HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

DATA STRUCTURES AND ALGORITHMS

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

WEEK 14 : GRAPH (PART 2)

ONE LOVE. ONE FUTURE.

CONTENT

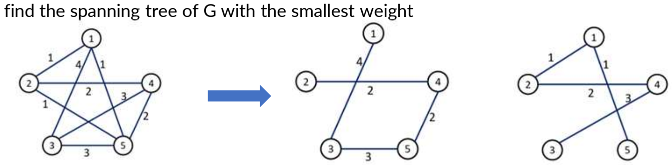
- Kruskal algorithm and Disjoint-Set in finding the minimum spanning tree problem
- Dijkstra's algorithm and Priority Queue in finding the shortest path problem

KRUSKAL ALGORITHM AND DISJOINT-SET

- The problem of the smallest spanning tree of a graph
  - Given a connected undirected graph G = (V, E) where V = {1, 2, . . ., n} is the set of vertices and E is the set of edges
    - *c*(*u*,*v*) is the edge weight (u,v), for all (u,v) $\in E$
  - A tree T = (V, F) where $F \subseteq E$ is called a spanning tree of G
  - Task: find the spanning tree of G with the smallest weight

KRUSKAL ALGORITHM AND DISJOINT-SET

- The problem of the smallest spanning tree of a graph
  - Given a connected undirected graph G = (V, E) where V = {1, 2, . . ., n} is the set of vertices and E is the set of edges
    - *c*(*u*,*v*) is the edge weight (u,v), for all (u,v) $\in E$
  - A tree T = (V, F) where $F \subseteq E$ is called a spanning tree of G
  - Task: find the spanning tree of G with the smallest weight



Graph *G* Spanning tree T_1, weight 11 Spanning treeT_2, weight 7

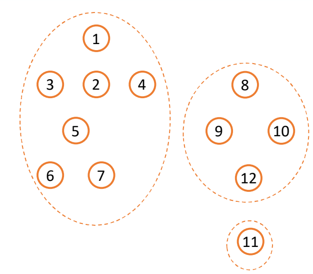
KRUSKAL ALGORITHM AND DISJOINT-SET

- Kruskal is a greedy algorithm:
  - Initially, the solution is just the set of vertices of G
  - For each iteration, we choose an edge with the smallest weight to add to the solution with the condition that no cycle is created.
  - The iteration process will end when all vertices of the graph are connected to each other

```
Kruskal(G = (V, E)) {
  T = {};
  L = sort E in a non-decreasing order of weight;
  for (u,v) in L do {
    if T ∪ {(u,v)} does not create a cycle then {
       T = T ∪ {(u,v)};
       if |T| = |V| - 1 then break;
    }
  }
  if |T| < |V| - 1 then
    return NULL;
  else
    return T;
}
```

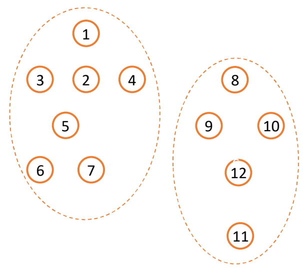

KRUSKAL ALGORITHM AND DISJOINT SETS

- Disjoint-Set: A data structure representing sets that do not intersect with two main operations
  - Find(x): returns the identifier of the set containing x
  - Unify(r1, r2): Merge two sets of identifiers r1 and r2 into one



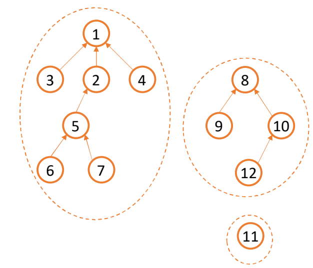
KRUSKAL ALGORITHM AND DISJOIN-SET

- Disjoint Set: A data structure representing sets that do not intersect with two main operations
  - Find(x): returns the identifier of the set containing x
  - Unify(u, v): Merge two sets of identifiers u and v into one



KRUSKAL ALGORITHM AND DISJOINT-SET

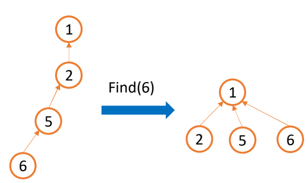
- Disjoint Set: A data structure representing sets that do not intersect with two main operations
  - Find(x): returns the identifier of the set containing x
  - Unify(u, v): Merge two sets of identifiers u and v into one
- Each set is represented by a rooted tree
  - Each node of the tree is an element
  - Each node x has a unique parent node p[x] (the parent of the root node is itself)
  - The root node is the identifier of the set



KRUSKAL ALGORITHM AND DISJOINT-SET

- Find(x) operation:
  - Starting from x, continuously access the parent node to find the root node
  - Complexity is proportional to the length of the path from x to the origin
  - Path Compression: in the process of going from x to the root, nodes on the path will be attached as direct children of the root node to reduce the path length from these nodes to the root in the following iterations.

