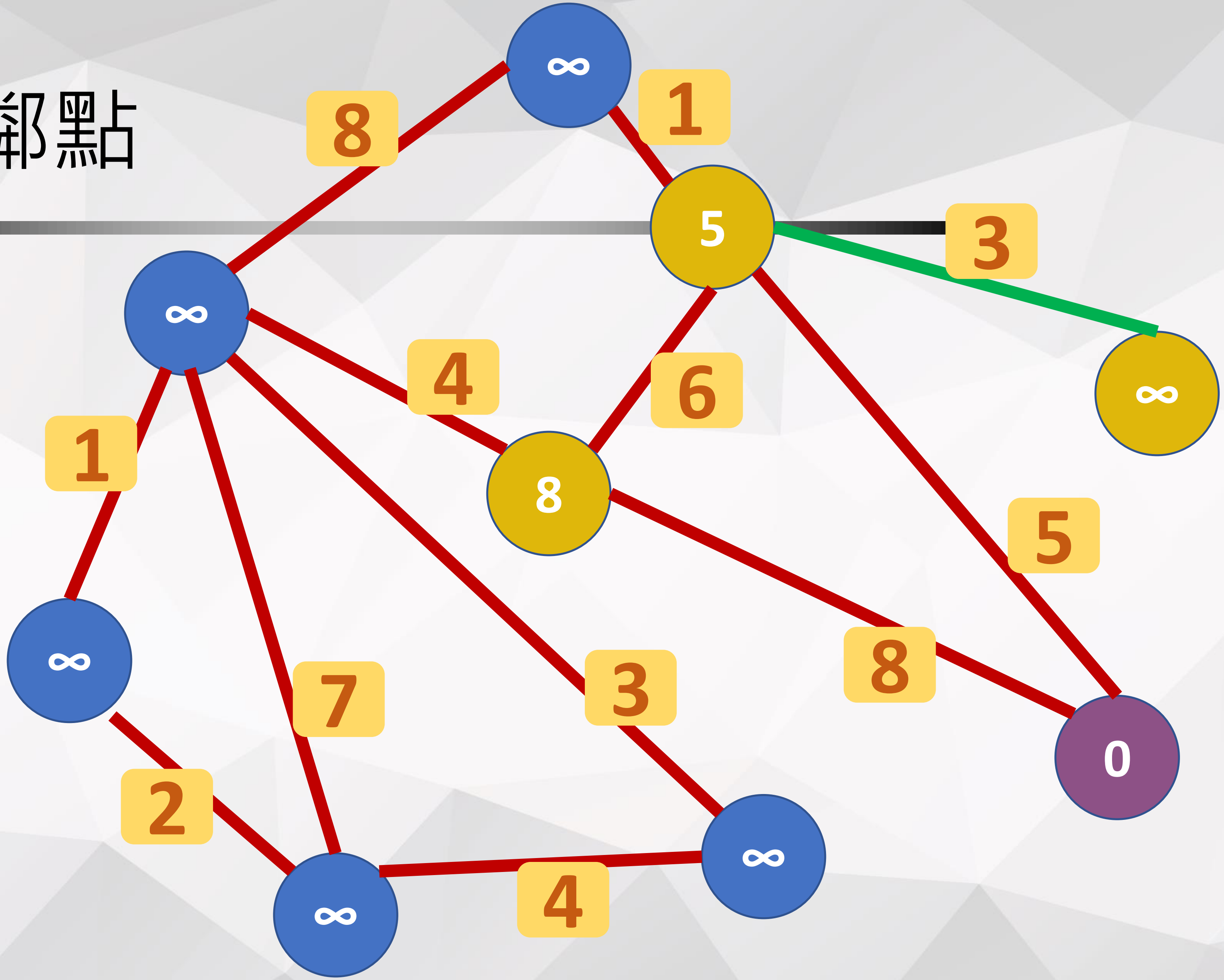
拜訪完



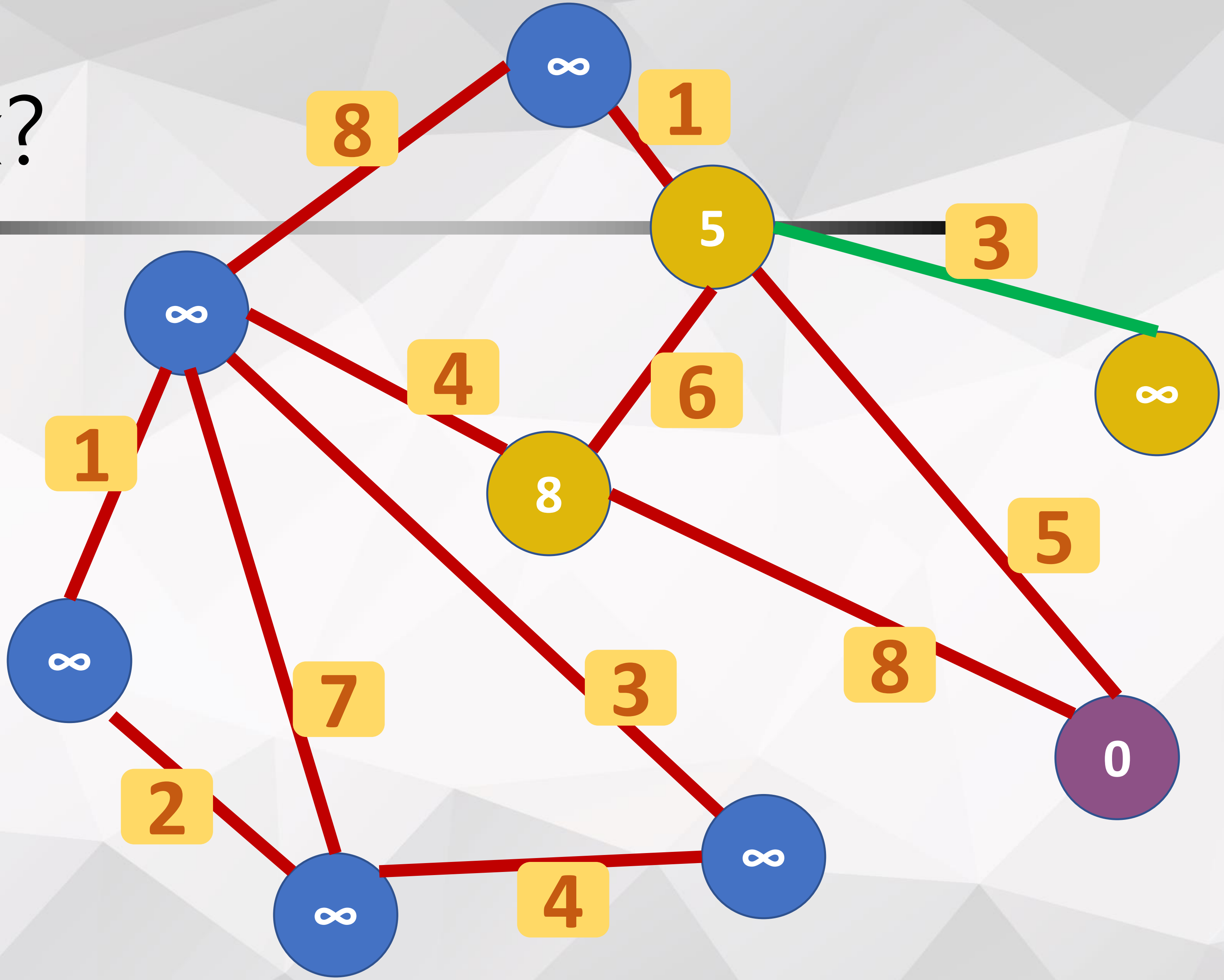
Queue



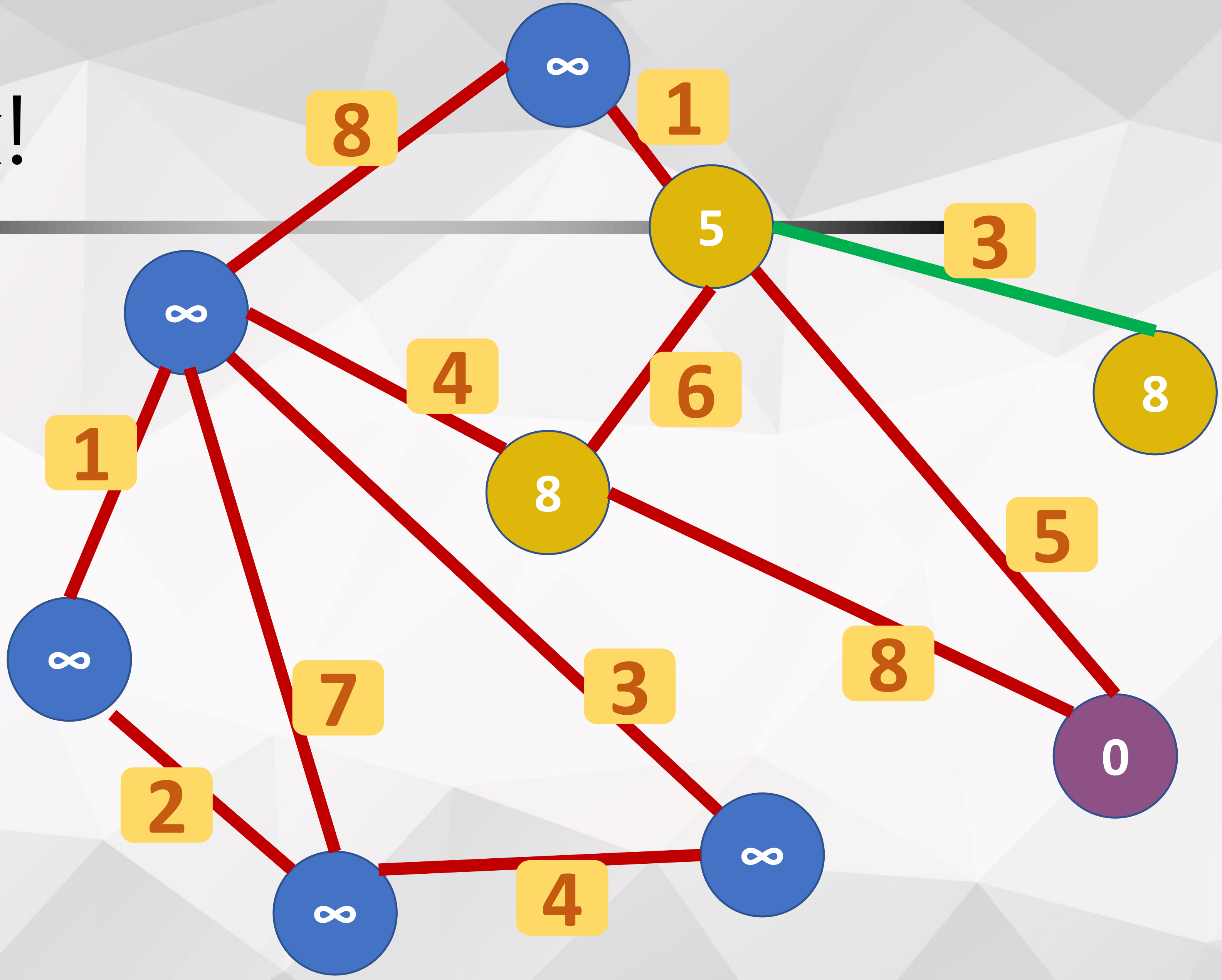
拜訪鄰點



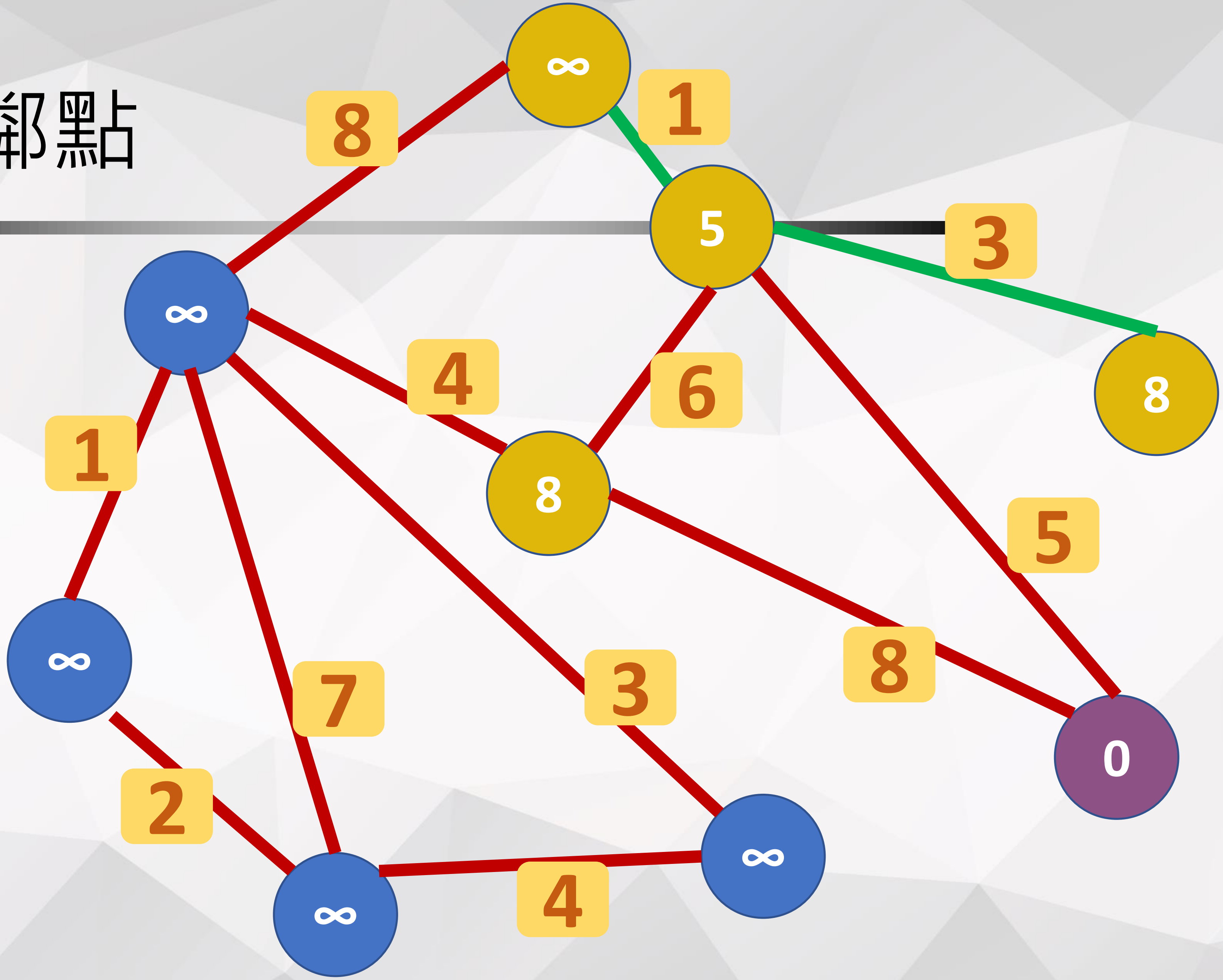
Relax?



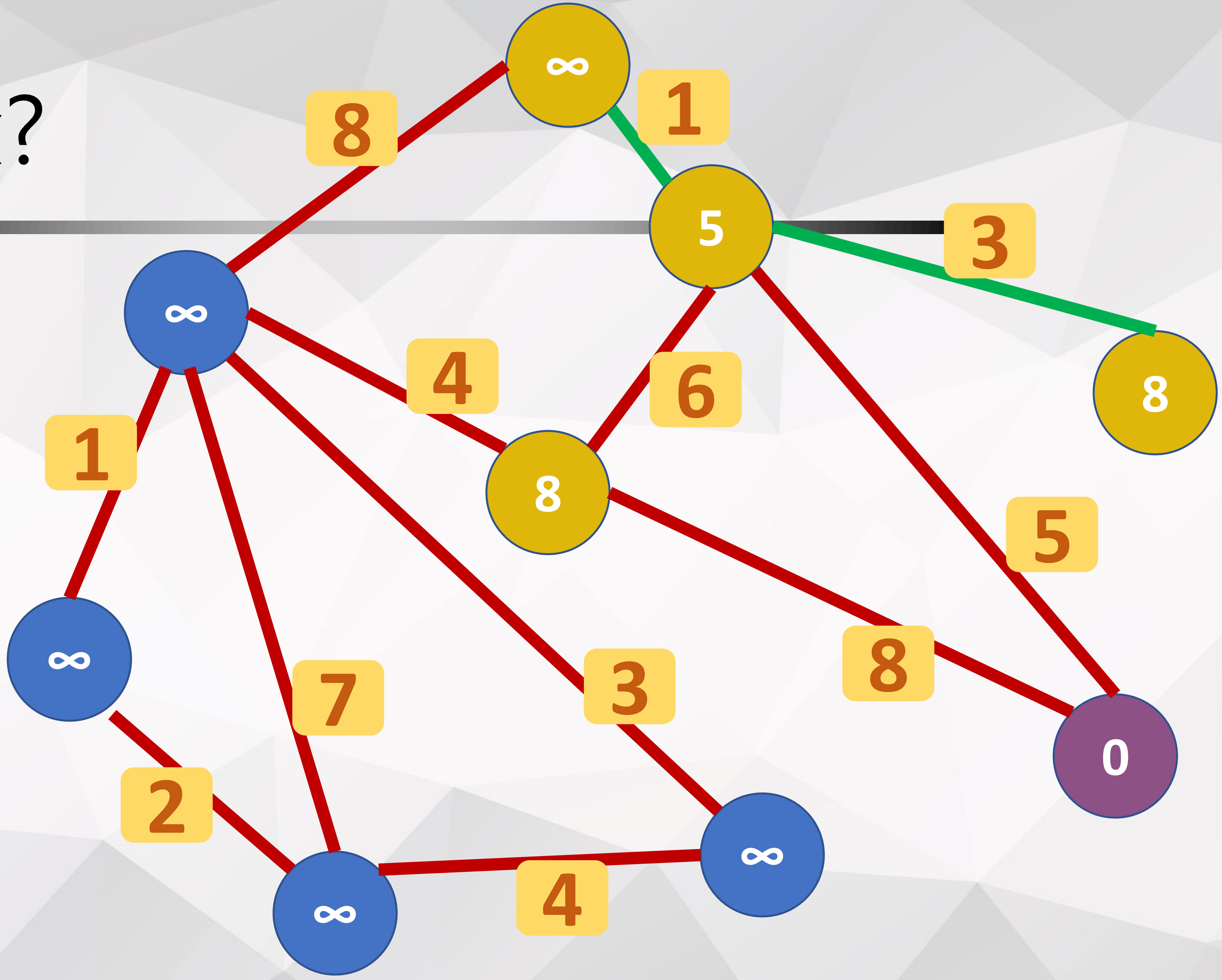
Relax!



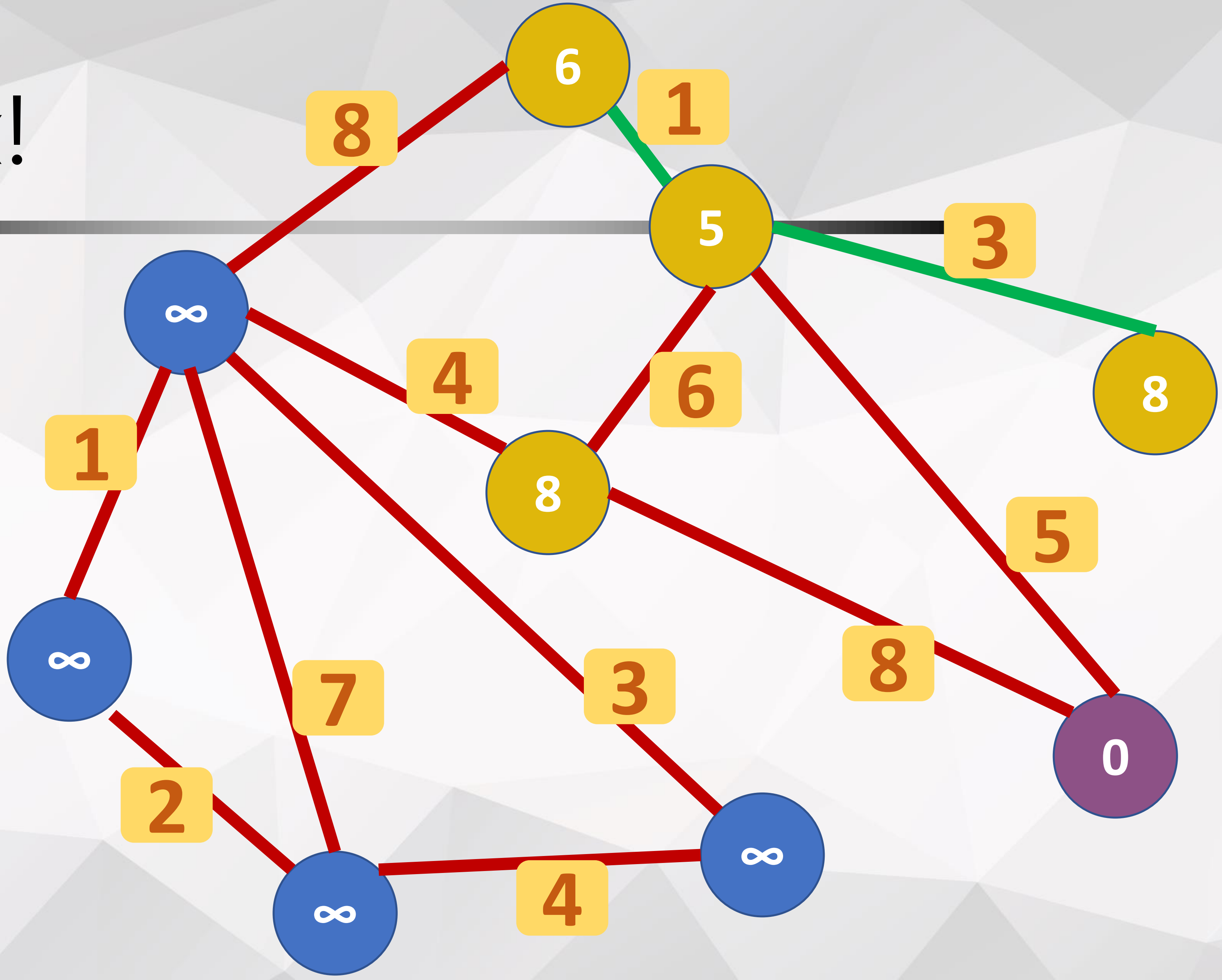
拜訪鄰點



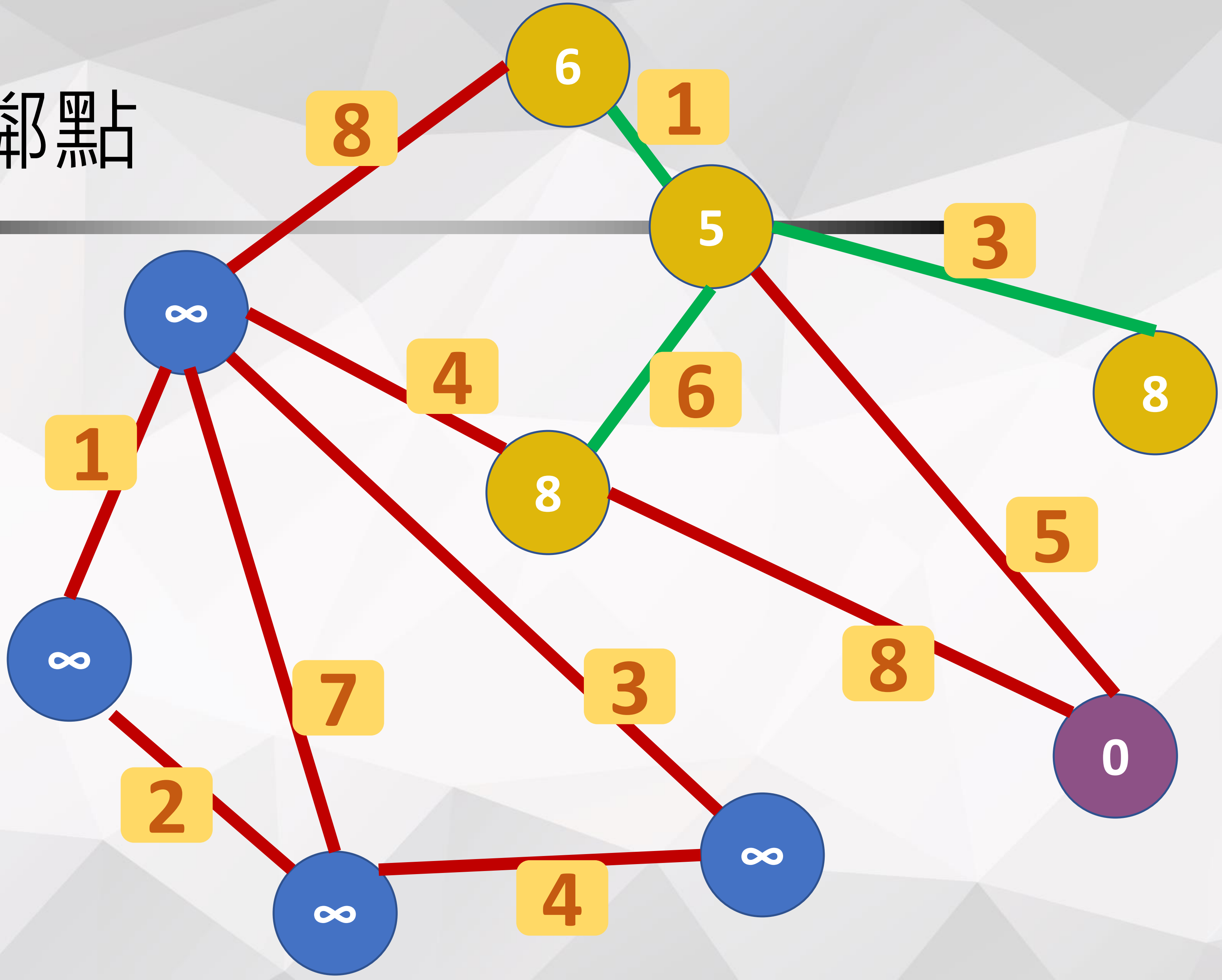
Relax?



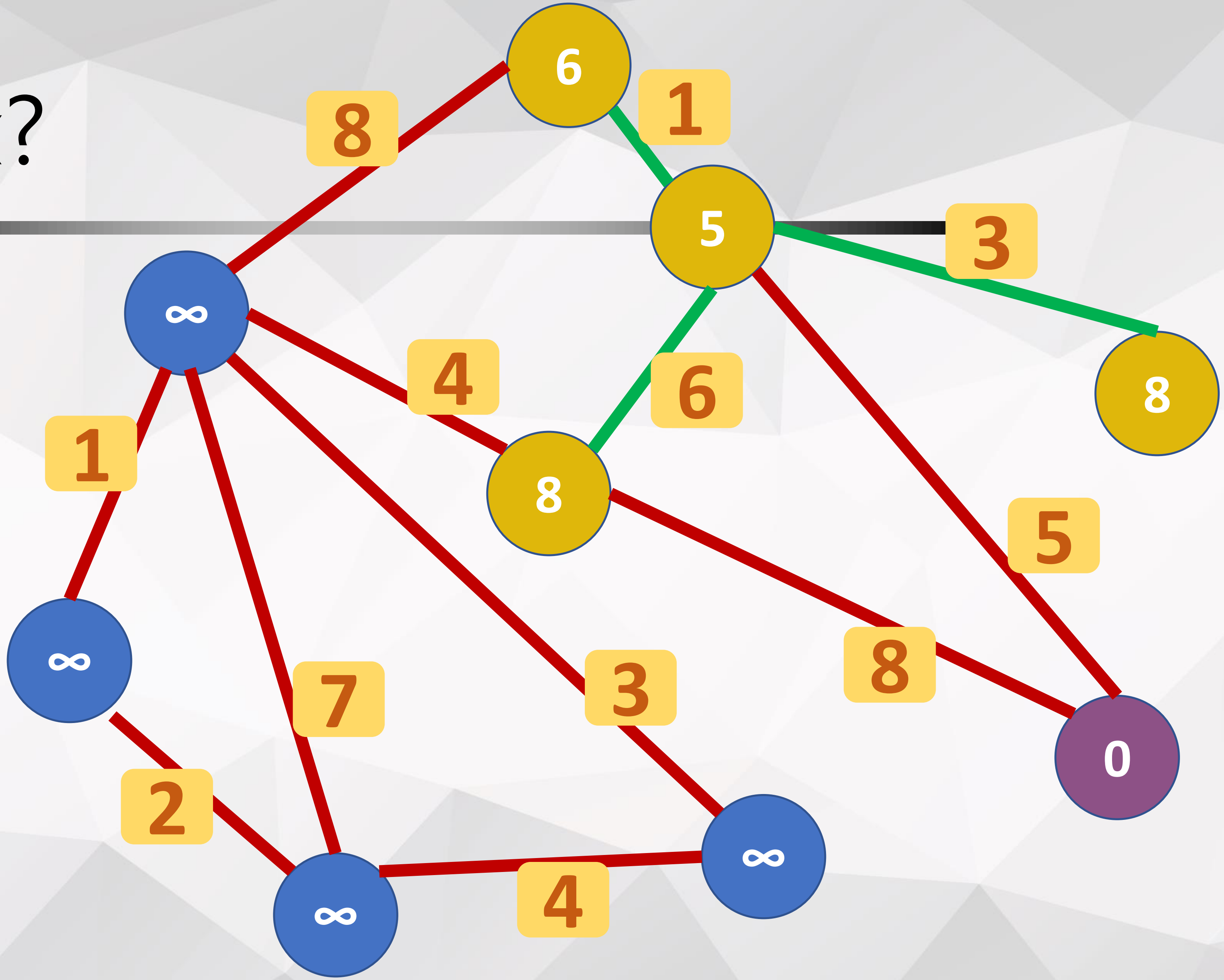
Relax!



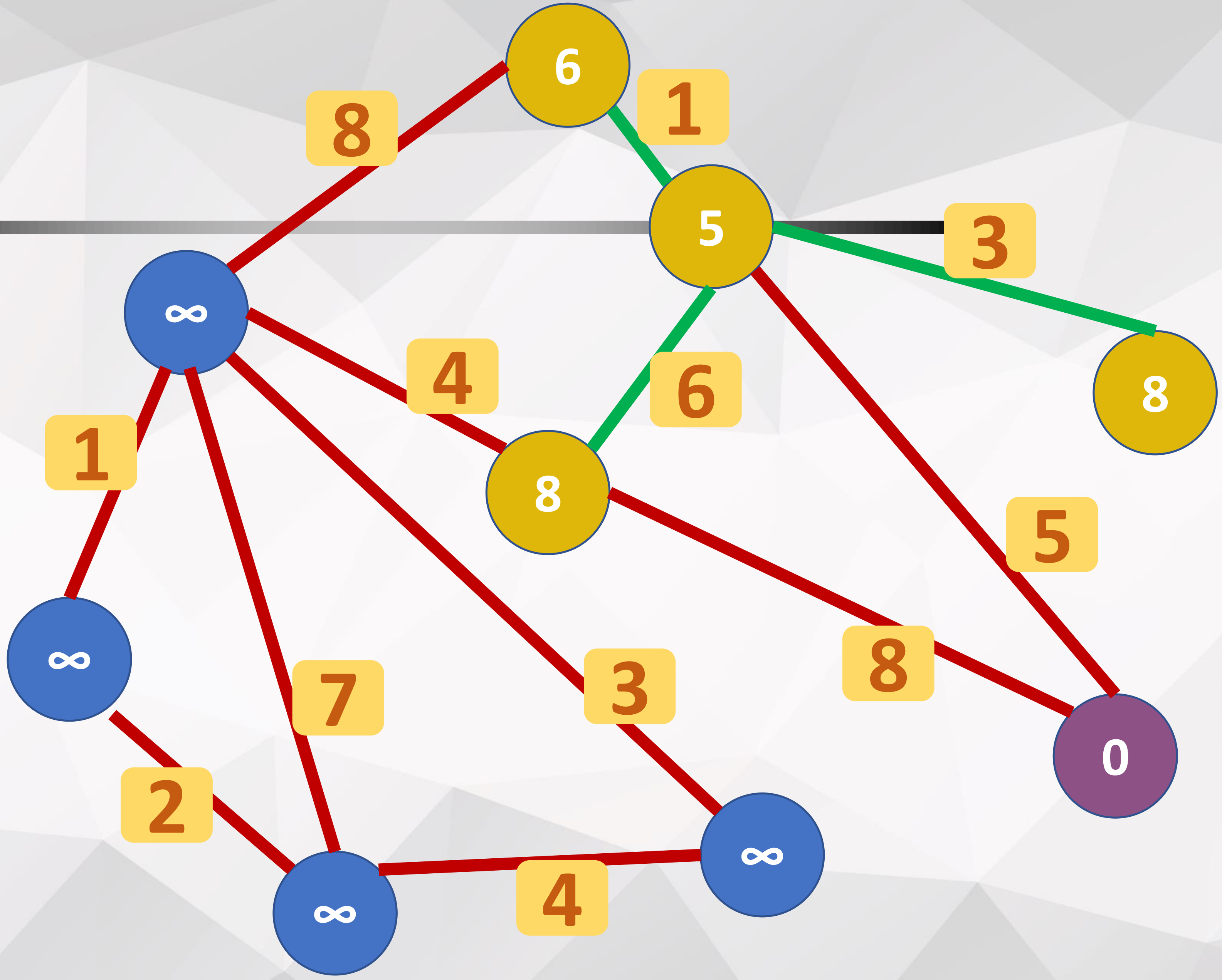
拜訪鄰點



Relax?

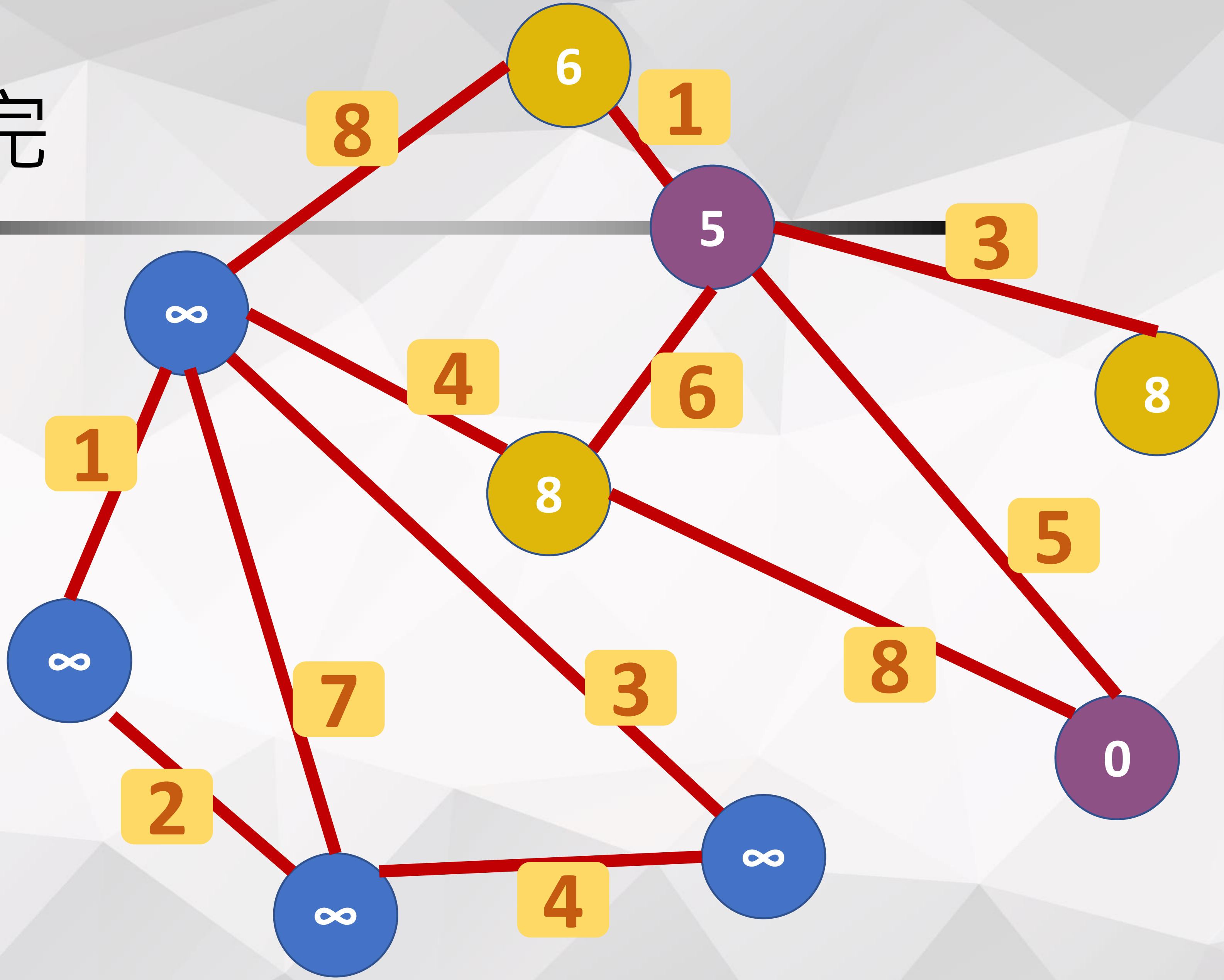


No



Competitive Programming

拜訪完

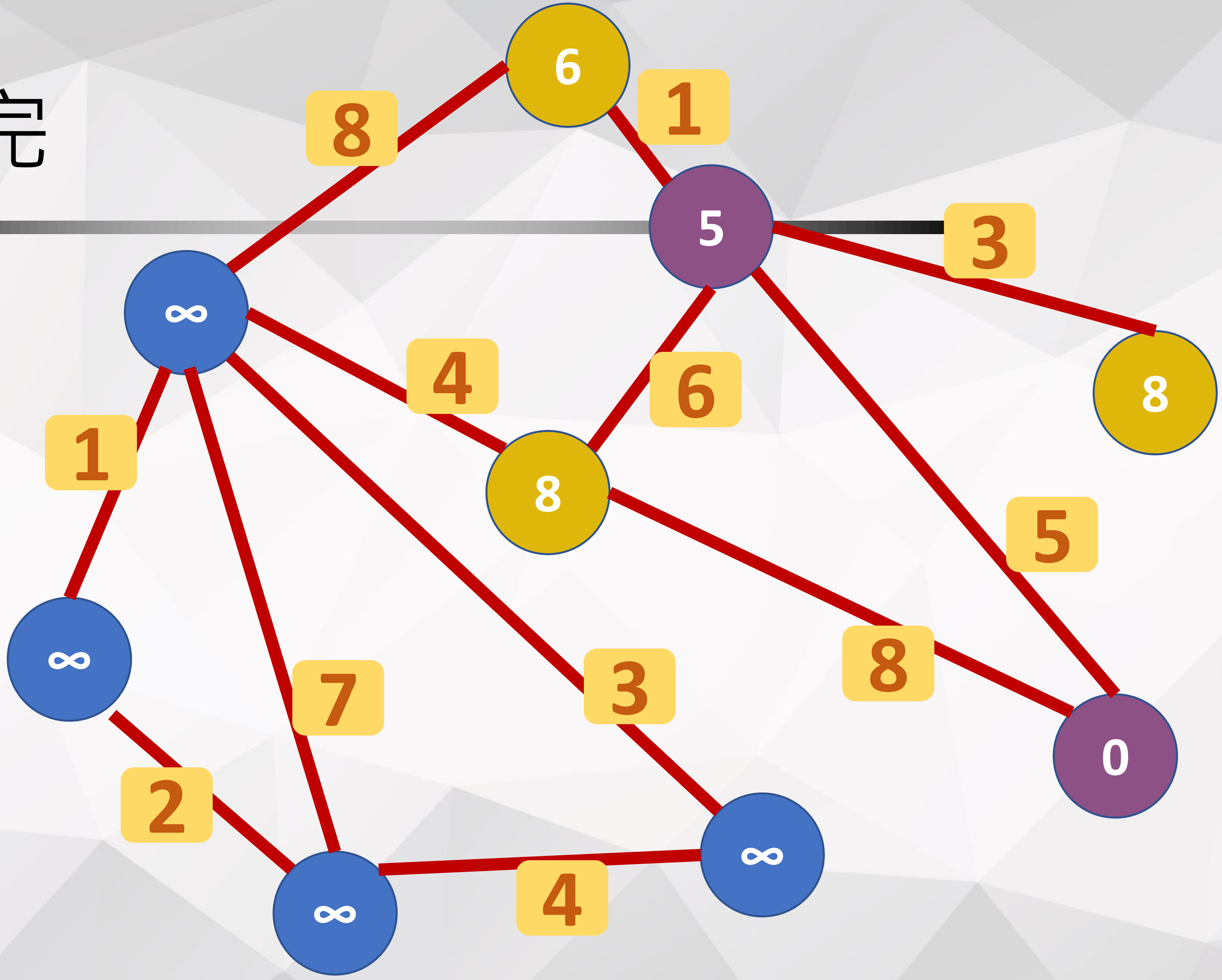


關於 relaxation

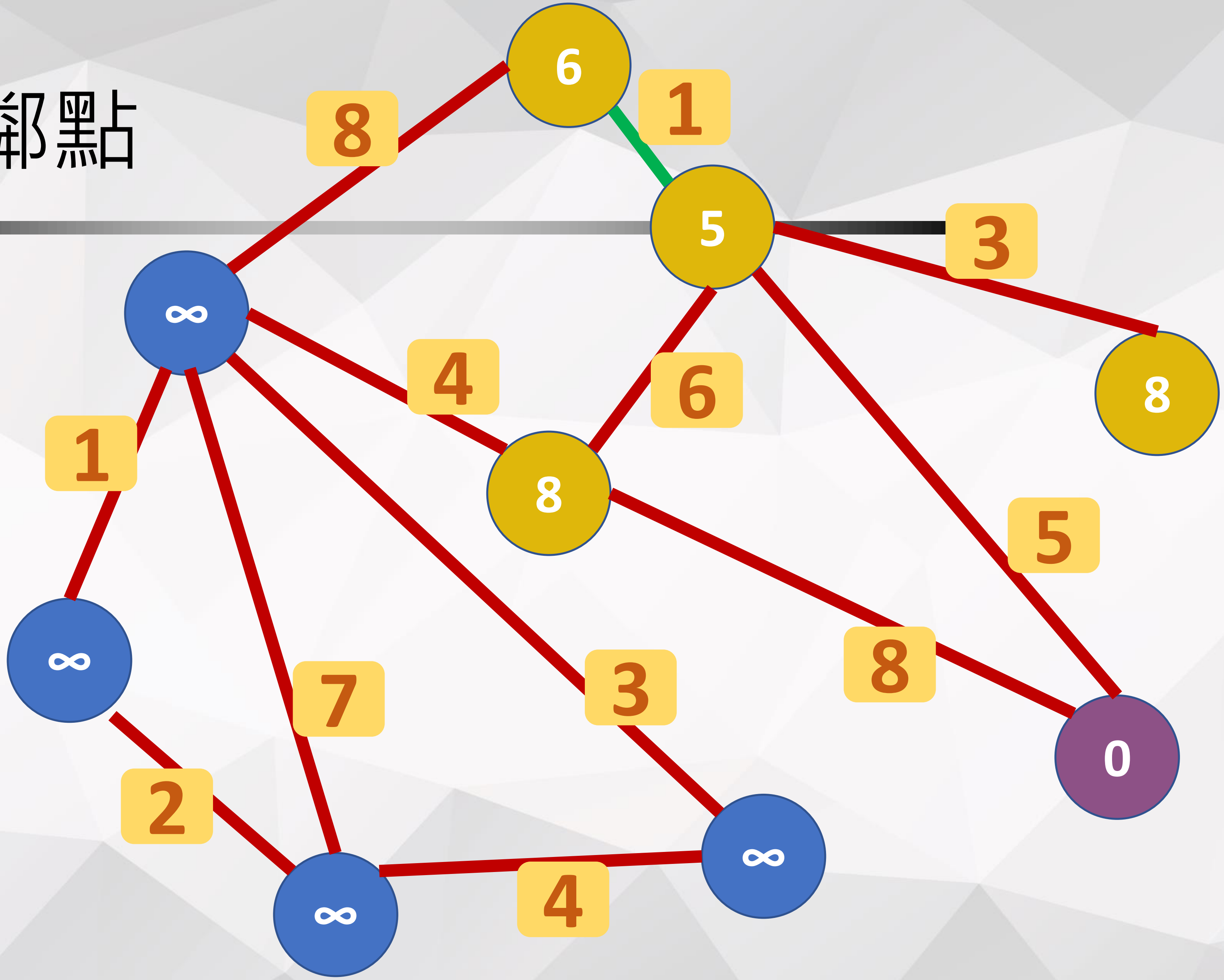
這個**拜訪完**的節點

在**未來**有可能再被 relax 嗎?

