# Datenstrukturen und Algorithmen

## Übung 10

FS 2018

# Programm von heute

1. Feedback letzte Übung
2. Wiederholung Theorie
3. Programmieraufgabe

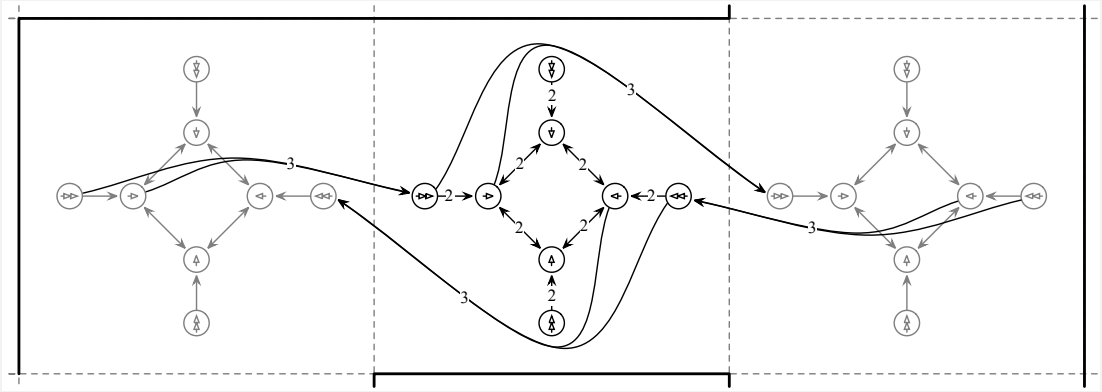
# 1. Feedback letzte Übung

# Aufgabe 9.1: Labyrinth

- Roboter muss anhalten um Richtung zu ändern
- Als shortest-path Problem auffassen

# Aufgabe 9.1: Labyrinth

- Position $\times$ Richtung $\times$ Geschwindigkeit




- Laufzeit?

# Aufgabe 9.1: Labyrinth

- Sei $n$ Anzahl Quadrate. Graph hat $|V| = 8n$ Knoten
- Graph hat $|E| \leq 20n$ Kanten
- Daher, hat Dijkstra $\mathcal{O}(|E| + |V|log|V|)$ Laufzeit $\mathcal{O}(nlogn)$

# Closeness Centrality

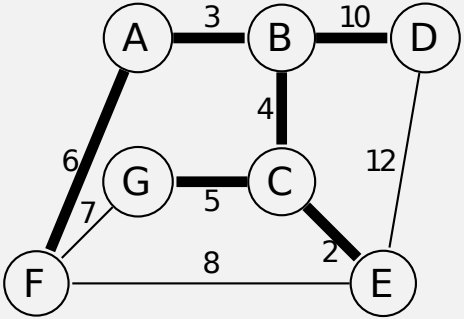
- Gegeben: Eine Adjazenzmatrix für einen *ungerichteten* Graphen auf $n$ Knoten.
- Aufgabe: für jeden Knoten $v$ die *Closeness Centrality* $C(v)$ von $v$.

$$C(v) = \sum_{u \in V \setminus \{v\}} d(v, u)$$

- Intuitiv: Wenn viele verbundene Knoten nahe bei $v$ liegen, dann ist $C(v)$ klein.
- „Wie zentral ist ein Knoten in seiner Zusammenhangskomponente?“

# Minimum Spanning Tree

Kruskal computes the following MST:



# Minimum Spanning Tree

Proof using induction over the number of vertices $|V|$:

- Hypothesis: undirected graph with $|V| - 1$ vertices has as most $|V| - 2$ edges.
- Induction base ($|V| = 1$): A graph with one node has no edges.
- Induction step ($|V| - 1 \rightarrow |V|$) Undirected cycle-free graph $G = (V, E)$. cycle-free $\Rightarrow$ there is at least one vertex $w$ with degree $0$ or $1$. Let $V' = V \setminus \{w\}$ and $E' = \{\{u, v\} \in E | u, v \in V'\}$. Because there is at most one edge incident to $w$, $|E'| \geq |E| - 1$. Due to the induction hypothesis $|E'| \leq |V'| - 1$, we get

$$|E| \leq |E'| + 1 \leq |V'| - 1 + 1 = |V'| = |V| - 1$$

# Alle kürzesten Pfade

```
template<typename Matrix>
void allPairsShortestPaths(unsigned n, Matrix& m){
  for(unsigned k = 0; k < n; ++k) {
    for(unsigned i = 0; i < n; ++i) {
      for(unsigned j = i + 1; j < n; ++j) {
        if(k == i || k == j)
          continue;
        if(m[i][k] == 0 || m[k][j] == 0)
          continue; // no connection via k
        if(m[i][j] == 0 || m[i][k] + m[k][j] < m[i][j])
          m[i][j] = m[j][i] = m[i][k] + m[k][j];
      }
    }
  }
}
```

