Grafuri (Graphs)

Reprezentări

- Matrice de adiacență: /dəːə/
- Liste de adiacență: `vector<Node> list[NodeCount];`
- Lista muchiilor/arcelor: `vector<pair<Node, Node>> list;`

Grafuri neorientate (Undirected Graphs)

Noduri (Vertices); Muchii (Edges)

Graf complet (Complete Graph): toate muchiile adiacente

Graf partial (partial?): ștergem muchii

Subgraf (Subgraph): ștergem noduri

Lanț (Walk / Chain): adiacente 2 cate 2

Lanț elementar (Trail): lanț cu noduri distincte

Ciclu (Cycle): lant cu primu' = ultimu'

Ciclu elementar (elementary?): You get the point

Graf aciclic (Acyclic): natürlich

Conex (Connected): $\forall\, v, w\ \exists$ lanț

Componenta conexă (Connected component): subgraf maximal (aka tătî bucata)

Lanț eulerian (Eulerian Path/ just 'Path'): lanț care vizitează fiecare muchie o singura dată

Ciclu eulerian (Eulerian cycle/circuit): możesz to rozgryźć

Graf eulerian (Eulerian Graph): Graf cu ciclu eulerian

Bipartit (Bipartite): putem împarți nodurile în 2 partiții, în care 2 noduri din aceeași partiție nu sunt adiacente

Lanț hamiltonian (Hamiltonian path): lanț elementar cu toate nodurile

Ciclu hamiltonian (Hamiltonian cycle): duh

Grafuri orientate (Directed Graphs)

Vârfuri (Vertices); Arce (Edges/arc)

Grad interior: câte intră

Grad exterior: câte ies

Drum (Directed Walk): "lanț"

Graph Turneu (Tour): intre oricare 2 varfuri distincte exista un arc

Slab Conex (Weakly connected): daca-l transformam în neorientat, e conex

Tare Conex (Strongly connected): există drum intre oricare 2 varfuri, dus-intors

Modalitati de parcurgere: Breadth first, Depth first

Grafuri ponderate (Weighted Graphs)

Dijkstra

```
function Dijkstra(Graph, source):
    create vertex set Q
    for each vertex v in Graph:
        dist[v] = INFINITY
        prev[v] = UNDEFINED
        add v to Q
    dist[source] = 0
```

```

    while Q is not empty:
        u = vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u:           # only v that are still in Q
            alt = dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] = alt
                prev[v] = u

    return dist[], prev[]
```

Roy-Floyd

```
let dist be a |V| × |V| array of minimum distances initialized to ∞
let next be a |V| × |V| array of vertex indices initialized to null

procedure FloydWarshallWithPathReconstruction():
    for each edge (u, v) do
        dist[u][v] = w(u, v) # The weight of the edge (u, v)
        next[u][v] = v
    for each vertex v do
        dist[v][v] = 0
        next[v][v] = v
    for k from 1 to |V| do # standard Floyd-Warshall implementation
        for i from 1 to |V|
            for j from 1 to |V|
                if dist[i][j] > dist[i][k] + dist[k][j] then
                    dist[i][j] = dist[i][k] + dist[k][j]
                    next[i][j] = next[i][k]

procedure Path(u, v)
    if next[u][v] = null then
        return []
    path = [u]
    while u ≠  v
        u = next[u][v]
        path.append(u)
    return path
```

Arbori (Trees)

Subarbori (Subtree): fii

Nivelul unui nod (Level): distanta pana la radacina

Inaltimea unui nod (Height): cel mai lung lant pana la un nod terminal

Arbore partial al unui graf: graf partial care este arbore

Arbore partial de cost minim (minimum-spanning-tree): avem o functie f care determina costul fiecarei muchii, gasim un arbore partial aî suma costurilor sale sa fie minima.

Reprezentări

- like a graph: /dəːə/
- Ref descendente: `struct Node { int data; Vector<Node*> children; };`
- Ref ascendente: `int parinte[NodeCount] = {-1, ...};`

Kruskal

```
algorithm Kruskal(G) is
A := ∅
for each v ∈ G.V do
    MAKE-SET(v)
for each (u, v) in G.E ordered by weight(u, v), increasing do
    if FIND-SET(u) ≠ FIND-SET(v) then
       A := A ∪ {(u, v)}
       UNION(FIND-SET(u), FIND-SET(v))
return A
```

Prim

```
void prim(double mat[sz][sz], ssize_t len) {
    int* s = new int[len] {-1}, j;
    for (int i = 1; i < len; ++i) s[i] = 0;
    for (int k = 1; k < len; ++k) {
        double min = inf;
        for (int i = 0; i < len; ++i)
            if (s[i] != -1 && min > mat[i][s[i]]) {
                min = mat[i][s[i]];
                j = i;
            }
        std::cout << "edge: " << j << "-" << s[j] << "\n";
        for (int i = 0; i < len; ++i)
            if (s[i] != -1 && mat[i][s[i]] > mat[i][j])
                s[i] = j;
        s[j] = -1;
    }
}
```

Arbori binari (Binary Tree)

Arbori binari completi (Perfect binary tree): pe fiecare nivel s are exact 2^s noduri

Aproape complet (Complete binary tree): pe ultimul nivel lipsesc doar primele din stanga/ ultimele din dreapta

Ansamblul Heap binar minim (Min-heap): Arbore aproape complet cu cheia oricarui parinte mai mica sau egala cu a fiului

Heap sort: stergem si punem intr-un vector

Coada de priorități (Priority queue): tl;dr a fancy heap

AVL Trees

```
struct Node {
    Key key;
    Node *left, *right;
```

```
    int h = 1;
    static int height(const Node* p) { return (p == nullptr) ? 0 : p->h; }
    void updateHeight() { h = std::max(height(left), height(right)) + 1; }
    int balanceFactor() const { return height(left) - height(right); }
    static void rightRotate(Node*& n) {
        Node* t = n->left;
        n->left = t->right;
        t->right = n;
        n->updateHeight();
        t->updateHeight();
        n = t;
    }
    static void leftRotate(Node*& n) { /* like above but swap right and left*/ }
    static void balance(Node*& node) {
        node->updateHeight();
        int factor = node->balanceFactor();
        if (factor > 1) {
            if (node->left->balanceFactor() < 0) leftRotate(node->left);
            rightRotate(node);
        } else if (factor < -1) {
            if (node->right->balanceFactor() > 0) rightRotate(node->right);
            leftRotate(node);
        }
    }
    static void add(Node*& node, const Key& key) {
        if (node == nullptr) { node = new Node(key); return; }
        assert(key != node->key, "Value already in tree");
        if (key < node->key) add(node->left, key);
        else if (key > node->key) add(node->right, key);
        balance(node);
    }

    static void remove(Node*& n, const Key& key) {
        if (n == nullptr) { printf("not found?\n"); return; }
        if (key < n->key) remove(n->left, key);
        else if (key > n->key) remove(n->right, key);
        else { // key == n->key
            if (n->left == nullptr) {
                if (n->right == nullptr) { n = nullptr; return; }
                Node* tmp = n;
                n = n->right;
                tmp->right = nullptr;
                delete tmp;
            } else if (n->right == nullptr) {
                Node* tmp = n;
                n = n->left;
                tmp->left = nullptr;
                delete tmp;
            } else {
                constexpr auto findMax = [](Node*& r, auto& findMax) -> Node*& {
                        if (r->right == nullptr) return r;
                        return findMax(r->right, findMax);
```

```
                    };
                Node*& tmp = findMax(n->left, findMax);
                n->key = tmp->key;
                remove(n->left, tmp->key);
            }
        }
        if (n == nullptr) return;
        balance(n);
    }
};
```

Binary heap

```
Vector<Val> data;
constexpr static int parent(int i) { return (i - 1) / 2; }
constexpr static int left(int i) { return (2 * i + 1); }
constexpr static int right(int i) { return (2 * i + 2); }
constexpr bool empty() const { return data.empty(); }
void push(const val& v) {
    data.push_back(v);
    for (size_t i = data.size()-1; ;) {
        if (i == 0) break;
        size_t j = parent(i);
        if (data[i] > data[j]) {
            std::swap(data[i], data[j]);
            i = j;
        }
    }
}
val pop() {
    val res = std::move(data[0]);
    data[0] = data.back();
    data.pop_back();
    size_t i = 0;
    size_t sz = data.size();
    while (i < sz) {
        size_t j = left(i);
        if (j < sz) {
            if (j+1 < sz && data[j+1] > data[j]) ++j;
            if (data[i] < data[j]) {
                std::swap(data[i], data[j]);
                i = j;
            } else break;
        } else break;
    }
    return res;
}
```

Tablele de dispersie (Hash tables)

functii: $h(k) = k \mod m$; $\quad h(k) = \lfloor m(k\varphi - [k\varphi]) \rfloor$, $\quad \varphi = \frac{\sqrt{5}-1}{2}$
Or: do it again: $h(k,i) = h_1(k) + i \mod m$; $\quad h(k,i) = h_1(k) + c_1 i + c_2 i^2 \mod m$