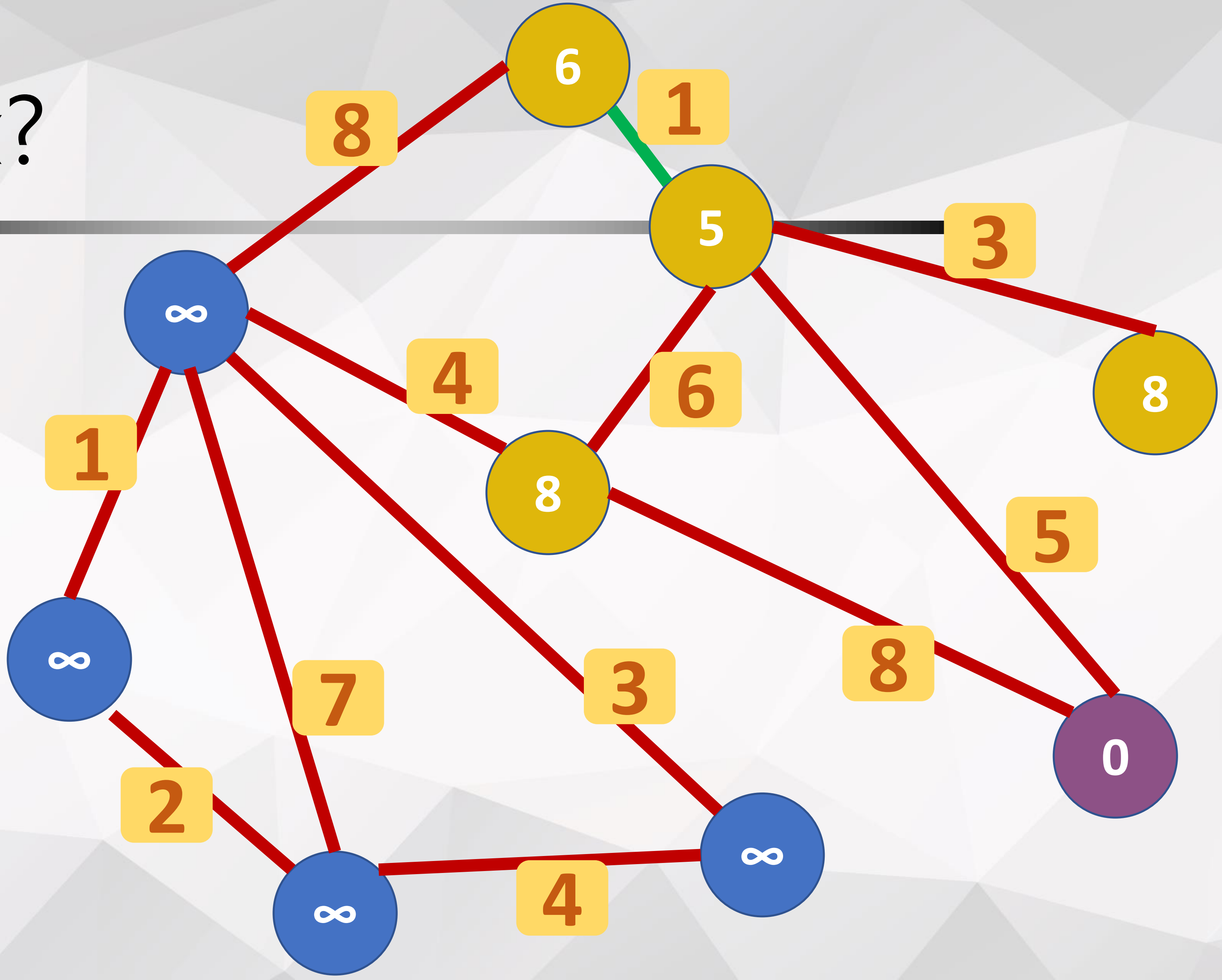
拜訪完



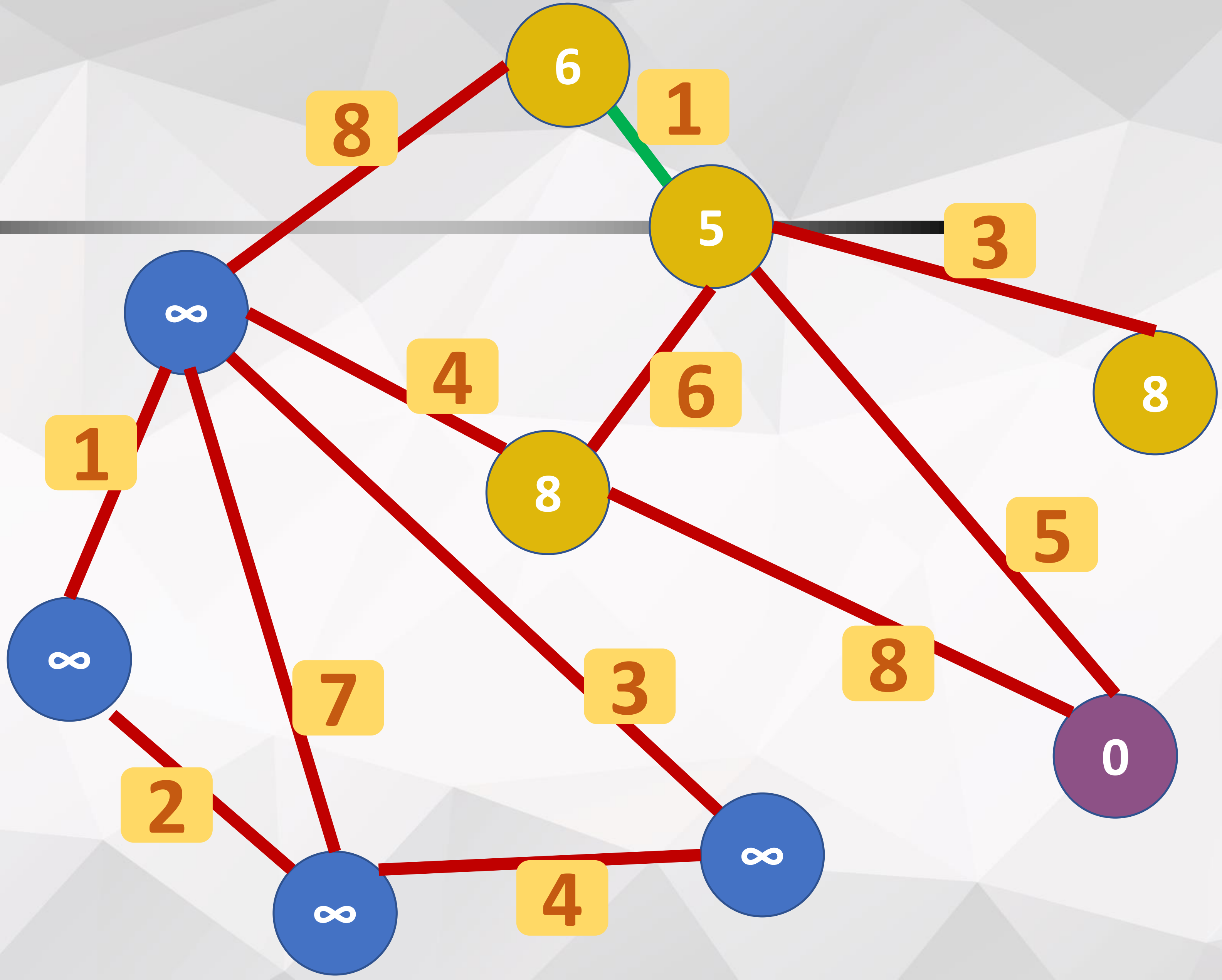
拜訪鄰點



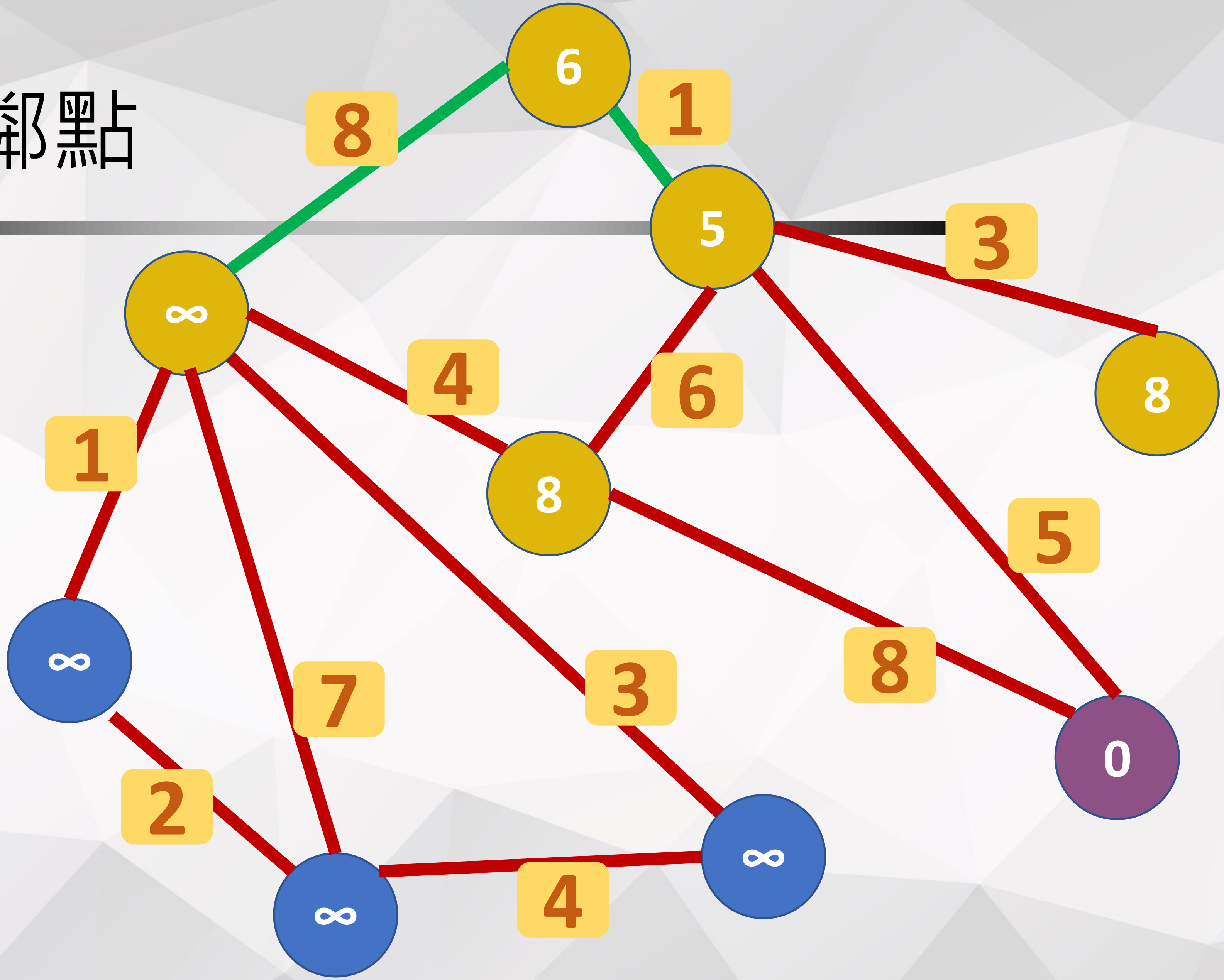
Relax?



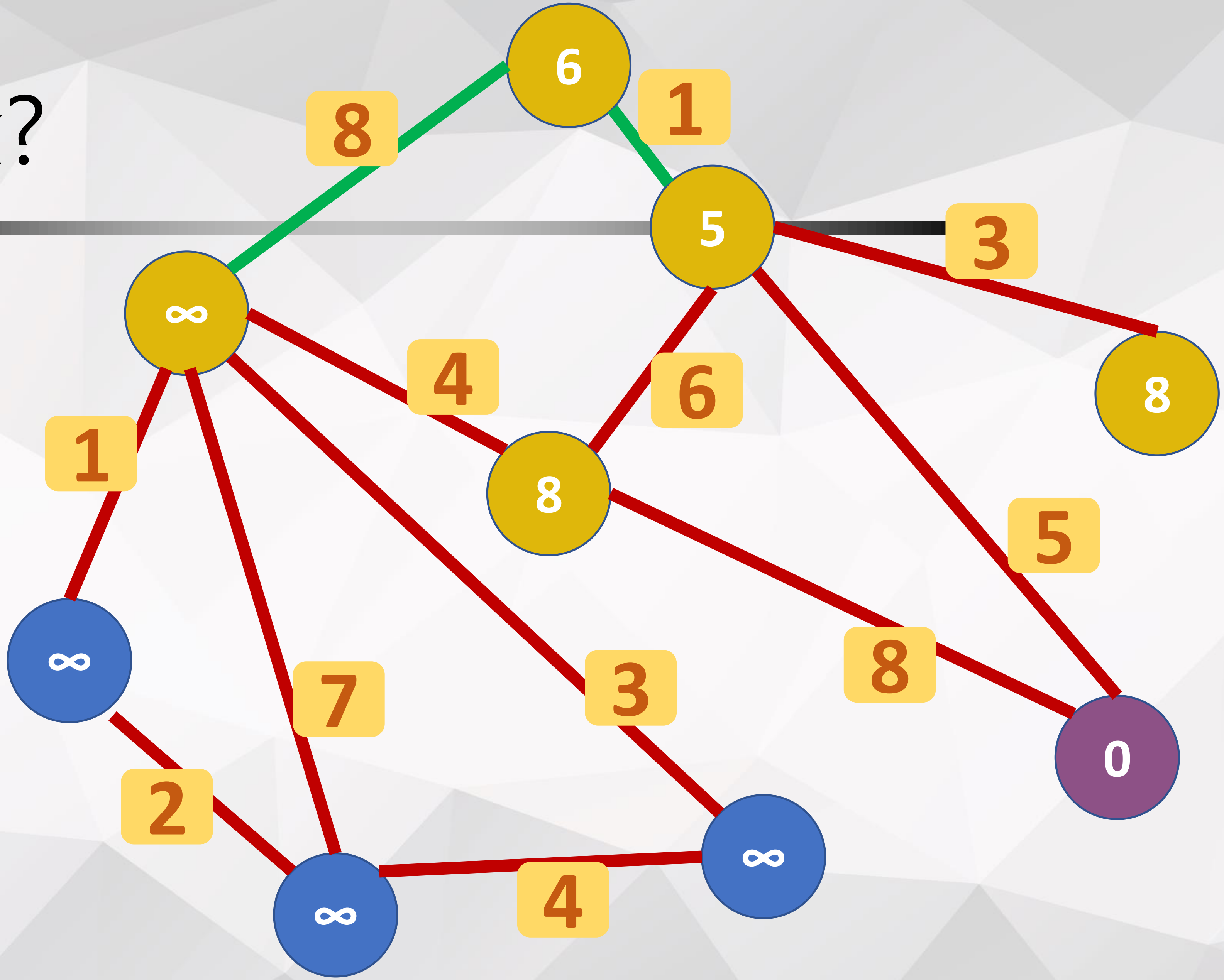
No



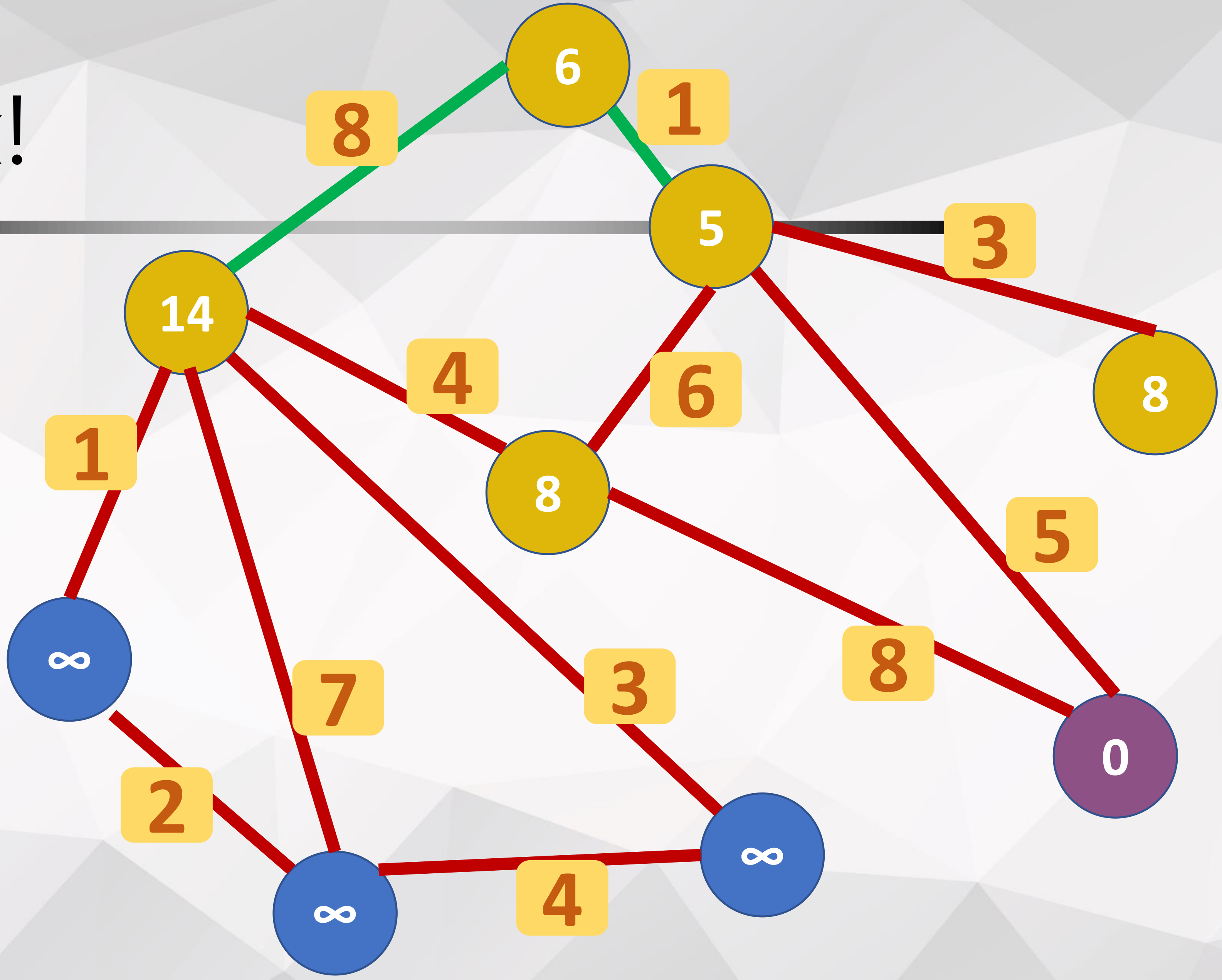
拜訪鄰點



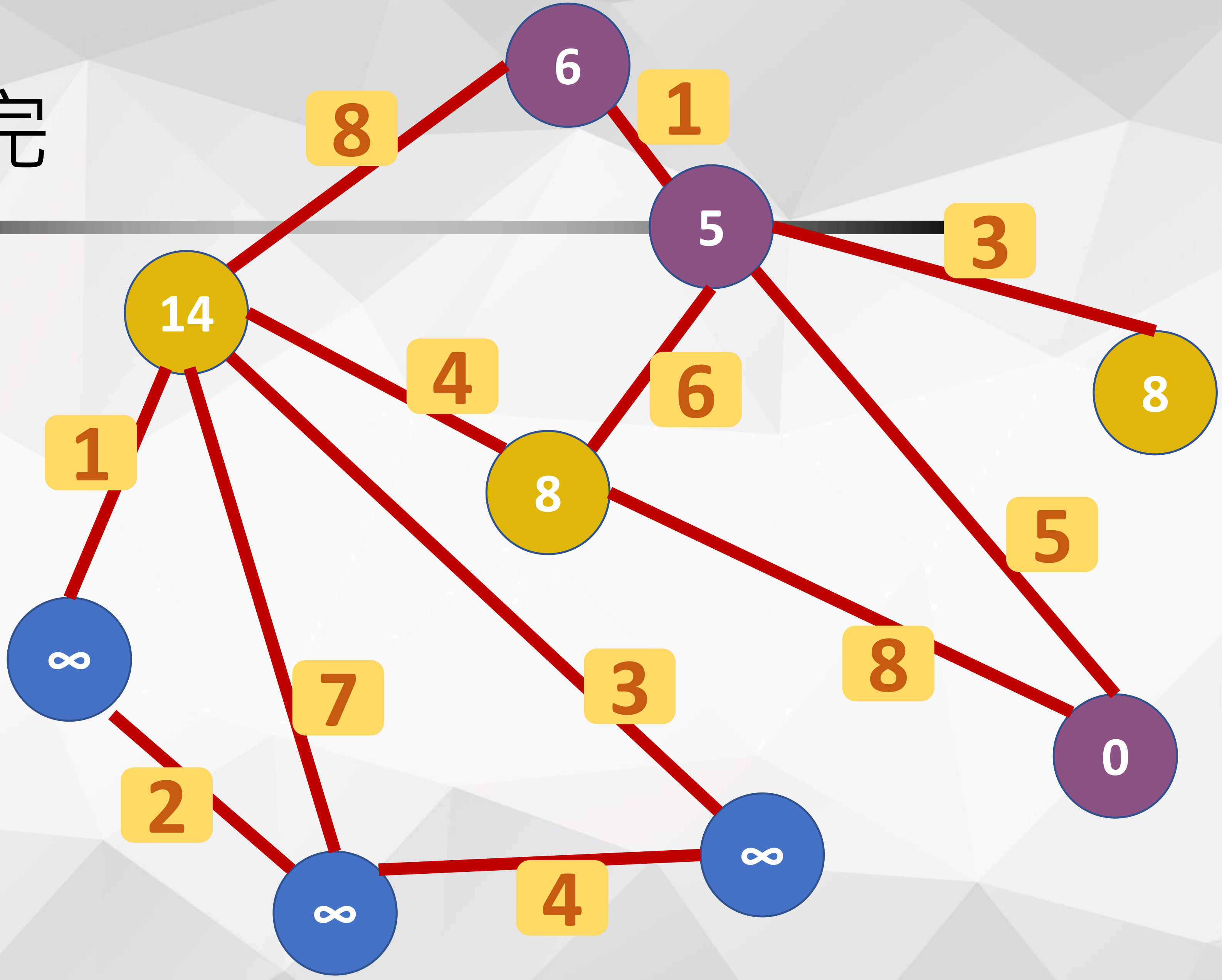
Relax?



Relax!



拜訪完



關於 relaxation

6

5

這些**拜訪完**的節點

在**未來**有可能再被 relax 嗎?

0

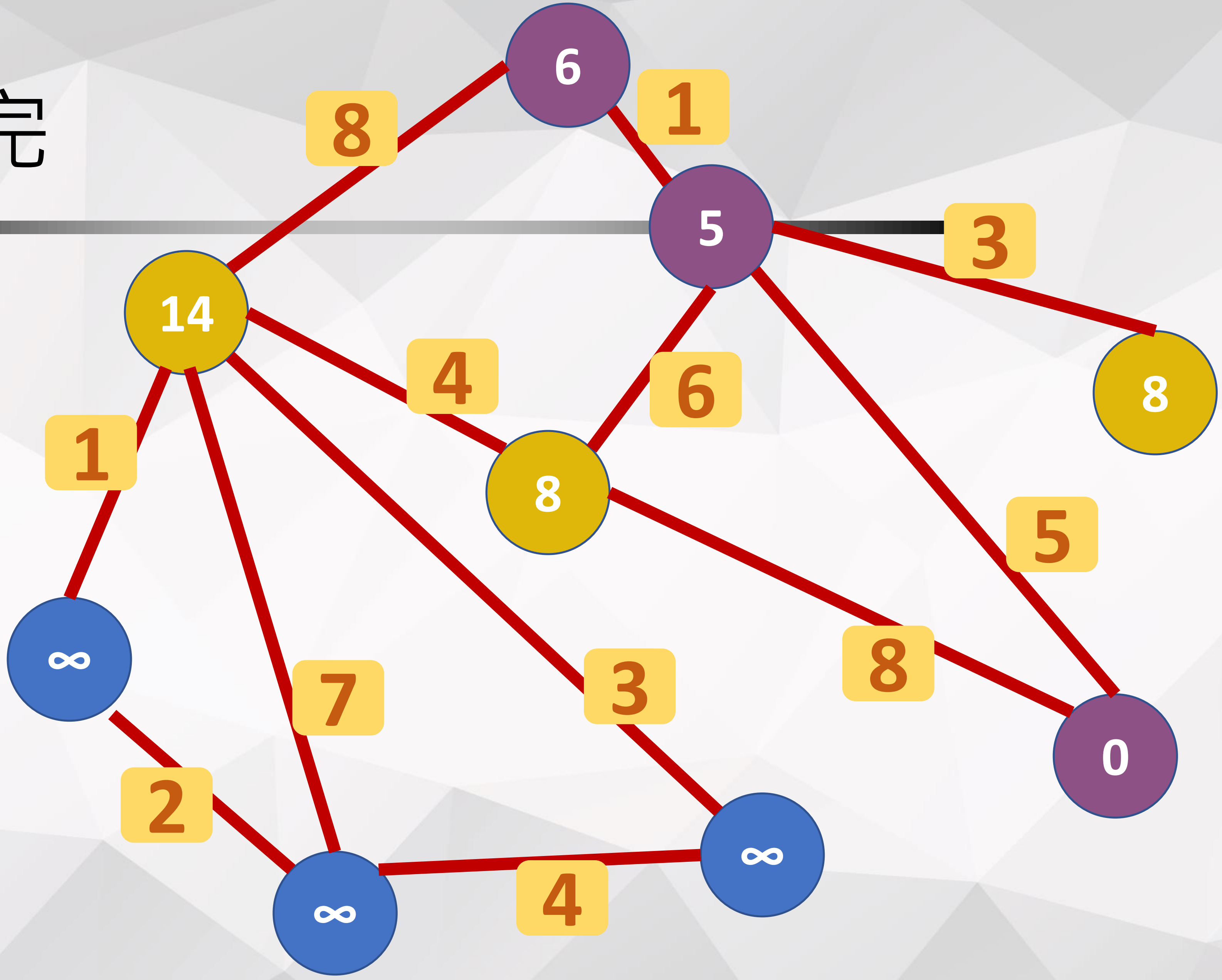
關於 relaxation

這些**拜訪完**的節點

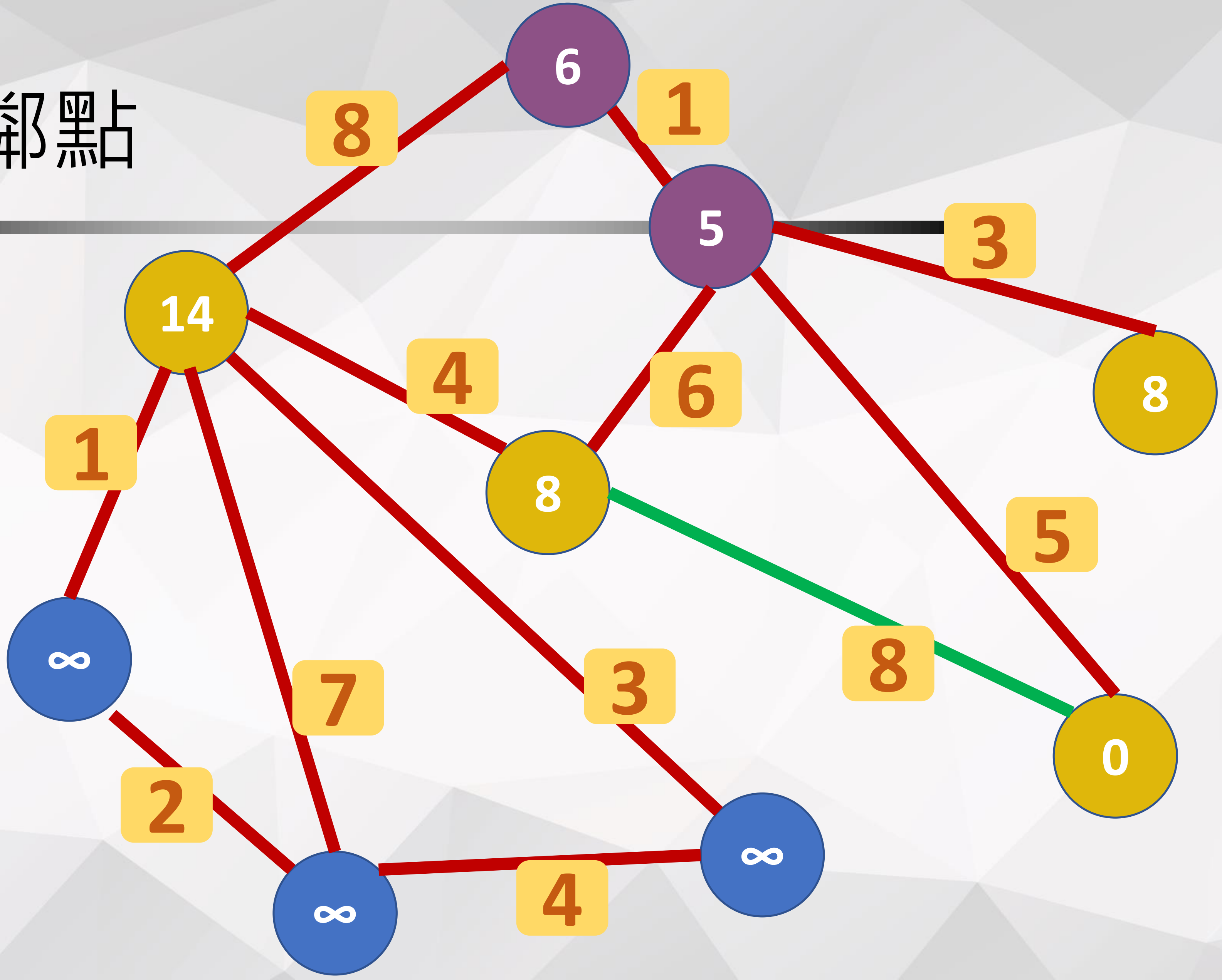
在**未來**有可能再被 relax 嗎?

若答案為**否定**的，
這似乎叫做: **無後效性**

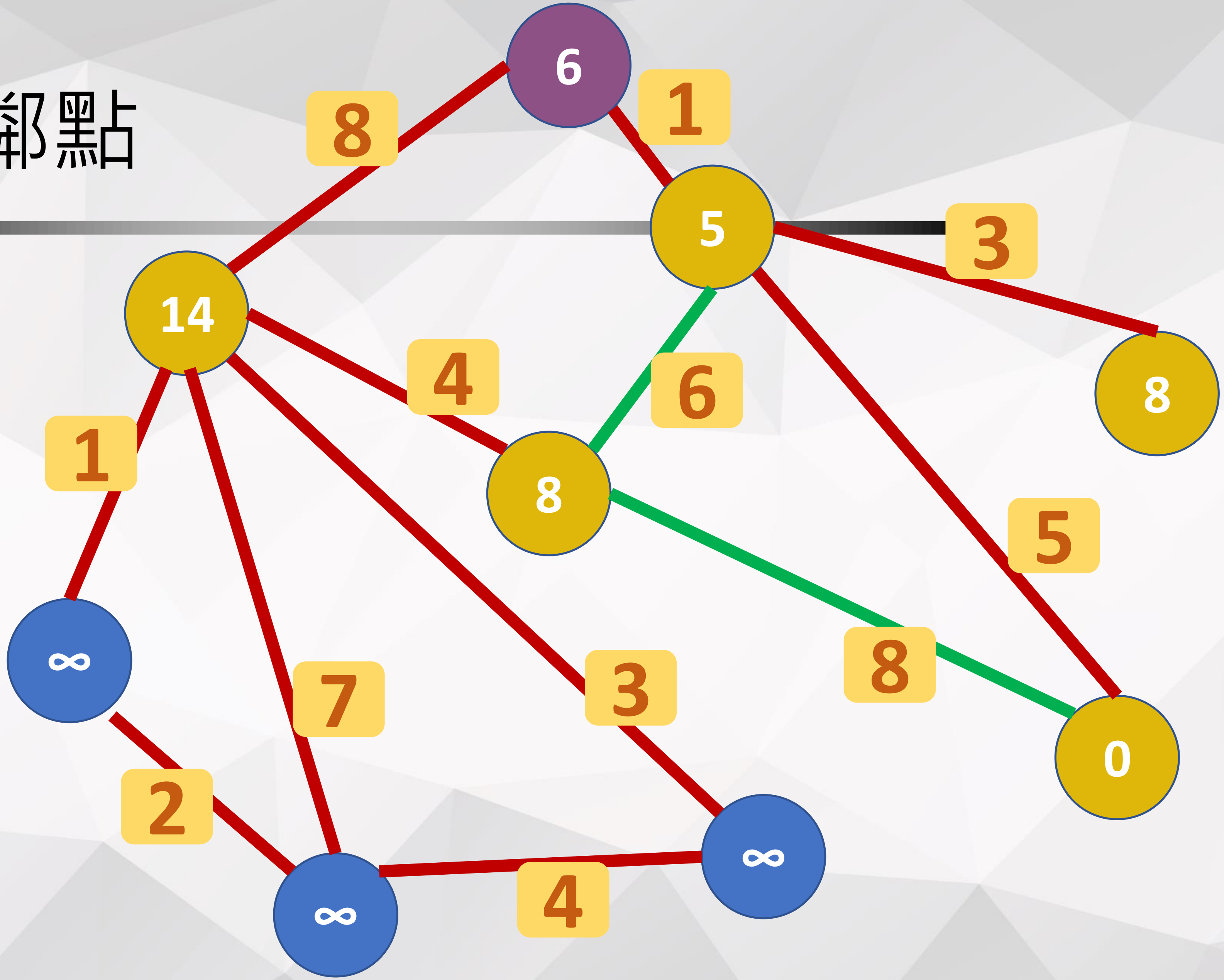
拜訪完



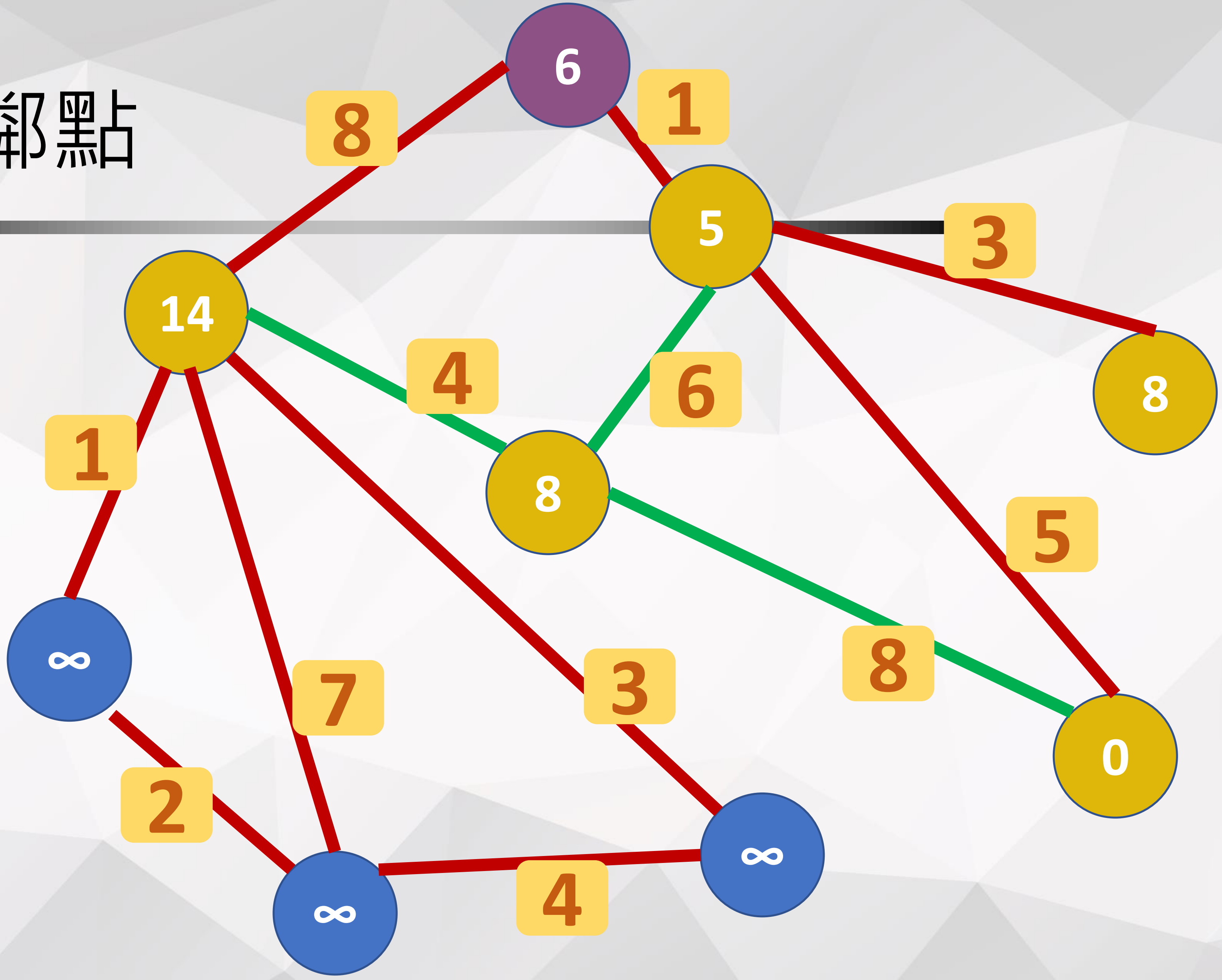
拜訪鄰點



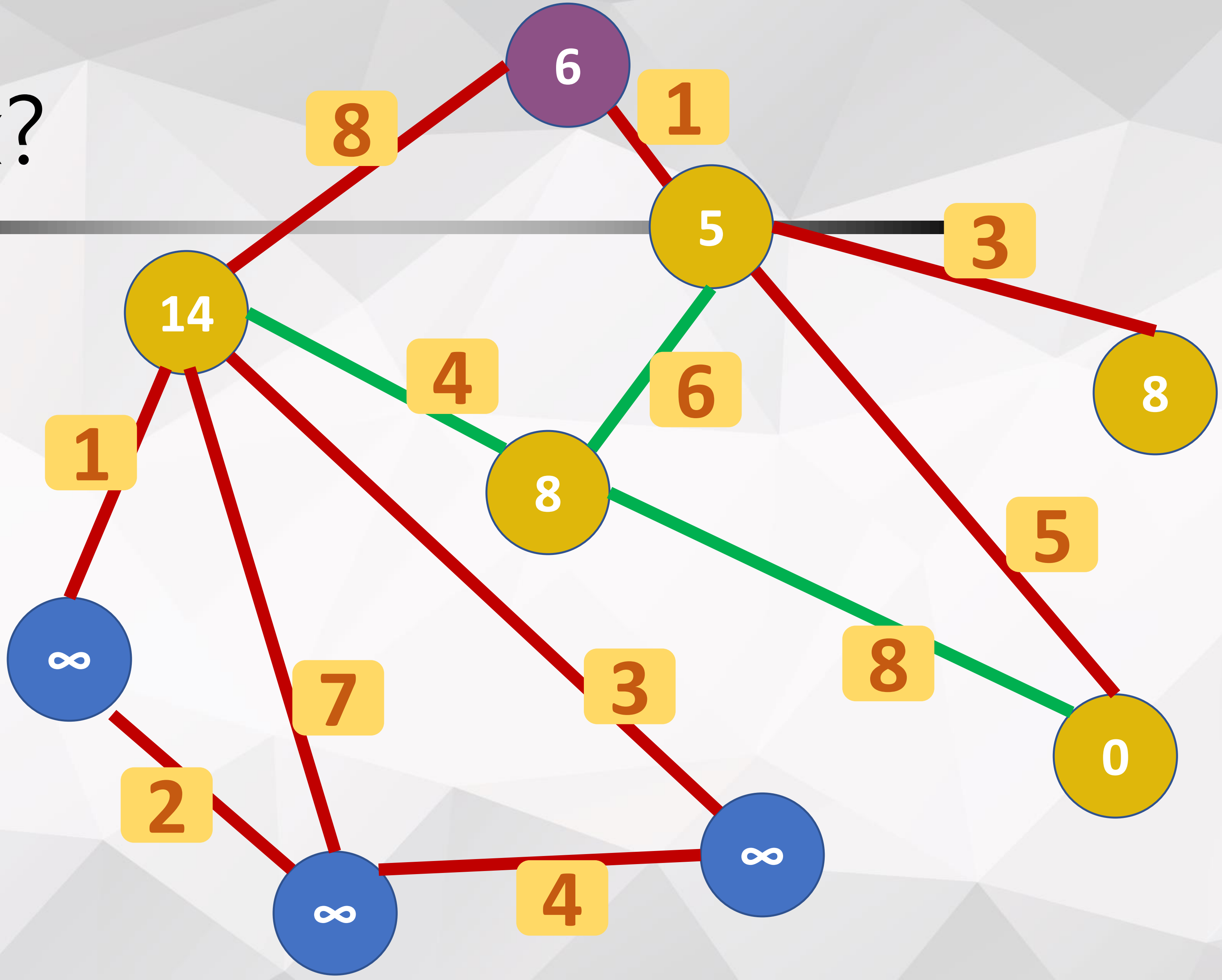
拜訪鄰點



拜訪鄰點



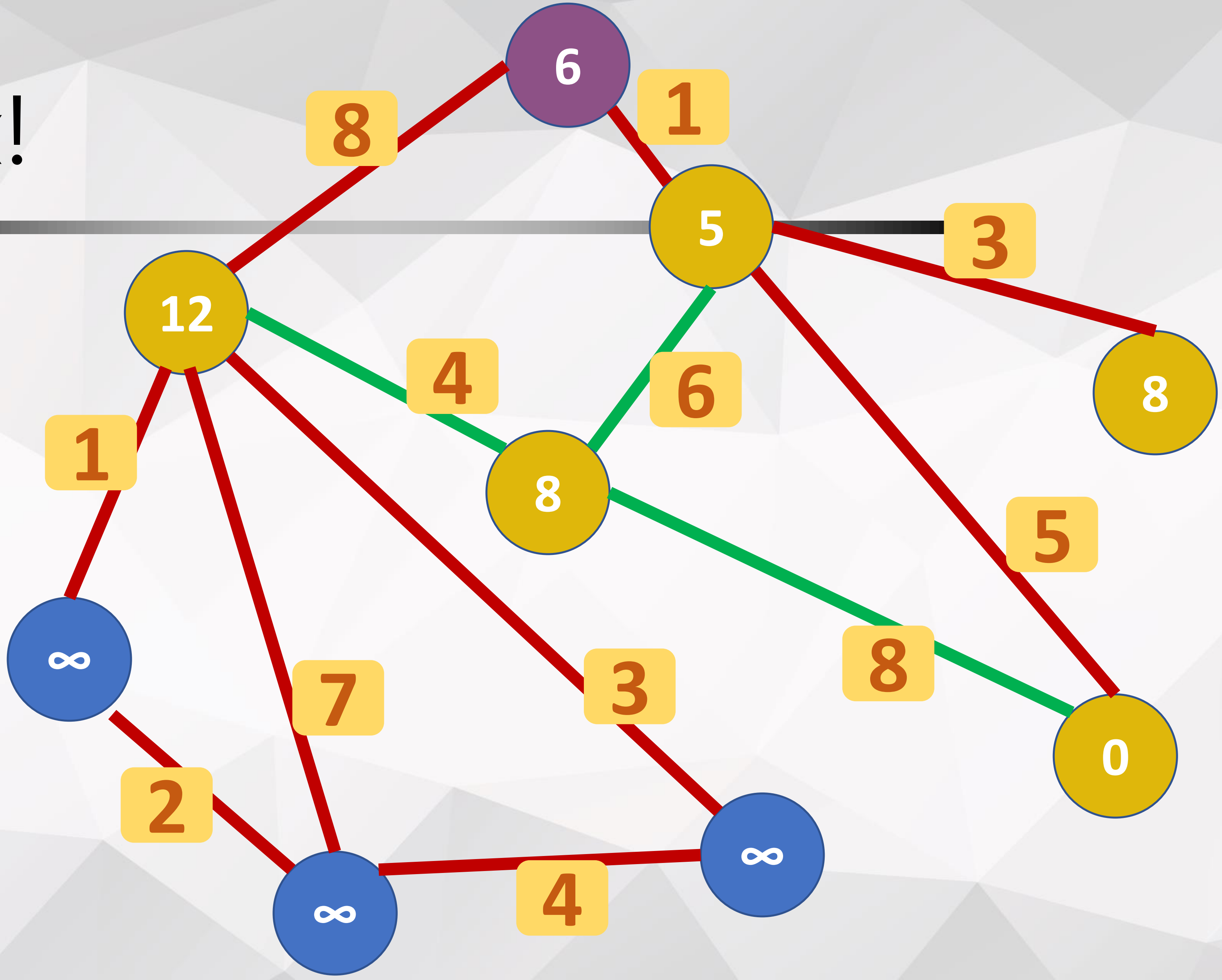
Relax?



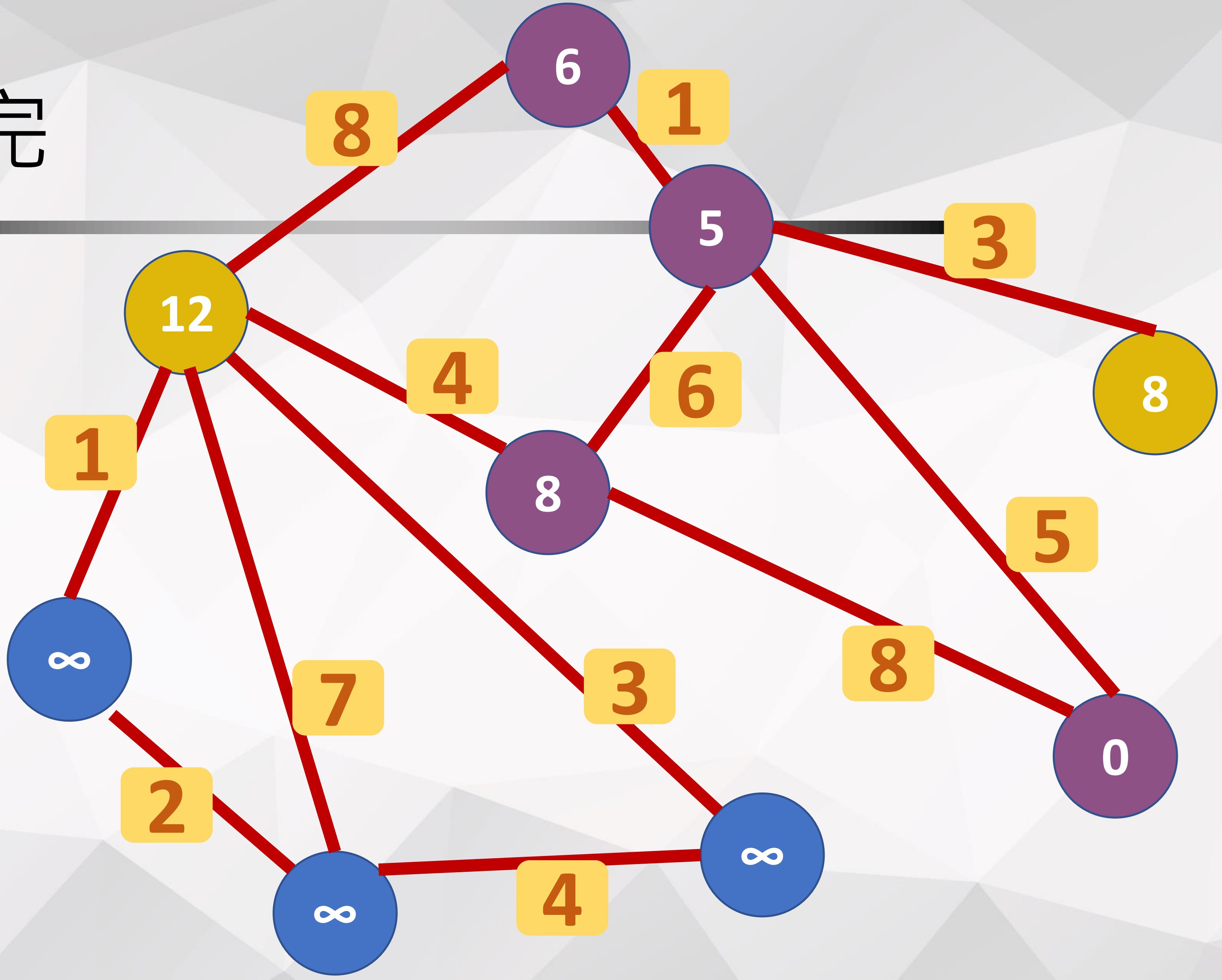
Competitive Programming

Made by 培訓團隊群 CCNS

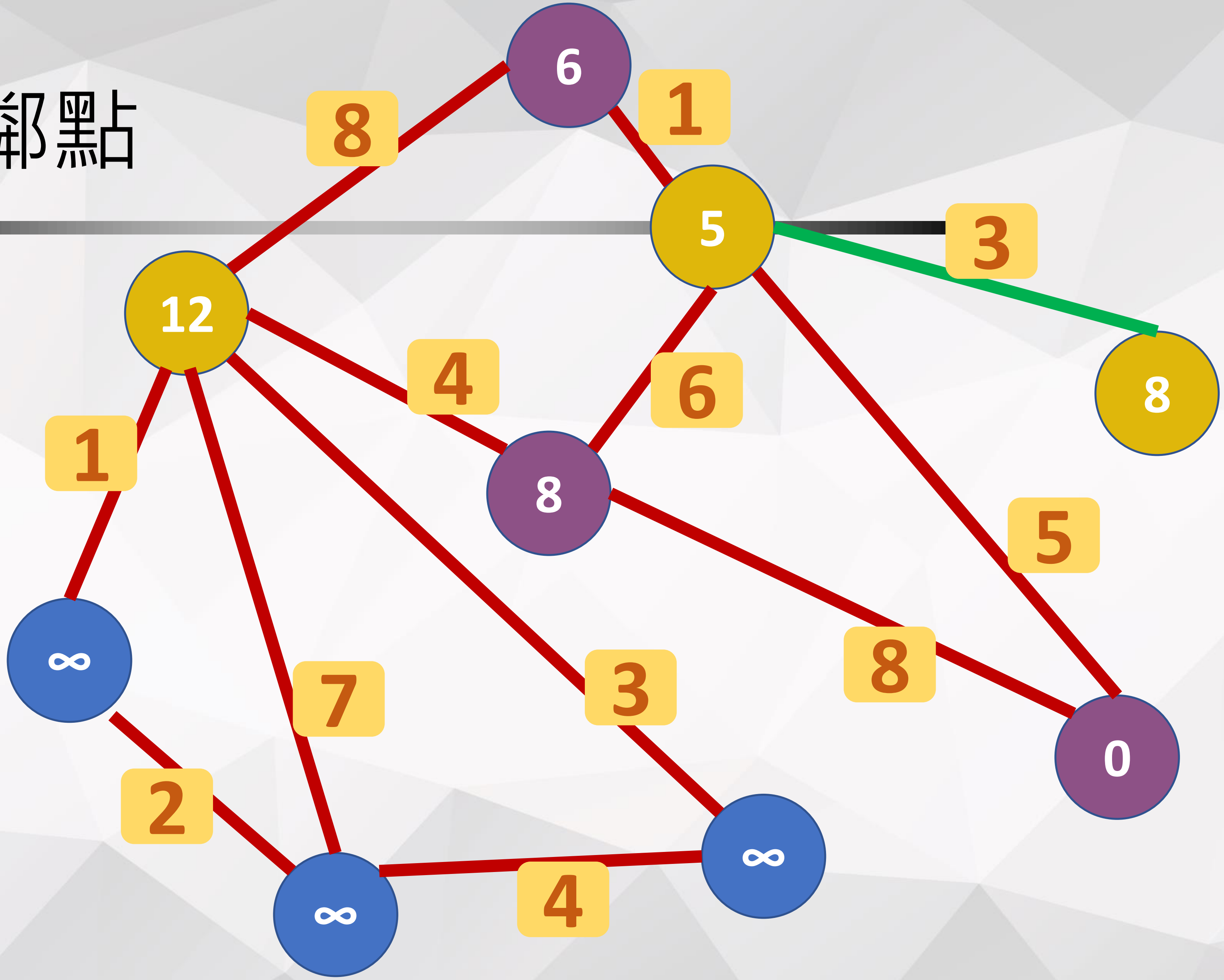
Relax!



