

# 1 Graphen

1.  $e = (v, w) : e$  inzident zu  $v$
2. "einfacher Pfad": Pfad ohne Schleifen, d.h. knotendisjunkt ( $v_i \neq v_j$  für  $i \neq j$ )
3.  $m \leq n^2$ ,  $m = |E|$ ,  $n = |N|$   
falls  $m = n^2 \Rightarrow E = V \times V$ ,  $G$  heißt vollständig

Die Kanten eines Graphen lassen sich wie folgt einteilen. Sie  $e = (v, w) \in E$ ,

1.  $e \in T$  falls  $w$  noch nicht besucht
2.  $e \in F$  falls  $w$  besucht und  $\exists$  Pfad aus Kanten aus  $T$ -Knoten der Länge  $\leq 1$ ,  $v \neq w$ .
3.  $e \in B$  falls  $w$  besucht und  $\exists$  Baumpfad von  $w$  nach  $v$  der Länge  $\leq 0$  (evtl.  $v = w$ ).
4.  $e \in C$  falls  $w$  besucht und  $\nexists$  Baumpfad von  $v$  nach  $w$  (Länge  $\leq 1$ ) und  $\nexists$  Baumpfad von  $w$  nach  $v$  (Länge  $\leq 0$ ).

## 1.1 DFS

Zeitkomplexität von  $\text{dfs}() \in \mathcal{O}(n + m)$

```
dfs(v) {
  besucht[v] = true;
  dfsnum[v] = ++dfsct;
  forall (w ∈ V, (v, w) ∈ E) {
    if (!besucht[w]) {
      // (v,w) ∈ T
      dfs(w);
    } else if (dfsnum[v] < dfsnum[w] && comp[v] > comp[w]) {
      // (v,w) ∈ F,T
    } else if (dfsnum[w] < dfsnum[v]) {
      if (comp[w] == 0) {
        // (v,w) ∈ B (Kreis)
      } else { // comp[v] < comp[w]
        // (v,w) ∈ C
      }
    }
  }
  comp[v] = true;
}
```

## 1.2 Topologische Sortierung

Zeitkomplexität von  $\text{topsort}() \in \mathcal{O}(n + m)$

```
topsort() {
  list l = {v ∈ V | indeg(v) = 0}
  int i = 0;
  while (!l.empty()) {
    node n = l.pop();
    topsort[n] = i++;
    forall (v ∈ V, (n,v) ∈ E) {
      indeg[v]--;
      if (indeg[v] == 0)
        l.push_back(v);
    }
  }
}
```

## 1.3 Gegenkanten

Zeitkomplexität von  $\text{rev}()$  und  $\text{bucketsort}() \in \mathcal{O}(n)$

```
list E1; // Kanten aus E sortiert nach source, target
list E2; // Kanten aus E sortiert nach target, source
rev[]; // Array von Pointern auf Kantern ueber Kanten

rev() {
  while (!E1.empty() && !E2.empty()) {
    e1 = E1.head();
    e2 = E2.head();
```

```

    if (source(e1) < target(e2)) {
        E1.pop_front();
    } else if (target(e1) < source(e2)) {
        E2.pop_front();
    } else {
        rev[e1] = e2;
        E1.pop_front();
        E2.pop_front();
    } } }

```

```

bucketsort(list e1) { // e1: Liste aus Kanten
    buckets[]; // Feld von Listen von Kanten ueber Knoten
    while (!e1.empty()) {
        e = e1.pop_front();
        buckets[source(e)].push_back(e);
    }
    forall (n ∈ V)
        e1.concat(buckets[v]);
    return e1;
}

```

## 1.4 Transitiv Hülle

Zeitkomplexität von `trans_hull()`  $\in \mathcal{O}(n^3)$

```

queue M; // init: ∅
visited[]; // init: 0
round = 0;

trans_hull() {
    forall (v ∈ V) {
        visited[v] = ++round;
        M.push(v);
        while (!M.empty()) {
            w = M.pop();
            forall (x ∈ V, (w,x) ∈ E)
                if (visited[x] < round) {
                    M.push(x);
                    visited[x] = round;
                    E_ = E_ ∪ {(v,x)}
                }
        }
    }
    return E_;
}