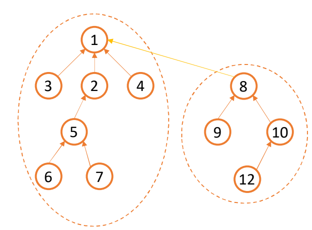
```
Find(x){
    if(x != p[x]) p[x] = Find(p[x]);
    return p[x];
}
```



KRUSKAL ALGORITHM AND DISJOINT-SET

- Unify(u, v) operation:
  - Let u be a child of v or v be a child of u depending on which node's height is smaller
  - Each node,maintains r[x] representing the height of node x

```
Unify(x, y){
    if(r[x] > r[y]) p[y] = x;
    else{
        p[x] = y;
        if(r[x] == r[y]) r[y] = r[y] + 1;
    }
}
```



KRUSKAL ALGORITHM AND DISJOINT-SET

```
makeSet(x){
    p[x] = x; r[x] = 0;
}
Find(x){
    if(x != p[x]) p[x] = Find(p[x]);
    return p[x];
}
Unify(x, y){
    if(r[x] > r[y]) p[y] = x;
    else{
        p[x] = y;
        if(r[x] == r[y]) r[y] = r[y] + 1;
    }
}
```

```
KruskalWithDisjointSset(G = (V, E)){
  T = {};
  for v in V do makeSet(v);
  L = sort E in a non-decreasing order of weight;
  for (u,v) in L do {
     ru = Find(u);
     rv = Find(v);
     if ru ≠ rv then {
        Unify(ru, rv);
        T = T ∪ {(u,v)};
        if |T| == |V| - 1 then break;
     }
  }
  if |T|  < |V| - 1 then return {};
  return T;
}
```


KRUSKAL ALGORITHM AND DISJOINT-SET

- Illustrated with C language
- Data
  - Line 1: write two positive integers n and m representing the number of vertices and edges of G, respectively (1 ≤ n, m ≤ 105)
  - Line i (i = 1, 2, ..., m): records 3 positive integers u, v and c in which c is the edge weight (u,v)
- Result
  - Write the weight of the smallest spanning tree found

stdin	stdout
5 8 1 2 1 1 3 4 1 5 1 2 4 2 2 5 1 3 4 3 3 5 3 4 5 2	7

KRUSKAL ALGORITHM AND DISJOINT-SET - CODE

```
#include <stdio.h>
#define MAX 100001
// data structure for input graph
int N, M;
int u[MAX];
int v[MAX];
int c[MAX];
int ET[MAX];
int nET;

// data structure for disjoint-set
int r[MAX];// r[v] is rank of set v
int p[MAX];// p[v] is parent of v
long long rs;
```

```
void unify(int x, int y){
    if(r[x] > r[y]) p[y] = x;
    else{
        p[x] = y;
        if(r[x] == r[y]) r[y] = r[y] + 1;
    }
}
void makeSet(int x){
    p[x] = x;  r[x] = 0;
}
int findSet(int x){
    if(x != p[x])  p[x] = findSet(p[x]);
    return p[x];
}
```


KRUSKAL ALGORITHM AND DISJOINT-SET - CODE

```
void swapEdge(int i, int j){
  int tmp = c[i]; c[i] = c[j]; c[j] = tmp;
  tmp = u[i]; u[i] = u[j]; u[j] = tmp;
  tmp = v[i]; v[i] = v[j]; v[j] = tmp;
}
int partition(int L, int R, int index){
  int pivot = c[index]; swapEdge(index,R);
  int storeIndex = L;
  for(int i = L; i <= R-1; i++){
    if(c[i] < pivot){
      swapEdge(storeIndex,i); storeIndex++;
    }
  }
  swapEdge(storeIndex,R); return storeIndex;
}
```

```
void quickSort(int L, int R){
    if(L < R){
        int index = (L+R)/2;
        index = partition(L,R,index);
        if(L < index) quickSort(L,index-1);
        if(index < R) quickSort(index+1,R);
    }
}
void sort(){
    quickSort(0,M-1);
}
```