拜訪完

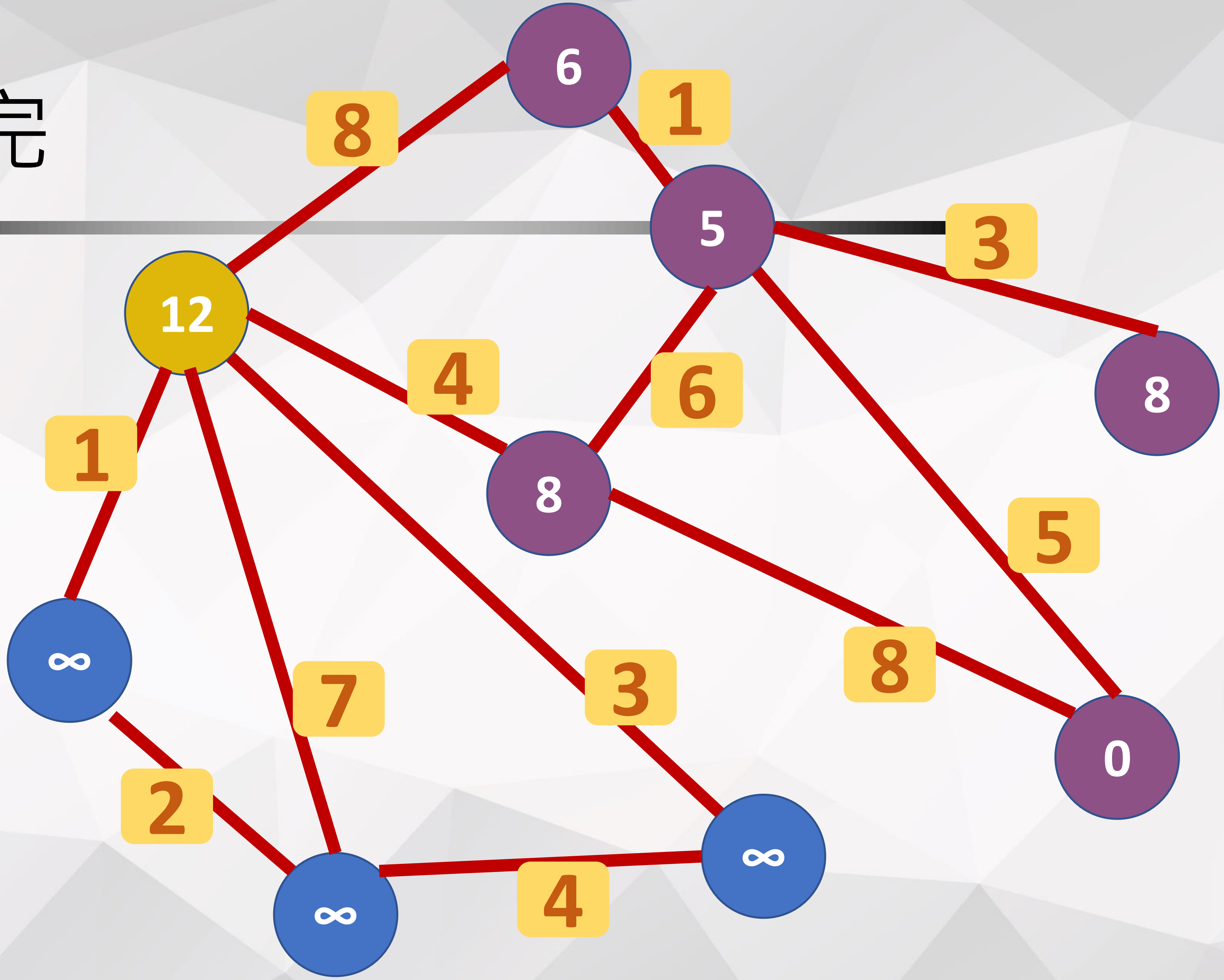


拜訪鄰點

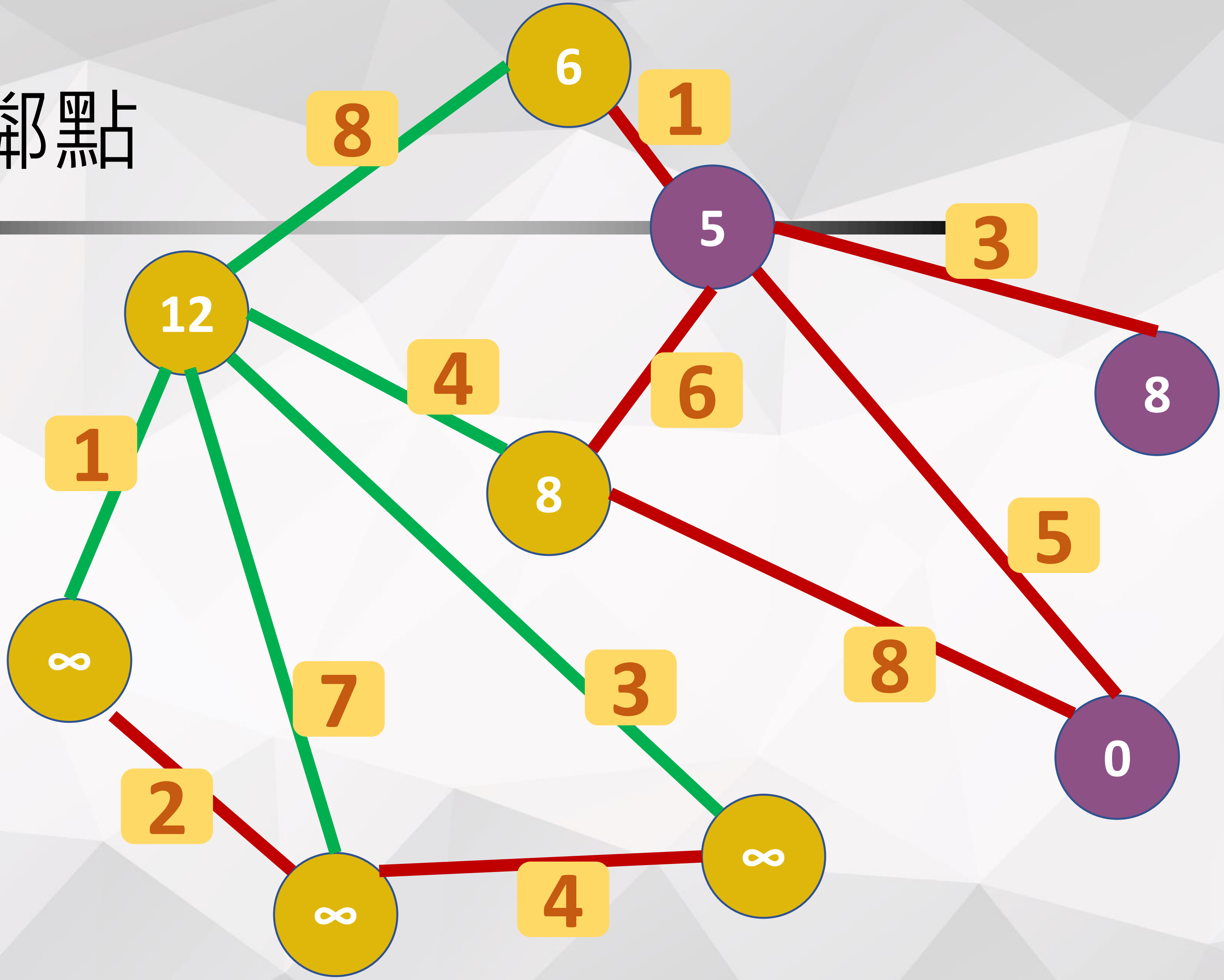


Competitive Programming

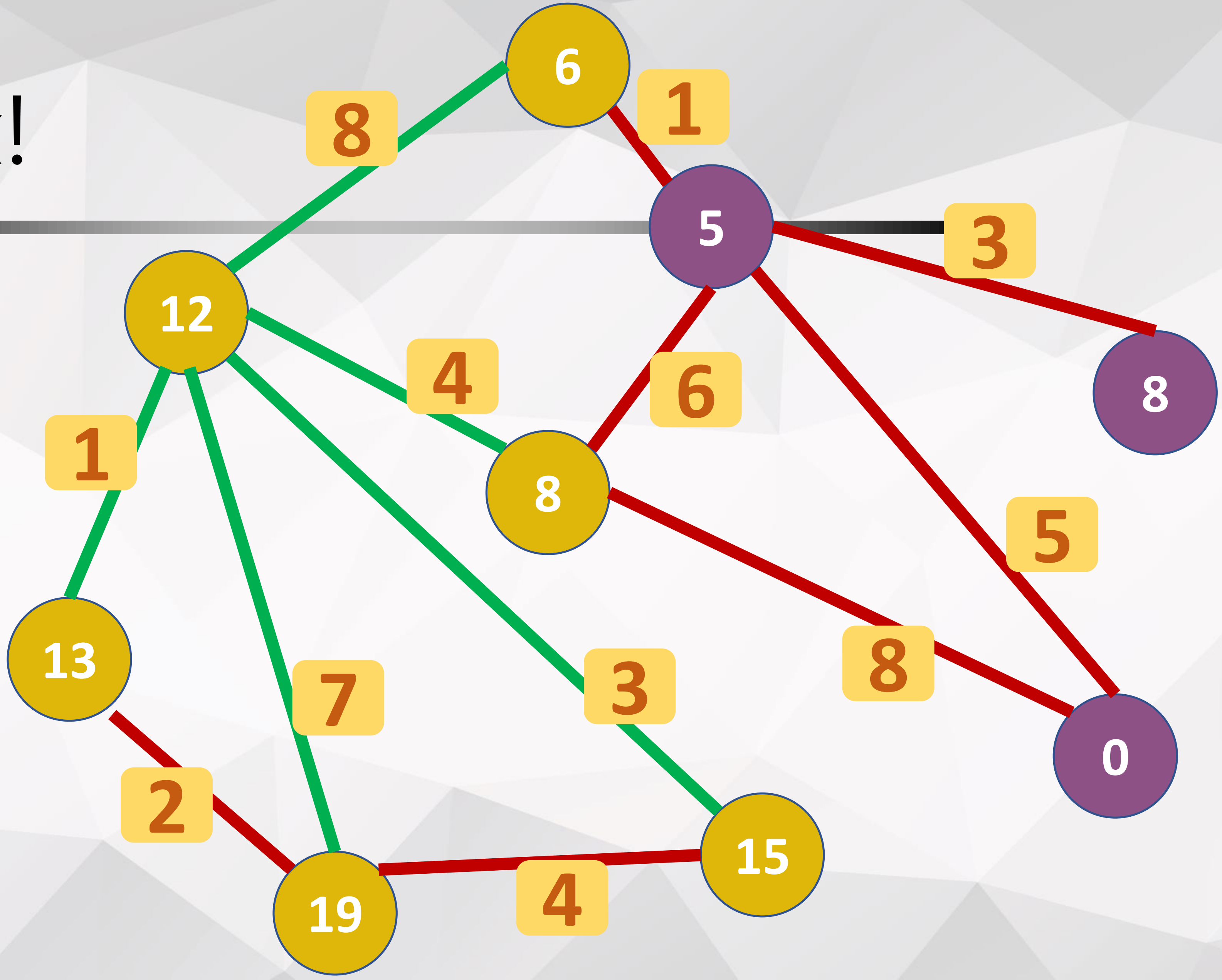
拜訪完



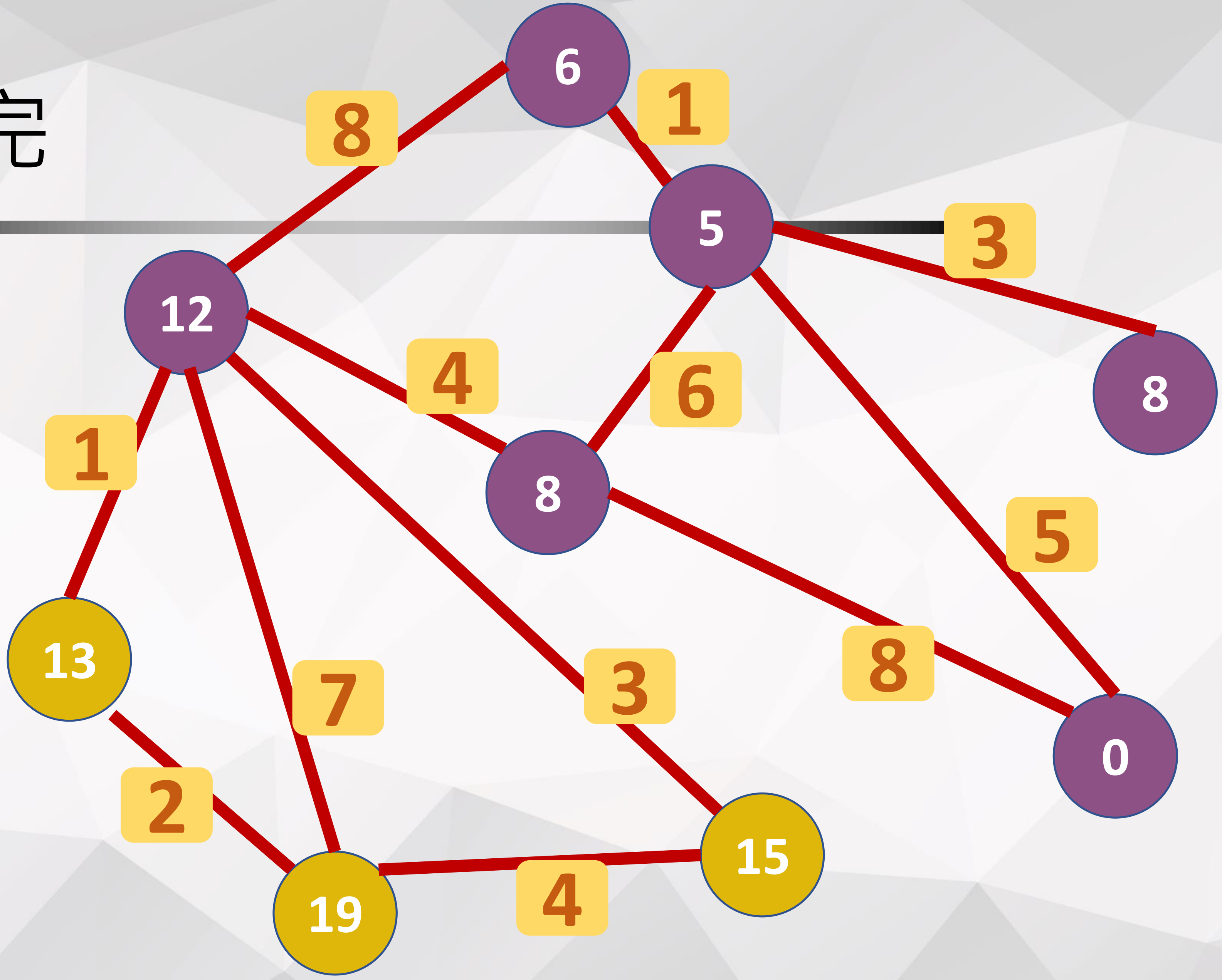
拜訪鄰點



Relax!

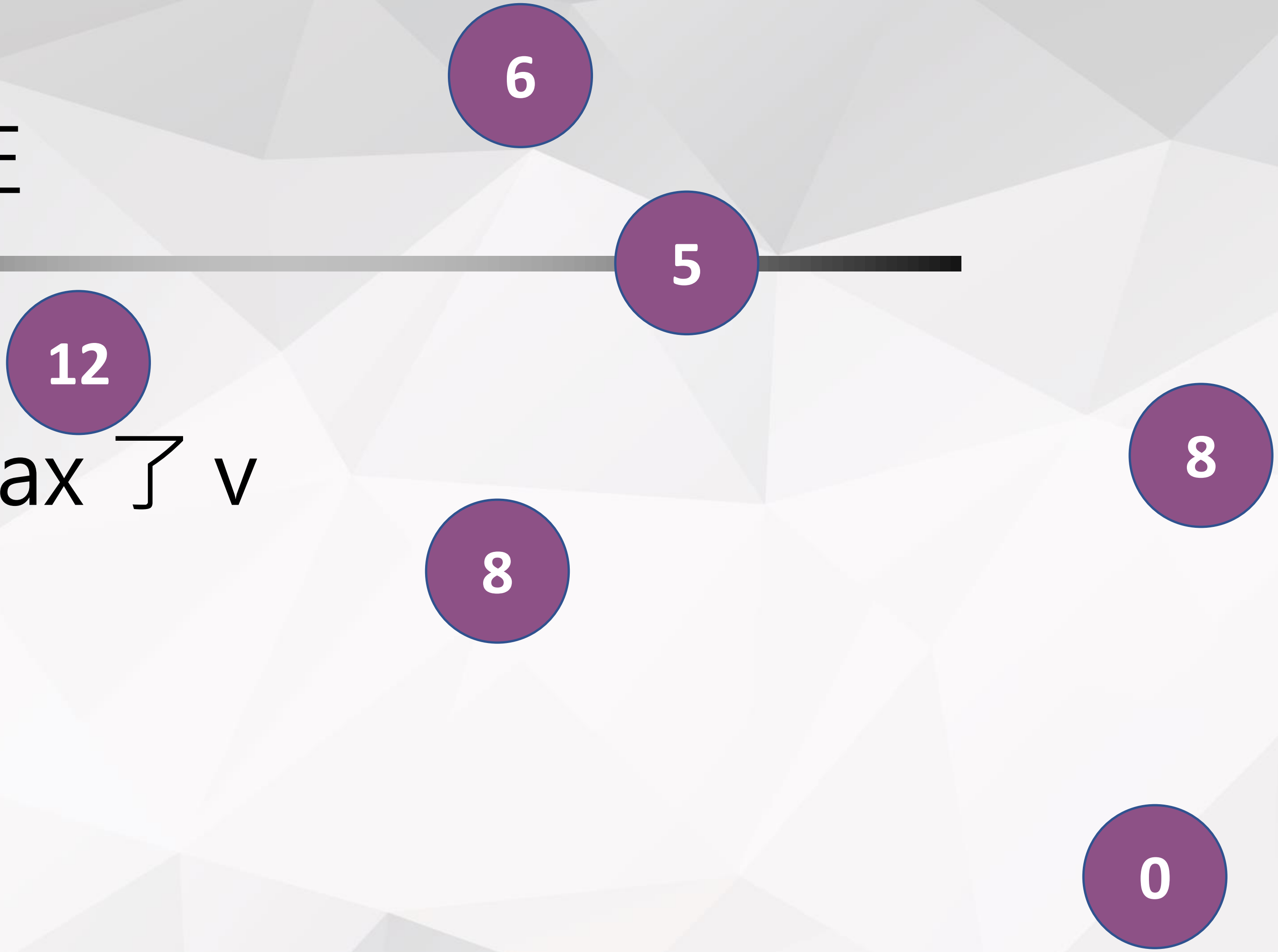


拜訪完



無後效性

若 u 去 relax 了 v



無後效性

若 u 去 relax 了 v

則未來不管如何，

v 不可能更新 u

無後效性

若 u 去 relax 了 v
則未來不管如何，

v 不可能更新 u

因為 u 先來的

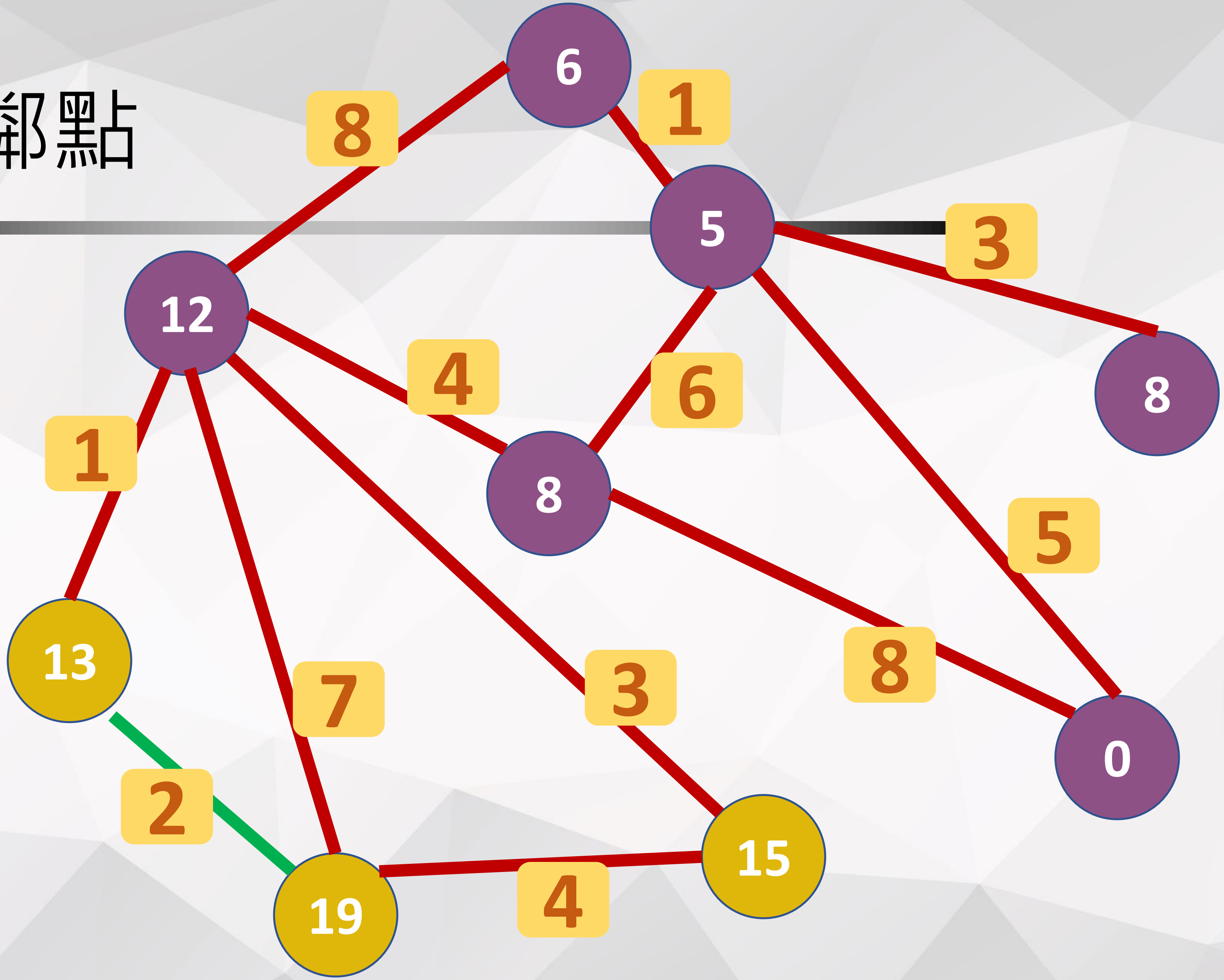
無後效性

若 u 去 relax 了 v
則未來不管如何，

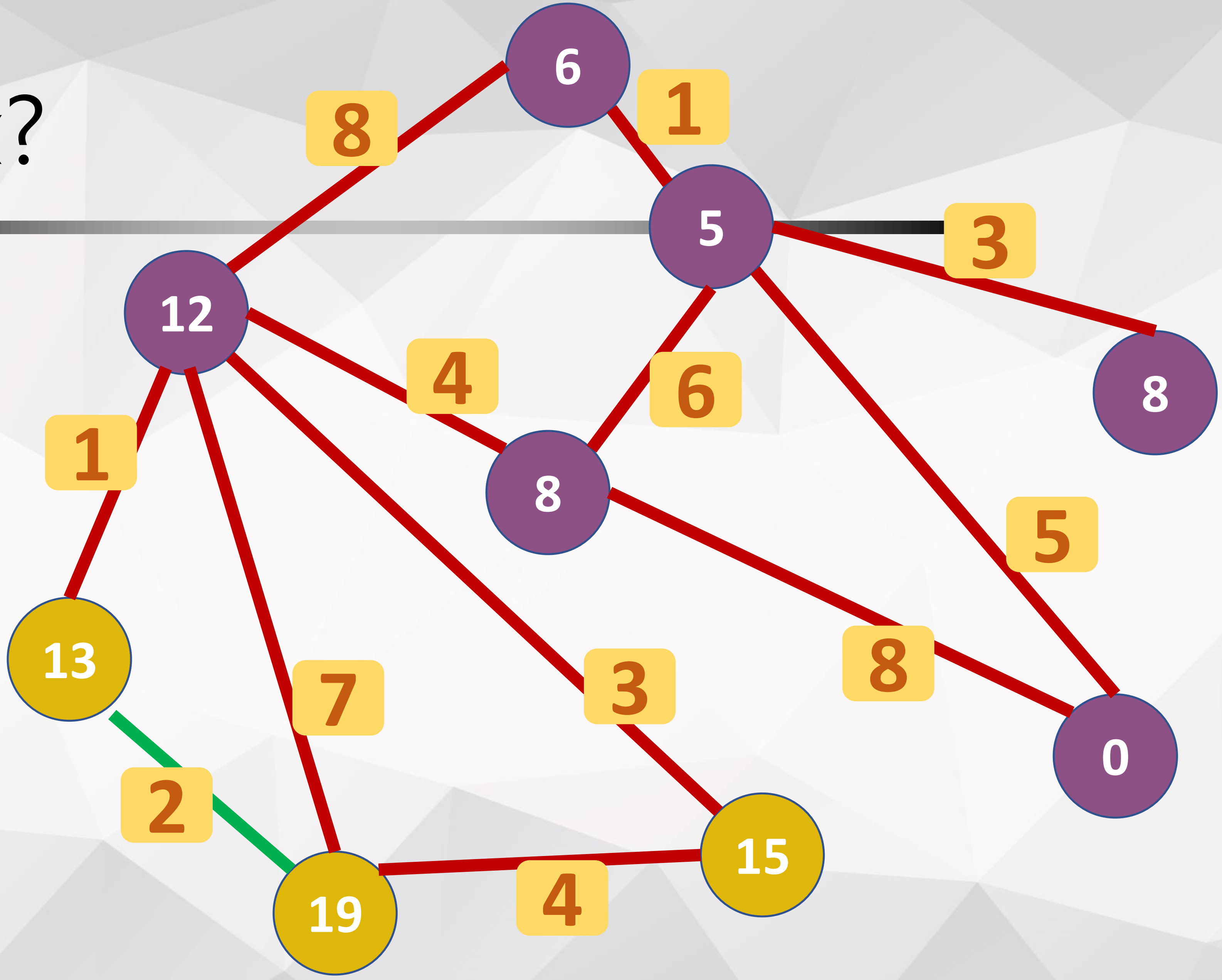
v 不可能更新 u

因為 u **先來的**，在 u 之後挑的**任何點**
其值都比 u 大，並且**邊權重恆正**！
怎麼**加**都比 u 大

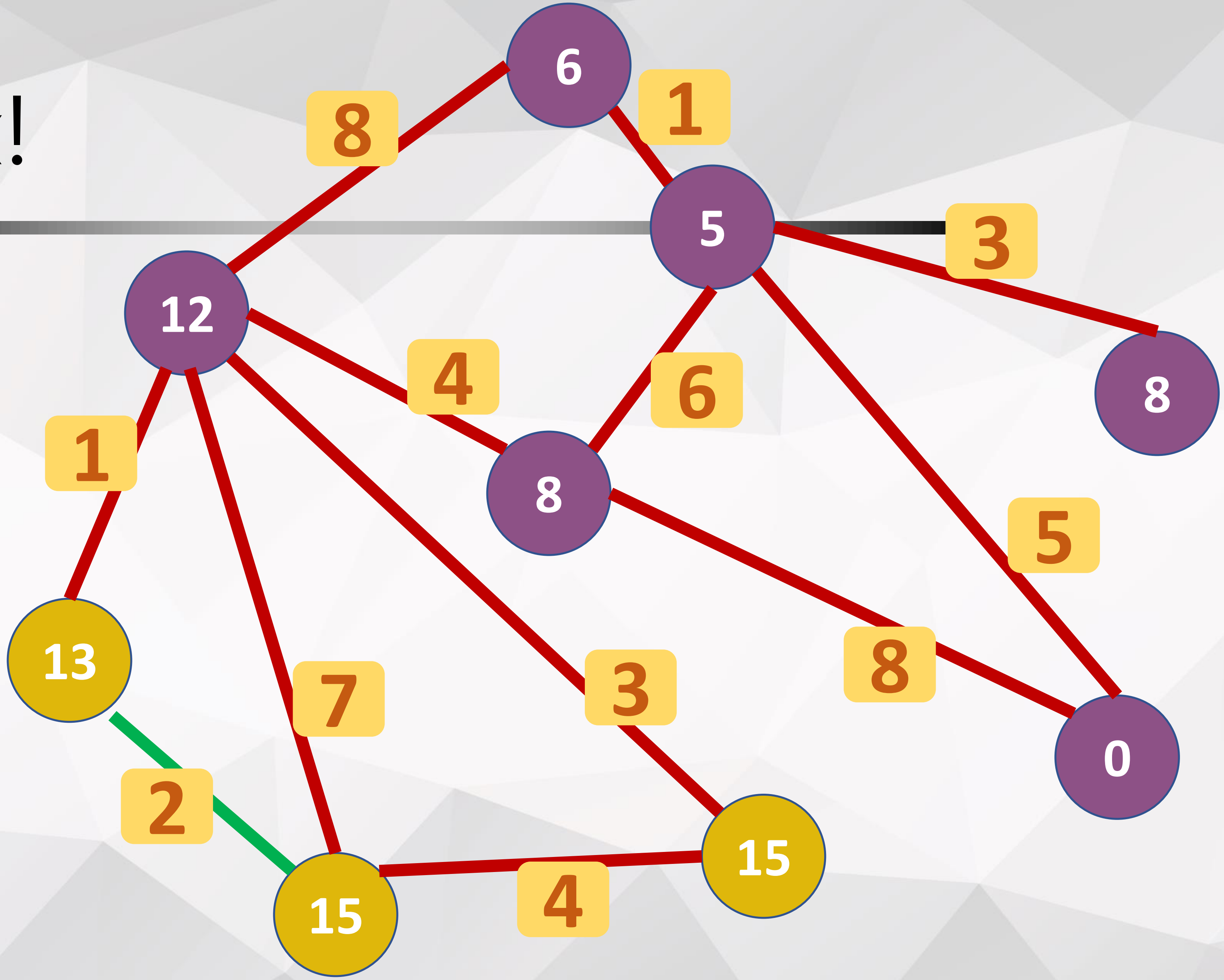
拜訪鄰點



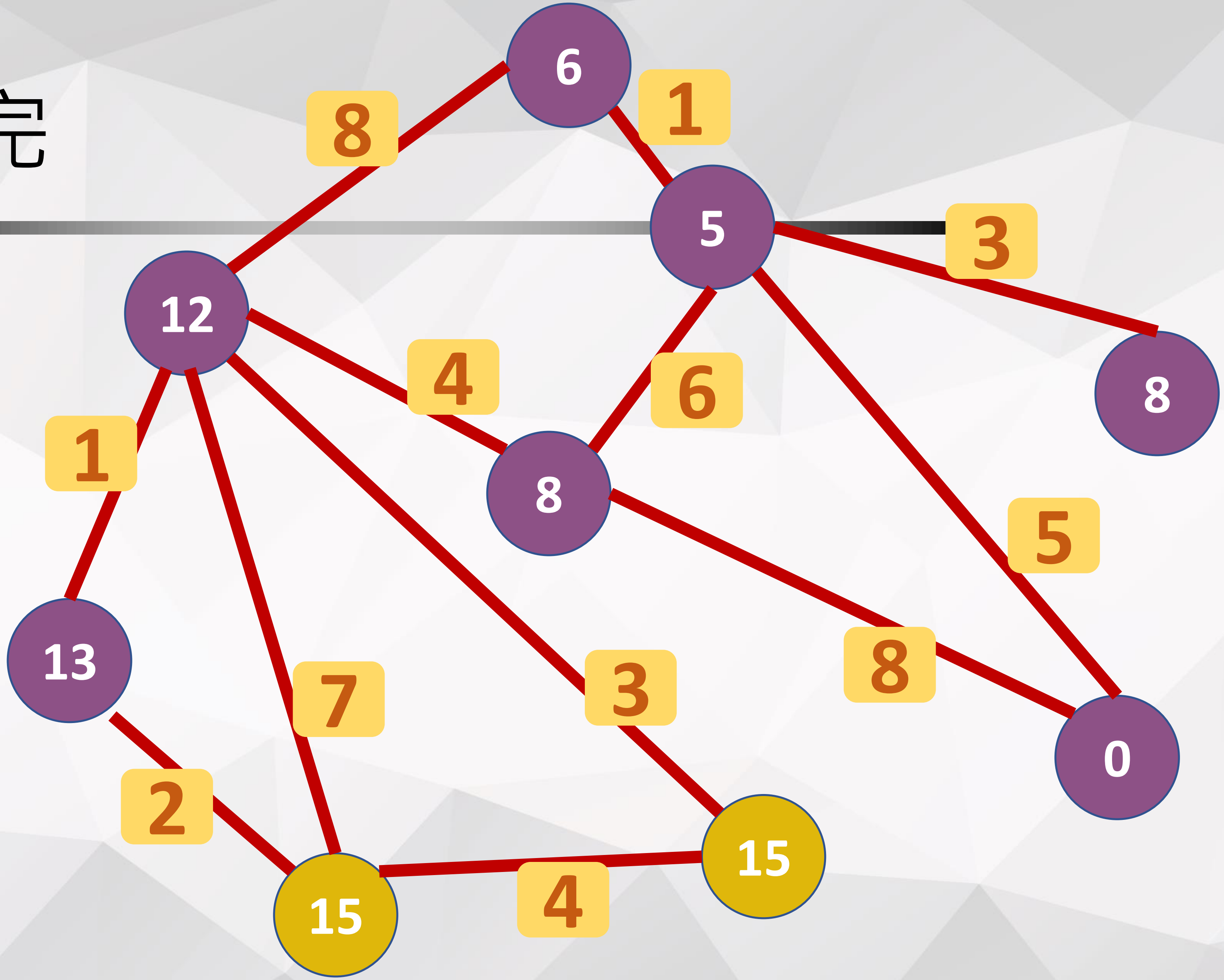
Relax?



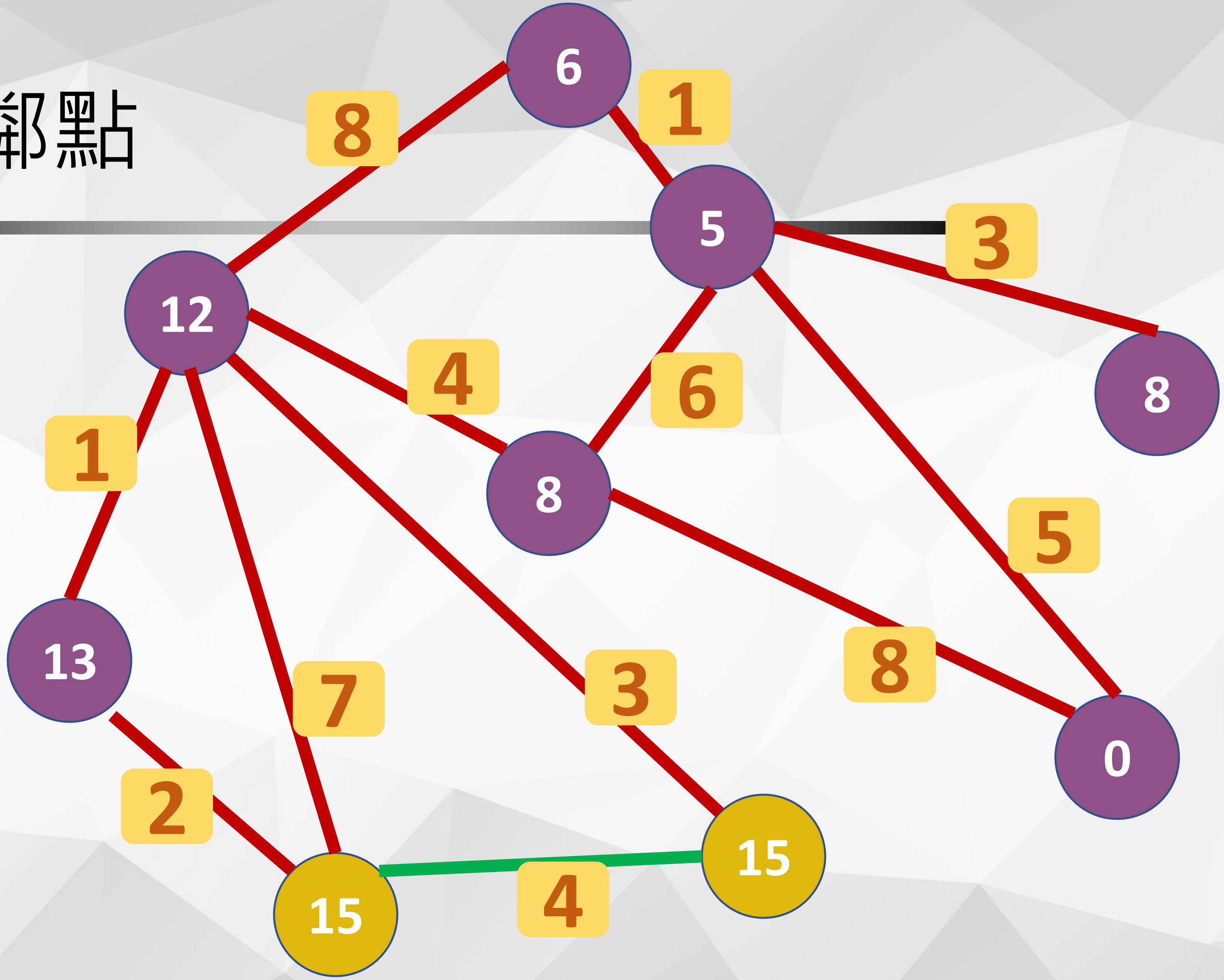
Relax!



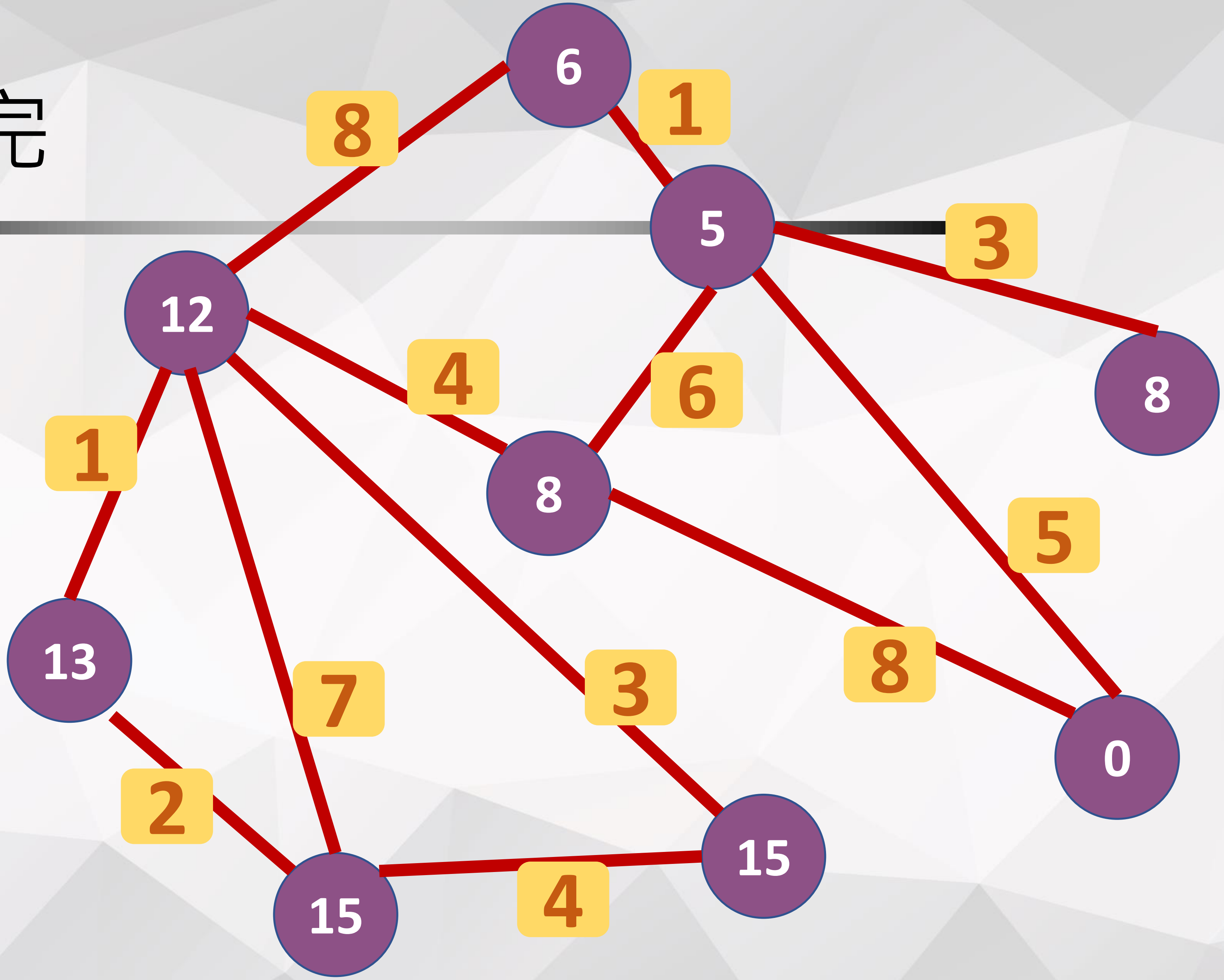
拜訪完



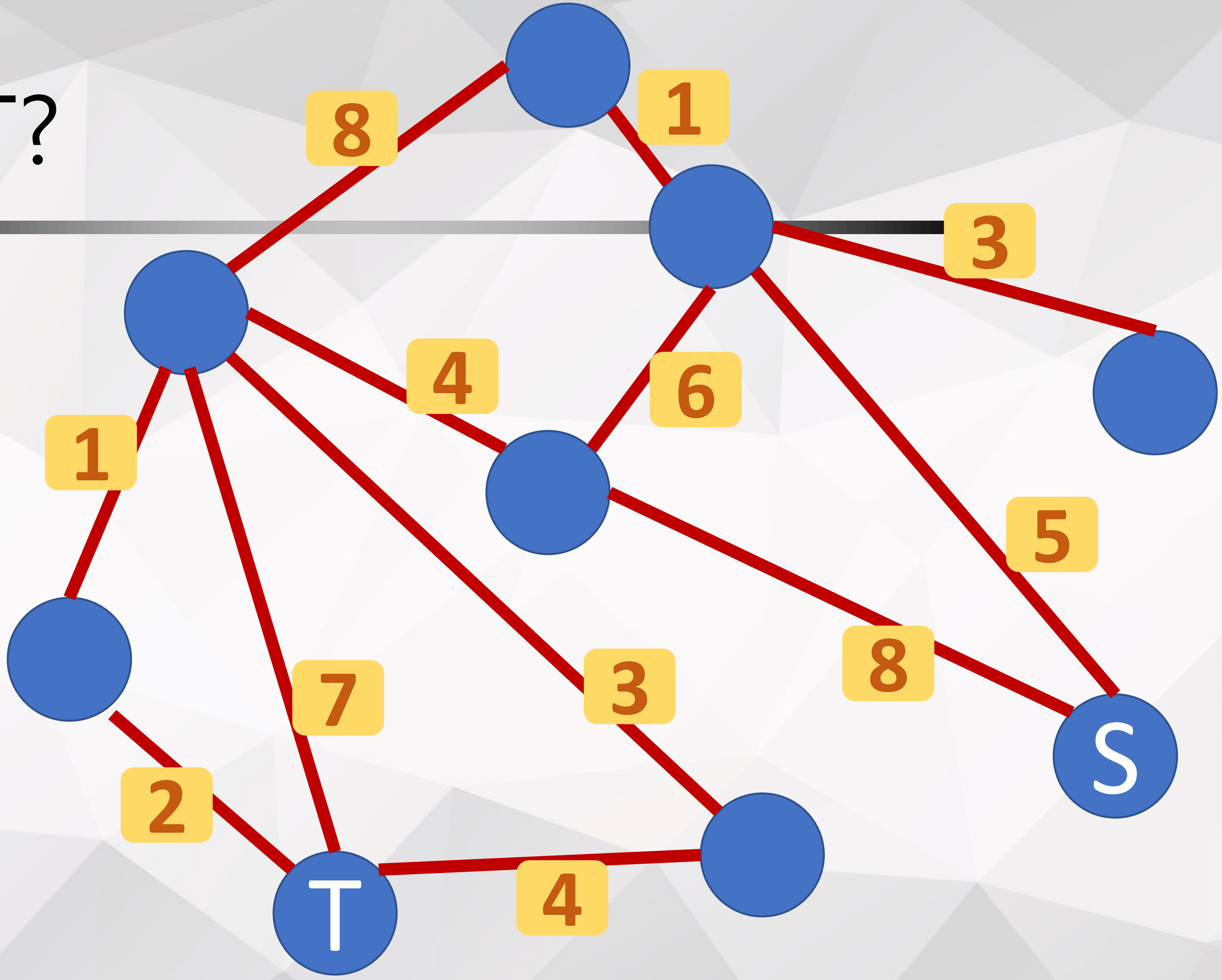
拜訪鄰點



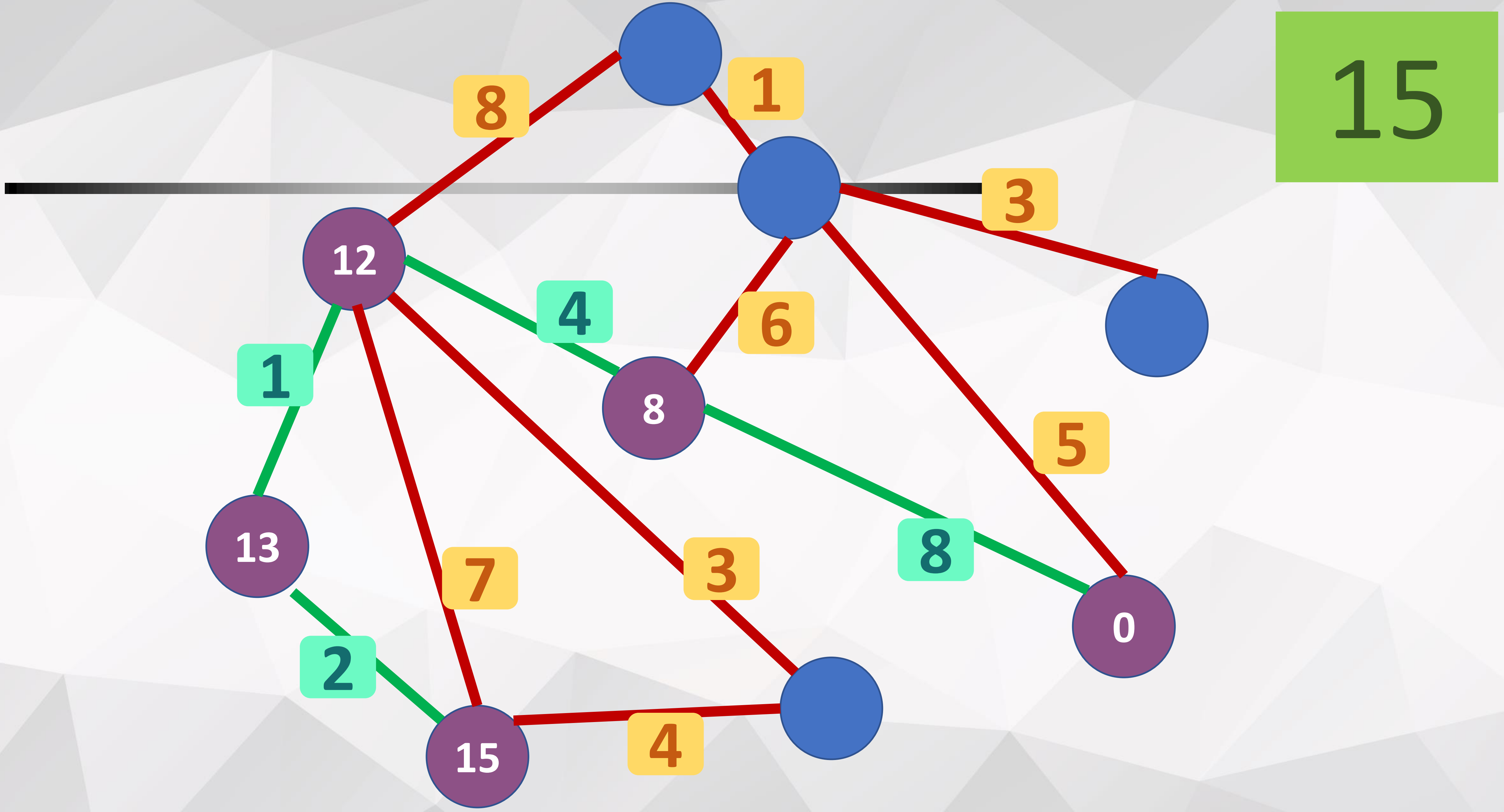
拜訪完



S 到 T?