```

## 1.5 Berechnung starker Zusammenhangskomponenten

Zeitkomplexität von `szk()`  $\in \mathcal{O}(n + m)$ , da `dfs()`  $\in \mathcal{O}(n + m)$  und  $\forall v \in V$  wird  $v$  genau einmal in unfertig und wurzeln eingefügt und entfernt ( $\in \mathcal{O}(n)$ ).

```

stack unfertig; // init: ∅
stack wurzeln; // init: ∅
besucht[]; // init: false
dfsnum[];
scc_num[]; // Nummer der SZK
in_unfertig[]; // v ∈ unfertig? init: false

szk(v) {
    dfsnum[v] = ++dfs_count;
    besucht[v] = true;
    unfertig.push(v); // \
    wurzeln.push(v); // + Neue SZK { v }
    in_unfertig[v] = true; // /

    forall (w ∈ V, (v,w) ∈ E)
        if (!besucht[w]) {
            szk(w);
        } else if (in_unfertig[w]) { // mischen
            while (dfsnum[wurzeln.top()] > dfsnum[w])
                wurzeln.pop();
        }
    }

    // Abschluss
    if (v == wurzeln.top()) {

```

```

    scc_count++;
    do {
        w = unfertig.pop();
        in_unfertig[w] = false;
        scc_num[w] = scc_count;
    } while (v != w);
    wurzeln.pop();
} }

```

## 1.6 Bipartitheit

Erweiterter dfs()  $\Rightarrow$  Laufzeit  $\in \mathcal{O}(n + m)$

```

color[]; // Feld von Knoten -> Farben, init: not_visited
stack s;

is_bipartit() {
    forall (v  $\in$  V) {
        if (color[v] == not_visited) {
            color[v] = red;
            s.push(v);
            while (!s.empty()) {
                w = s.pop();
                forall (v  $\in$  V, (w,v)  $\in$  E) {
                    if (color[v] == not_visited) {
                        if (color[w] == red) color[v] = blue;
                        else color[v] = red;
                    }
                    s.push(v);
                } else if (color[v] == color[w]) {
                    return false;
                }
            }
        }
    }
    return true;
}

```

## 1.7 Distanzwerte von $s$ ausgehend $\forall v \in V$

Erweitertes bfs()  $\Rightarrow$  Laufzeit  $\in \mathcal{O}(n + m)$

```

dist[]; // Init: -1
queue q;

dist(s) {
    dist[s] = 0;
    q.push(s);
    while (!q.empty()) {
        v = q.pop();
        forall (w  $\in$  V, (v,w)  $\in$  E) {
            if (dist[w] == -1) {
                dist[w] = dist[v] + 1;
                q.push(w);
            }
        }
    }
}

```

## 1.8 Knotendisjunkte Pfade von $s$ nach $t$

```

pred[]; // Feld von Knoten auf Knoten, init: 0
dist[]; // Init mit Distanzwerten von s (siehe dist())
result; // Liste v Knoten

v: wird durchsucht
x: Vorgaenger von v im DFS-Baum
t: Zielknoten
dfs_next_path(v, x, t) {
    if (v == t) {
        result.push(x);
        return;
    }
    pred[v] = x;
    forall (w  $\in$  V, (v,w)  $\in$  E) {
        if (pred[w] == 0 && dist[v] == dist[w] - 1 && dist[w]  $\leq$  dist[t])
            dfs_next_path(w, v, t);
    }
}

```

## 1.9 Dijkstras Algorithmus

Zeitkomplexität von `dijkstra()` abhängig von Implementierung von PQ. Generell:

$$\mathcal{O}(n \cdot (1 + T_{\text{insert}} + T_{\text{empty}} + T_{\text{delmin}}) + m \cdot (1 + T_{\text{decrease\_p}}))$$

- unsortierte Liste:  $\mathcal{O}(n^2 + m)$ :

- $T_{\text{insert}} \in \mathcal{O}(1)$
- $T_{\text{empty}} \in \mathcal{O}(1)$
- $T_{\text{delmin}} \in \mathcal{O}(n)$
- $T_{\text{decrease\_p}} \in \mathcal{O}(1)$

- Fibonacci-Heap:  $\mathcal{O}(n \cdot \log n + m)$

- $T_{\text{insert}} \in \mathcal{O}(1)$
- $T_{\text{empty}} \in \mathcal{O}(1)$
- $T_{\text{delmin}} \in \mathcal{O}(\log n)$  (amortisiert)
- $T_{\text{decrease\_p}} \in \mathcal{O}(1)$  (amortisiert)

```
c(v, w); // Kostenfunktion: E -> R_0^+ fuer Kante (v, w)
DIST[]; // Feld Knoten -> reelle Zahl
PQ; // priority-queue (theor. optimal: Fibonacci-Heap)

dijkstra(s) {
  forall (v in V)
    DIST[v] = infinity;
  DIST[s] = 0;
  PQ.insert(s, 0);

  while (!PQ.empty()) {
    u = PQ.delmin();
    forall (v in V, (u,v) in E) {
      d = DIST[u] + c(u, v);
      if (d < DIST[v]) {
        if (DIST[v] == infinity) {
          PQ.insert(v, d);
        } else {
          PQ.decrease_p(v, d);
        }
      }
      DIST[v] = d;
    }
  }
}
```