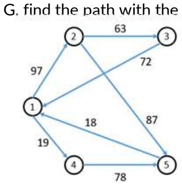
KRUSKAL ALGORITHM AND DISJOIN-SET - CODE

```
void kruskal(){
    for(int x = 1; x <= N; x++)  makeSet(x);
    sort(); rs = 0; int nET = 0;
    for(int i = 0;i < M; i++){
        int ru = findSet(u[i]);
        int rv = findSet(v[i]);
        if(ru != rv){
            unify(ru,rv);
            nET++;   rs += c[i];
            if(nET == N-1) break;
        }
    }
    printf("%lld",rs);
}
```

```
void input(){
    scanf("%d%d",&N,&M);
    for(int i = 0; i < M; i++){
        scanf("%d%d%d",&u[i],&v[i],&c[i]);
    }
}
int main(){
    input();
    kruskal();
}
```


DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE

- The problem of the shortest path between two vertices on a non-negative weight graph
  - Given a connected directed graph G = (V, E) where V = {1, 2, . . ., n} is the set of vertices and E is the set of arcs
    - *c(u,v) is the non-negative weight of arc (u,v), for all (u,v) ∈ E*
  - Task: Given two vertices s and t in G, find the path with the smallest total weight on G



Graph G: the shortest path from 1 to 5 is 1 - 4 - 5 with length equal to 19 + 78 = 97

DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE

- The data structure represents the graph G using an adjacency list
  - *A[u] is the set of arcs e that go out from the point u, every u ∈ V.*
  - For each arc e leaving point u, e.id is the remaining vertex of e and e.w is the arc weight
    - For example: arc e = (u, v) is an arc leaving u with weight 10, then e.id = v and e.w = 10

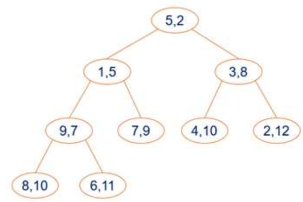
DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE

- Dijkstra's algorithm
  - For each vertex v of G, we maintain an upper bound path P[v] of the shortest path from s to v. The symbol d[v] is the length of P[v]. This upper bound path will be gradually improved (length decreasing) through iterations.
  - If there exists a point u such that d[v] > d[u] + c(u,v), then we can make a good upper bound path P[v] by path P[u] connecting the arc (u,v).), and update d[v] = d[u] + c(u,v)

```
Dijkstra(G = (V, E)){
  for v in V do d[v] = ∞;
  d[s] = 0; F = V;
  while F not empty do {
     u = select u from F s.t. d[u] is minimal;
     if u = t then break;
     F = F \ {u};
     for e in A[u] do {
       if d[e.id] > d[u] + e.w then
          d[e.id] = d[u] + e.w;
     }
  }
  return d[t];
}
```

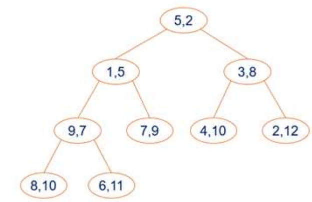
DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE

- Priority Queue (priority queue)
  - The data structure stores pairs of elements v and its key d[v], with 2 main operations:
    - push(v, d[v]): put the pair (v, d[v]) into the priority queue or update v's key if the element (pair) identifying v already exists
    - deleteMin(): removes the pair (v, d[v]) with the smallest key d[v] among the pairs in the priority queue and returns element v.