0


15
8
1
3
12
4
6
1
8
5
13
7
3
8
0
2
15
4

Questions?

練習

- [POJ 3255 Roadblocks]

單源最短路徑

- Relaxation
- Dijkstra's algorithm
- Bellman-Ford's algorithm

Bellman-Ford's algorithm

Bellman-Ford 實作

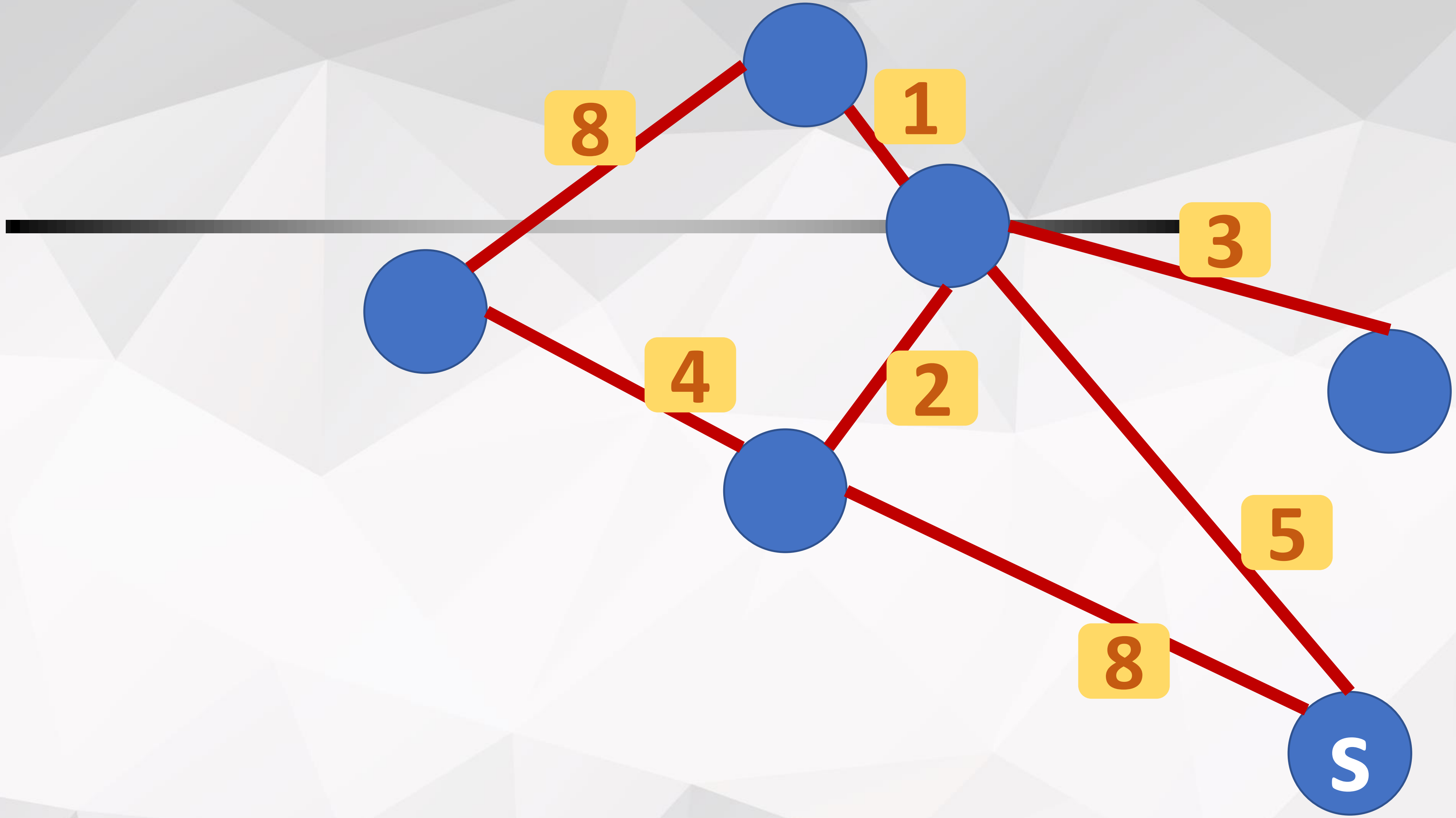
```
vector<edge> E;
    :
    .

/* 假設輸入完邊的資訊了 */
```

Bellman-Ford 實作

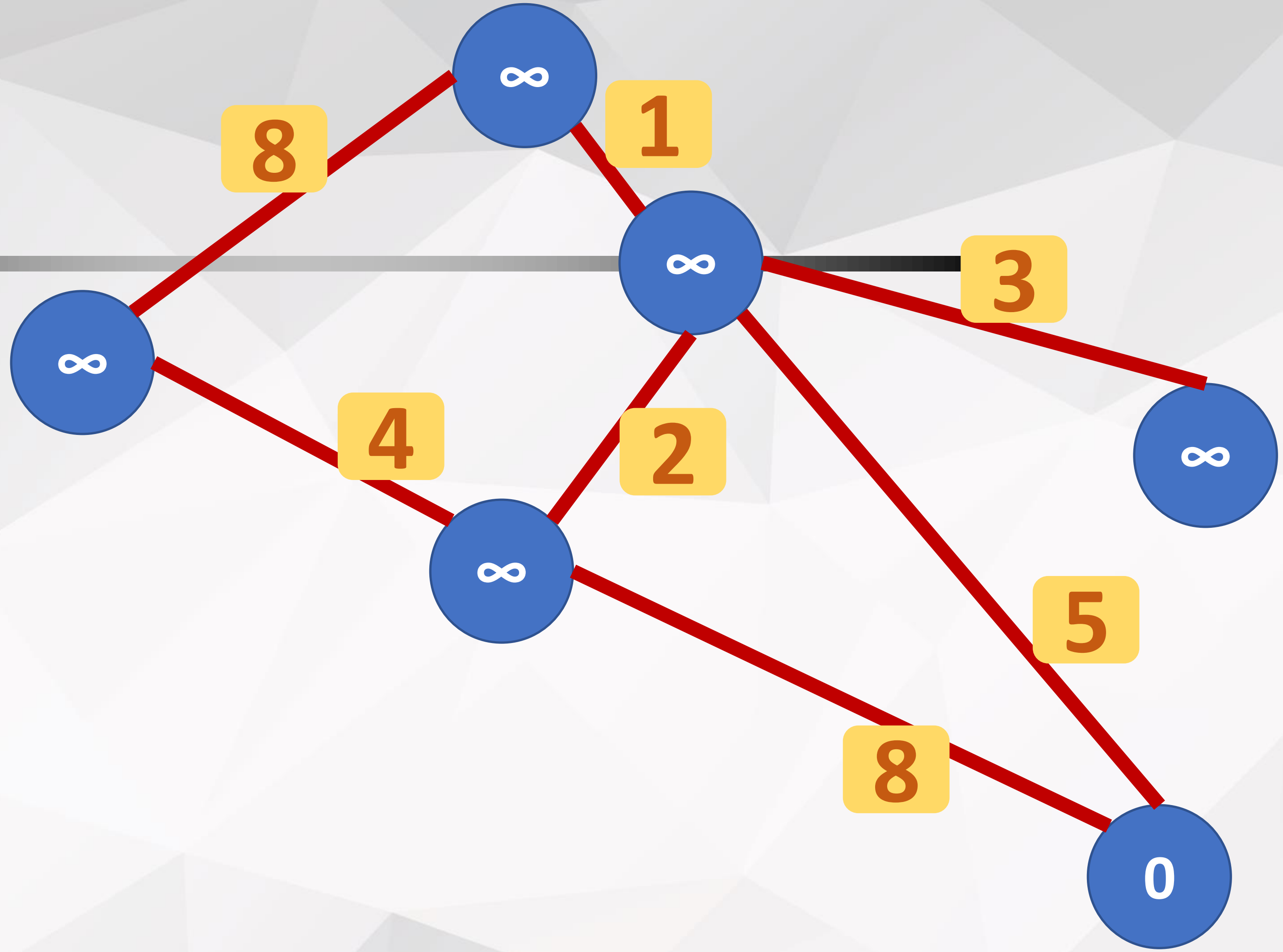
```
memset(s, 0x3f, sizeof(s)); // 初始無限大

s[source] = 0;
```

Bellman-Ford 實作

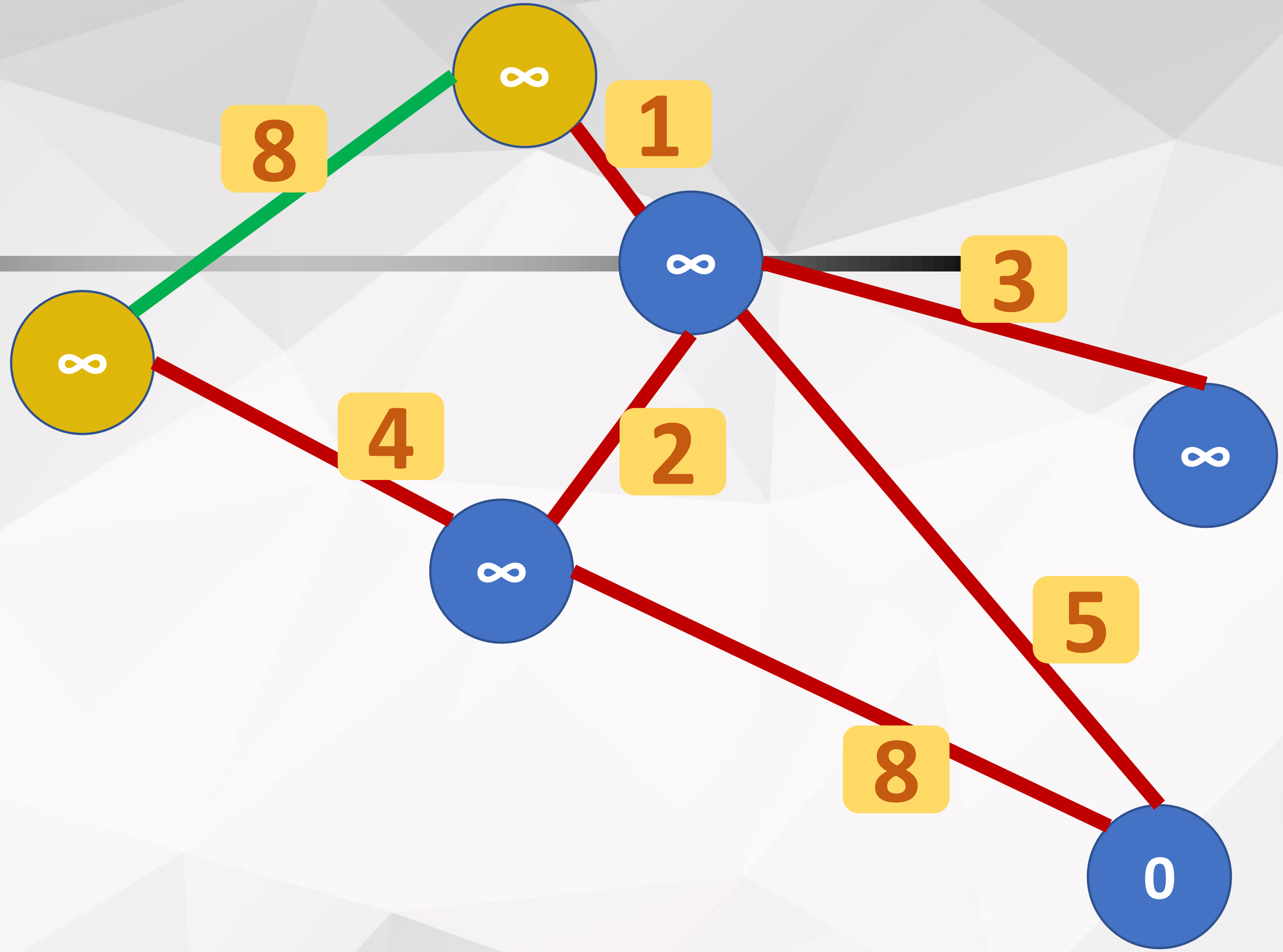
```
for (int i = 0; i < V.size(); i++)
    for (edge e: E)
        s[e.v] = min(s[e.v], s[e.u] + e.w);
```


8
1
3
4
2
5
8
S

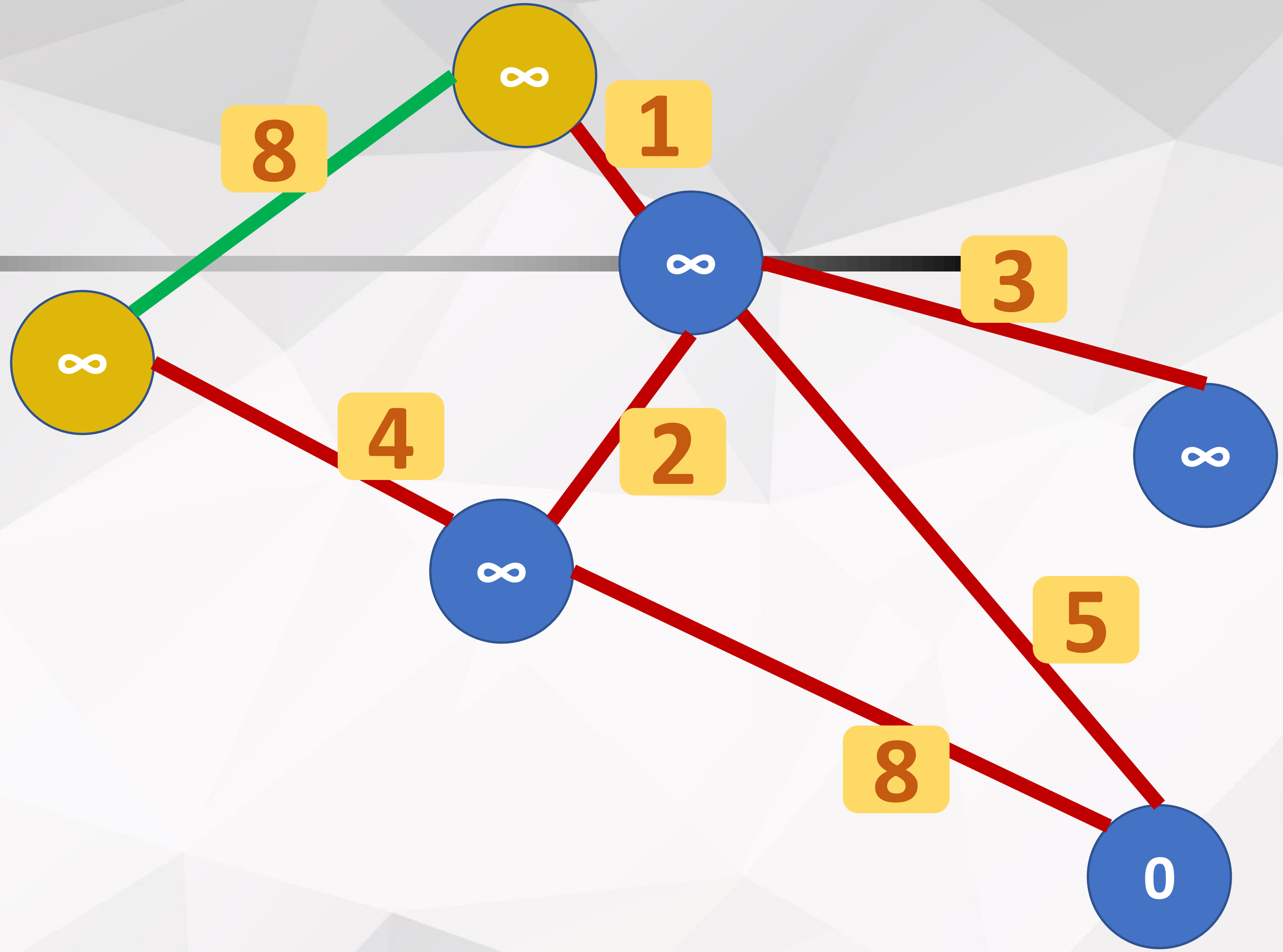
初始化

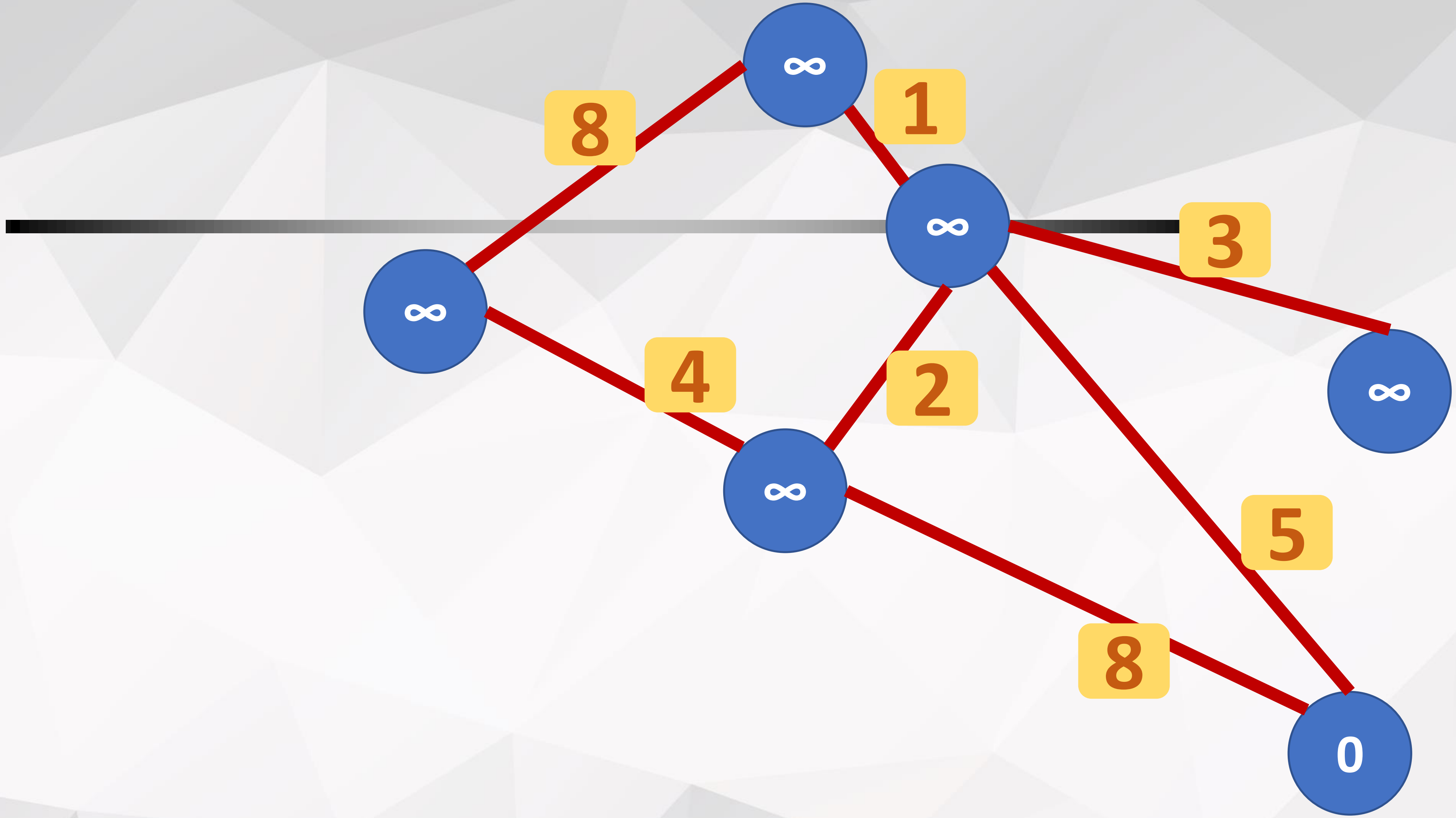


Relax?

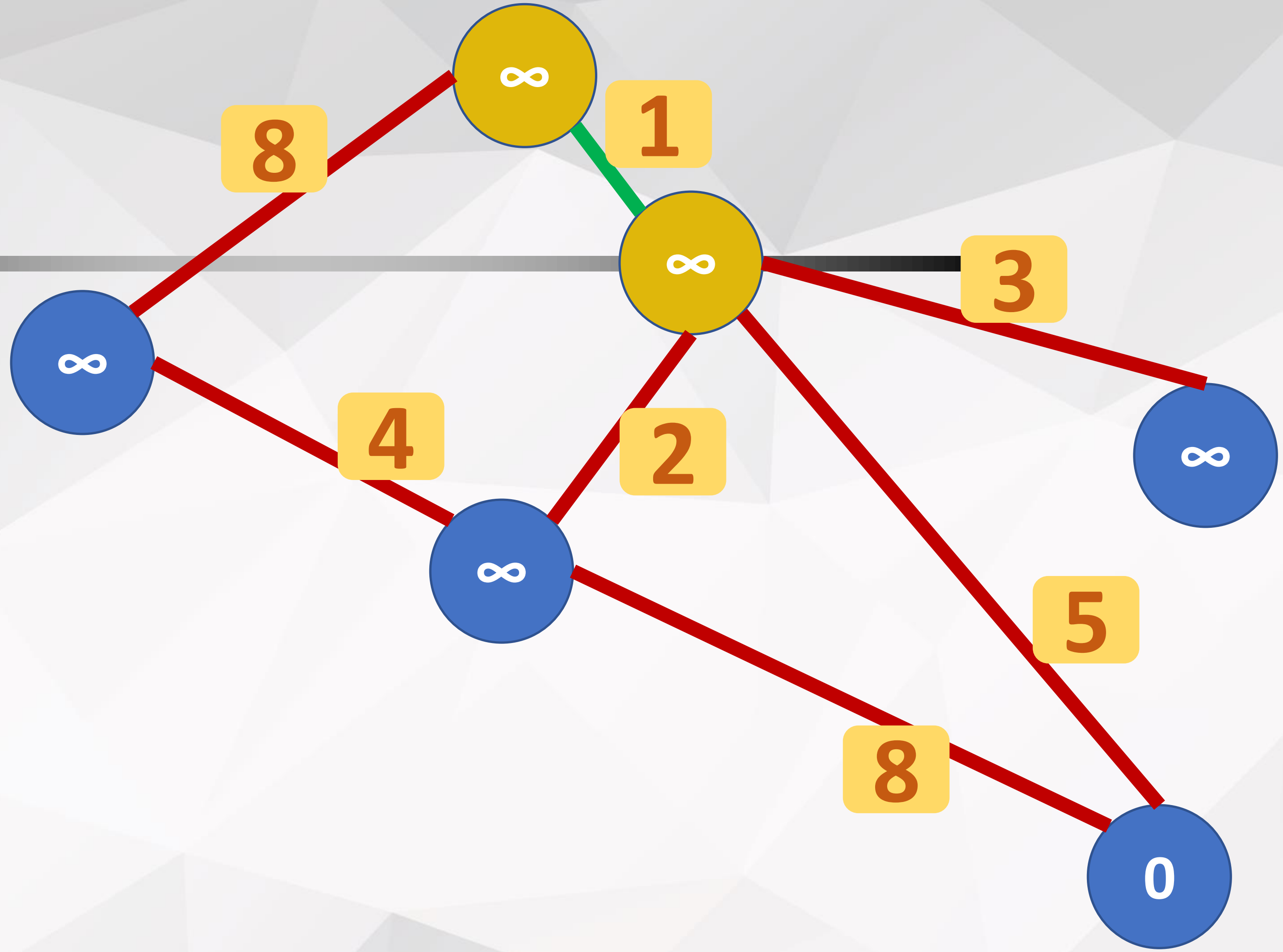


No

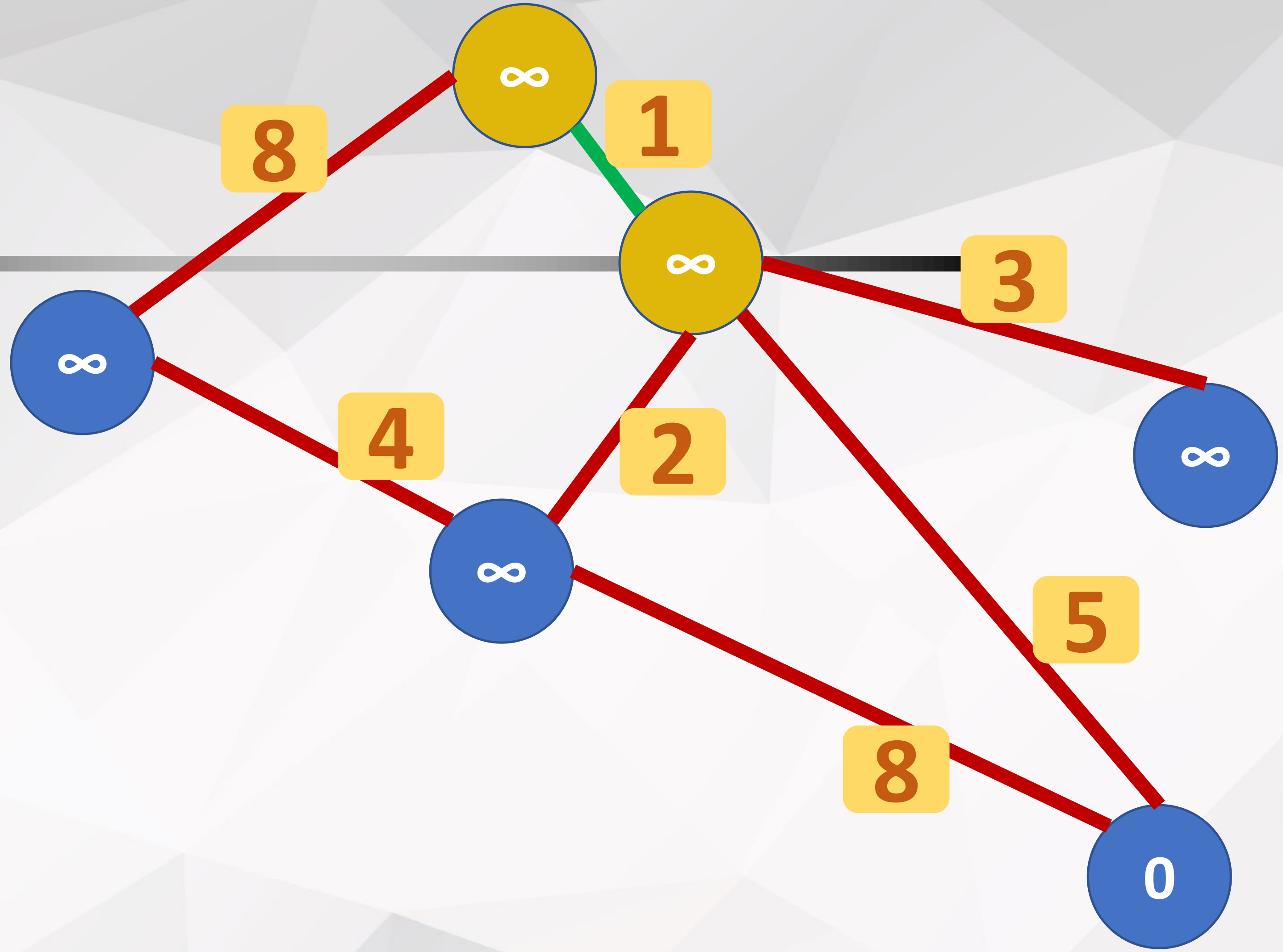


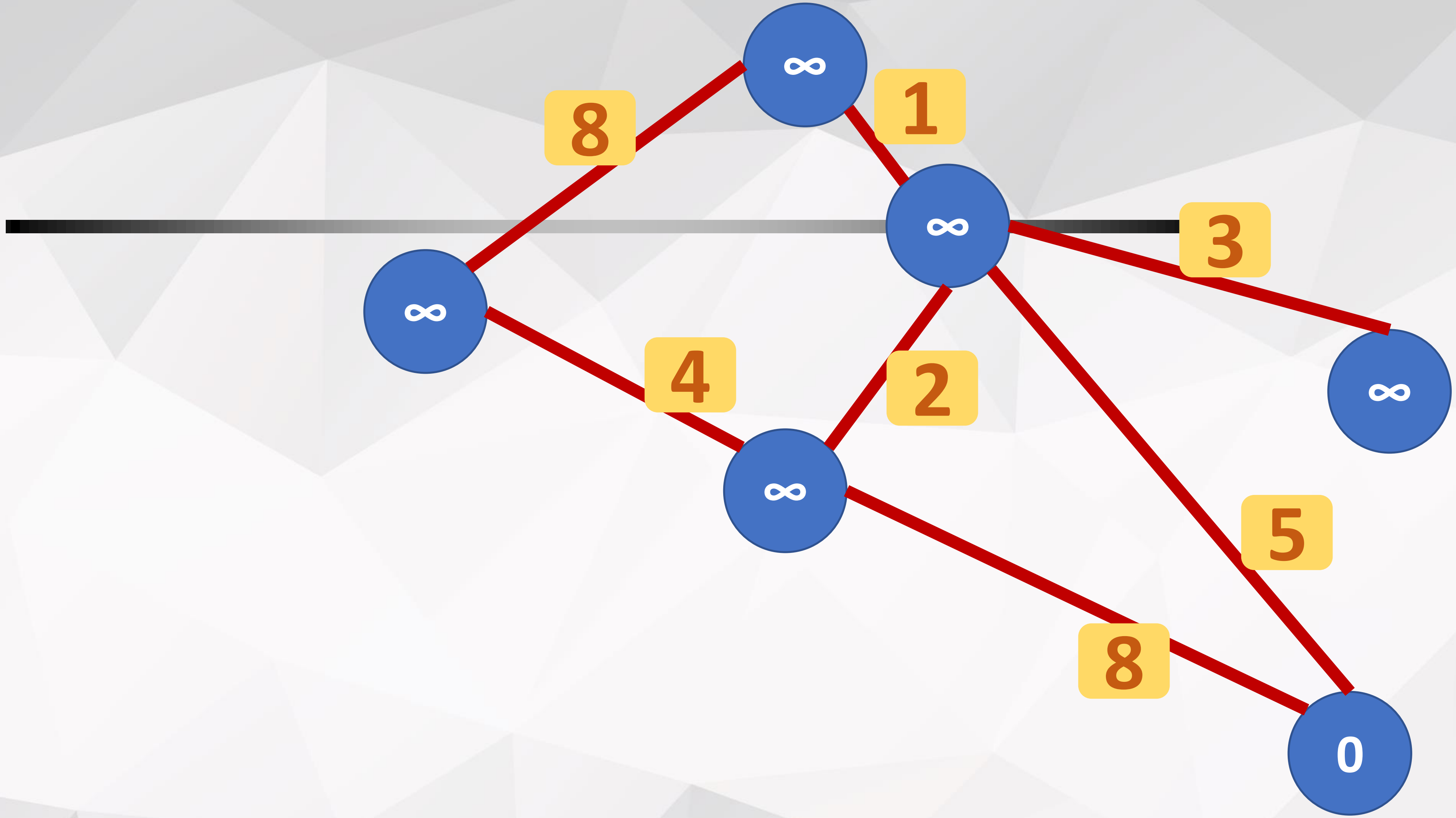

∞
8
1
∞
3
∞
4
2
∞
∞
5
8
0

Relax?



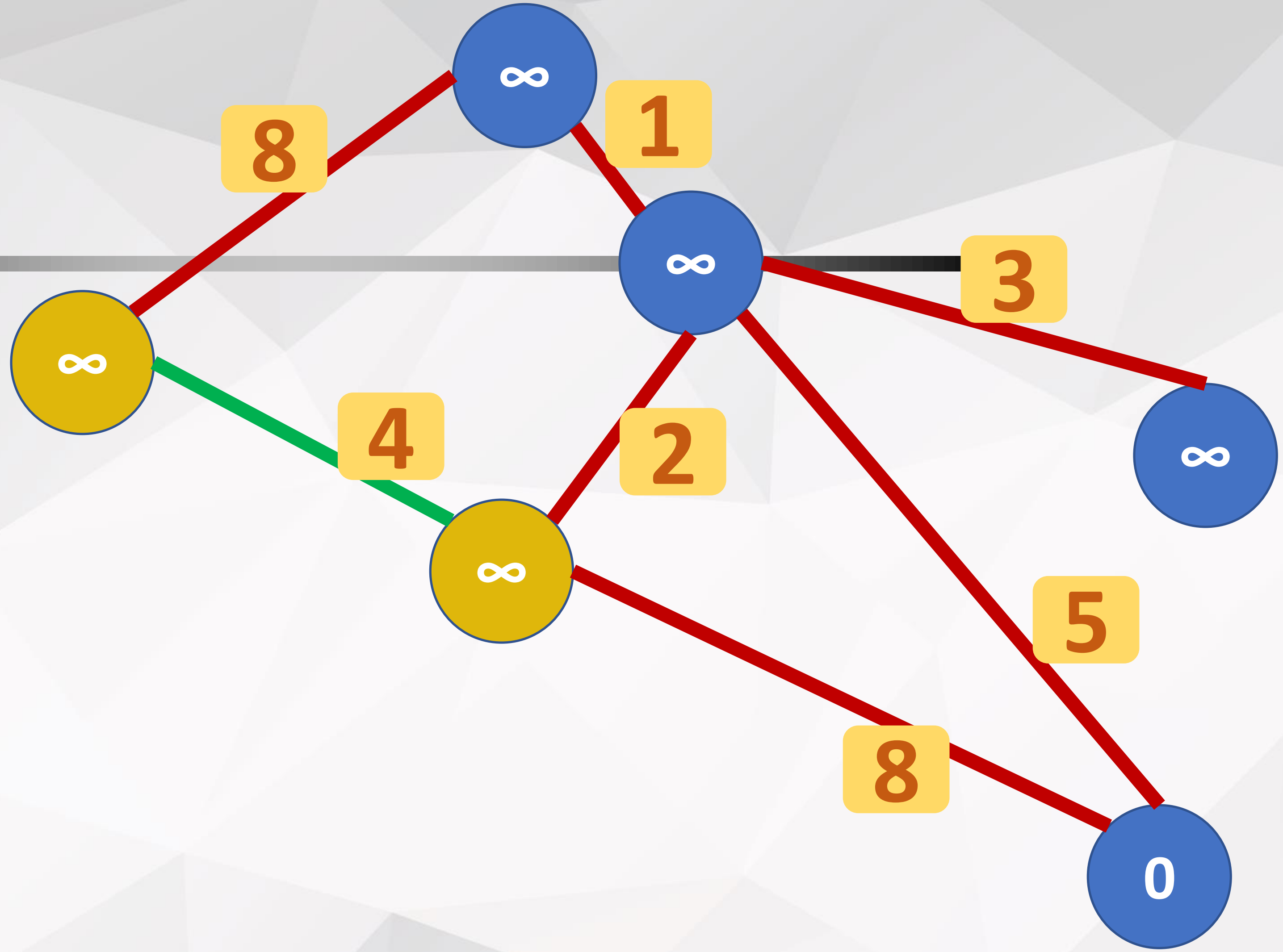
No



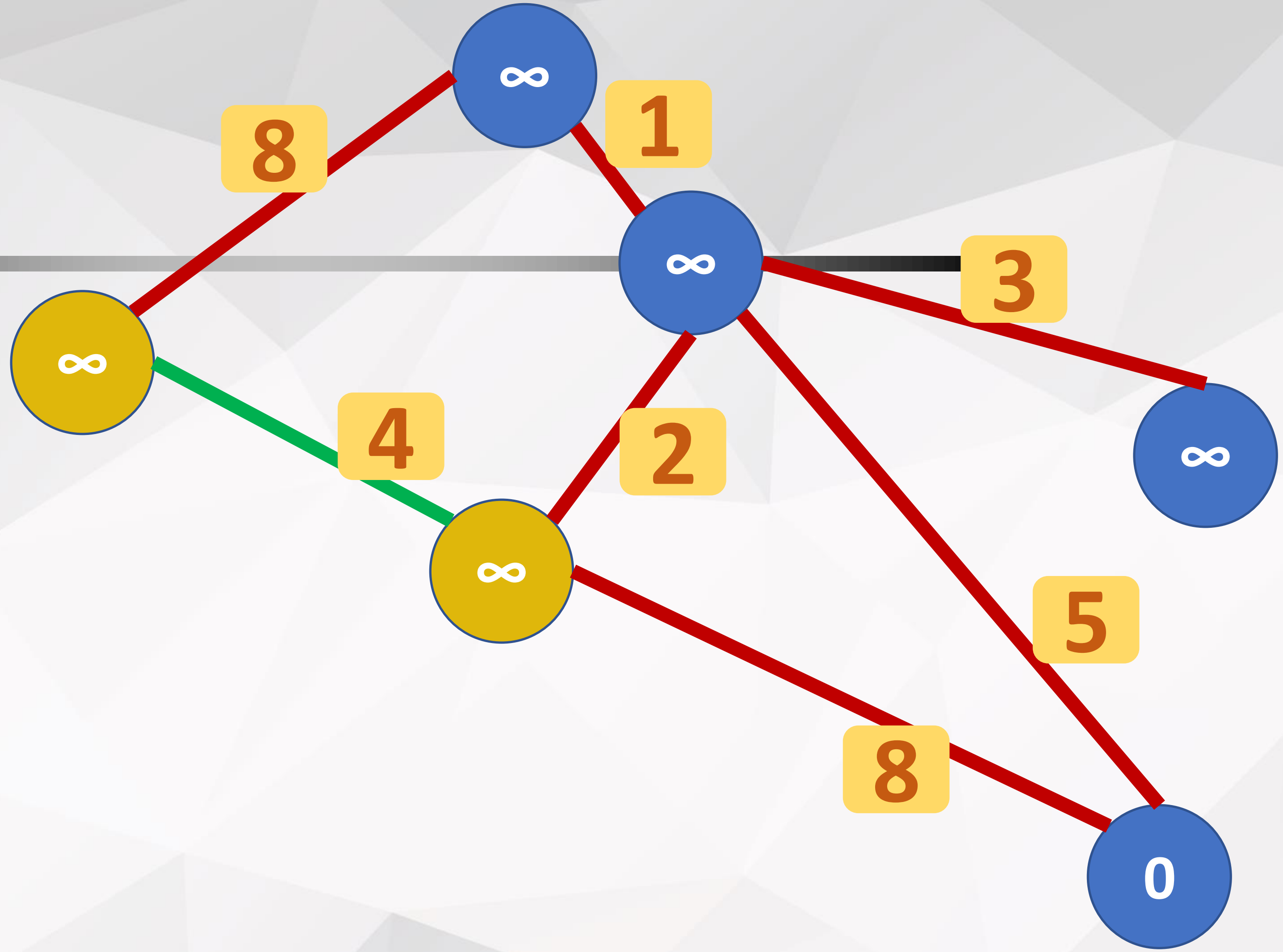


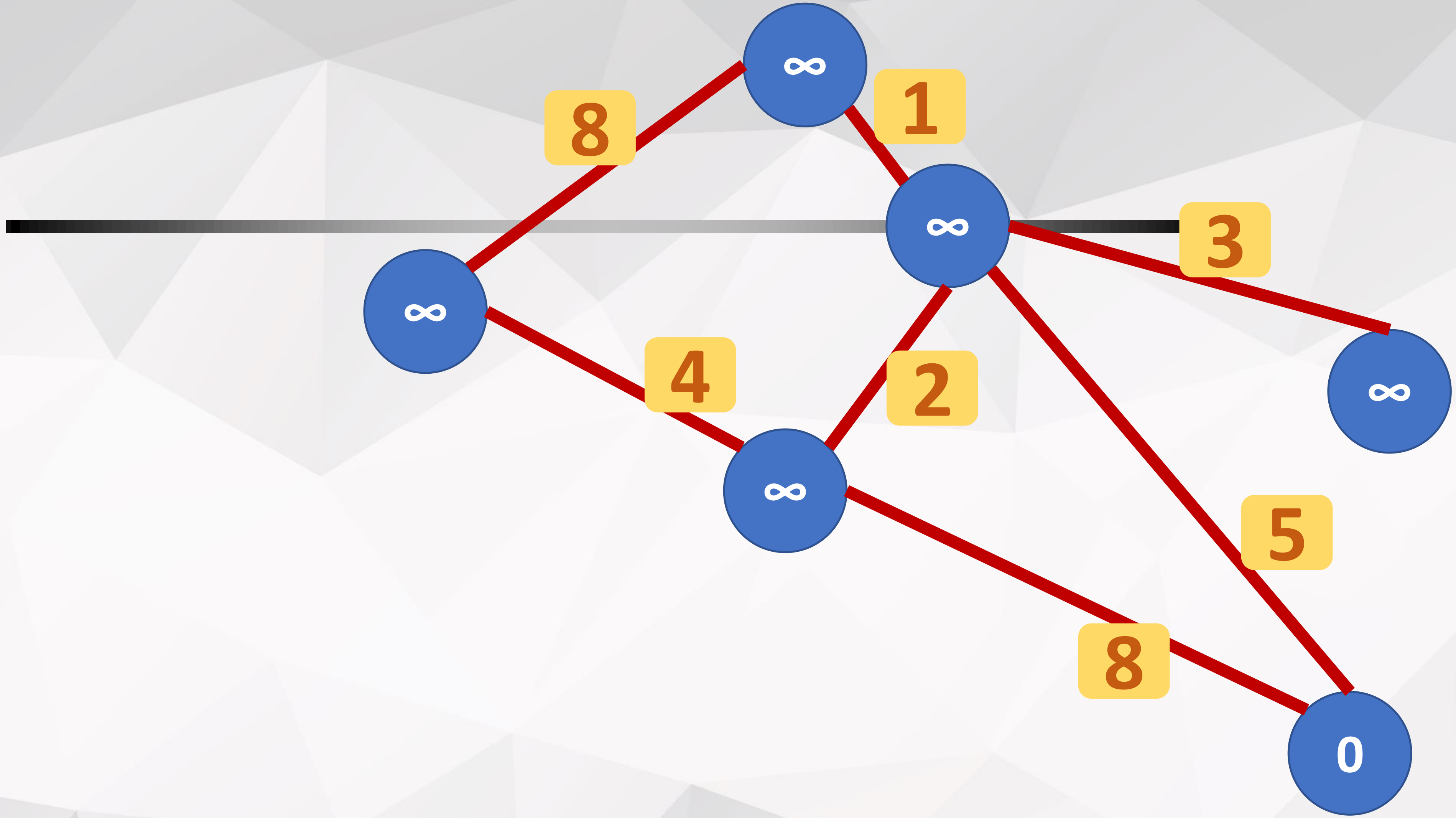
∞
8
1
∞
3
∞
4
2
∞
∞
5
8
0

Relax?