## 2 Fibonacci-Heap

Warteschlange, realisiert als Menge sog. "Heap-ordered Trees". Ein Baum dessen Knoten mit Information ( $Inf : V \rightarrow \mathbb{R}$ ) beschränkt ist, heißt "heap-ordered", falls für alle Knoten  $v \in V : Inf(v) \geq Inf(Eltern(v))$ . Fibonacci-Heap ist ein Wald von heap-ordered trees und einem Zeiger auf die Wurzel mit der kleinsten Information. Die trees sind Binomische Bäume.

Die Knoten auf einer Bauebene bilden mit ihren Pointern `leftsib` und `rightsib` eine zirkuläre, doppelt verkettete Liste.

$\forall T_i \in f \rightarrow \text{roots} : \max \text{Rang}(T_i) \leq \log n$  ( $n = \text{Anzahl Knoten in } T_i$ ).

```
struct node : dlist_item {
  K key;
  I inf;
  node *parent, *child, *leftsib, *rightsib;
  int rang = 0;
  bool marked = false;
};

struct fibheap {
  struct node *min; // Zeiger auf item in Wurzelliste mit minimalem inf-Eintrag
  struct dlist *roots; // Liste von Wurzeln struct node*
}

void init(struct fibheap *f) {
  f->min = NULL;
}
```

```

K find_min(f) {                               // in O(1)
    return f->min->key;
}

struct node * insert(f, K k, I i) {           // in O(1)
    struct node *n = new node;
    n->key = k;
    n->inf = i;
    put_root(f, n);
    return n;
}

K delete_min(f) {
    struct node *n = dlist_del(f->min);
    K r = n->key;
    struct node *m = n->child;
    delete n;
    n = m;
    dlist_append_list(f->roots, n->child);    // in O(1)
    verschmelze_roots(f);                    // in O(max rang)
    f->min = dlist_find_min(f->roots);        // in O(max rang)
    return r;
}

void verschmelze_roots(f) {
    struct node *rang_arr[f->roots->size()]; // Array von Zeigern auf Wurzelemente, init: NULL
    struct node *w, *v;
    dlist_forall(f->roots, w) {
        while ((v = f->root_arr[w->rang]) != NULL) {
            f->root_arr[w->rang] = NULL;
            w = verschmelze(f, v, w);         // in O(1)
        }
        f->root_arr[w->rang] = w;
    } }

struct node * verschmelze(f, v, w) {
    if (v->inf > w->inf)
        swap(v, w);
    dlist_remove(f->roots, w);
    w->parent = v;
    v->rang++;
    dlist_insert(v->child, w);
}

void decrease_p(f, struct node *v, I i) {
    v->inf = i;
    if (v->parent != NULL)
        markiere(f, v->parent);
    put_root(f, v);
}

void markiere(f, v) {                          // worst-case in O(log|Ti|), v ∈ Ti, amortisiert in O(1)
    struct node *p;
    while (v->marked && (p = v->parent) != NULL) {
        p->rang--;
        put_root(f, v);
        v = p;
    }
    if (v->parent != NULL)
        v->marked = true;
}

void put_root(f, v) {
    v->parent = NULL;
    v->marked = false;
    struct node *it = dlist_append(f->roots, v);
    if (f->min && f->min->inf < v->inf)
        f->min = it;
}

```

### 3 Amortisierte Analyse

- Datenstruktur  $D$
- Potentialfunktion  $pot : D \rightarrow \mathbb{R}_0^+$  ordnet Zuständen von  $D$  einen nicht-negativen reellen Wert zu.

- Operation  $op$  überführt  $D$  in neuen Zustand  $D'$ :  $D \xrightarrow{op} D'$

### 3.1 Definition

- $T_{\text{tats}}(op) :=$  tatsächliche Kosten (Ausführungszeit) der Operation  $op$
- $T_{\text{amort}}(op) := T_{\text{tats}} + \underbrace{pot(D') - pot(D)}_{\Delta pot}$

Betrachte Folge von Operationen  $D_0 \xrightarrow{op_1} D_1 \xrightarrow{op_2} \dots \xrightarrow{op_n} D_n$ .