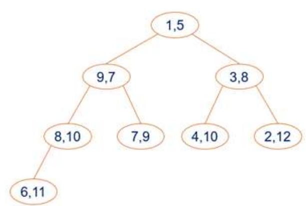
DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE

- Priority Queue
- Implementation:
  - Use arrays with elements numbered 0, 1, 2, ...
    - Each element contains 2 pieces of information: v is the identifier and d[v] is the key of the element
  - Arrays are viewed from the perspective of a complete binary tree
    - For an element with number i, the left child has number 2i+1 and the right child has number 2i+2.
    - The key of an element is less than or equal to the keys of 2 child elements (if any) → Min-Heap structure



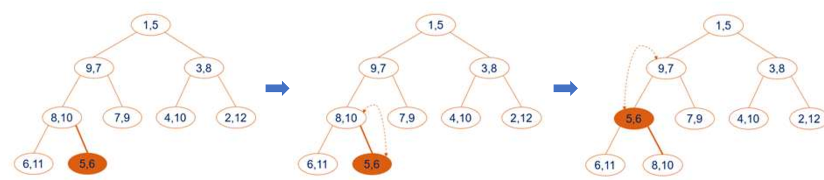
DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE

- The push(v, d[v]) operation
  - Creates a new element with identifier v and key d[v]
  - Add this new element to the end of the Min-Heap (end of the array)
  - Repeat swapping this element with its parent as long as this element's key is less than the parent's key (this operation is called UpHeap).



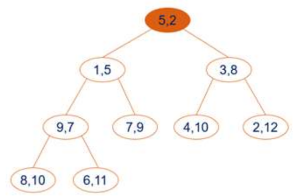
Execute push(5, 6)

DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE

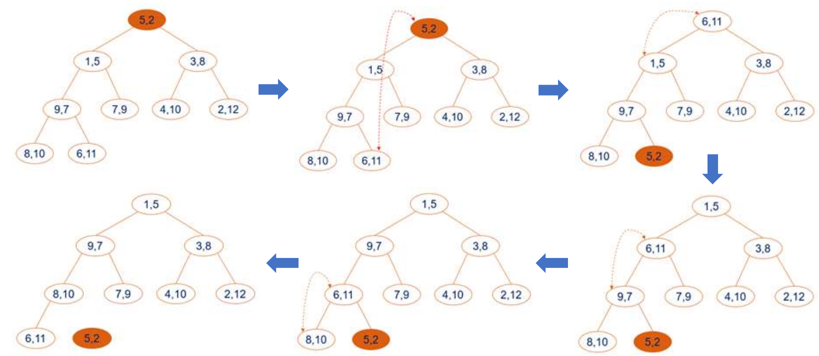


DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE

- Thao tác deleteMin()
  - Hoán đổi phần tử đầu tiên (gốc của Min-Heap) với phần tử cuối cùng của mảng
  - Thực hiện lặp lại việc hoán đổi phần tử hiện tại (xuất phát từ gốc) với phần tử có khóa nhỏ hơn trong số 2 phần tử con (nếu có) chừng nào khóa của phần tử hiện tại còn chưa nhỏ hơn hoặc bằng khóa của các phần tử con (thao tác này còn được gọi là DownHeap)
  - Loại bỏ phần tử cuối cùng của mảng đi



DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE



DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE

- Thuật toán Dijkstra sử dụng hàng đợi ưu tiên
  - Khởi tạo hàng đợi ưu tiên **pq** chứa các đỉnh đã tìm được đường đi cận trên đồng thời chưa tìm được đường đi ngắn nhất
  - Mỗi bước lặp sẽ dùng thao tác deleteMin() để lấy ra đỉnh *u* từ **pq** có *d*[*u*] nhỏ nhất và cập nhật lại *d*[*v*] với mỗi đỉnh *v* kề với *u* nếu d[v] > *d*[*u*] + *c*(*u*,*v*) sau đó push(*v*, *d*[v]) vào **pq**

```
Dijkstra(G = (V, A), s, t){
  for v in V do d[v] = +∞;
  pq = initPQ(); pq.push(s, 0);
  while(pq not empty){
    u = pq.deleteMin();
    for e in A[u] do {
       v = e.id; w = e.w;
       if d[v] > d[u] + w then
          pq.push(v,d[u] + w);
    }
  }
  return d[t];
}
```


DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE

- Minh họa với ngôn ngữ C
- Dữ liệu
  - Dòng 1: ghi 2 số nguyên dương *n* và *m* tương ứng là số đỉnh và số cạnh của G ($1 \le n, m \le 10^5$)
  - Dòng *i* (*i* = 1, 2, ..., m): ghi 3 số nguyên dương *u*, *v* and *c* trong đó *c* là trọng số cung (*u*,*v*)
  - Dòng cuối cùng chứa 2 số nguyên dương là đỉnh *s* (đỉnh đầu) và *t* (đỉnh cuối)
- Kết quả
  - Ghi ra trọng số của đường đi ngắn nhất từ *s* đến *t*

stdin	stdout
5 7 2 5 87 1 2 97 4 5 78 3 1 72 1 4 19 2 3 63 5 1 18 1 5	97

DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE - CODE

```
#include <stdio.h>
#include <stdlib.h>
#define N 100001
#define INF 1000000
typedef struct Arc{
  int id;
  int w;
  struct Arc* next;
}Arc;
```

```
int n,m; // number of nodes and arcs of the given graph
int s,t; // source and destination nodes
Arc* A[N]; // A[v] is the pointer to the first item of the adjacent arcs
           // of node v

// priority queue data structure (implemented using BINARY HEAP)
int d[N]; // d[v] is the upper bound of the length of the shortest path
         // from s to v (key)
int node[N]; // node[i] the i^th element in the HEAP
int idx[N]; // idx[v] is the index of v in the HEAP (idx[node[i]] = i)
int sH; // size of the HEAP
```


DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE - CODE

```
void swap(int i, int j){
  int tmp = node[i]; node[i] = node[j];
  node[j] = tmp;
  idx[node[i]] = i; idx[node[j]] = j;
}

void upHeap(int i){
  if(i == 0) return;
  while(i > 0){
    int pi = (i-1)/2;
    if(d[node[i]] < d[node[pi]]) swap(i,pi);
    else break;
    i = pi;
  }
}
```

```
int inHeap(int v){
    return idx[v] >= 0;
}
void downHeap(int i){
  int L = 2*i+1; int R = 2*i+2;
  int maxIdx = i;
  if(L < sH && d[node[L]] < d[node[maxIdx]])
    maxIdx = L;
  if(R < sH && d[node[R]] < d[node[maxIdx]])
    maxIdx = R;
  if(maxIdx != i){
    swap(i,maxIdx); downHeap(maxIdx);
  }
}
```


DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE - CODE

```
void updateKey(int v, int k){
  if(d[v] > k){ d[v] = k; upHeap(idx[v]); }
  else{  d[v] = k; downHeap(idx[v]);  }
}

void pushPQ(int v, int k){
  if(!inHeap(v)){
    d[v] = k;      node[sH] = v;
    idx[node[sH]] = sH;     upHeap(sH);
         sH++;
  }else updateKey(v,k);
}
```

```
int pqEmpty(){
         return sH <= 0;
}

int deleteMin(){
    int sel_node = node[0];
    swap(0,sH-1);  sH--;  downHeap(0);
    return sel_node;
}
```


DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE - CODE

```
Arc* makeArc(int id, int w){
  Arc* a = (Arc*)malloc(sizeof(Arc));
  a->id = id; a->w = w; a->next = NULL; return
a;
}
void addArc(int u, int v, int w){
  Arc* a = makeArc(v,w);
  a->next = A[u]; A[u] = a;
}
```

```
void input(){
  scanf("%d%d",&n,&m);
  for(int v = 1; v <= n; v++) A[v] = NULL;
  for(int k = 1; k <= m; k++){
    int u,v,w;
    scanf("%d%d%d",&u,&v,&w);
    addArc(u,v,w);
  }
  scanf("%d%d",&s,&t);
}
```

DIJKSTRA'S ALGORITHM AND PRIORITY QUEUE - CODE

```
void initPQ(){
  sH = 0;
  for(int v = 1; v <= n; v++)
    idx[v] = -1;
}
```

```
int main(){
   input();
   solve();
   return 0;
}
```

```
void solve(){
    for(int v = 1; v <= n; v++) d[v] = INF;
    initPQ(); pushPQ(s,0);
    while(!pqEmpty()){
        int u = deleteMin();
        for(Arc* a = A[u]; a != NULL; a = a->next){
            int v = a->id; int w = a->w;
            if(d[v] > d[u] + w)      pushPQ(v,d[u]+w);
        }
    }
    int rs = d[t]; if(d[t]==INF) rs = -1;
    printf("%d",rs);
}
```


HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !