GRAPH

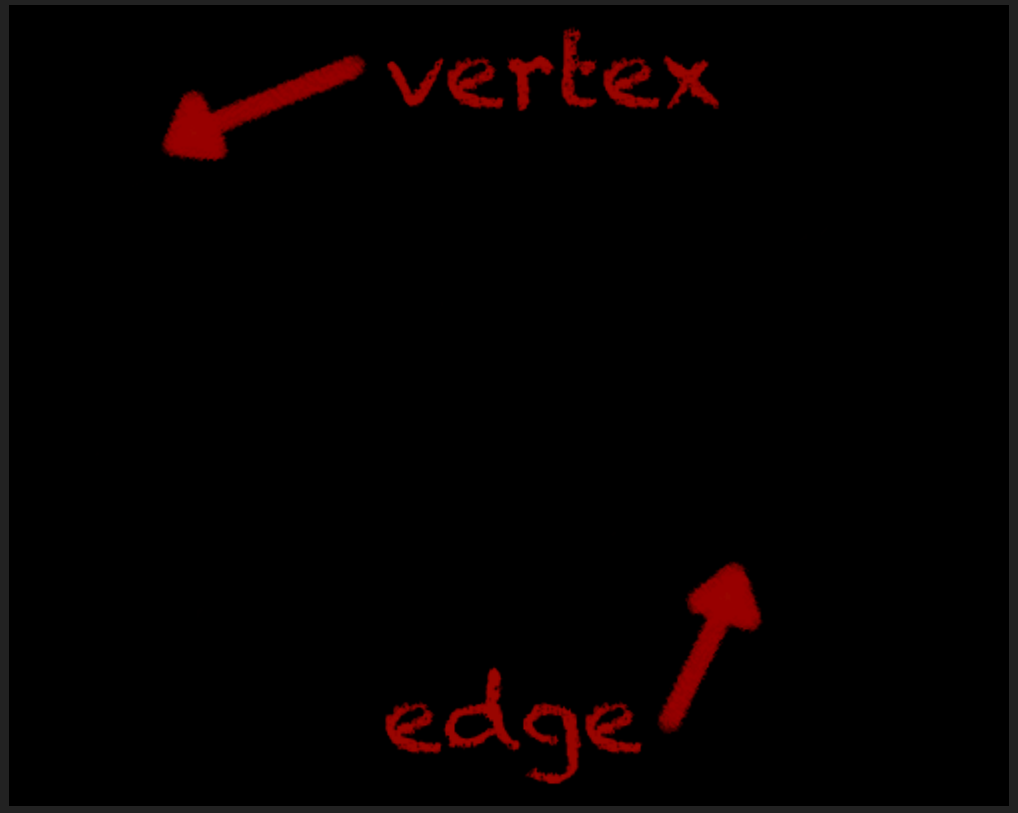
COURSE OUTLINE

- Basic Graph (review)
- Shortest Path
- Minimum Spanning Tree

BASIC GRAPH

DEFINITION

- $G = (V, E)$



MORE DEFINITIONS

- weighted vs unweighted
- directed vs undirected

IMPLEMENTATION

UNDIRECTED GRAPH

```
vector< int > v[ N ] ;
// an edge between a and b
v[ a ].push_back( b ) ;
v[ b ].push_back( a ) ;
```

DIRECTED GRAPH

```
vector< int > v[ N ] ;
// an edge from a to b
v[ a ].push_back( b ) ;
```

UNWEIGHTED GRAPH

```
vector< int > v[ N ] ;
// an edge between a and b
v[ a ].push_back( b ) ;
v[ b ].push_back( a ) ;
```

WEIGHTED GRAPH

```
typedef pair< int , int > ii ;
vector< ii > v[ N ] ;
// an edge between a and b with weight w
v[ a ].push_back( ii( b , w ) ) ;
v[ b ].push_back( ii( a , w ) ) ;
// v[ a ].emplace_back( b , w ) ;
// v[ b ].emplace_back( a , w ) ;
```

TRAVERSING NEIGHBOR

```
vector< int > v[ N ] ;
// go through all vertices adjacent to a
for ( int x : v[ a ] ) {
  // x: a vertice directly linked to by "a"
}
```

```
typedef pair< int , int > ii ;
vector< ii > v[ N ] ;
// go through all vertices adjacent to a
for ( ii x : v[ a ] ) {
  int b = x.first , w = x.second
  // b: a vertice directly linked to by "a"
  // w: weight(a, b)
}
```

SHORTEST PATH

PATH

- A path on a graph $G = (V, E)$ is a sequence of vertices
- $v_1, v_2, v_3, \ldots, v_{n-1}, v_n \in V$
- satisfying $(v_i, v_{i+1}) \in E \, (1 \leq i < n)$

WEIGHT OF A PATH

- The weight of a path $v_1, v_2, v_3, \ldots, v_{n-1}, v_n \in V$
- is the sum $\sum_{i=1}^{n-1} \text{weight}(v_i, v_{i+1})$