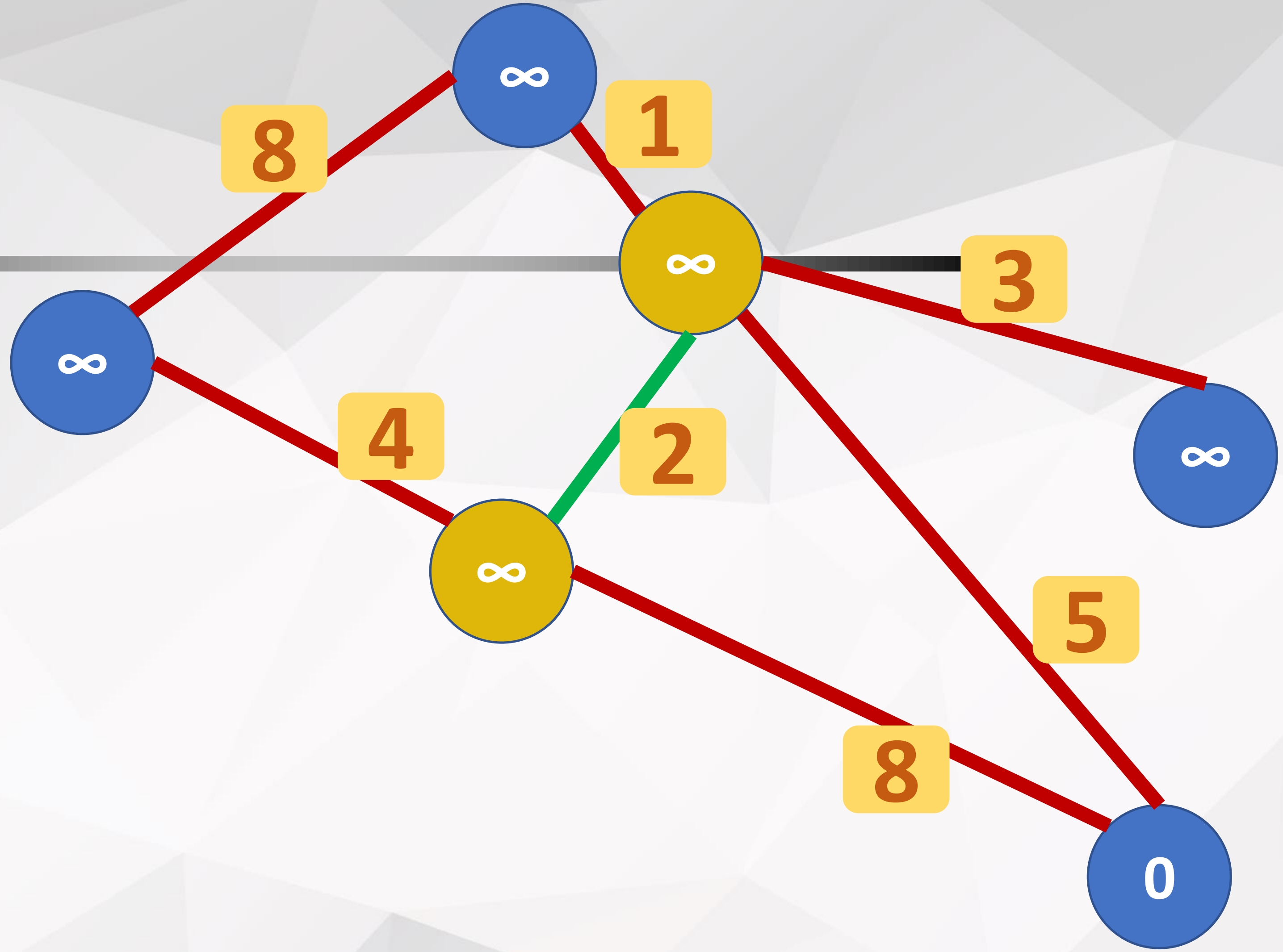


No

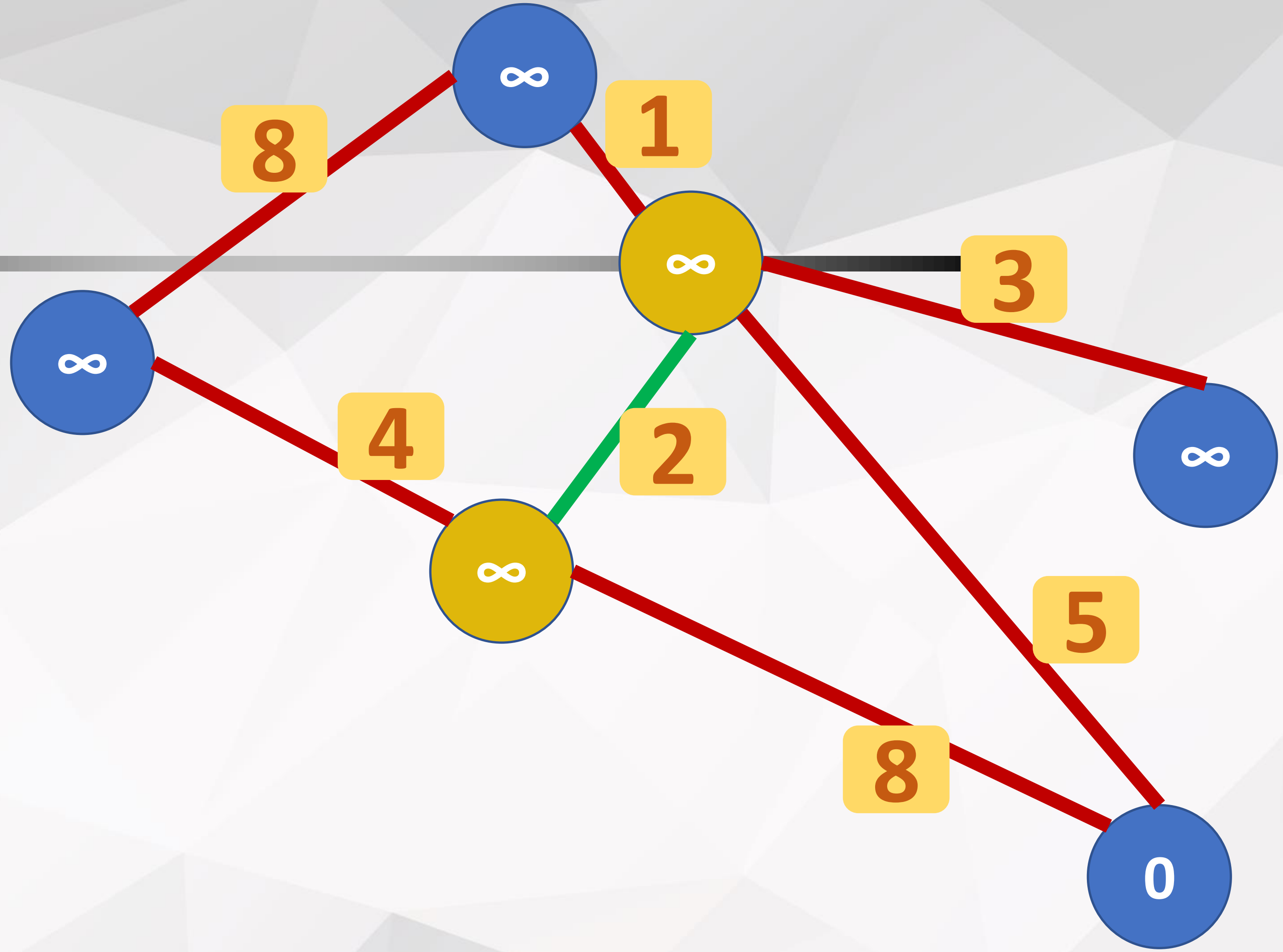


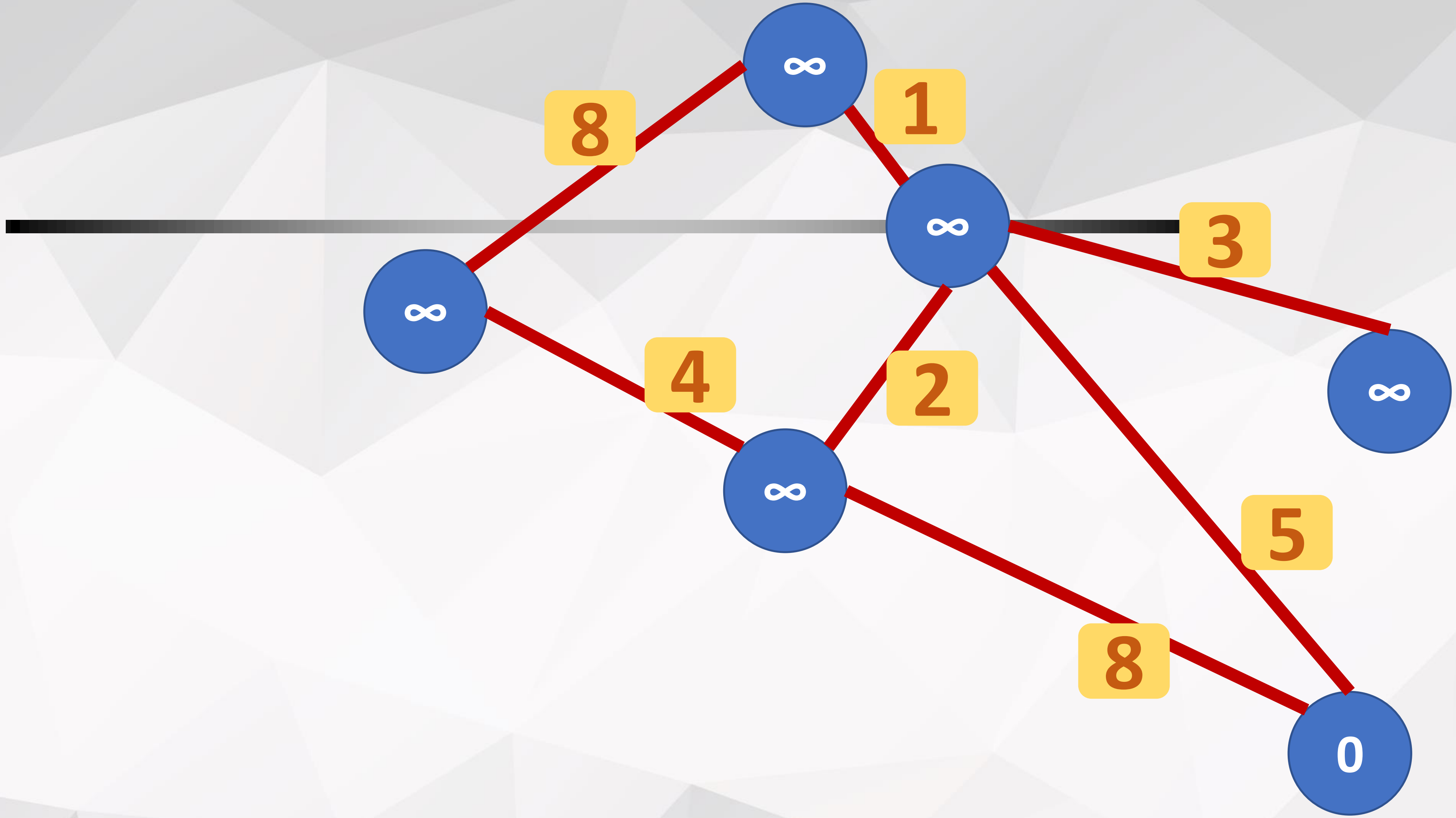

∞
8
1
∞
3
∞
4
2
∞
∞
5
8
0

Relax?

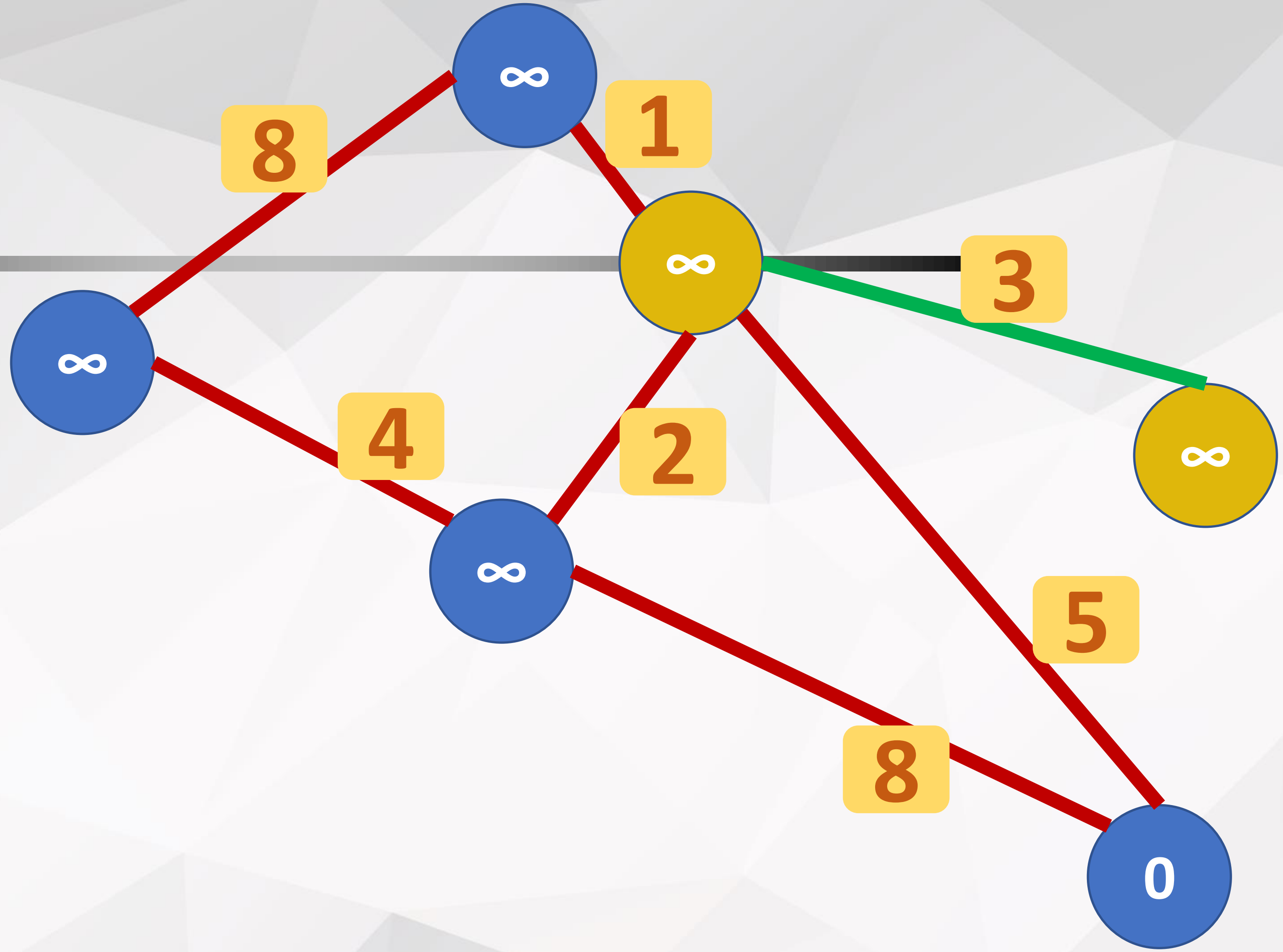


No

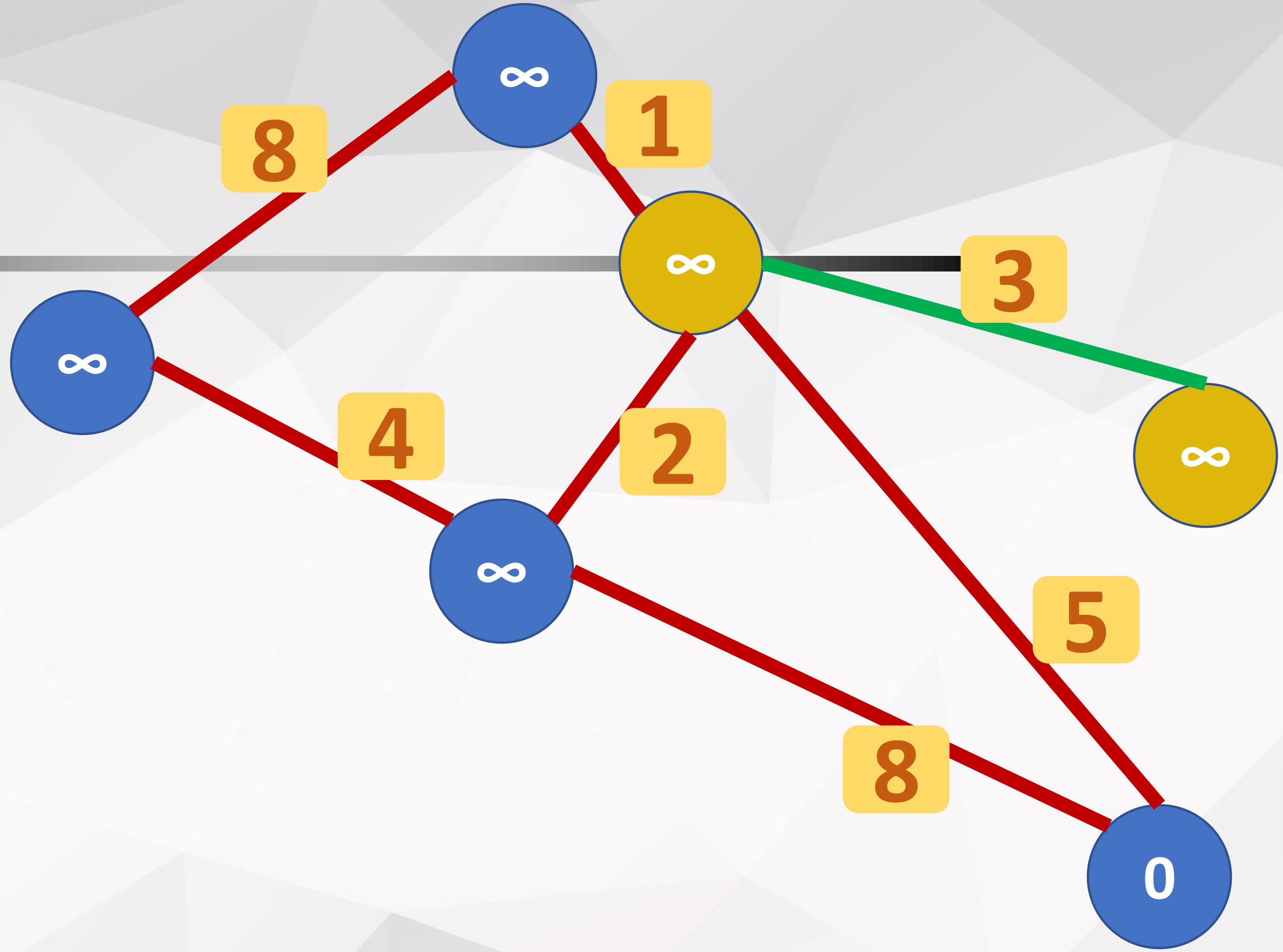


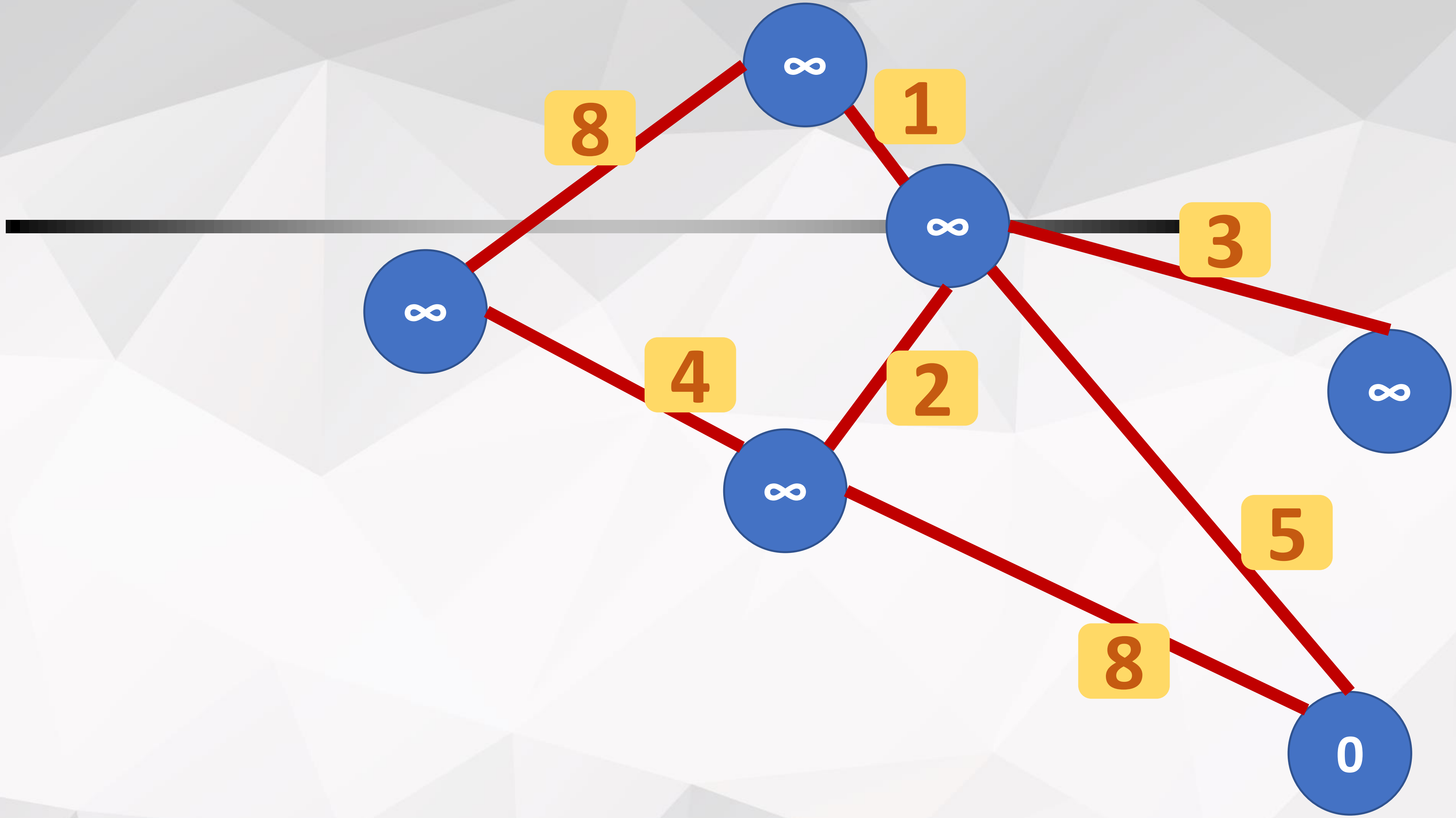

∞
8
1
∞
3
∞
4
2
∞
∞
5
8
0

Relax?

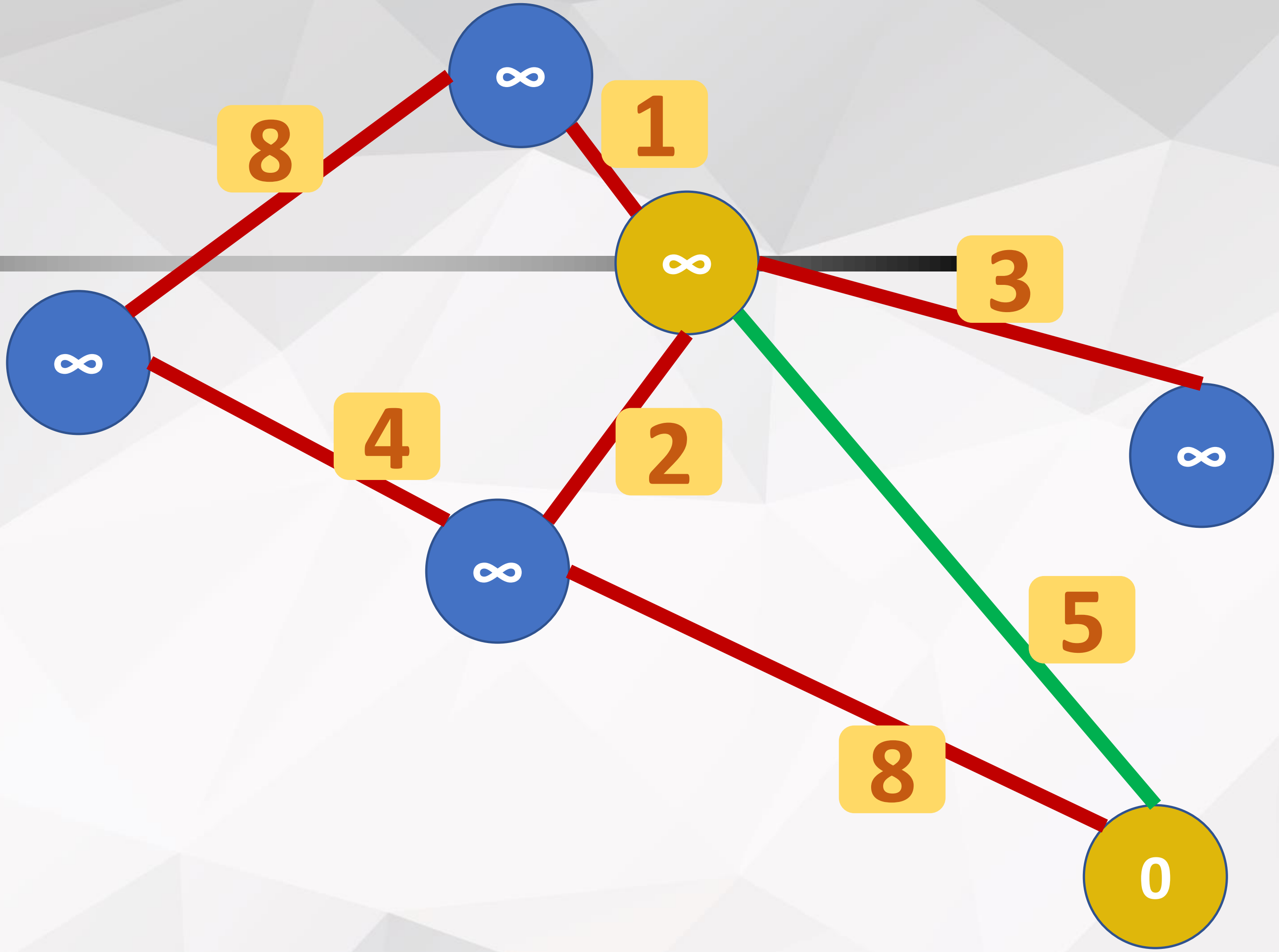


No

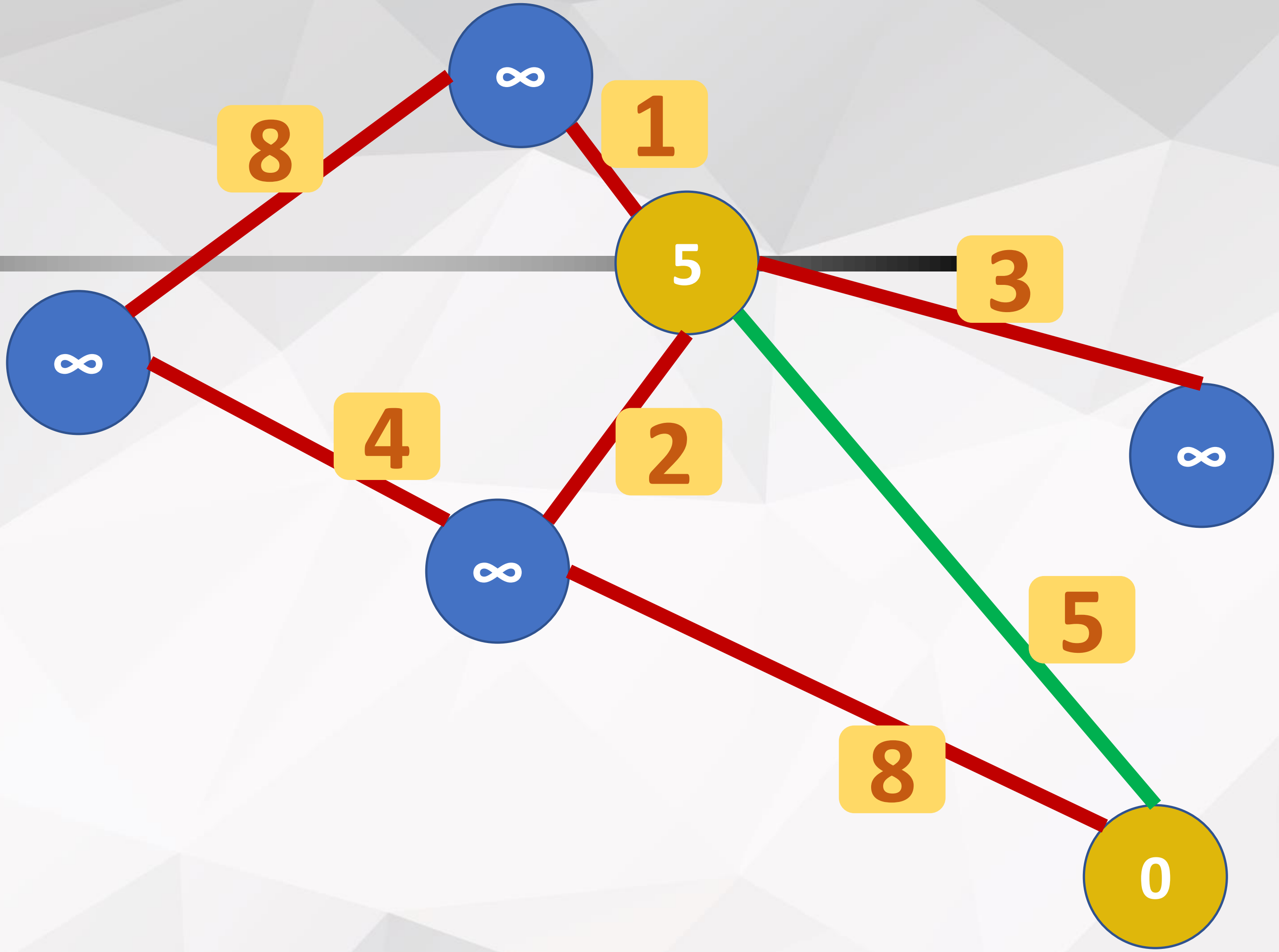


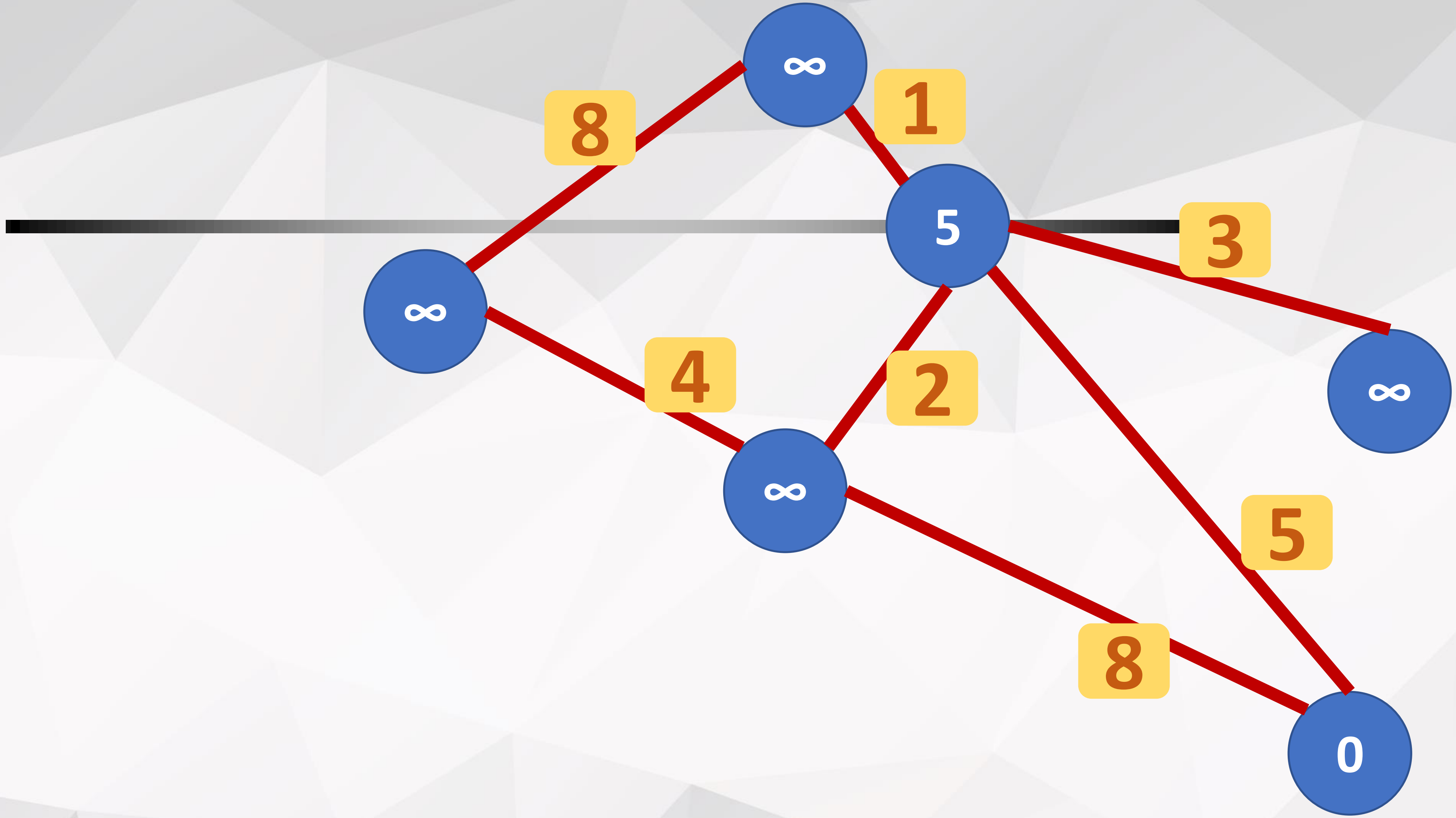

∞
8
1
∞
3
∞
4
2
∞
∞
5
8
0

Relax?

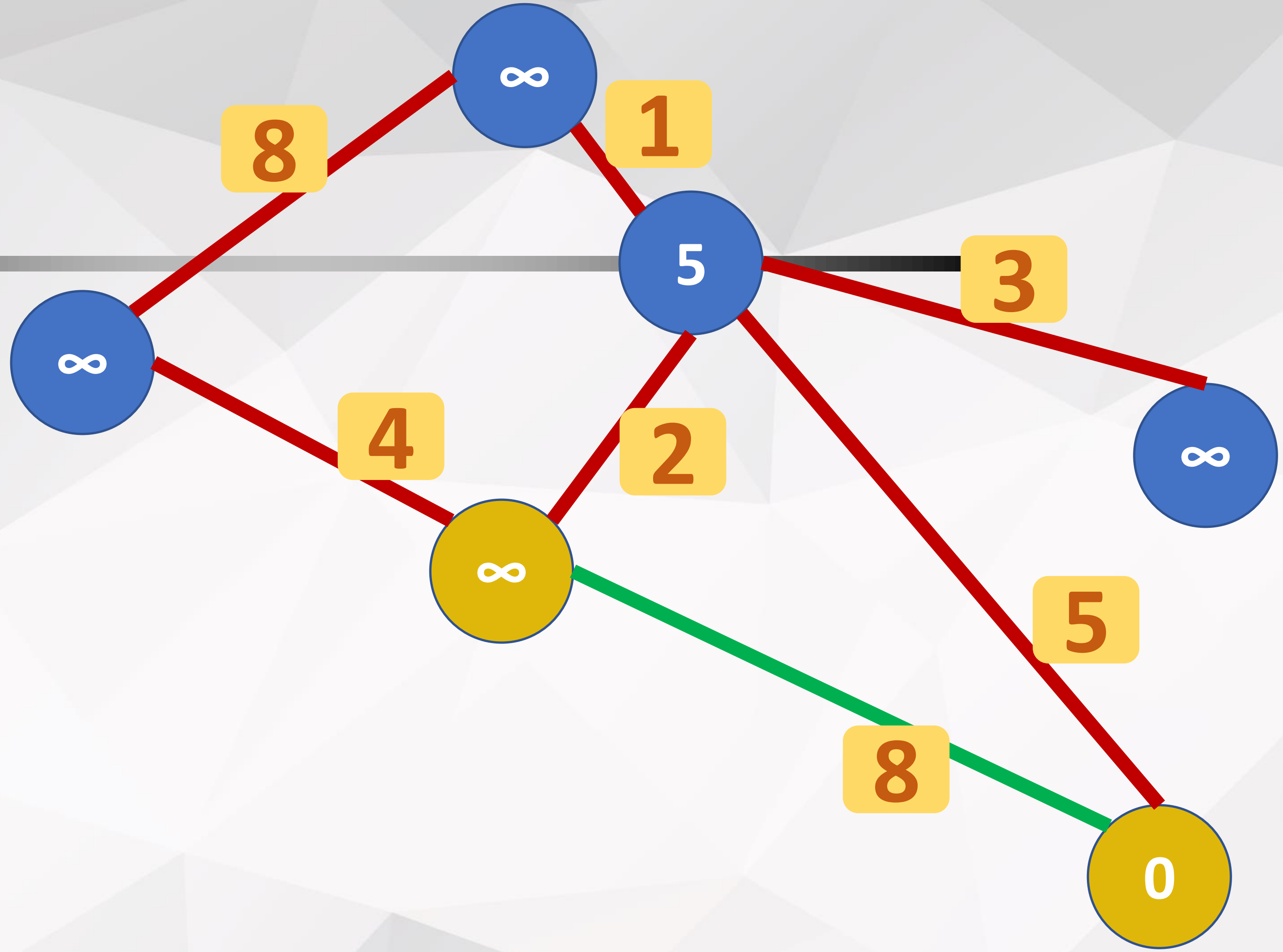


Relax!

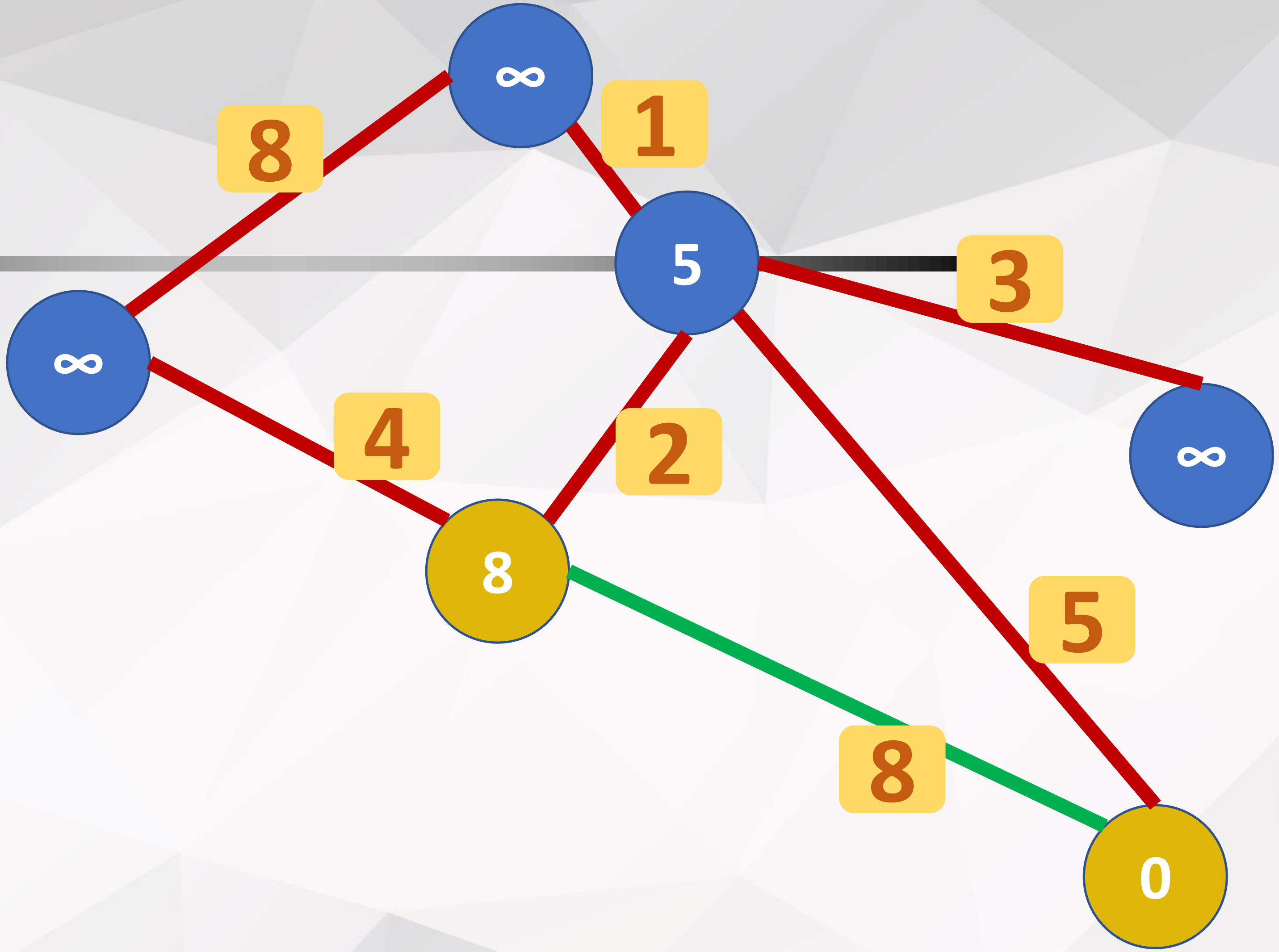


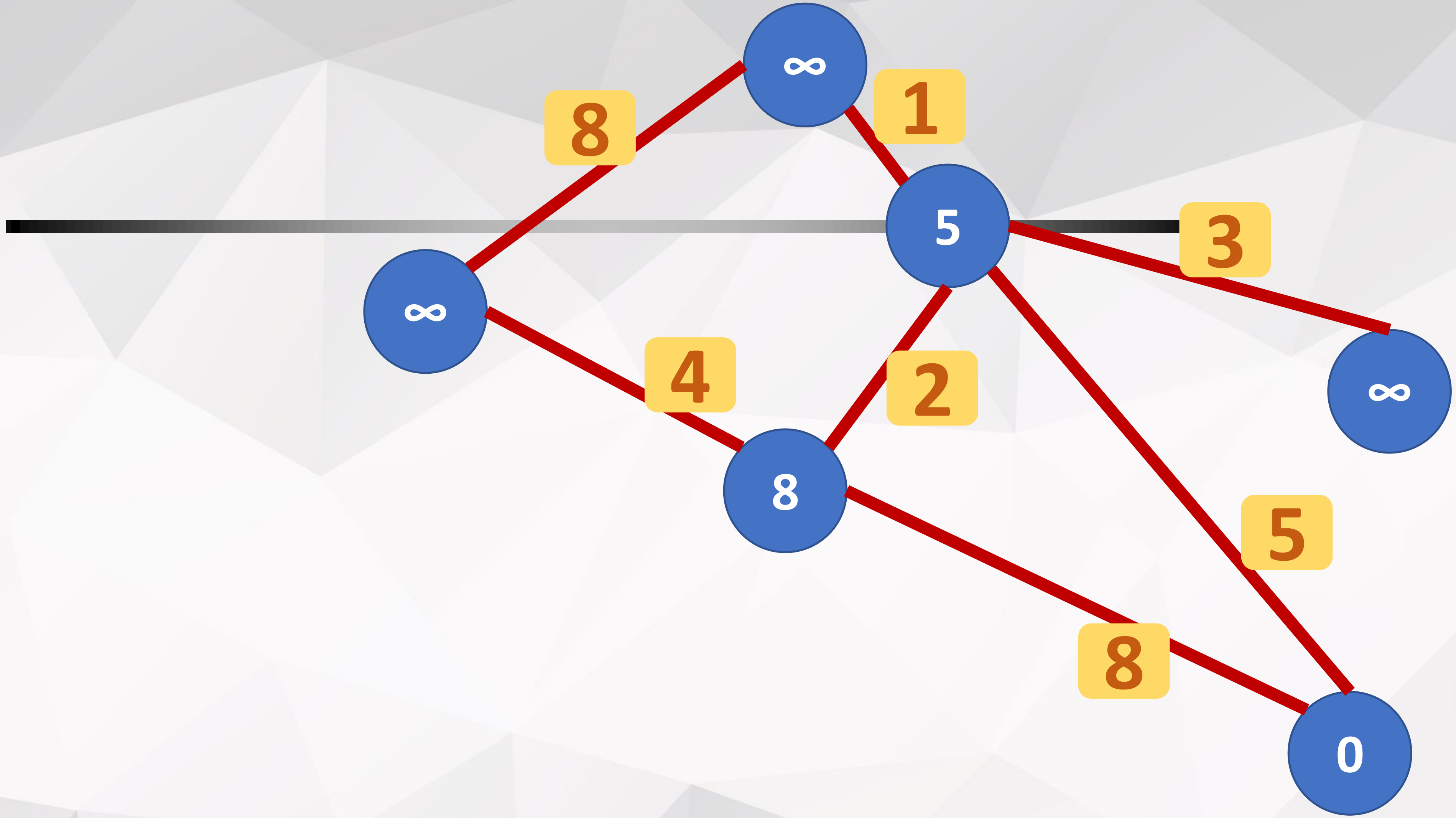

∞
8
1
5
3
∞
4
2
∞
∞
5
8
0

Relax?

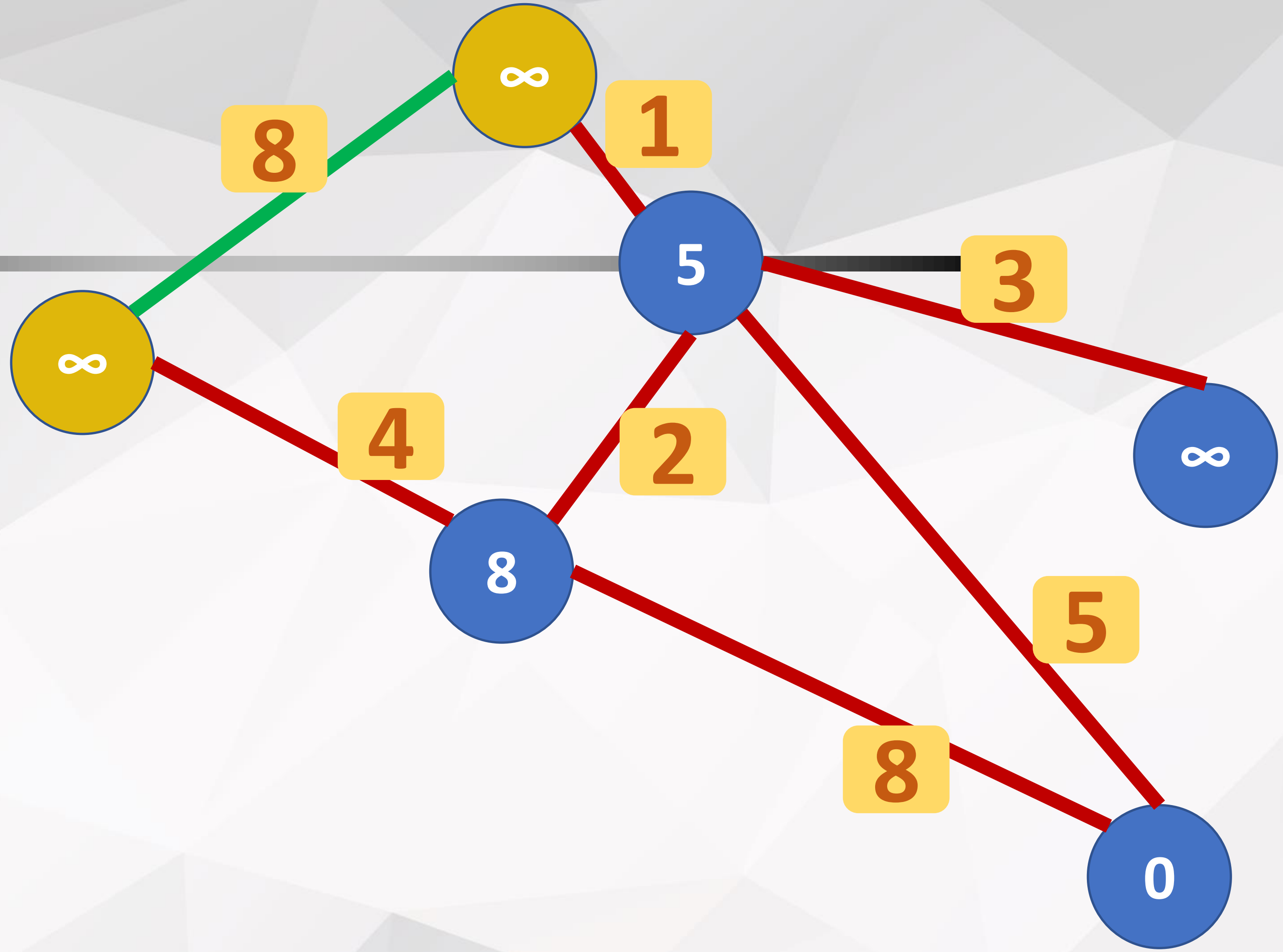


Relax!

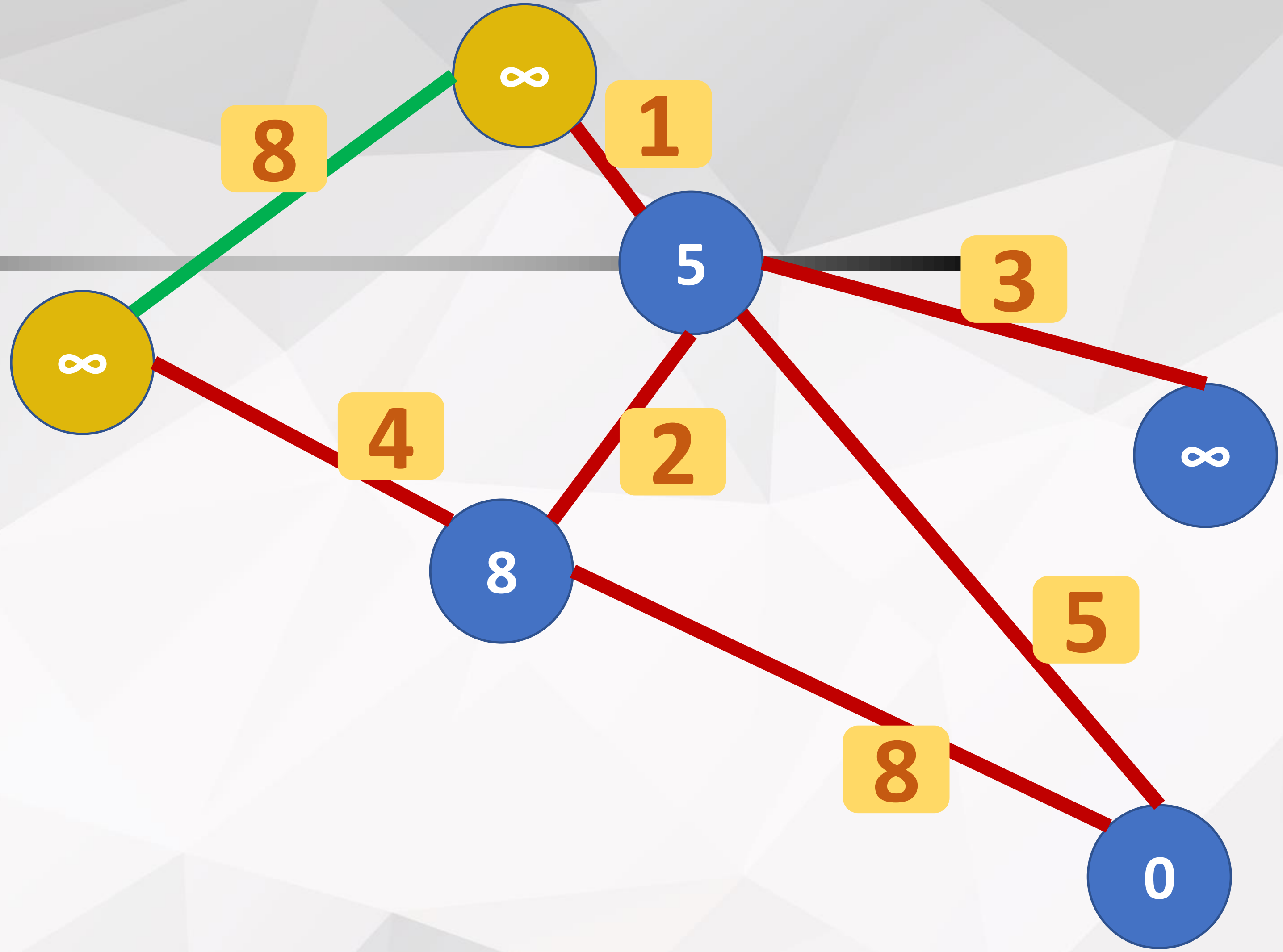


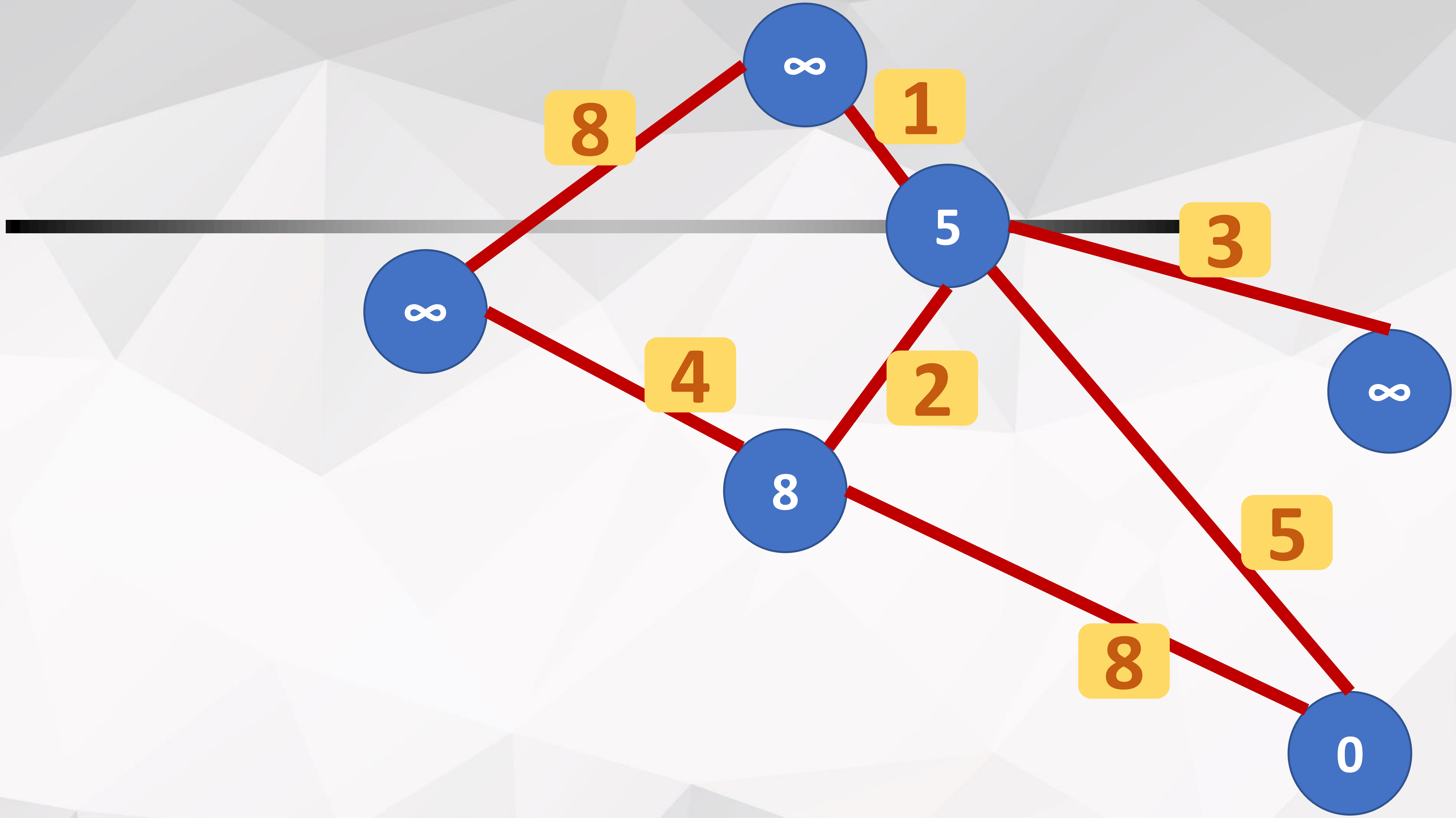

∞
8
1
5
3
∞
4
2
∞
8
5
8
0

Relax?

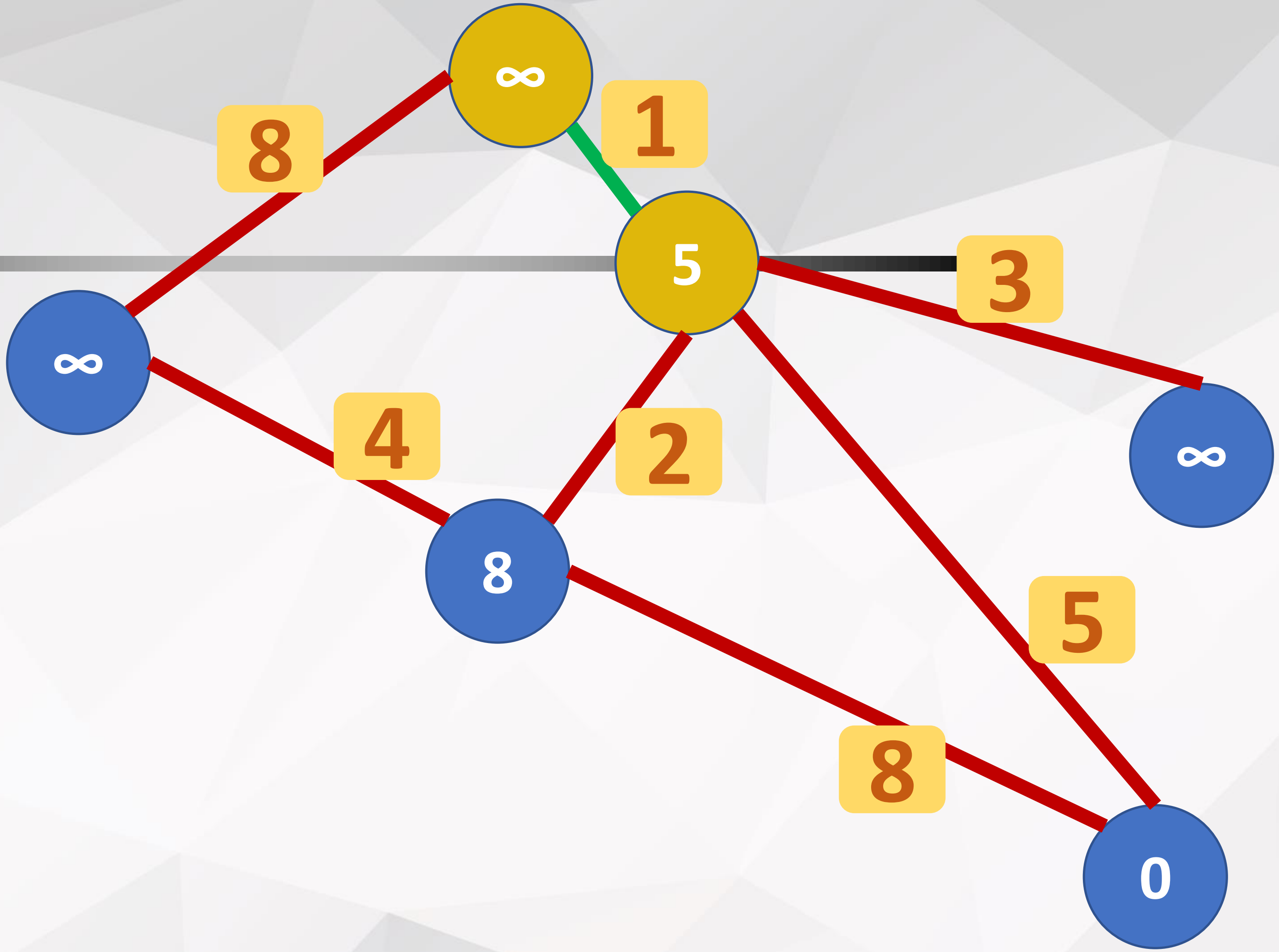


No

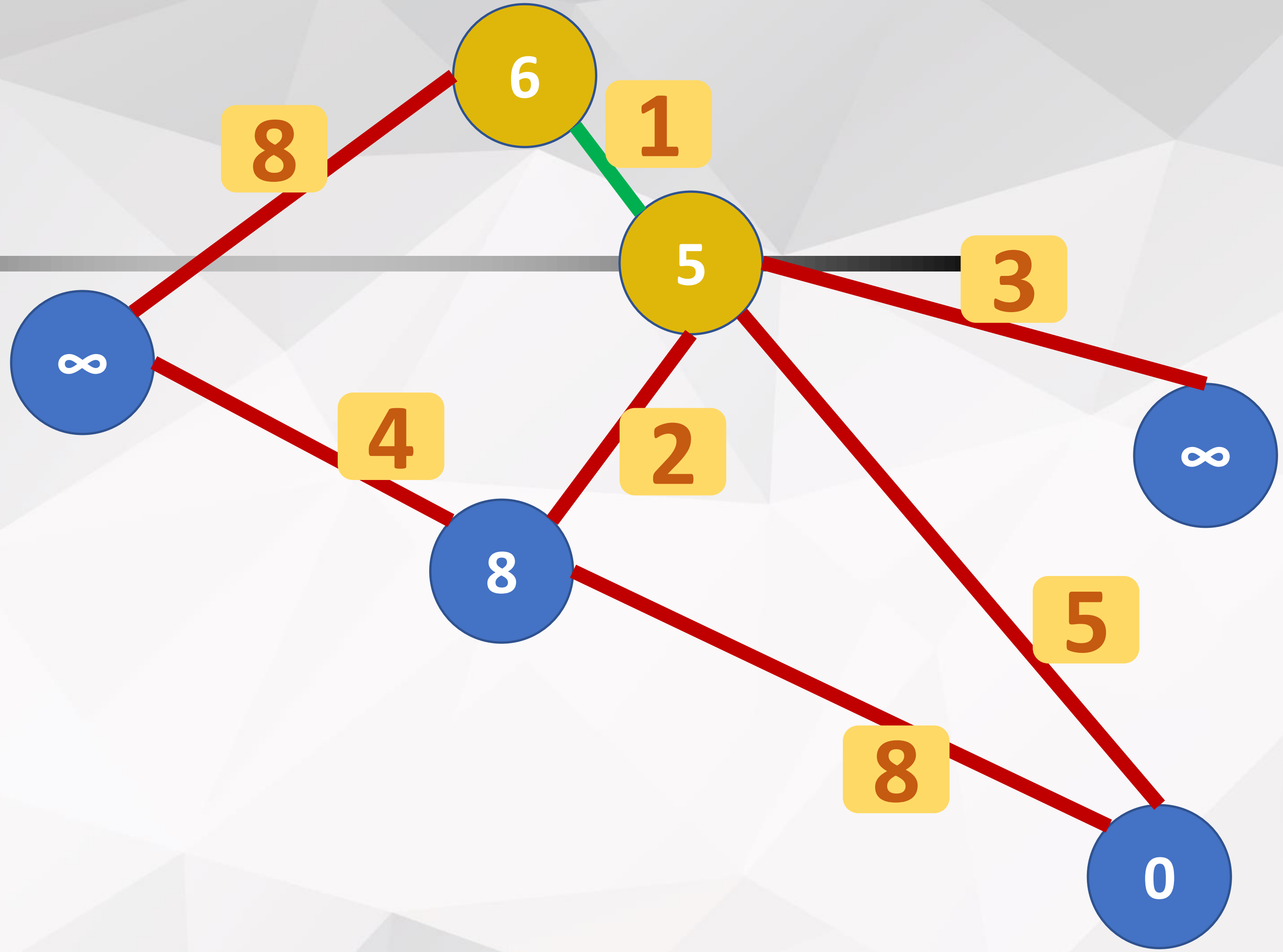


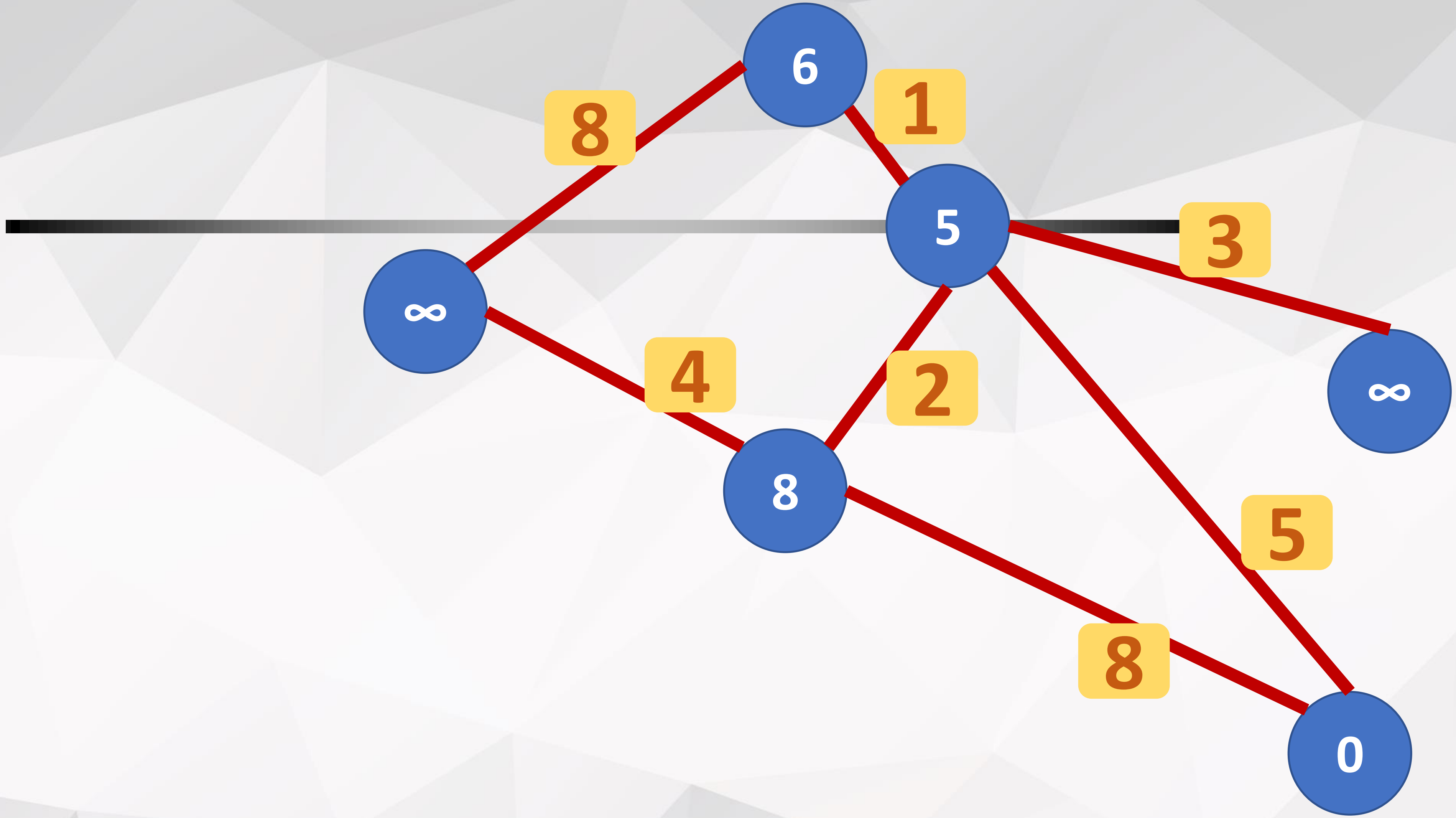

∞
8
1
5
3
∞
4
2
∞
8
5
8
0

Relax?

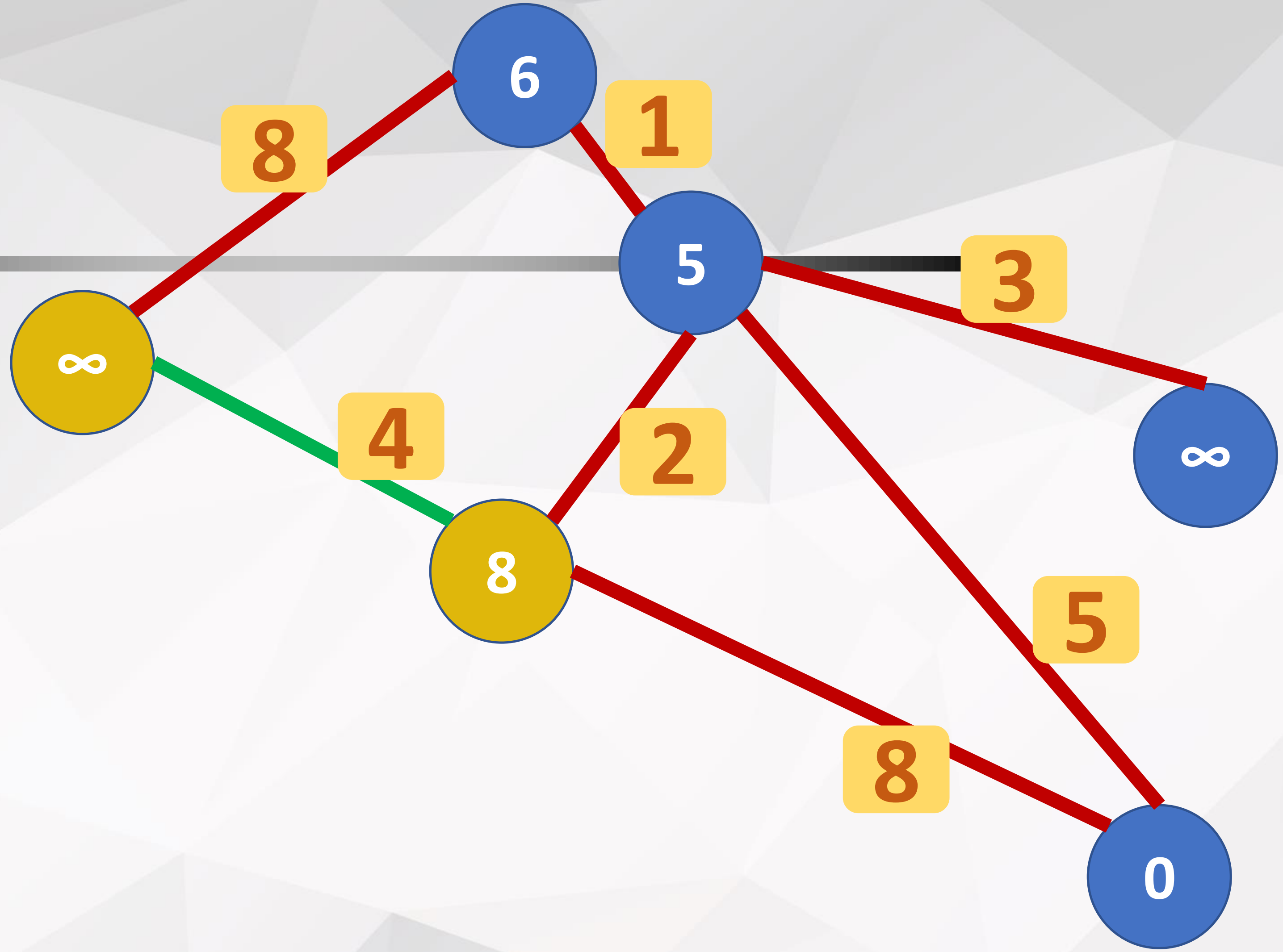


Relax!

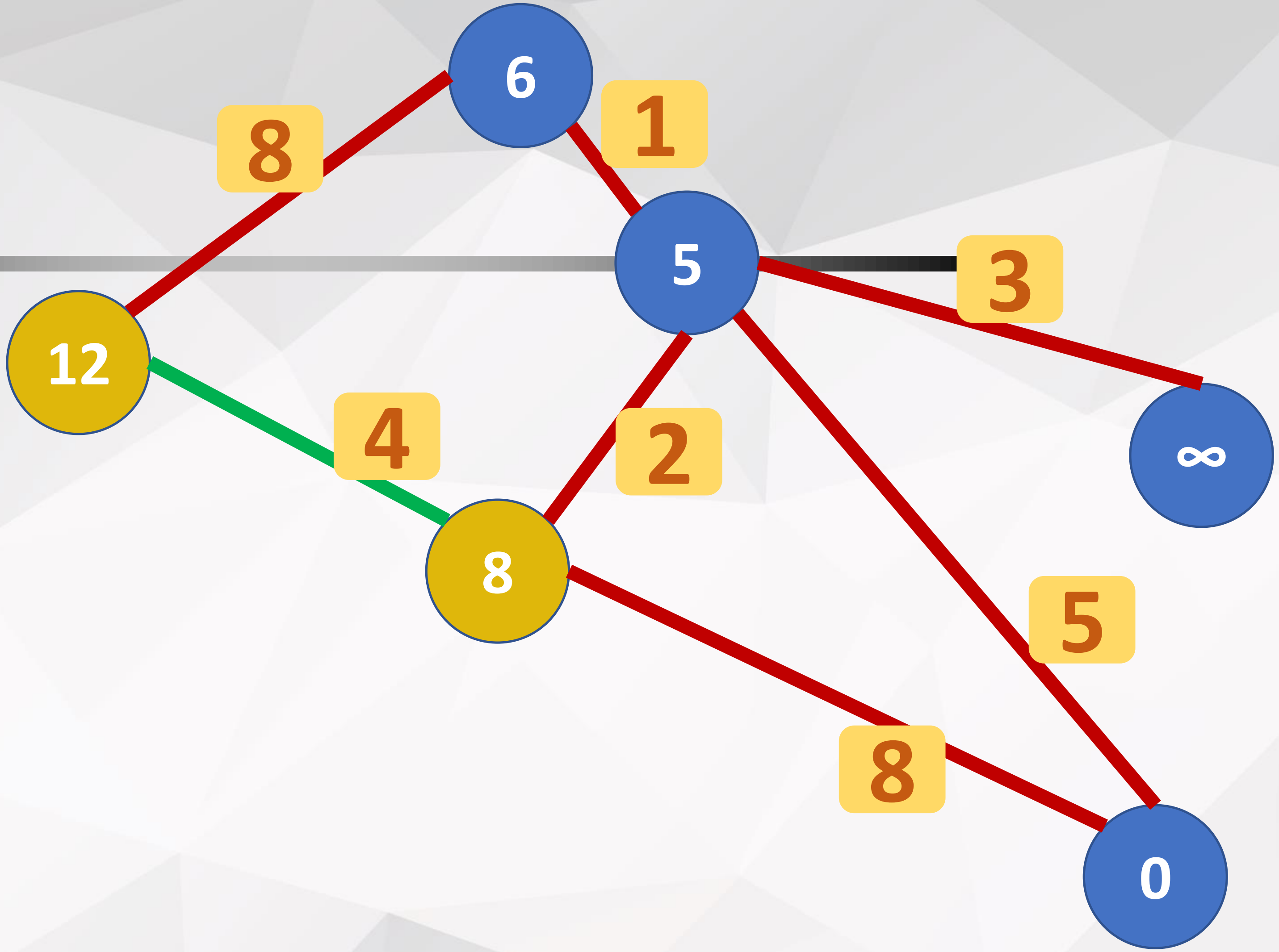


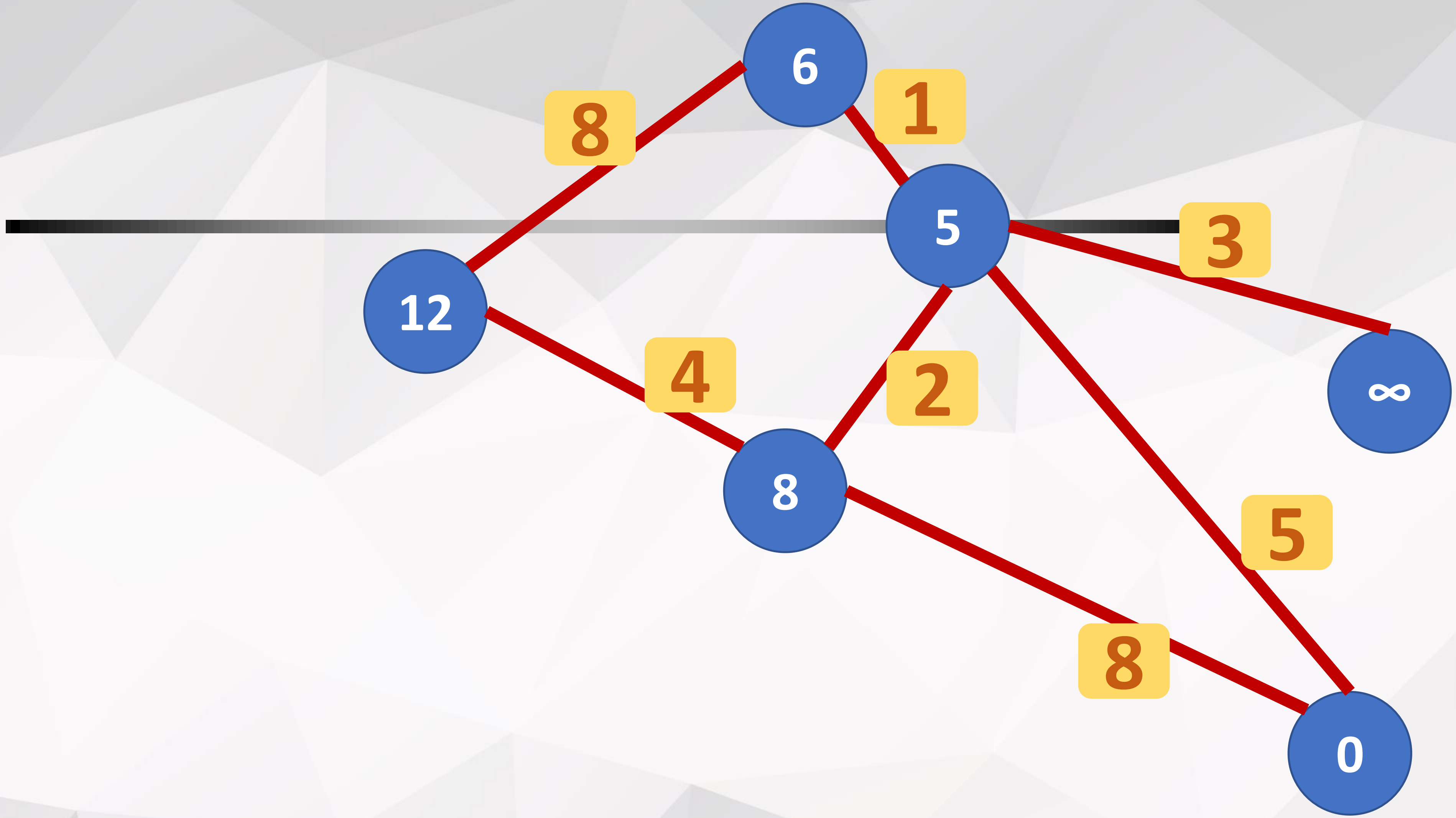

6
8
1
5
3
∞
4
2
∞
8
5
8
0

Relax?

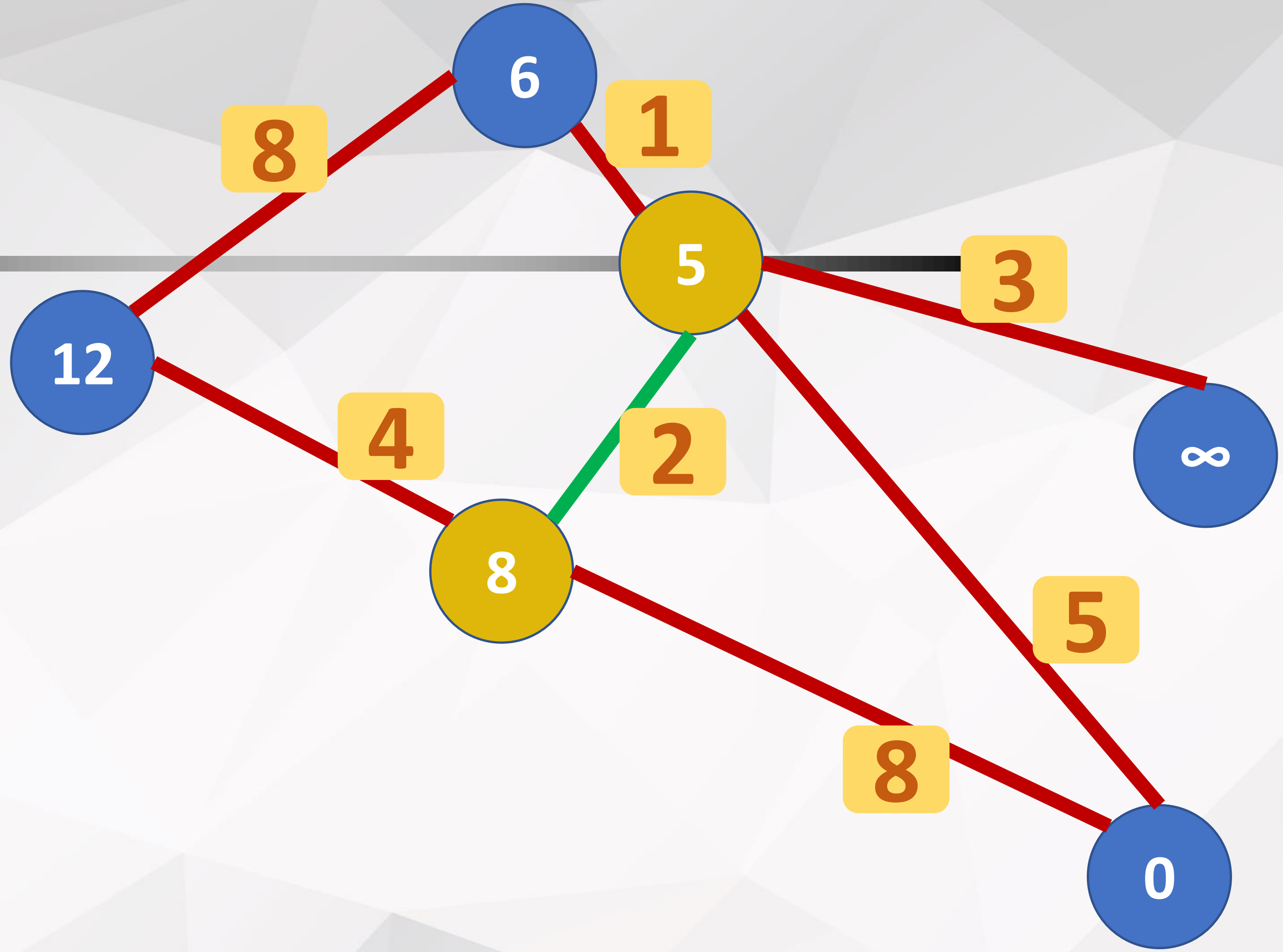


Relax!

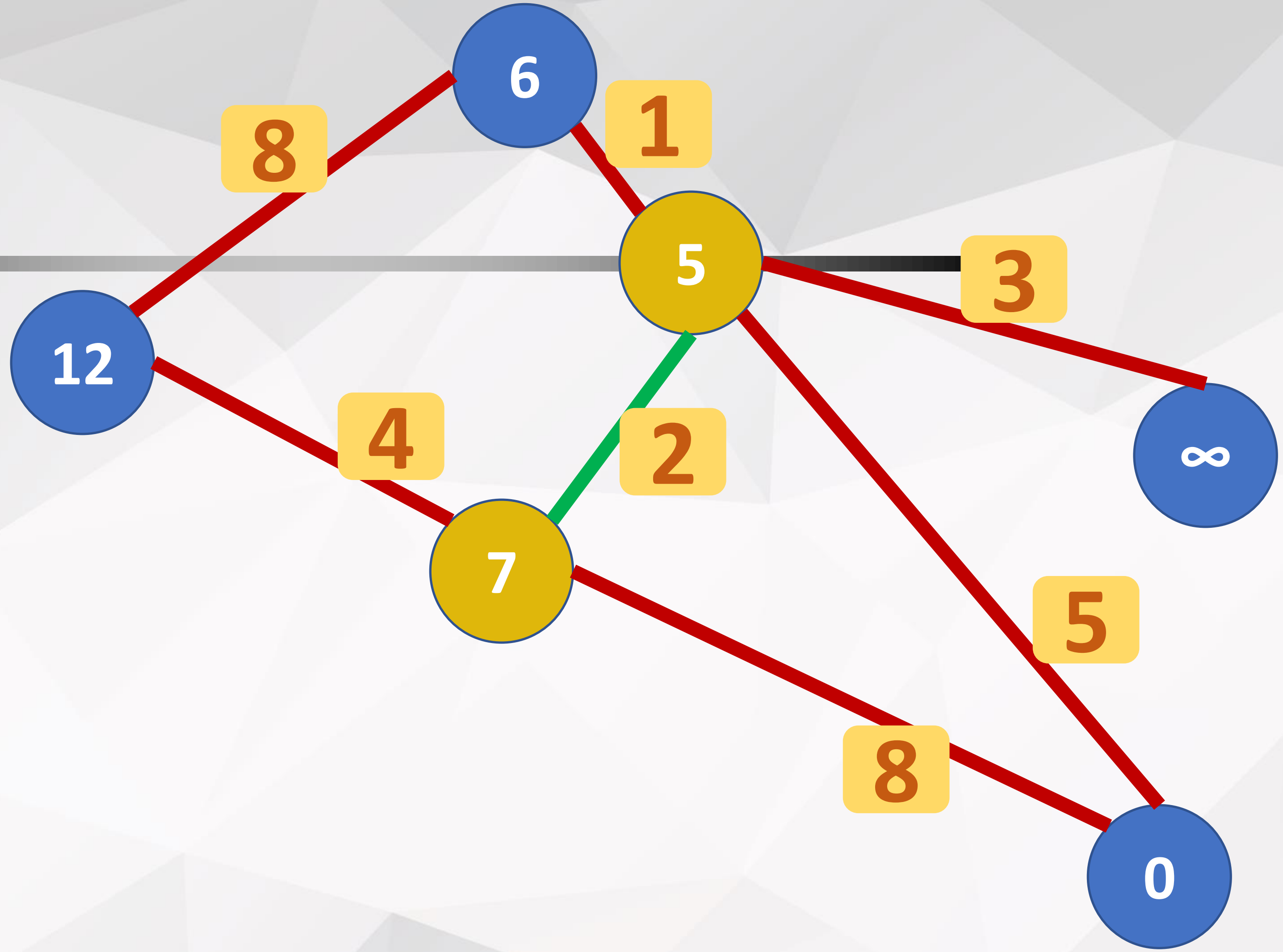


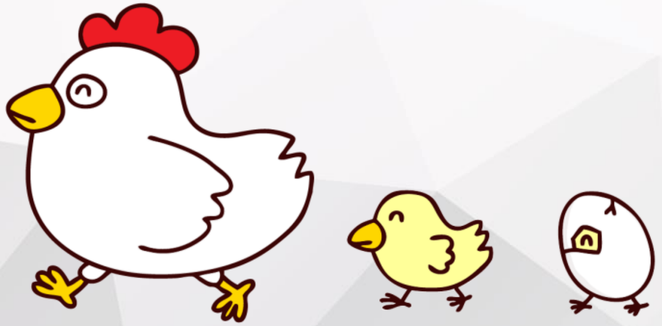

6
8
1
5
3
12
4
2
∞
8
5
8
0

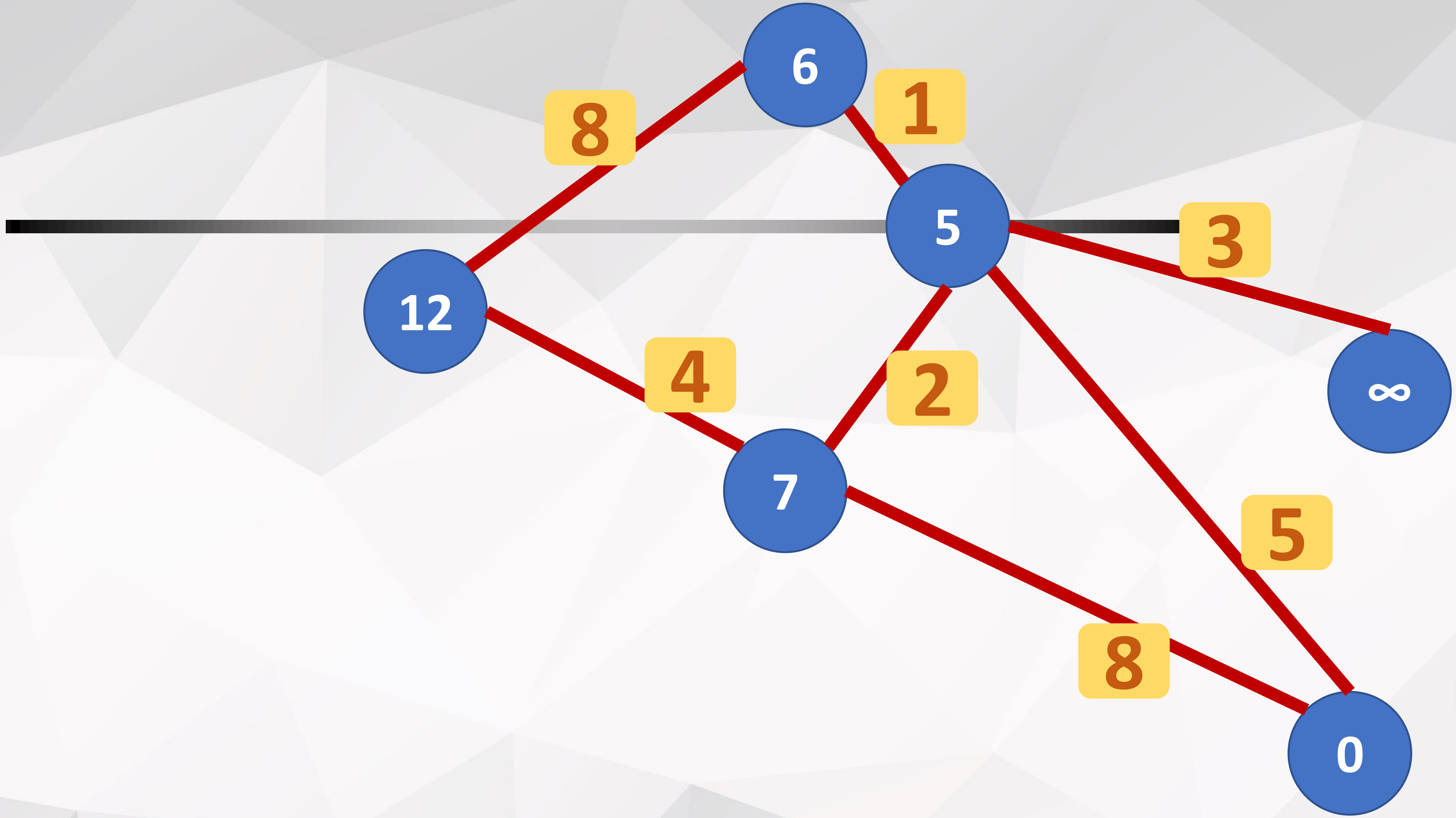
Relax?



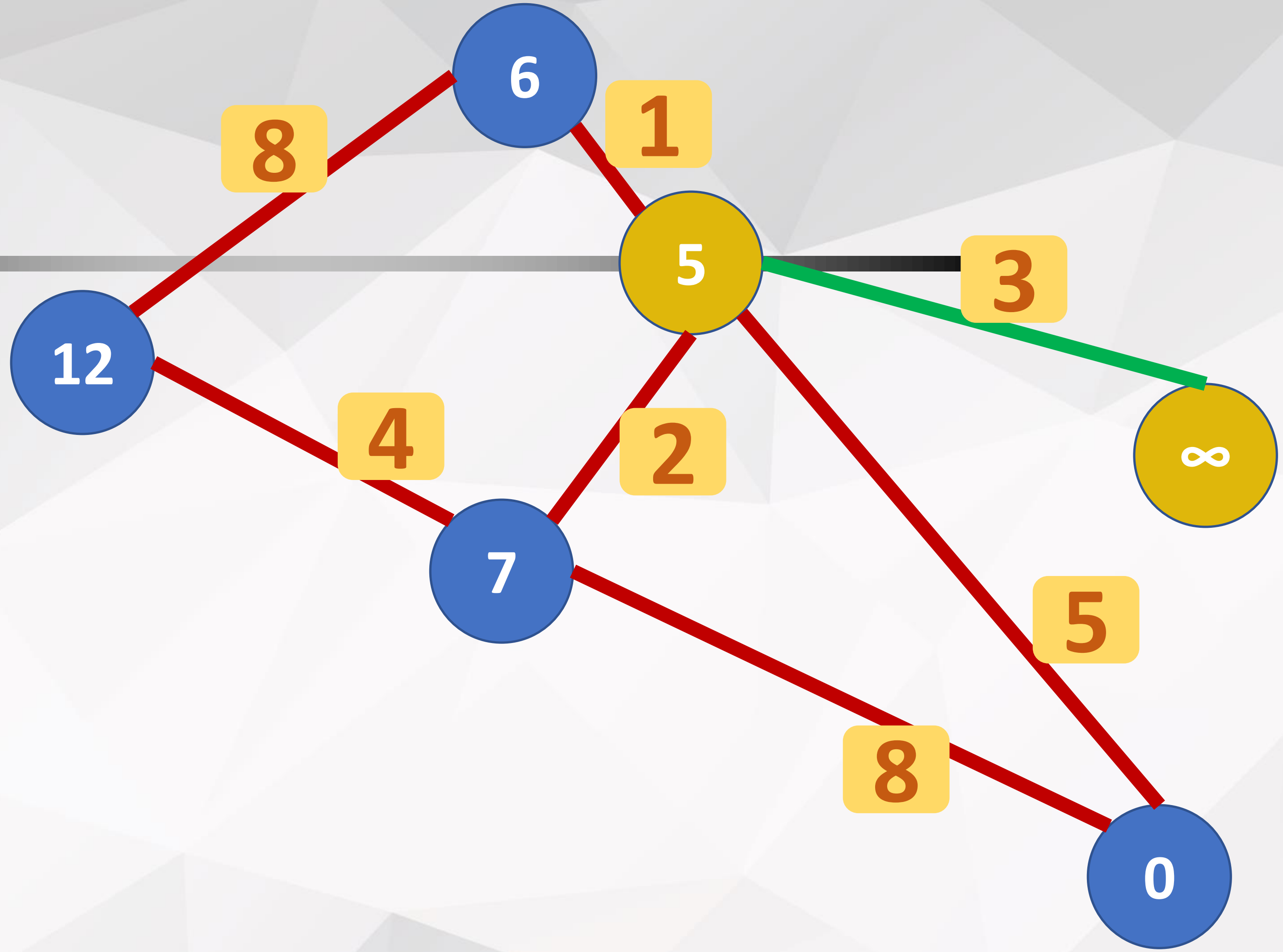
Relax!



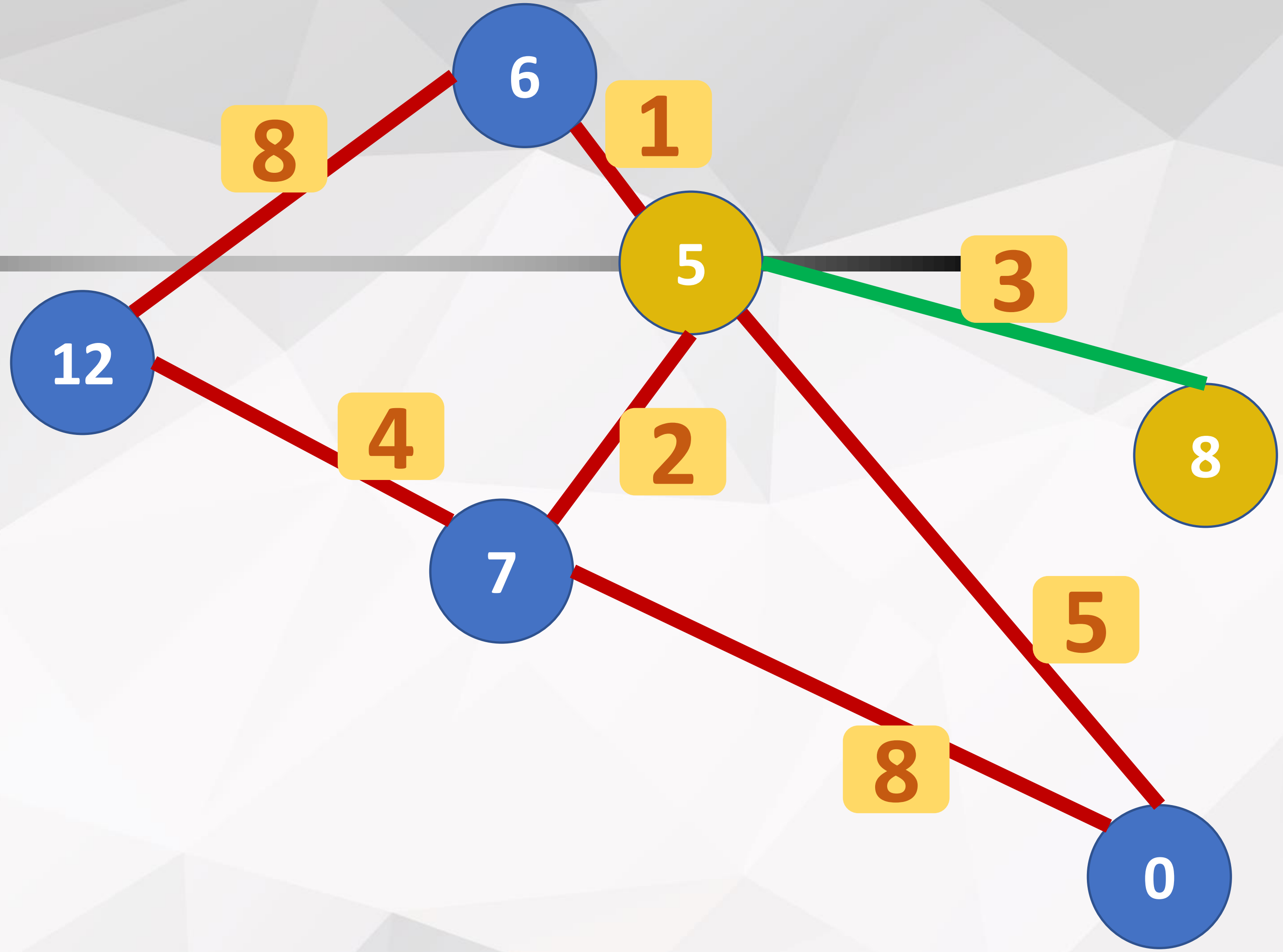


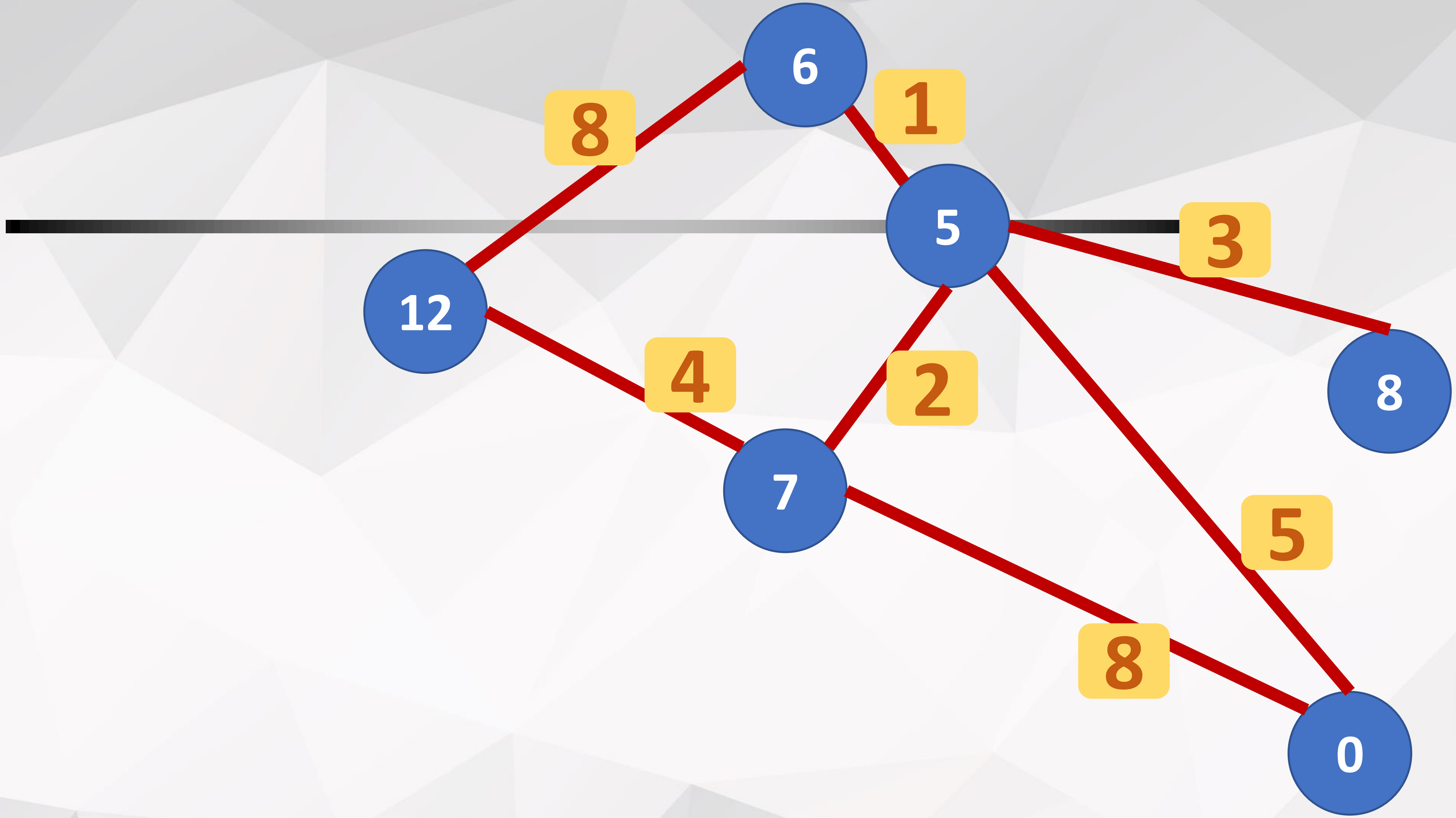

6
1
8
5
3
12
4
2
∞
7
5
8
0

Relax?

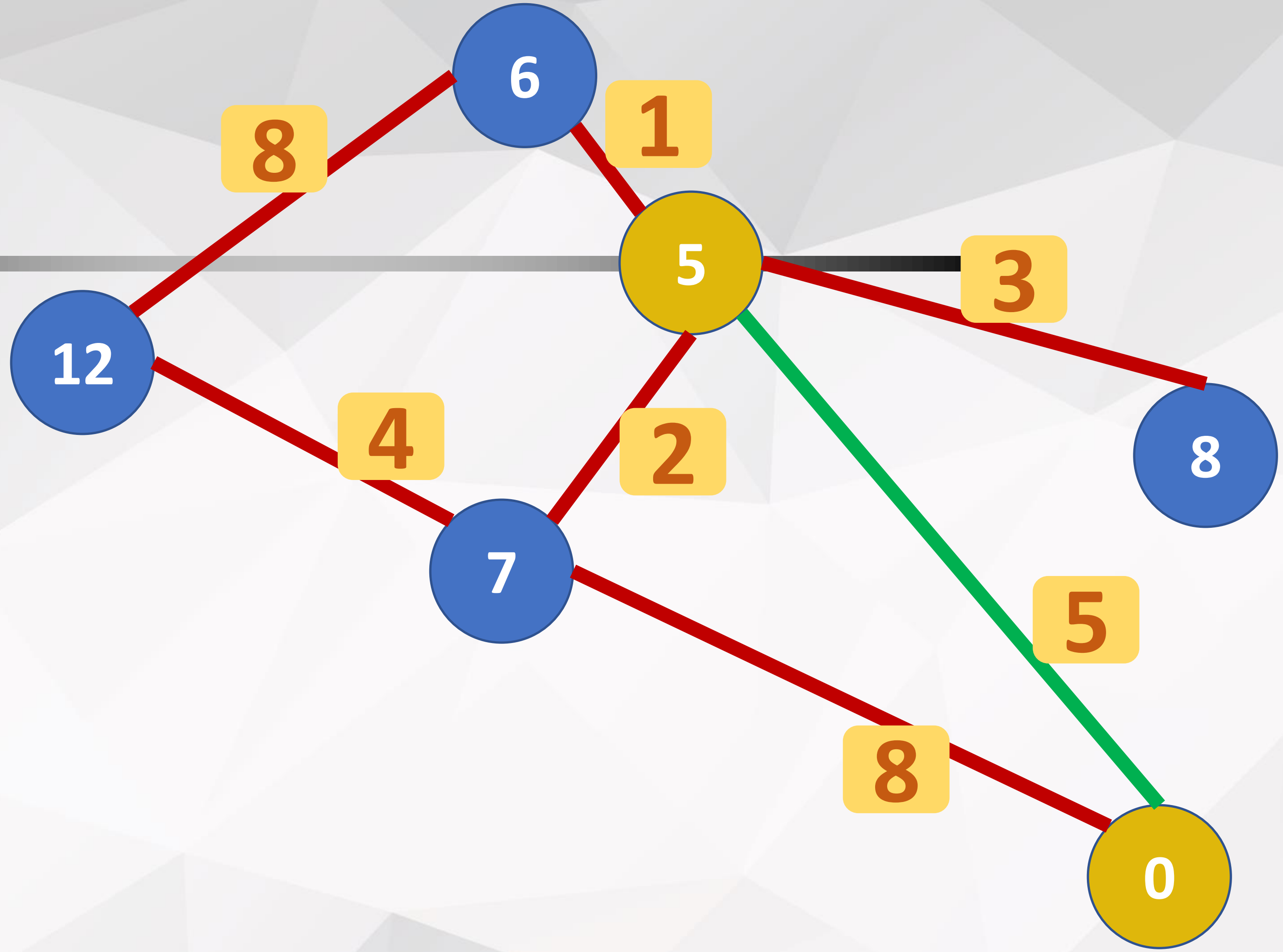


Relax!

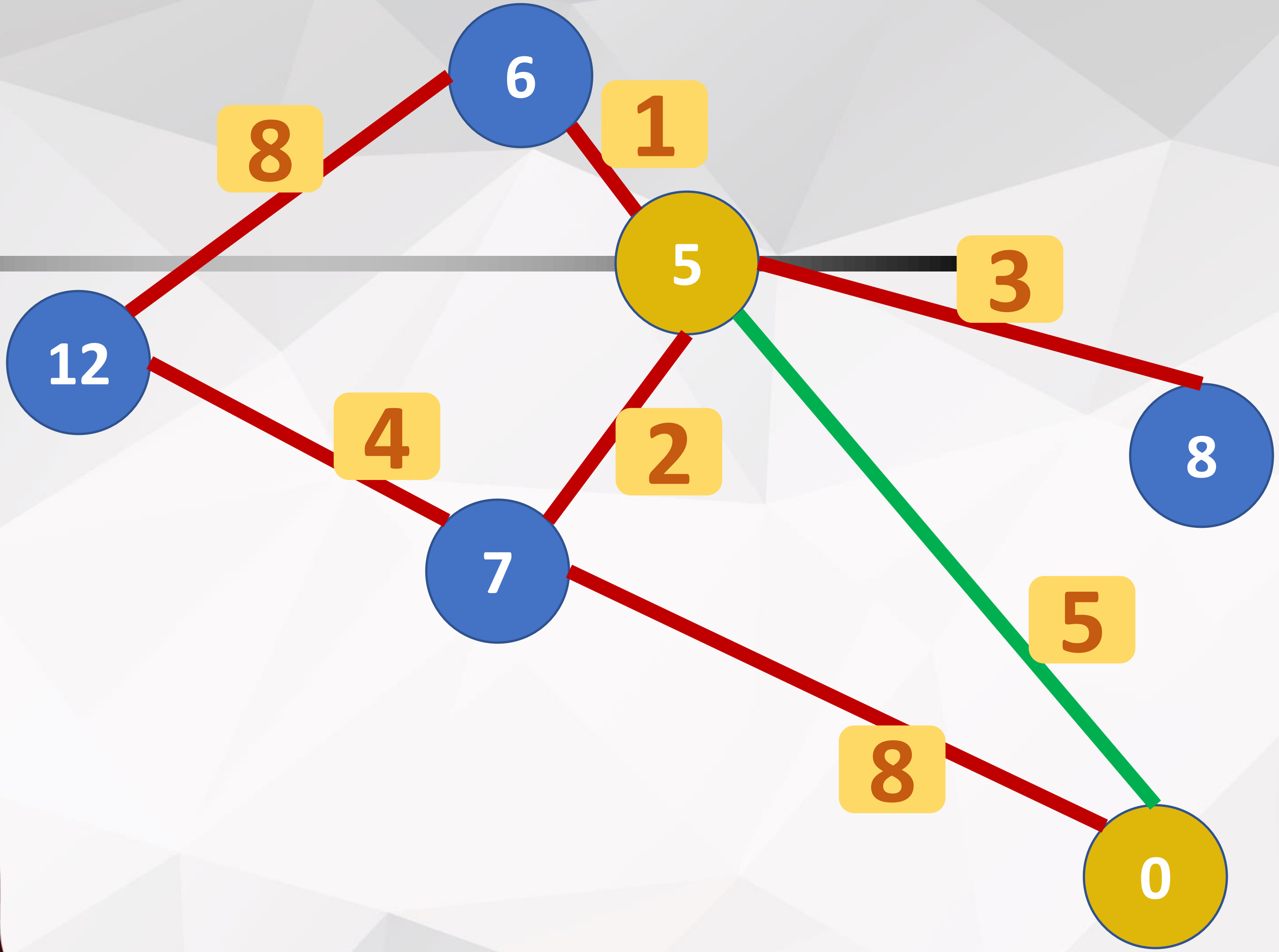


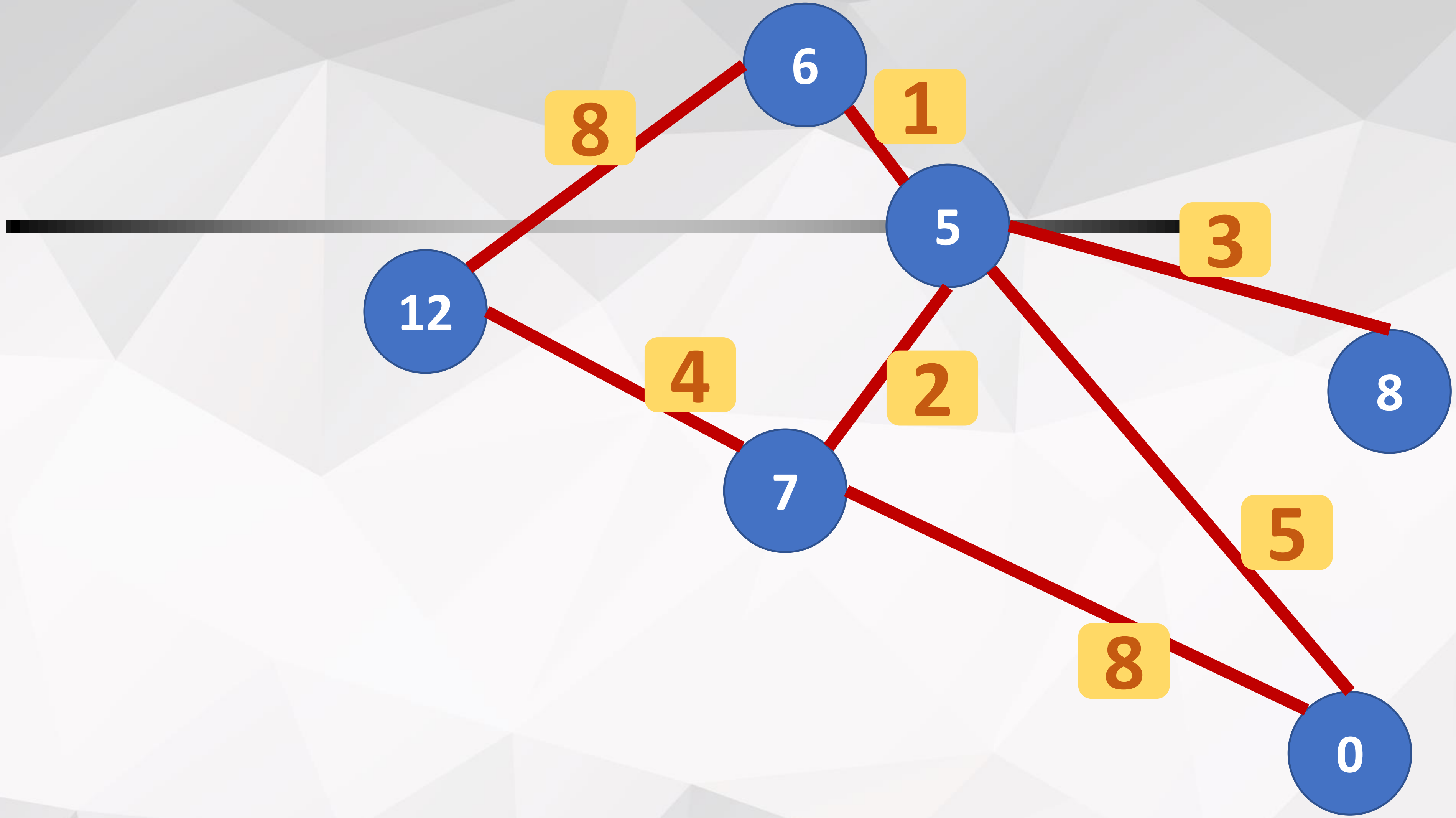

6
8
1
5
3
12
4
2
8
7
5
8
0

Relax?

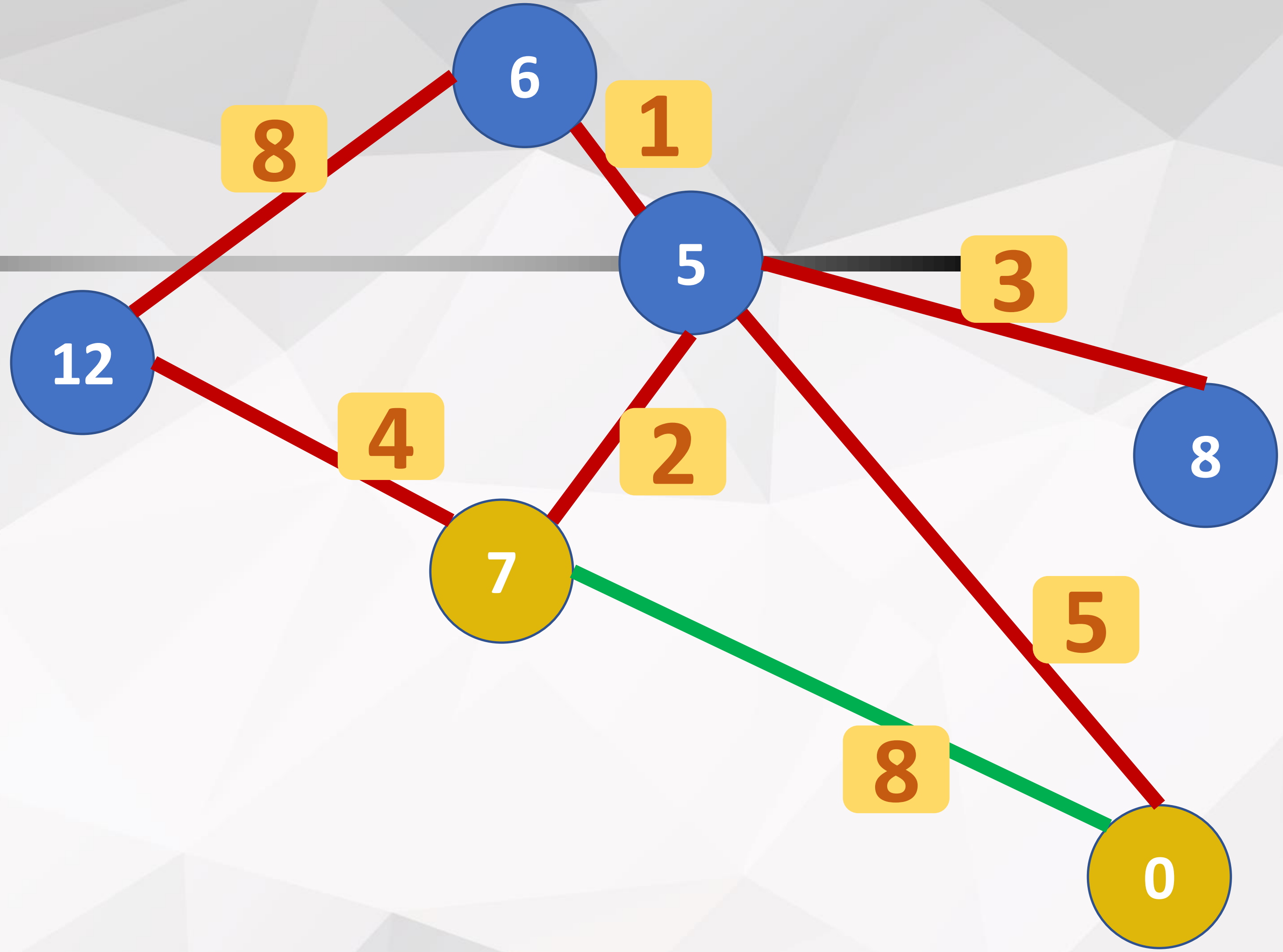


No

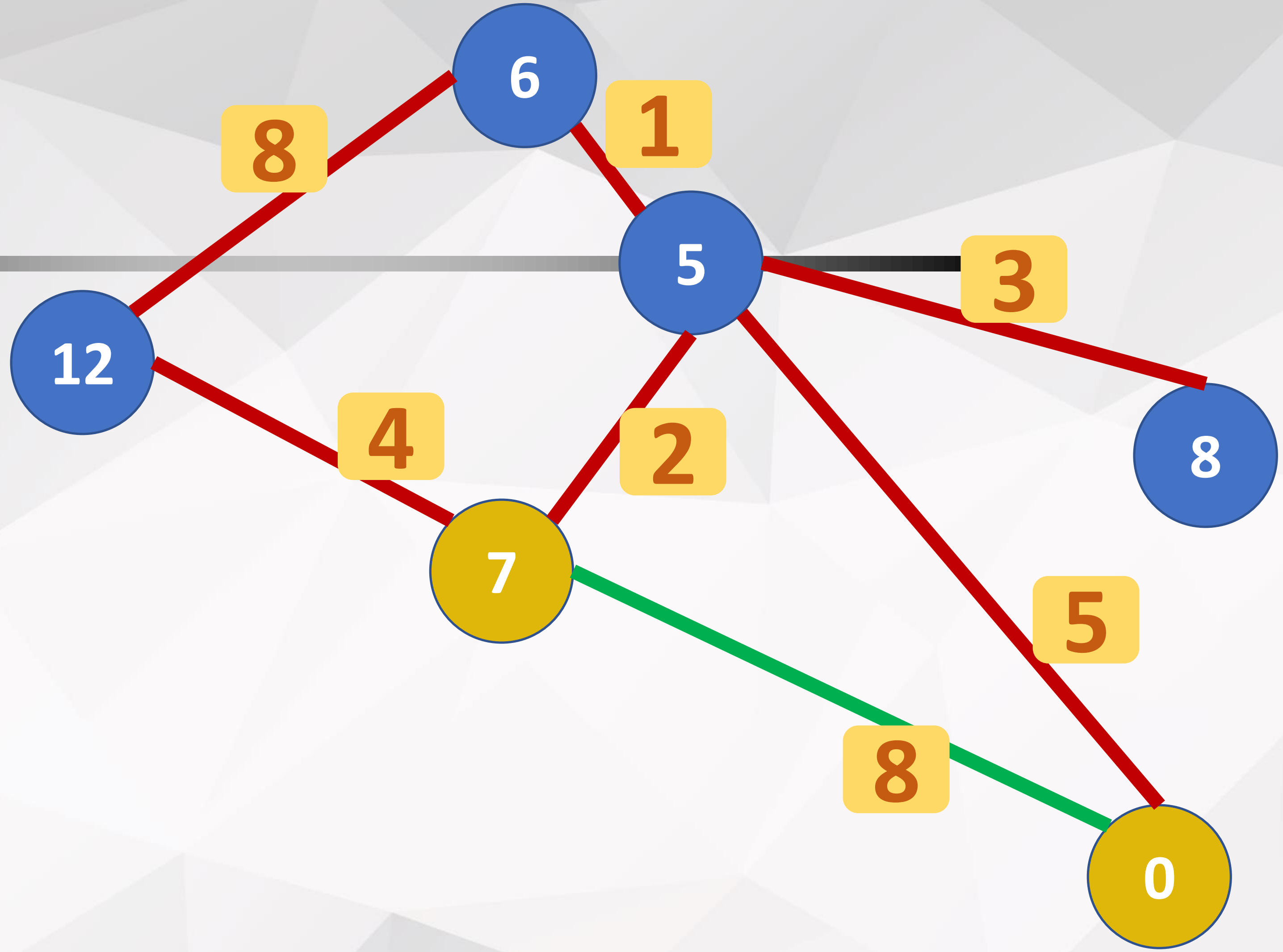


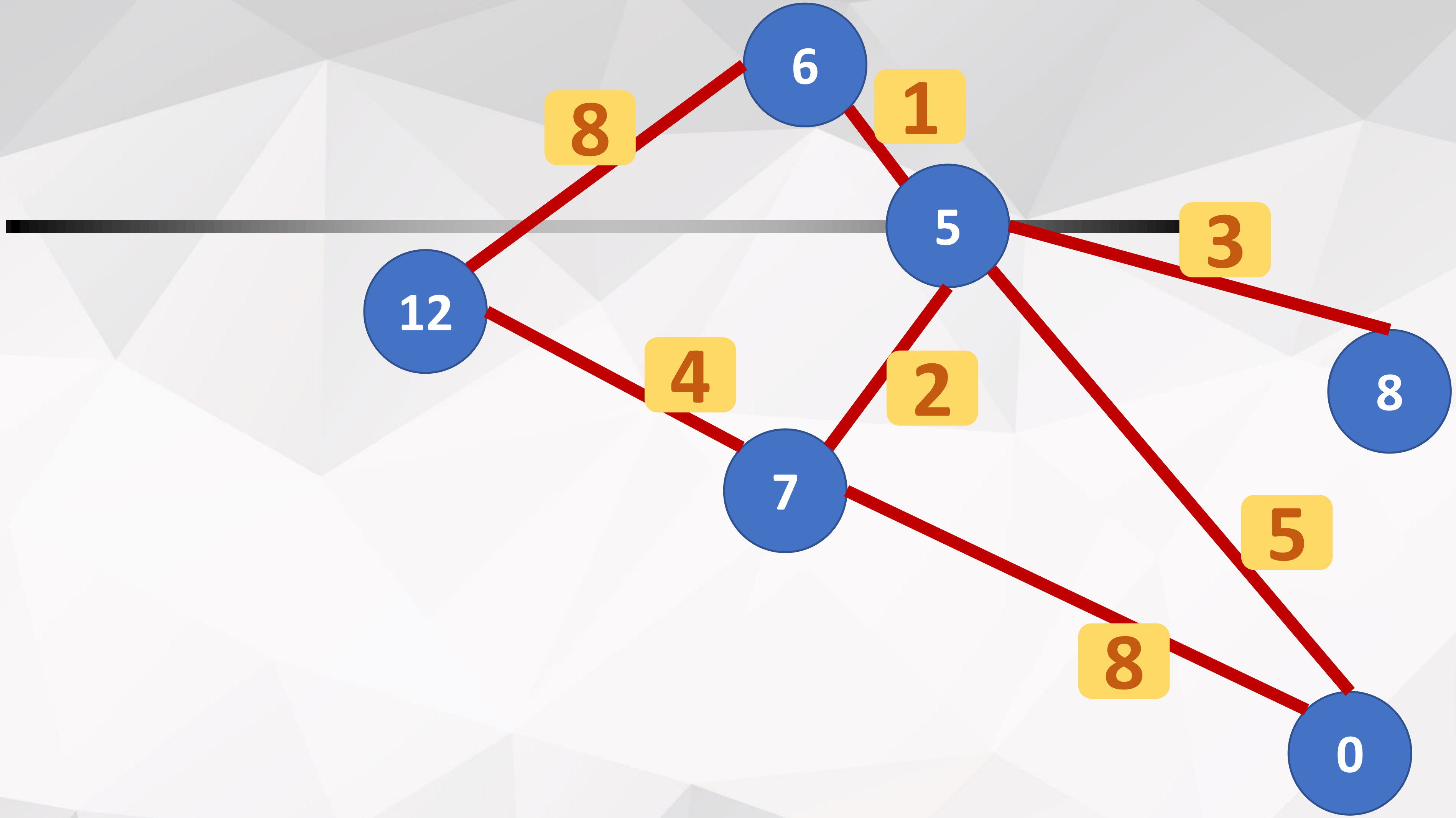

6
8
1
5
3
12
4
2
8
7
5
8
0

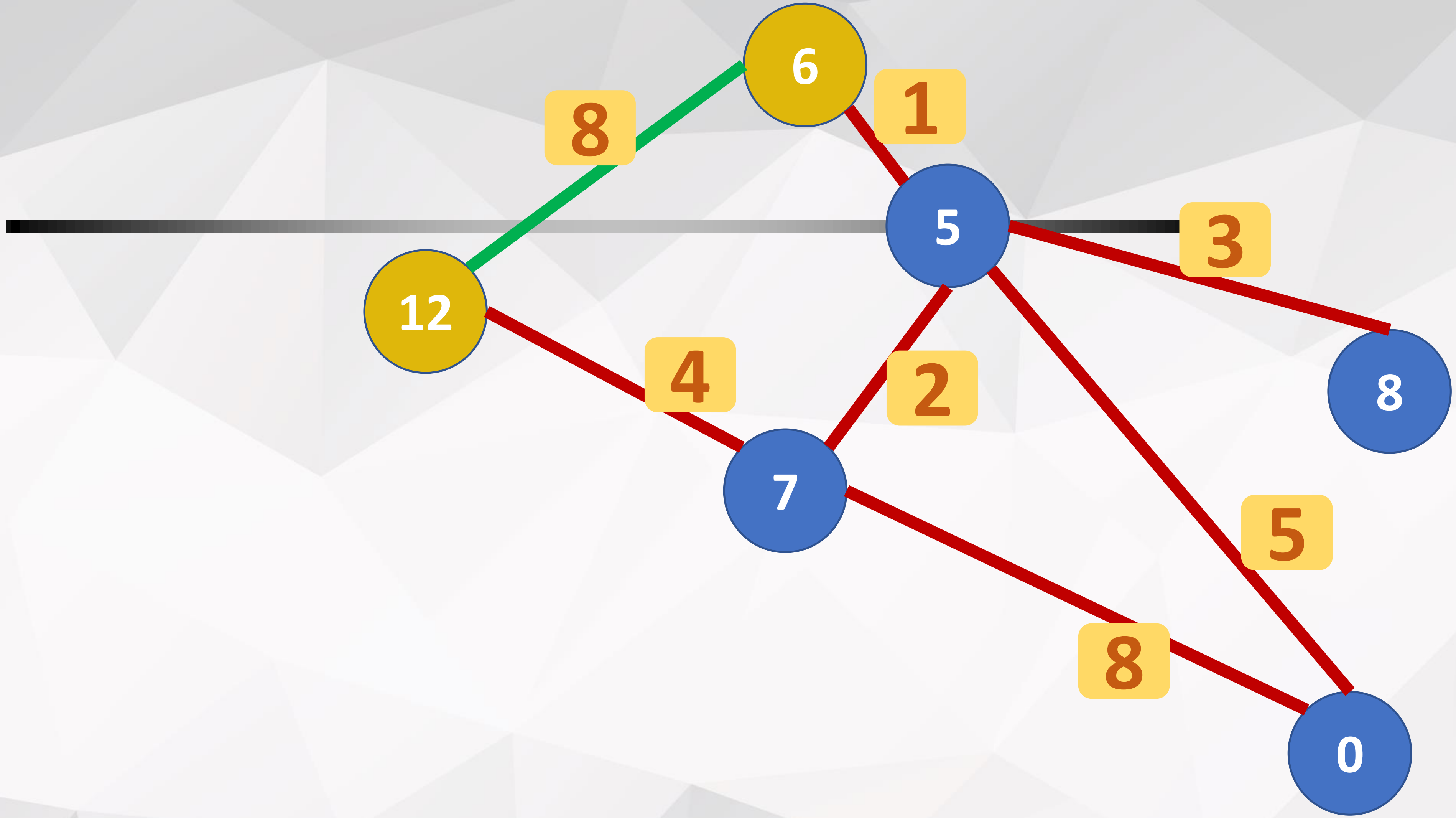
Relax?

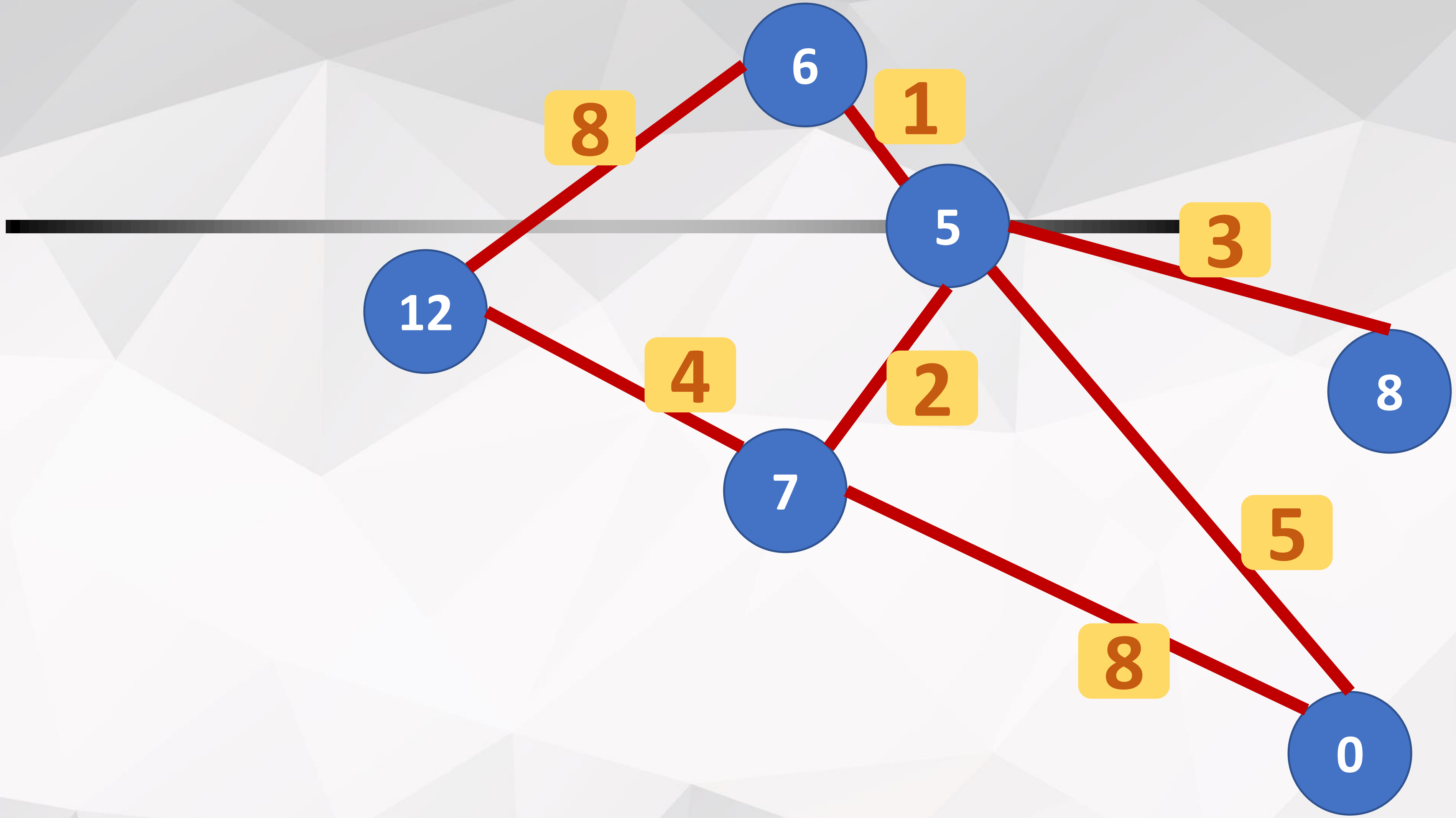


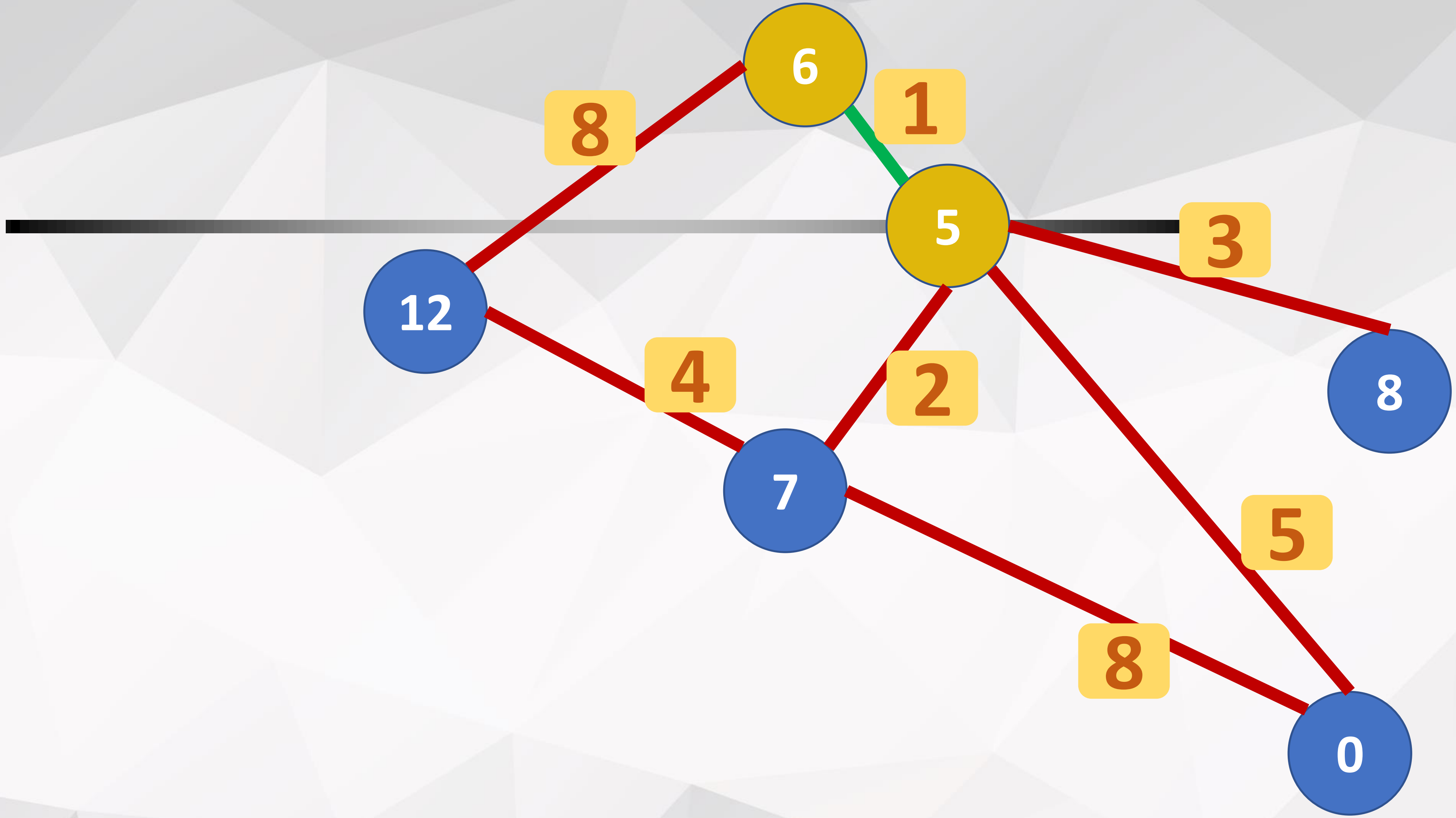
No

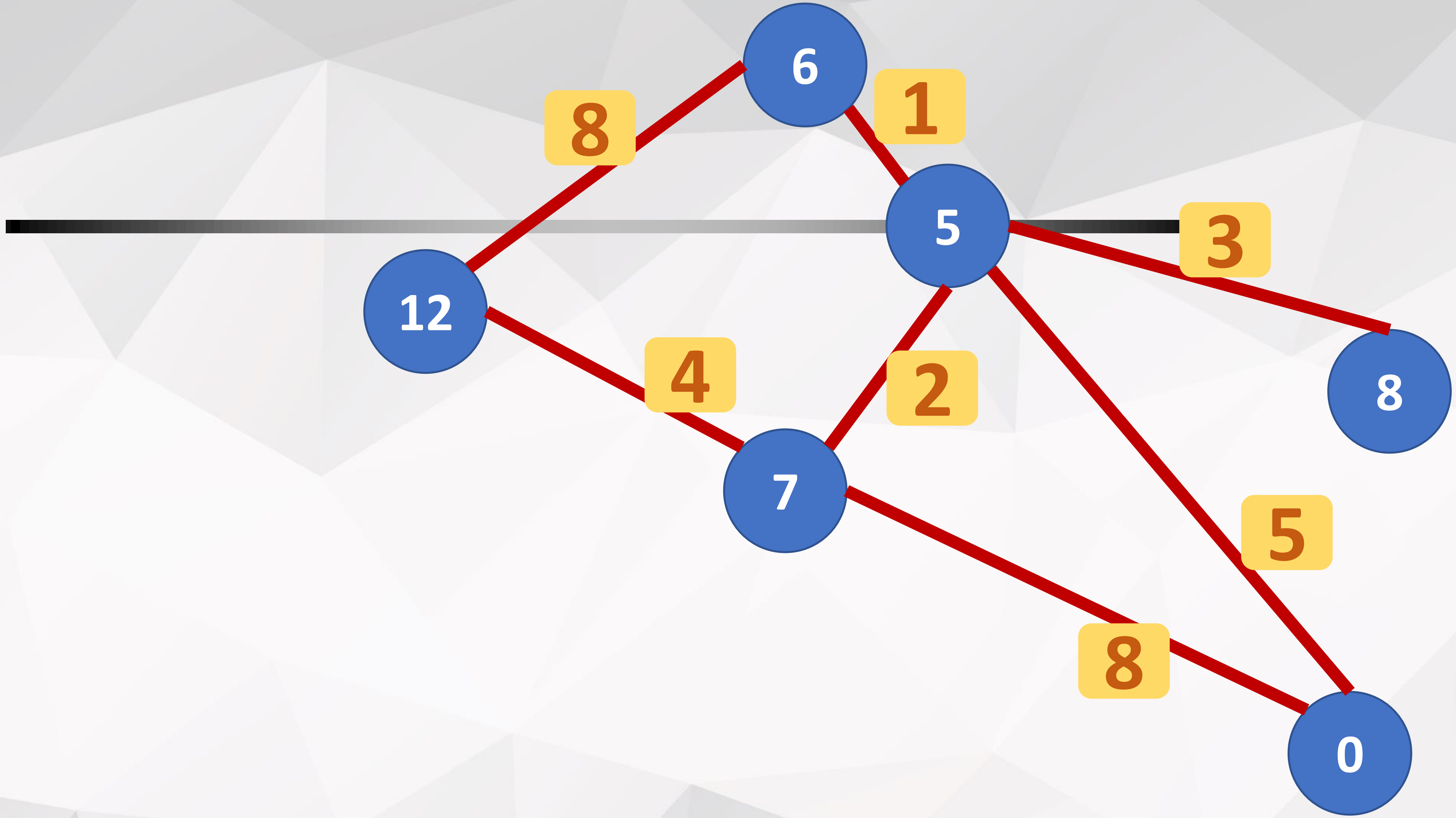


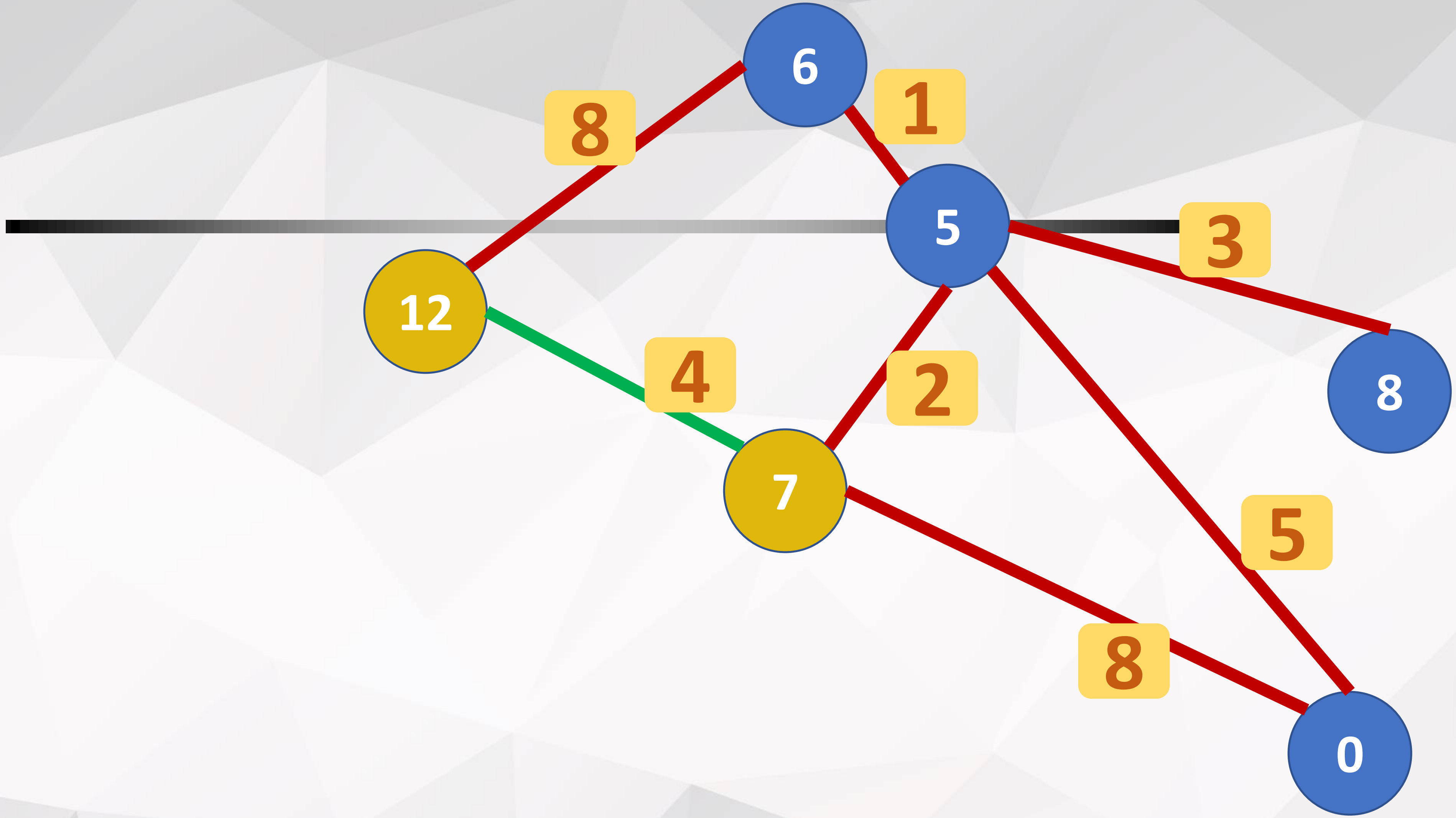

6
8
1
5
3
12
4
2
8
7
5
8
0


6
1
8
5
3
12
4
2
8
7
5
8
0

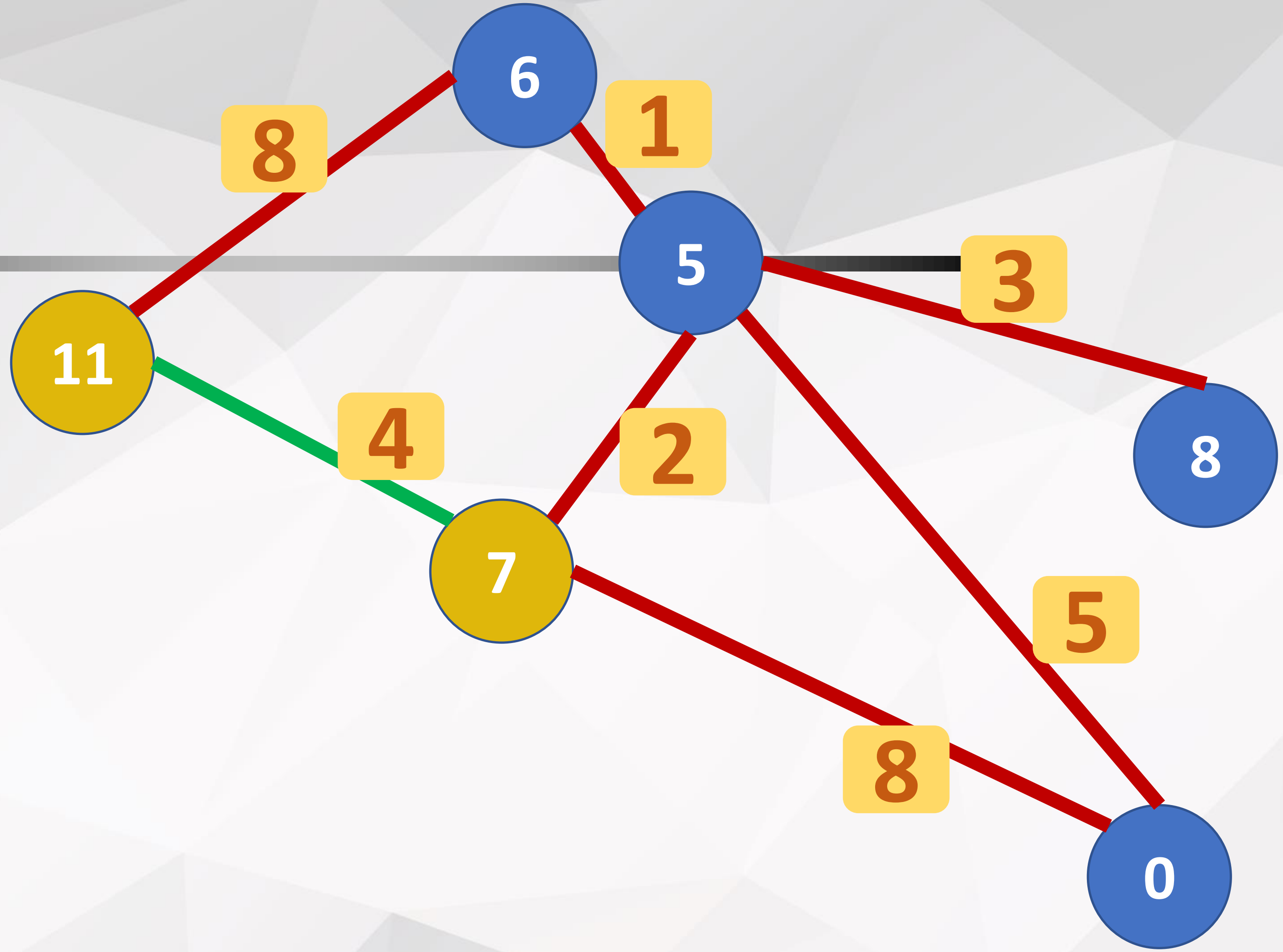

6
8
1
5
3
12
4
2
8
7
5
8
0

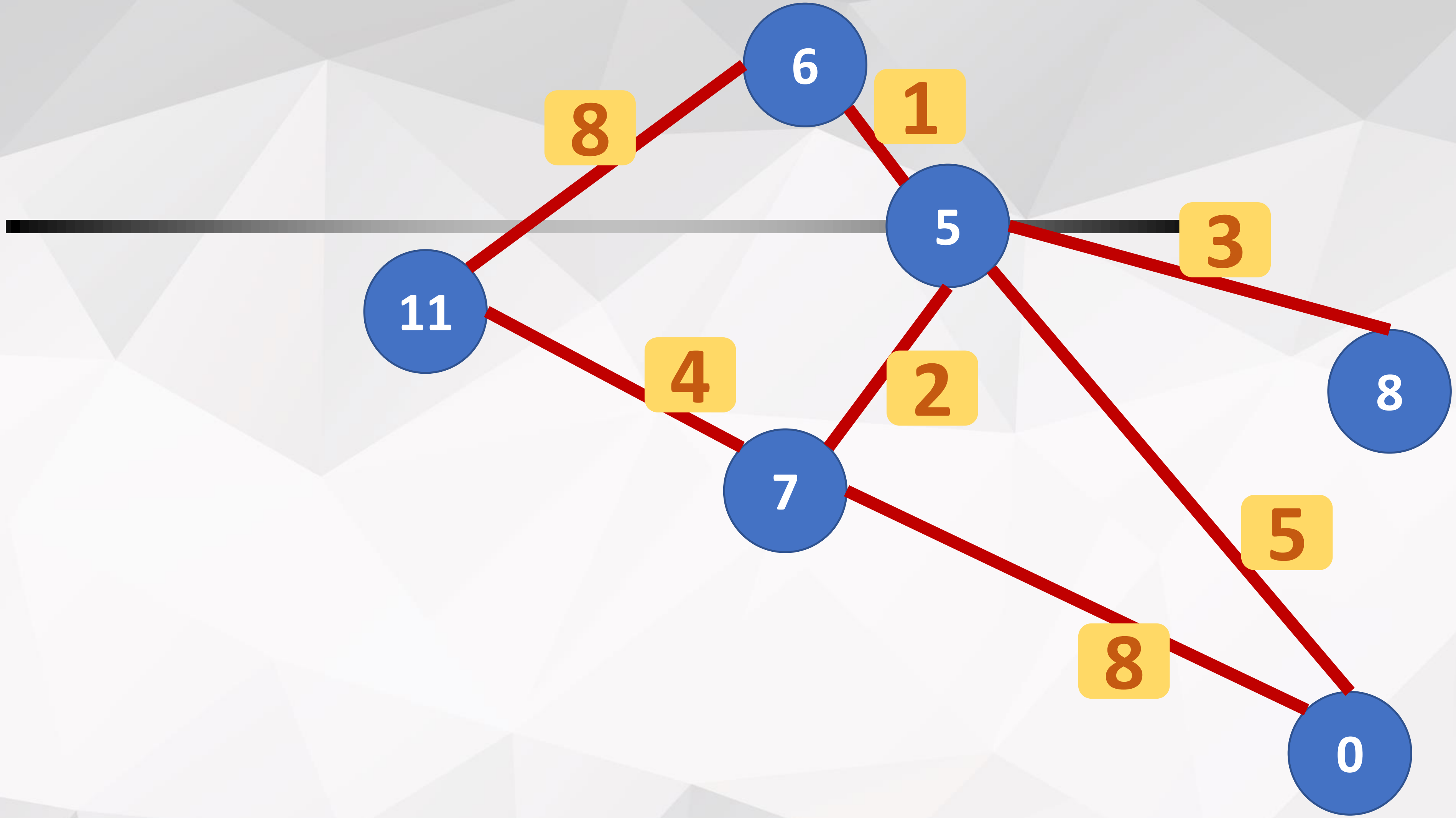

6
8
1
5
3
12
4
2
8
7
5
8
0

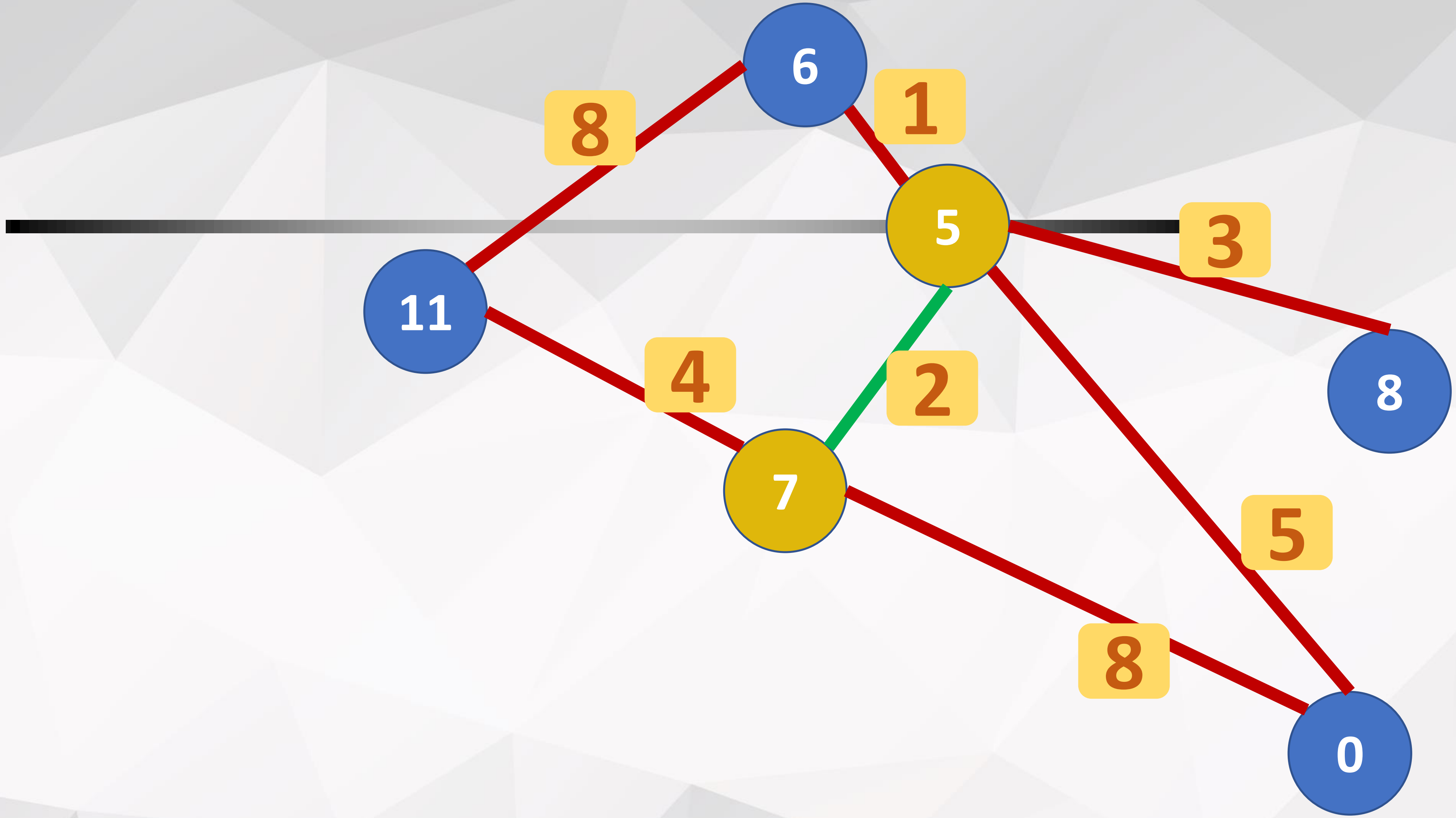

6
8
1
5
3
12
4
2
8
7
5
8
0

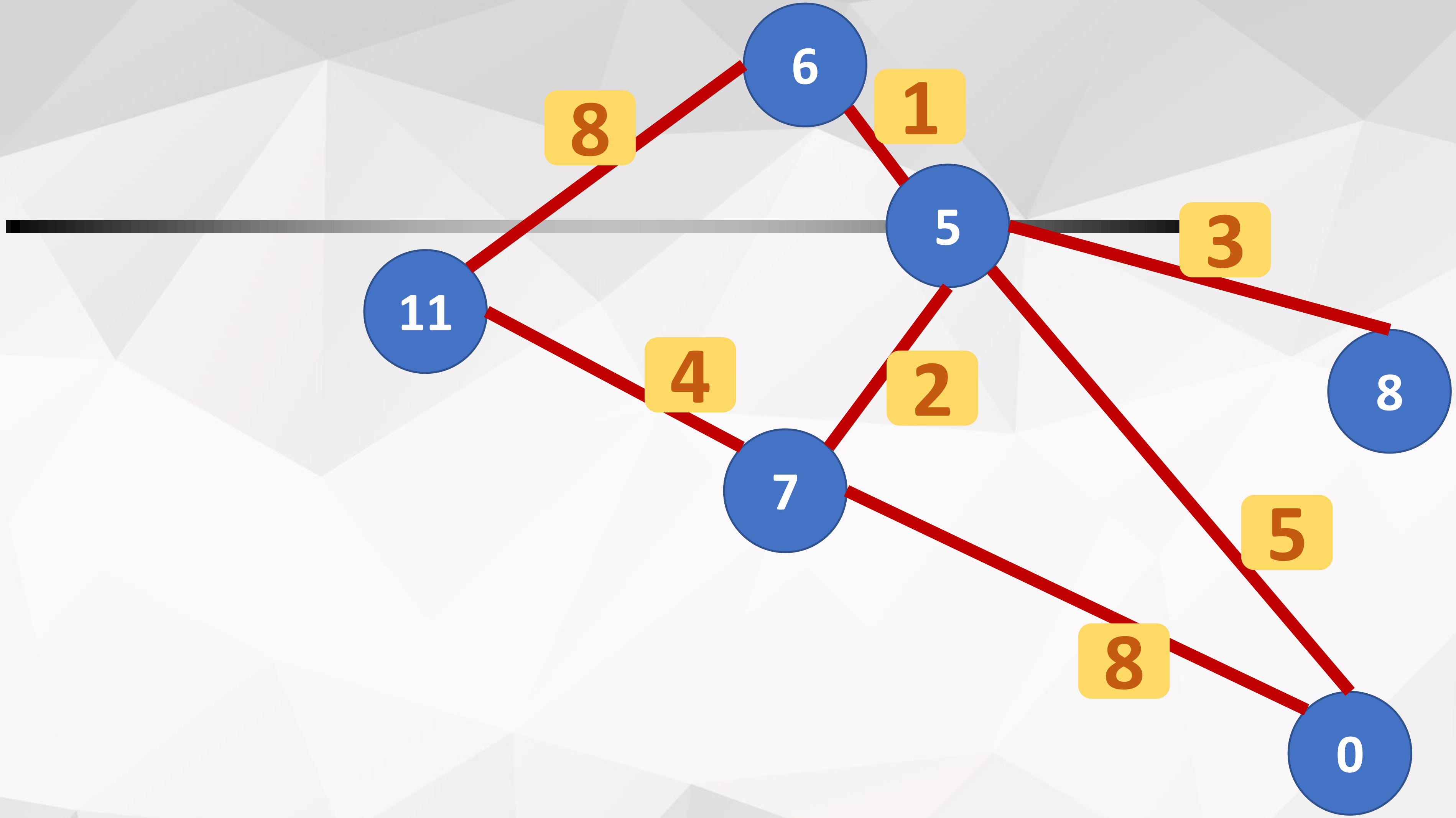

6
1
8
5
3
12
4
2
8
7
5
8
0

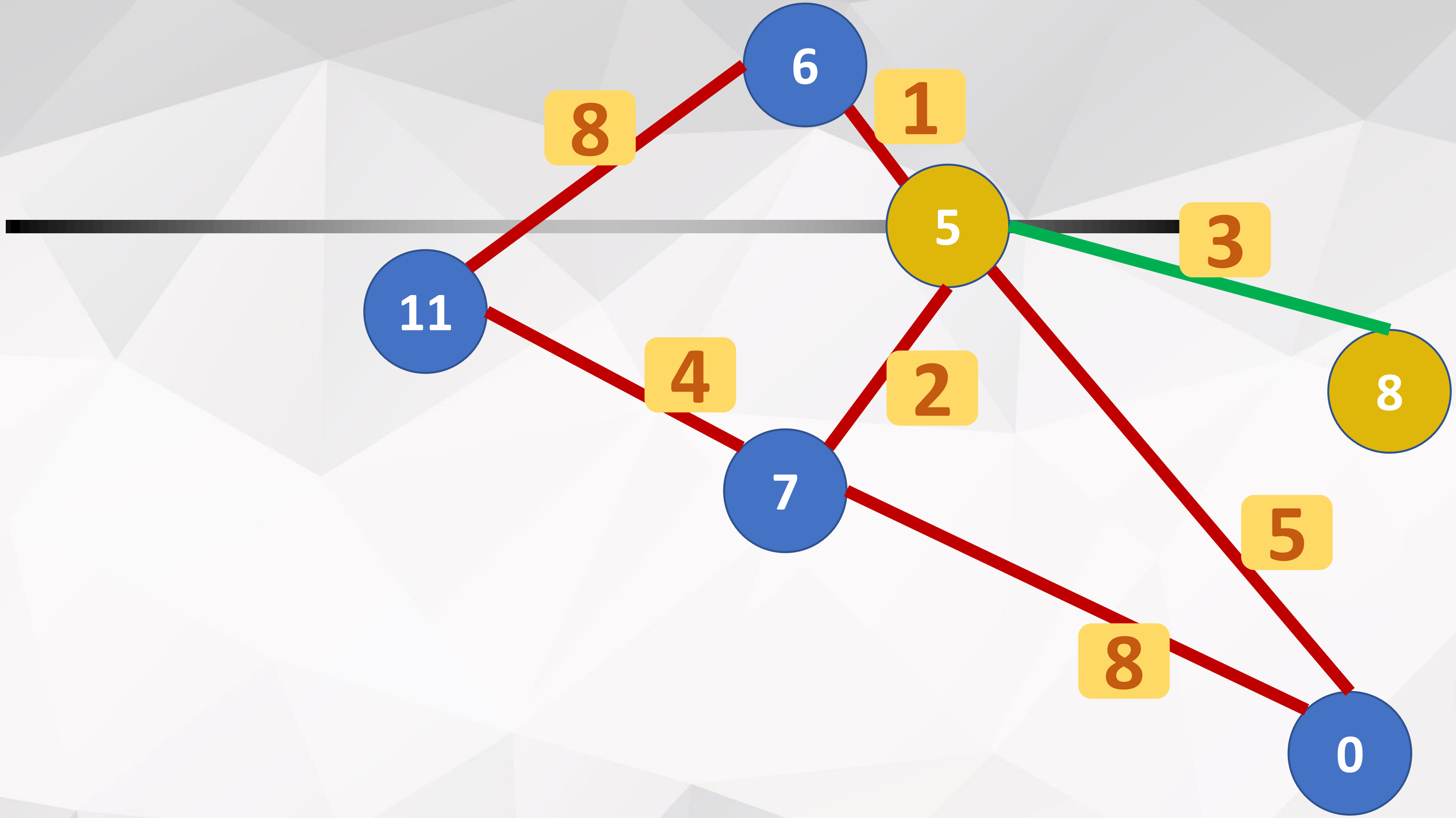
Relax!

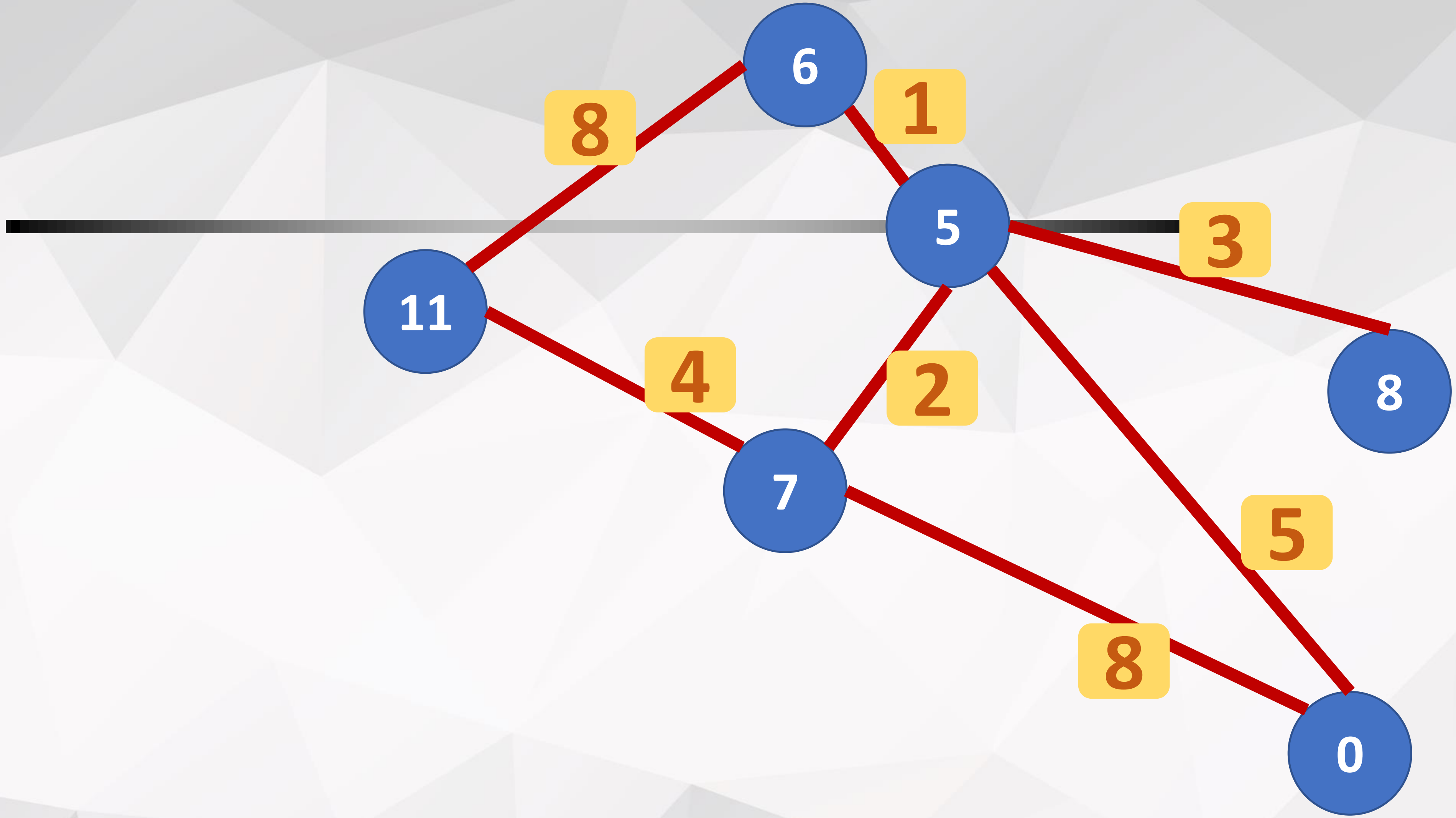


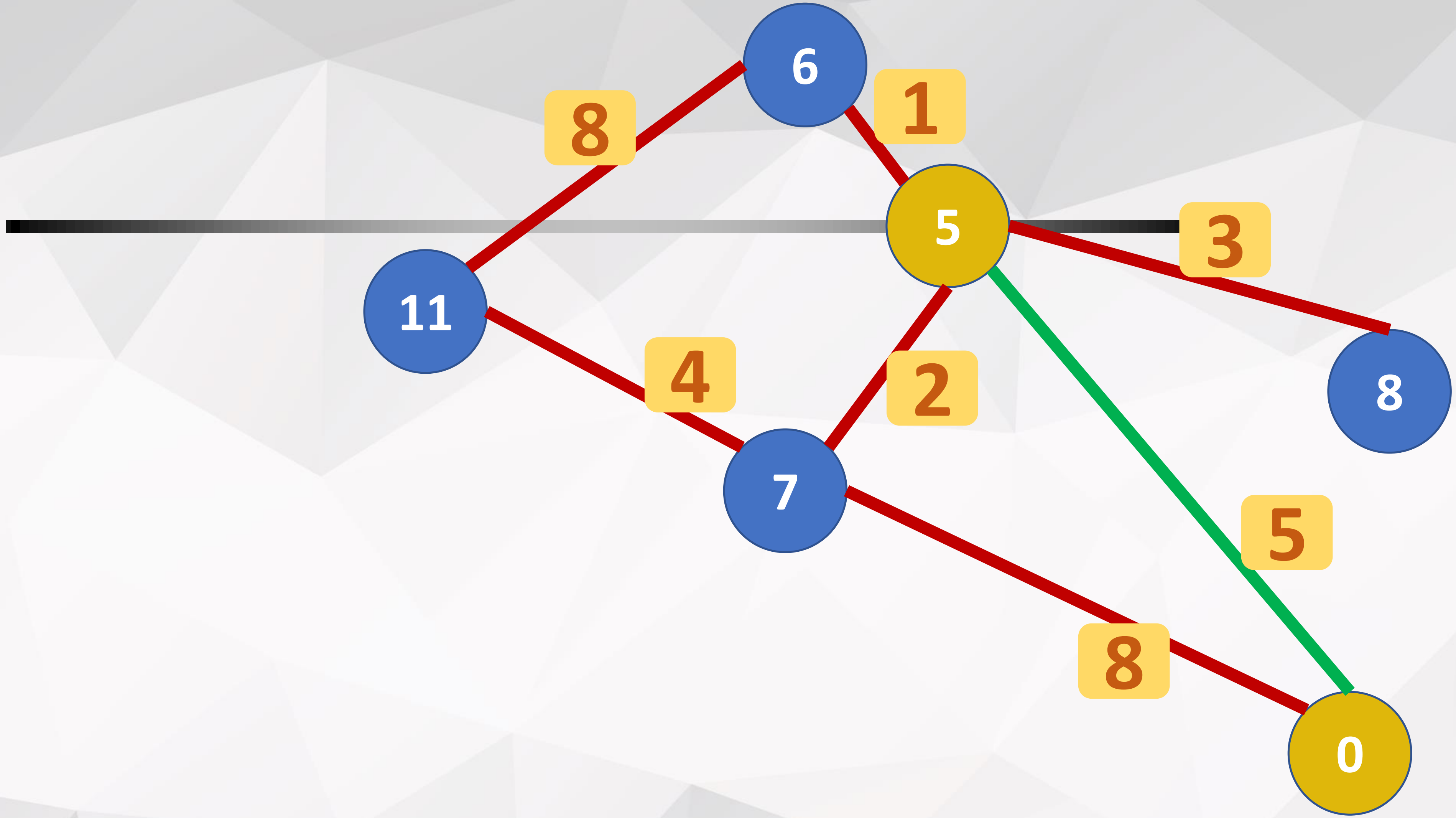

6
1
8
5
3
11
4
2
8
7
5
8
0

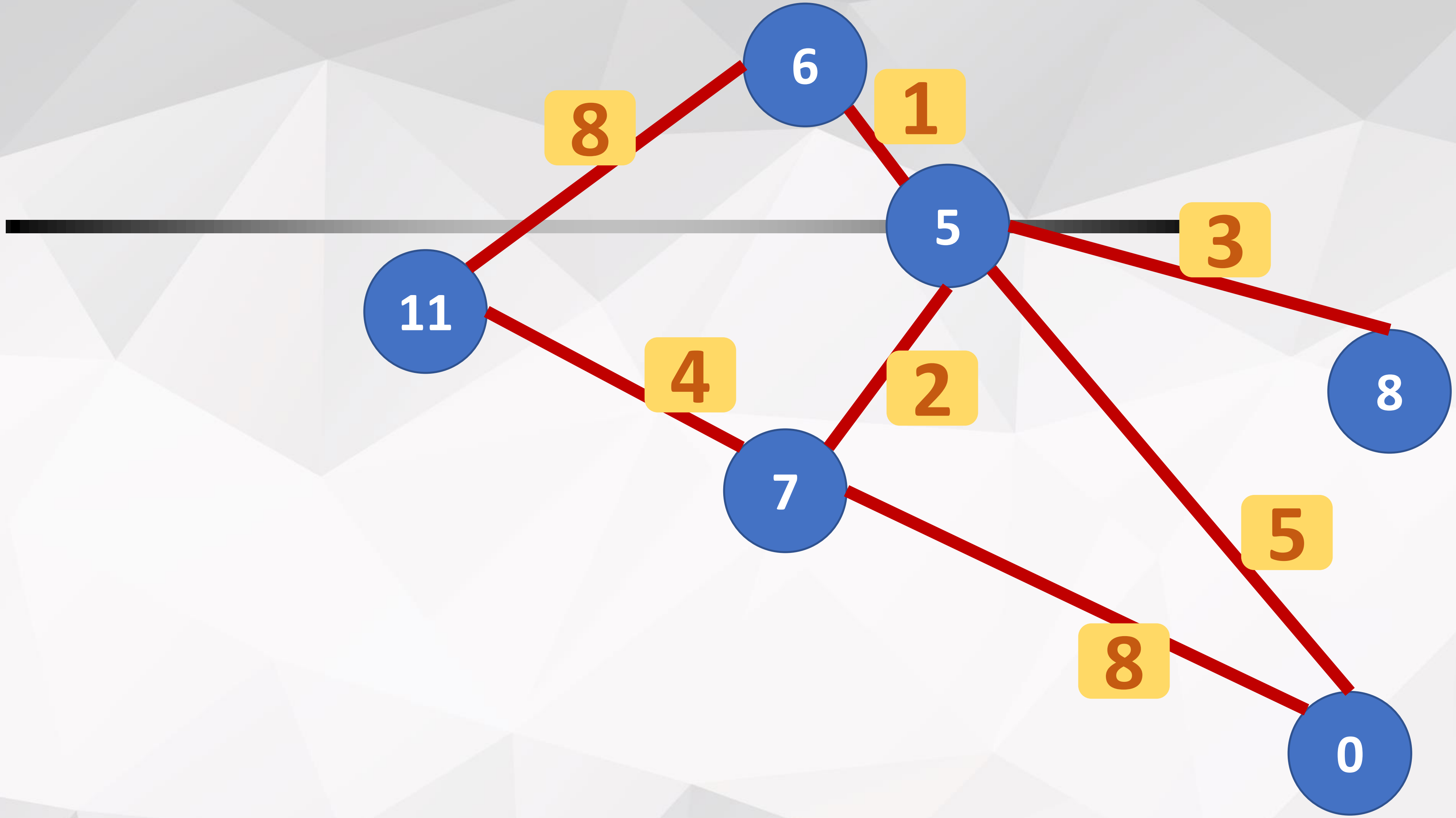

6
1
8
5
3
11
4
2
8
7
5
8
0

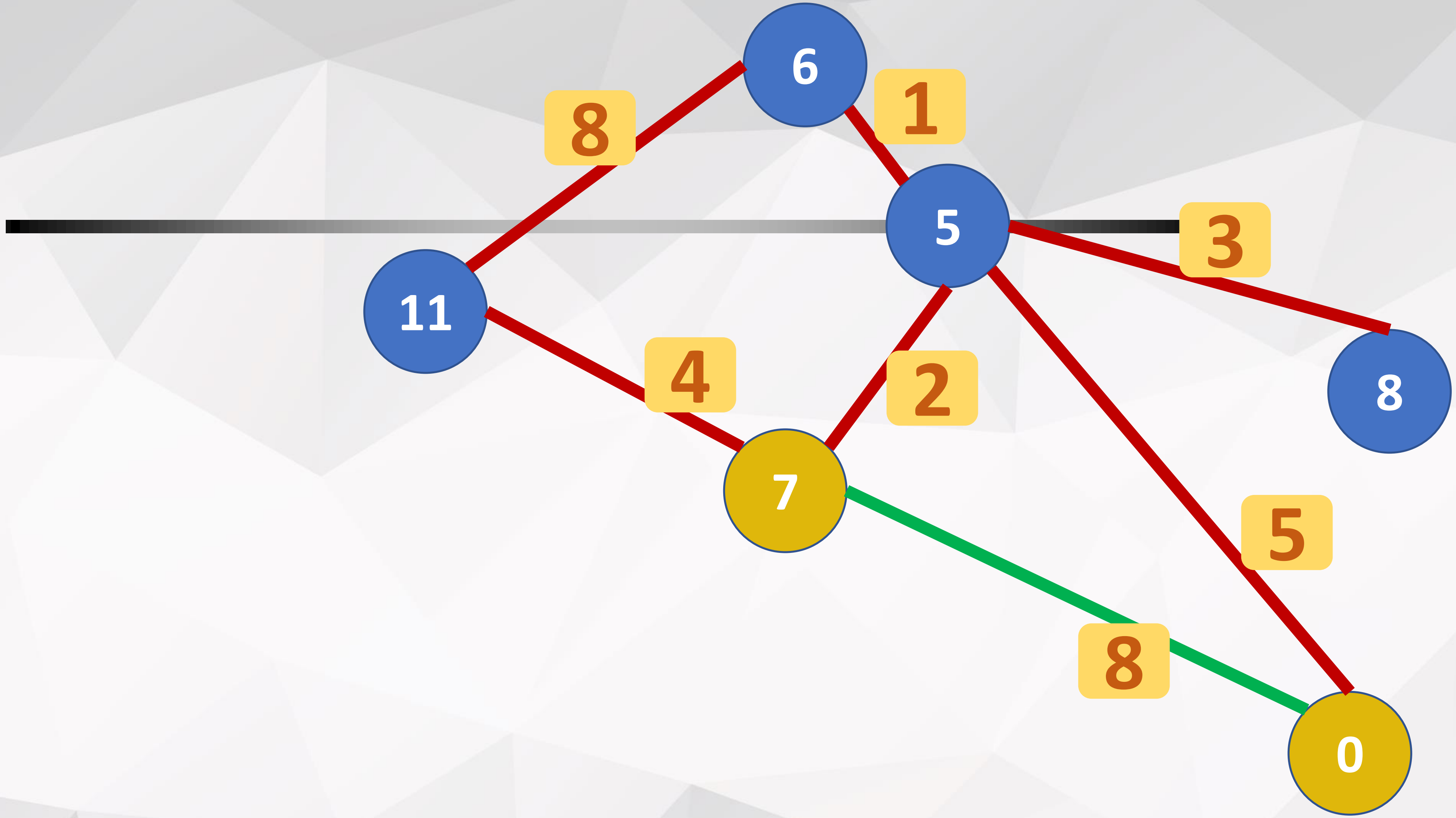

6
8
1
5
3
11
4
2
8
7
5
8
0

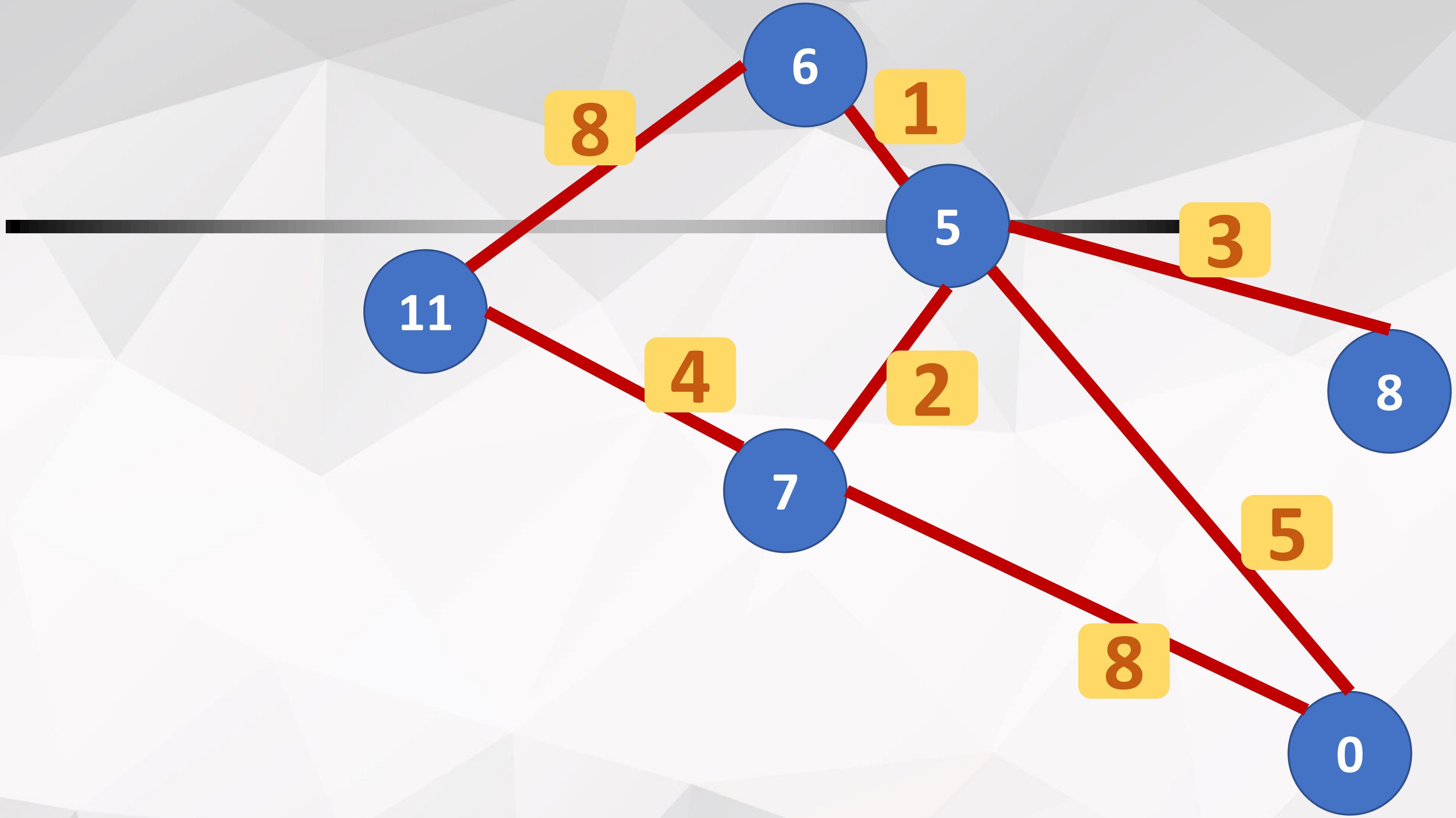

6
8
1
5
3
11
4
2
8
7
5
8
0


6
8
1
5
3
11
4
2
8
7
5
8
0

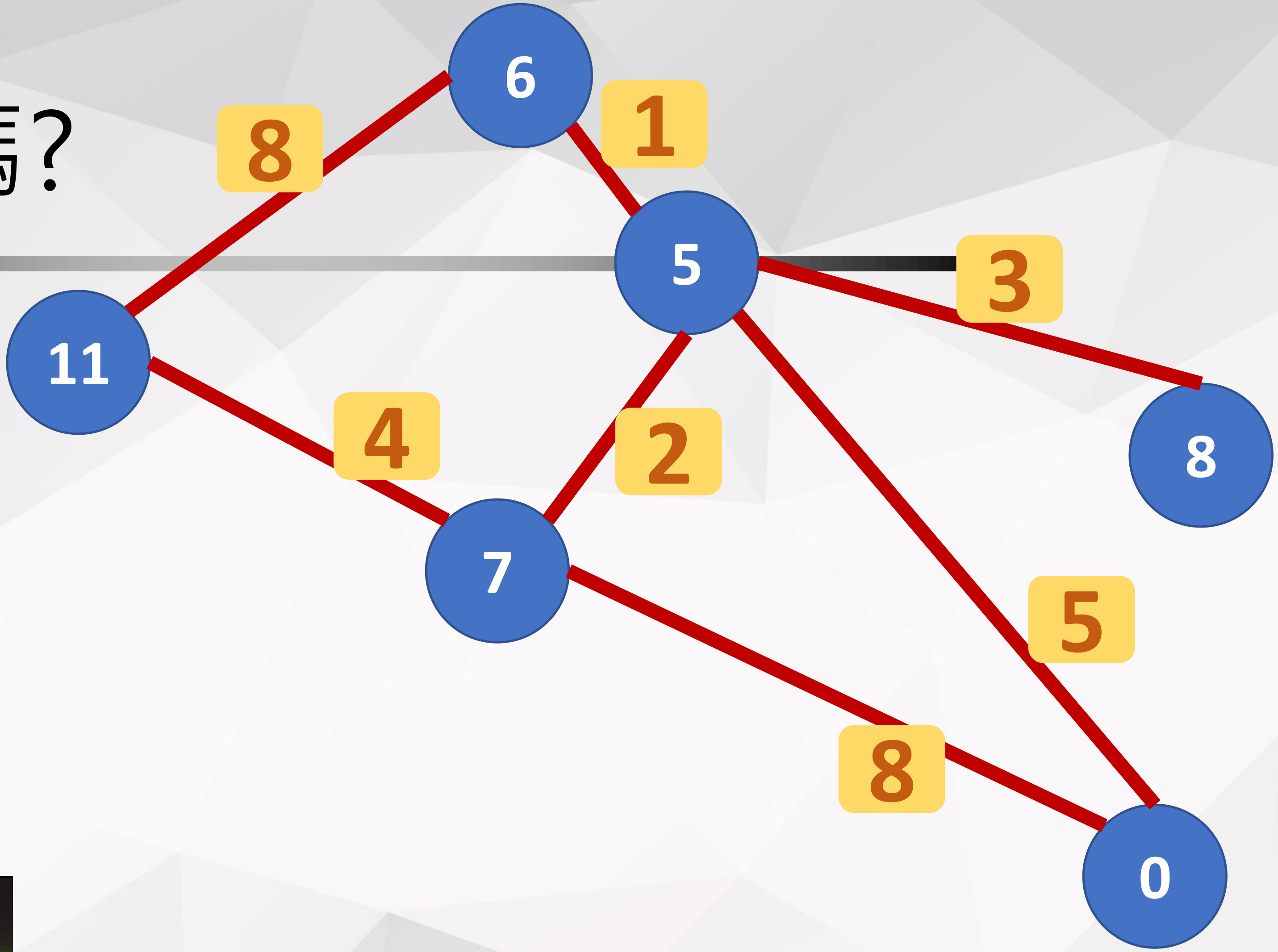

6
8
1
5
3
11
4
2
8
7
5
8
0


6
8
1
5
3
11
4
2
8
7
5
8
0


6
1
8
5
3
11
8
4
2
7
5
8
0


6
8
1
5
3
11
4
2
8
7
5
8
0

結束了嗎?



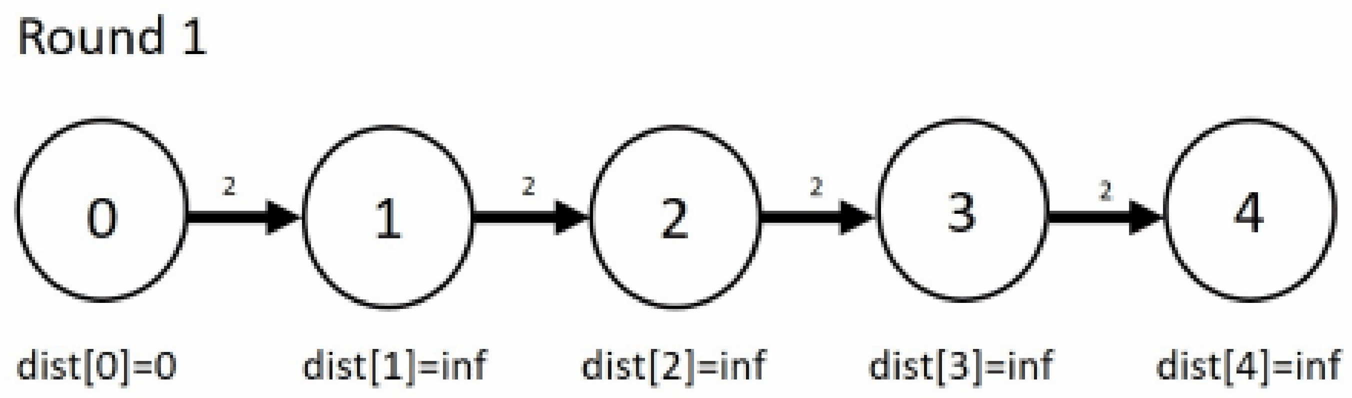
結束了嗎?

結束了。

可是

要**怎麼判斷**，每個點都得到**最短路**了?

路就是一條直直的


Round 1
0
2
1
2
2
2
3
2
4
dist[0]=0
dist[1]=inf
dist[2]=inf
dist[3]=inf
dist[4]=inf

所以

- 至多要做 |V| - 1 次的全部邊 relaxtion
  - 剛剛的例子只做了 3 遍
  - 自行證明吧

負權重邊

對每邊做至多 |V|−1 次 relaxation 後，

要是某邊還能 relaxation，

負權重邊

對每邊做至多 $|V|-1$ 次 relaxation 後，

要是某邊還能 relaxation，
就表示有**負權重邊**能使路徑成本一直降低。

Questions?

練習

UVa OJ 558 Wormholes

Outline

- 術語複習
  - Graph
  - Tree
- 最小生成樹
- A* 搜尋法則
- 單源最短路徑
- 全點對最短路徑

A* search

評估函數

下一步到底該往哪走?

評估函數

下一步到底該往哪走? (透過轉移方程找可走鄰點)

評估函數

下一步到底該往哪走?

走下去，會更好嗎?

評估函數

下一步到底該往哪走?

走下去，會更好嗎?

好或不好，就是由評估函數決定

得自行設計

評估函數

- g(n): 從**起**點到 n 點的成本
- h(n): 從 n 點到**終**點的成本

f(n) = g(n) + h(n): 評估函數

例如

當求帶權重圖的單源最短路徑

g(n) = 從起點到 n 的最小成本
h(n) = 0

這個是， Dijkstra 演算法

例如

當求二維平面圖的單點到單點最短路徑

g(n) = 0
h(n) = n 點到終點的歐幾里得距離

這個是， Best-first search 演算法
不是 Breadth-first search

Outline

- 術語複習
  - Graph
  - Tree
- 最小生成樹
- A* 搜尋法則
- 單源最短路徑
- 全點對最短路徑

All-Pairs Shortest Paths

Competitive Programming

Made by 培訓團隊群 CCNS.

APSP

- 問**任意點**到**任意點**的最小成本

樸素解

- 利用剛才教的 SSSP 演算法們
- 對每個點都設定為源點 (source)

樸素解

- 利用剛才教的 SSSP 演算法們
- 對每個點都設定為源點 (source)
- 當然可以!

全點對最短路徑

- Floyd-Warshall's Algorithm
- Johnson's Algorithm

全點對最短路徑

- Floyd-Warshall's Algorithm
- Johnson's Algorithm

Floyd-Warshall's Algorithm

Competitive Programming

Made by 培訓團隊群

Floyd-Warshall 實作

```
int s[maxn][maxn];

for (int i = 1; i <= N; i++)
  for (int j = 1; j <= N; j++)
    s[i][j] = G[i][j];

for (int k = 1; k <= N; k++)
  for (int i = 1; i <= N; i++)
    for (int j = 1; j <= N; j++)
      s[i][j] = min(s[i][j], s[i][k] + s[k][j]);
```

狀態/轉移方程

設定狀態 $s(\mathrm{i},\mathrm{j},k)$ 為 i 到 j 只以 $\{1,..,k\}$ 為中間點的最小成本

狀態/轉移方程

設定狀態 $s(\mathrm{i},\mathrm{j},k)$ 為 i 到 j 只以 $\{1,..,k\}$ 為中間點的最小成本

$$s(\mathrm{i},\mathrm{j},k)=\begin{cases} s(i,j,k-1) & \text{若無經過 } k \\ \mathrm{s}(\mathrm{i},k,k-1)+s(k,j,k-1) & \text{若有經過 } k \end{cases}$$

狀態/轉移方程

設定狀態 $s(\mathrm{i},\mathrm{j},k)$ 為 i 到 j 只以 $\{1,..,k\}$ 為中間點的最小成本

$$s(\mathrm{i},\mathrm{j},k)=\begin{cases} s(i,j,k-1) & \text{若無經過 } k \\ \mathrm{s}(\mathrm{i},k,k-1)+s(k,j,k-1) & \text{若有經過 } k \end{cases}$$

$$s(\mathrm{i},\mathrm{j},k)\ =min(s(i,j,k-1),\mathrm{s}(\mathrm{I},k,k-1)+s(k,j,k-1))$$

邊界

$$s(\mathrm{i},\mathrm{j},0)=\begin{cases}0 & \text{若 } i=j\\ weight(i,j) & \text{若有 } (i,j) \text{ 邊}\\ \infty & \text{若無 } (i,j) \text{ 邊}\end{cases}$$

全點對最短路徑

- Floyd-Warshall's Algorithm
- Johnson's Algorithm

Johnson's Algorithm

Johnson's Algorithm

結合了 Bellman-ford 與 Dijkstra 的演算法

其複雜度為兩個演算法複雜度相加

Johnson's Algorithm

自行研究吧

Questions?