Gesucht: Obere Abschätzung von  $\sum_{i=1}^n T_{\text{tats}}(op_i)$ . Betrachte nun Teleskopsumme

$$\begin{aligned} \sum_{i=1}^n T_{\text{amort}}(op_i) &\stackrel{\text{Def.}}{=} \sum_{i=1}^n (T_{\text{tats}}(op_i)) + pot(D_i) - pot(D_{i-1}) = \sum_{i=1}^n T_{\text{tats}}(op_i) + pot(D_n) - pot(D_0) \\ \Rightarrow \underbrace{\sum_{i=1}^n T_{\text{tats}}(op_i)}_{\text{gesucht}} &= \underbrace{\sum_{i=1}^n T_{\text{amort}}(op_i)}_{\text{leichte Abschätzung (*)}} + \underbrace{pot(D_0)}_{=0 (*)} - \underbrace{pot(D_n)}_{\geq 0 (*)} \end{aligned}$$

(\*): durch geeignete Wahl von  $pot$  ( $pot(D_i) \geq 0$  und  $pot(D_0) = 0$ ):

$$\Rightarrow \sum_{i=1}^n T_{\text{pot}}(op_i) \leq \sum_{i=1}^n T_{\text{amort}}(op_i)$$

### 3.2 Zu Fibonacci-Heaps

Ränge der Einträge in der Wurzelliste sind alle paarweise verschieden  $\Rightarrow pot(D_i) = D_i \rightarrow \text{roots} \rightarrow \text{size}$ .

- $\text{find\_min}()$ :  $T_{\text{tats}} = \mathcal{O}(1)$ ,  $\Delta pot = 0$
- $\text{insert}()$ :  $T_{\text{tats}} = \mathcal{O}(1)$ ,  $\Delta pot = 1$
- $\text{delete\_min}()$ :  $\Delta pot \leq \max \text{Rang} - \#\text{Verschmelzungen}$

$$\Rightarrow T_{\text{amort}} = \underbrace{\max \text{Rang} + \#\text{Verschmelzungen}}_{T_{\text{tats}}} + \Delta pot = \max \text{Rang} \leq 2 \cdot \log n \in \mathcal{O}(\log n)$$

- $\text{decrease\_p}()$ : Für Kinder  $w_1, \dots, w_i$  von Knoten  $v$  mit  $\text{Rang}(v) = i$ :  $\text{Rang}(w_j) \geq j - 2$  (da Kinder nur durch verschmelze-Operationen hinzukommen können).

$S_i :=$  minimale Zahl von Knoten im Unterbaum mit Wurzel  $v$  von Rang  $i$  (inklusive  $v$ )  $\Rightarrow S_0 = 1, S_1 = 2$ .

Für  $i \geq 2$ :  $S_i = 2 + S_0 + S_1 + \dots + S_{i-2} = 2 + \sum_{j=0}^{i-2} S_j$ .

1. Fibonacci-Zahlen:  $F_0 = 0, F_1 = 1, F_i = F_{i-2} + F_{i-1}$  für  $i \geq 2$

2.  $S_i \geq F_{i+2} \quad \forall i \geq 0$

3.  $\Phi = \frac{1+\sqrt{5}}{2}, \Phi^2 = \frac{1+2\cdot\sqrt{5}+5}{4} = \frac{3+\sqrt{5}}{2} = \Phi + 1$

4.  $F_{i+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^i$  als exponentielle untere Schranke für  $F_i$

$\Rightarrow$  exponentielle untere Schranke für  $S_i$ :  $S_i \geq F_{i+2} \geq \Phi^i \geq 1,618^i$

Sei  $r$  max Rang  $\Rightarrow$  Mindestens  $\Phi^r$  Knoten im Unterbaum mit Wurzel von Rang  $r \Rightarrow \Phi^i \leq n \Rightarrow r \leq \log_{\Phi} n = \frac{\log n}{\log \Phi} \leq 1,4404 \cdot \log_2 n$

Modifiziertes  $pot(D) := \#\text{Wurzeln} + 2 \cdot \text{Länge Sequenz markierter Knoten}$

Sei  $k = \#\text{markierte Knoten}$ , die in  $op$  zu Wurzeln werden, dann  $T_{\text{tats}}(op) \in \mathcal{O}(k + 1)$

Potentialänderung  $\Delta pot = k + 1 - 2 \cdot (k - 1) = 3 - k$ , da

-  $\#\text{Wurzeln to } \#\text{Wurzeln} + k + 1$

-  $\#\text{Markierungen to } \#\text{Markierungen} - k - 1$

$\Rightarrow T_{\text{amort}} = T_{\text{tats}} + \Delta pot = k + 1 + 3 - k = 4 \in \mathcal{O}(1)$