SHORTEST PATH

- The shortest path from a to b is the path with minimum
- weight among all paths $v_1, v_2, v_3, \ldots, v_{n-1}, v_n$
- satisfying $v_1 = a$ and $v_n = b$

HOW TO FIND THE SHORTEST PATH

SHORTEST PATH ON UNWEIGHTED GRAPH

- Breadth-First Search (BFS)
- Time: $O(V + E)$

0-1 BFS

- Implement with a deque
- Time: $O(V + E)$

SHORTEST PATH ON WEIGHTED GRAPH

- Floyd-Warshall Algorithm (All-pairs)
- Dijkstra's Algorithm (Single-source)
- Bellman-Ford Algorithm (Single-source)

NEGATIVE EDGE

vs

NEGATIVE CYCLE

FLOYD-WARSHALL

- No negative cycle
- Dynamic Programming

```
// dis[ i ][ j ]: initialized to weight of edge(i, j)
for ( int k = 1 ; k <= n ; k ++ )
  for ( int i = 1 ; i <= n ; i ++ )
    for ( int j = 1 ; j <= n ; j ++ )
      dis[ i ][ j ] = min( dis[ i ][ j ] ,
                          dis[ i ][ k ] + dis[ k ][ j ] ) ;
// dis[ i ][ j ]: shortest path from i to j
```

ADVANTAGE

- easy to implement

DISADVANTAGE

- $O(V^3)$
- Cannot handle negative cycle

DIJKSTRA'S ALGORITHM

- $O(E + V^2)$
- Cannot handle negative edge
- Single-source
- Greedy

ALGORITHM (INITIALIZE)

- Let the source be s
- Maintain an unvisited vertice set U (intially, $U = V$)
- Maintain an array $d[]$, where $d[i]$ stores the current known shortest path from the source to vertice i. (initially, $d[i] = \mathrm{INF}, \forall i$)
- Maintain a variable p storing the vertice currently being handled.
- Set $d[s] = 0$ and let $p = s$.

ALGORITHM (MAIN PART)

- Repeat the following procedure until U is empty
- For all vertices adjacent to p, let it be q, update $d[q]$ with $d[p] + \text{weight}(p, q)$
- Remove p from U.
- For all vertices in U, find the vertice with smallest $d[]$, and assign it to p.

ALGORITHM (FINALE)

- $d[i]$ now stores the shortest path from s to i
- Time complexity: $O(E + V^2)$ (Better for dense graph)
- Can also be implemented in time $O((E + V) \log V)$ with priority queue. (Better for sparse graph)

SUMMARY

- Why is it correct?
- Why not negative edges?

IMPLEMENTATION (INITIALIZE)

```
bool vis[ N ] ;
int vcnt = 0 ;
int dis[ N ] ;

memset( vis , false , sizeof vis ) ;
dis[ s ] = 0 ;
int p = s ;
```

IMPLEMENTATION (MAIN)

```
while ( true ) {
  for ( ii x : v[ p ] ) if ( !vis[ x ] ) {
    int q = x.first , w = x.second ;
    dis[ q ] = min( dis[ q ] , dis[ p ] + w ) ;
  }
  vis[ p ] = true ;
  int nxtp = -1 ;
  for ( int i = 0 ; i < n ; i ++ ) if ( !vis[ i ] ) {
    if ( nxtp == -1 || dis[ i ] < dis[ nxtp ] ) nxtp = i ;
  }
  if ( nxtp == -1 ) break ;
  else p = nxtp ;
}
```

BELLMAN-FORD ALGORITHM

- $O(V \cdot E)$
- Can handle negative cycles
- Single-source

核心概念：放鬆(RELAX)

- 對一條邊 (u, v)
- 如果滿足 $\text{dis}(u) + \text{weight}(u, v) < \text{dis}(v)$
- 就更新 $\text{dis}(v)$

算法

- 初始化所有 $\mathrm{dis}(u)$ 為無窮大（起點設為 0）
- 對圖上的每一條邊放鬆一次
- 重複上面的步驟 $V-1$ 次

複雑度

$O(V \cdot E)$

性質一

- 在沒負環的圖上
- 對任意節點 v 都有一條長度（經過的邊數）小於 V 的最短路

性質二

- 對於一個節點 v
- 若存在一條從 u 來（前一個點）的最短路
- 則從起點走 u 的最短路到 u 後再走 (u, v) 到 v 會是 v 的一條最短路

性質三

- 對於一個節點 v
- 若存在一條長度（經過的邊數）為 l 的最短路
- 則最多放鬆 l 輪後會得到這條最短路

負環怎麼辦?

- 再做一輪?
- 再做 $V-1$ 輪?

實作

```
struct edge {
  int u , v , w ;
} ;
vector< edge > E ;
int dis[ N ] ;

for ( int i = 1 ; i < n ; i ++ )
  for ( edge ed : E )
    dis[ ed.v ] = min( dis[ ed.v ] , dis[ ed.u ] + ed.w ) ;
```