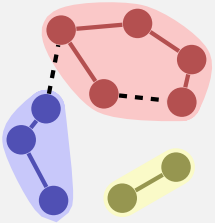
# Closeness Centrality

```
vector<vector<unsigned> > adjacencies(n,vector<unsigned>(n, 0));
vector<string> names(n);
// ...
allPairsShortestPaths(n, adjacencies);
for(unsigned i = 0; i < n; ++i) {
  cout << names[i] << ": "; unsigned centrality = 0;
  for(unsigned j = 0; j < n; ++j) {
    if(j == i) continue;
    centrality += adjacencies[i][j];
  }
  cout << centrality << endl;
}
```

# 2. Wiederholung Theorie

# Zur Implementation

Gegeben eine Menge von Mengen $i \equiv A_i \subset V$. Zur Identifikation von Schnitten und Kreisen: Zugehörigkeit der beiden Endpunkte einer Kante zu einer der Mengen.



# Union-Find Algorithmus MST-Kruskal($G$)

**Input :** Gewichteter Graph $G = (V, E, c)$
**Output :** Minimaler Spannbaum mit Kanten $A$.

Sortiere Kanten nach Gewicht $c(e_1) \leq ... \leq c(e_m)$
$A \leftarrow \emptyset$
**for** $k = 1$ **to** $m$ **do**
    MakeSet($k$)
**for** $k = 1$ **to** $m$ **do**
    $(u, v) \leftarrow e_k$
    **if** Find($u$) $\neq$ Find($v$) **then**
        Union(Find($u$), Find($v$))
        $A \leftarrow A \cup e_k$
**return** $(V, A, c)$

# Implementation Union-Find

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Parent | **1** | 1 | 1 | 6 | **5** | **6** | 5 | 5 | 3 | **10** |

Operationen:

- Make-Set($i$): $p[i] \leftarrow i$; **return** $i$
- Find($i$): **while** $(p[i] \neq i)$ **do** $i \leftarrow p[i]$ ; **return** $i$
- Union($i, j$): $p[j] \leftarrow i$; **return** $i$

# Optimierung der Laufzeit für Find

Baum kann entarten: Beispiel Union$(1,2)$, Union$(2,3)$, Union$(3,4)$, ...
Idee: Immer kleineren Baum unter grösseren Baum hängen.
Zusätzlich: Grösseninformation $g$

Operationen:

- Make-Set$(i)$: $p[i] \leftarrow i$; $g[i] \leftarrow 1$; **return** $i$
- Union$(i,j)$:
  **if** $g[j] > g[i]$ **then** swap$(i,j)$
  $p[j] \leftarrow i$
  $g[i] \leftarrow g[i] + g[j]$
  **return** $i$

# Weitere Verbesserung

Bei jedem Find alle Knoten direkt an den Wurzelknoten hängen.

Find($i$):

$j \leftarrow i$
**while** $(p[i] \neq i)$ **do** $i \leftarrow p[i]$
**while** $(j \neq i)$ **do**
    $t \leftarrow j$
    $j \leftarrow p[j]$
    $p[t] \leftarrow i$
**return** $i$

Amortisierte Laufzeit: amortisiert *fast* konstant (Inverse der Ackermannfunktion).

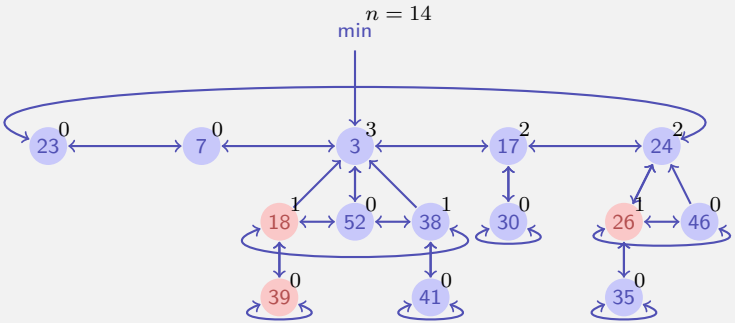
# Fibonacci Heaps

Datenstruktur zur Verwaltung von Elementen mit Schlüsseln.
Operationen

- MakeHeap(): Liefere neuen Heap ohne Elemente
- Insert($H, x$): Füge $x$ zu $H$ hinzu
- Minimum($H$): Liefere Zeiger auf das Element $m$ mit minimalem Schlüssel
- ExtractMin($H$): Liefere und entferne (von $H$) Zeiger auf das Element $m$
- Union($H_1, H_2$): Liefere Verschmelzung zweier Heaps $H_1$ und $H_2$
- DecreaseKey($H, x, k$): Verkleinere Schlüssel von $x$ in $H$ zu $k$
- Delete ($H, x$): Entferne Element $x$ von $H$

# Implementation

Doppelt verkettete Listen von Knoten mit marked-Flag und Anzahl Kinder. Zeiger auf das minimale Element und Anzahl Knoten.



# Einfache Operationen

- MakeHeap (trivial)
- Minimum (trivial)
- Insert$(H, e)$
  1. Füge neues Element in die Wurzelliste ein
  2. Wenn Schlüssel kleiner als Minimum, min-pointer neu setzen.
- Union $(H_1, H_2)$
  1. Wurzellisten von $H_1$ und $H_2$ aneinander hängen
  2. Min-Pointer neu setzen.
- Delete$(H, e)$
  1. DecreaseKey$(H, e, -\infty)$
  2. ExtractMin$(H)$

# ExtractMin

1. Entferne Minimalknoten $m$ aus der Wurzelliste
2. Hänge Liste der Kinder von $m$ in die Wurzelliste
3. Verschmelze solange heapgeordnete Bäume gleichen Ranges, bis alle Bäume unterschiedlichen Rang haben:
   Rangarray $a[1, \ldots, n]$ von Elementen, zu Beginn leer. Für jedes Element $e$ der Wurzelliste:
   a. Sei $g$ der Grad von $e$.
   b. Wenn $a[g] = nil$: $a[g] \leftarrow e$.
   c. Wenn $e' := a[g] \neq nil$: Verschmelze $e$ mit $e'$ zu neuem $e''$ und setze $a[g] \leftarrow nil$. Setze $e''$ unmarkiert Iteriere erneut mit $e \leftarrow e''$ vom Grad $g + 1$.

# DecreaseKey ($H, e, k$)

1. Entferne $e$ von seinem Vaterknoten $p$ (falls vorhanden) und erniedrige den Rang von $p$ um eins.
2. Insert($H, e$)
3. Vermeide zu dünne Bäume:
   a. Wenn $p = nil$ , dann fertig
   b. Wenn $p$ unmarkiert: markiere $p$, fertig.
   c. Wenn $p$ markiert: unmarkiere $p$, trenne $p$ von seinem Vater $pp$ ab und Insert($H, p$). Iteriere mit $p \leftarrow pp$.

# Laufzeiten

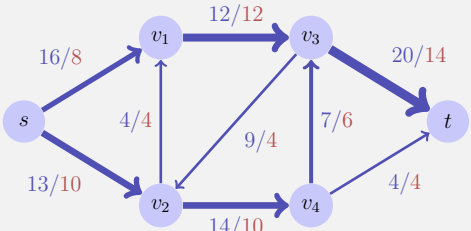
| | Binary Heap (worst-Case) | Fibonacci Heap (amortisiert) |
|---|---|---|
| MakeHeap | $\Theta(1)$ | $\Theta(1)$ |
| Insert | $\Theta(\log n)$ | $\Theta(1)$ |
| Minimum | $\Theta(1)$ | $\Theta(1)$ |
| ExtractMin | $\Theta(\log n)$ | $\Theta(\log n)$ |
| Union | $\Theta(n)$ | $\Theta(1)$ |
| DecreaseKey | $\Theta(\log n)$ | $\Theta(1)$ |
| Delete | $\Theta(\log n)$ | $\Theta(\log n)$ |

# Fluss

Ein *Fluss* $f : V \times V \to \mathbb{R}$ erfüllt folgende Bedingungen:

- *Kapazitätsbeschränkung*:
  Für alle $u, v \in V$: $f(u, v) \leq c(u, v)$.
- *Schiefsymmetrie*:
  Für alle $u, v \in V$: $f(u, v) = -f(v, u)$.
- *Flusserhaltung*:
  Für alle $u \in V \setminus \{s, t\}$:

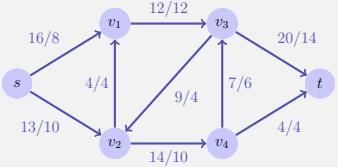
$$\sum_{v \in V} f(u, v) = 0.$$

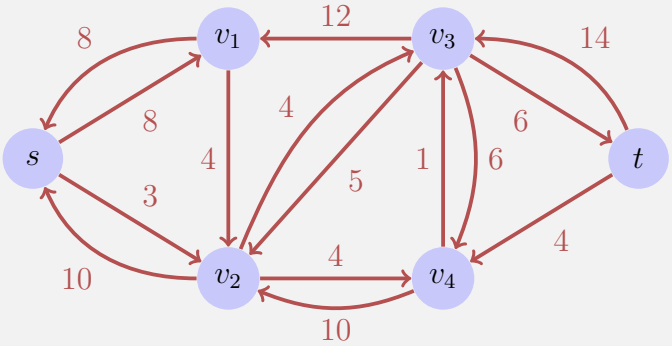



*Wert* $w$ des Flusses:
$|f| = \sum_{v \in V} f(s, v)$.
Hier $|f| = 18$.

# Restnetzwerk

*Restnetzwerk* $G_f$ gegeben durch alle Kanten mit Restkapazität:




Restnetzwerke haben dieselben Eigenschaften wie Flussnetzwerke, ausser dass antiparallele Kanten zugelassen sind.

# Beobachtung

**Theorem**

*Sei $G = (V, E, c)$ ein Flussnetzwerk mit Quelle $s$ und Senke $t$ und $f$ ein Fluss in $G$. Sei $G_f$ das dazugehörige Restnetzwerk und sei $f'$ ein Fluss in $G_f$. Dann definiert $f \oplus f'$ mit*

$$(f \oplus f')(u, v) = f(u, v) + f'(u, v)$$

*einen Fluss in $G$ mit Wert $|f| + |f'|$.*

# Erweiterungspfade

*Erweiterungspfad* $p$: einfacher Pfad von $s$ nach $t$ im Restnetzwerk $G_f$.

*Restkapazität* $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ Kante in } p\}$

# Fluss in $G_f$

**Theorem**

*Die Funktion* $f_p : V \times V \to \mathbb{R}$,

$$f_p(u,v) = \begin{cases} c_f(p) & \text{wenn } (u,v) \text{ Kante in } p \\ -c_f(p) & \text{wenn } (v,u) \text{ Kante in } p \\ 0 & \text{sonst} \end{cases}$$

*ist ein Fluss in* $G_f$ *mit dem Wert* $|f_p| = c_f(p) > 0$.

$f_p$ ist ein Fluss (leicht nachprüfbar). Es gibt genau einen Knoten $u \in V$ mit $(s,u) \in p$. Somit $|f_p| = \sum_{v \in V} f_p(s,v) = f_p(s,u) = c_f(p)$.

# Max-Flow Min-Cut Theorem

**Theorem**

*Wenn $f$ ein Fluss in einem Flussnetzwerk $G = (V, E, c)$ mit Quelle $s$ und Senke $t$ is, dann sind folgende Aussagen äquivalent:*

1. *$f$ ist ein maximaler Fluss in $G$*
2. *Das Restnetzwerk $G_f$ enthält keine Erweiterungspfade*
3. *Es gilt $|f| = c(S, T)$ für einen Schnitt $(S, T)$ von $G$.*

# Algorithmus Ford-Fulkerson($G, s, t$)

**Input :** Flussnetzwerk $G = (V, E, c)$
**Output :** Maximaler Fluss $f$.

**for** $(u, v) \in E$ **do**
    $f(u, v) \leftarrow 0$

**while** Existiert Pfad $p : s \rightsquigarrow t$ im Restnetzwerk $G_f$ **do**
    $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \in p\}$
    **foreach** $(u, v) \in p$ **do**
        **if** $(u, v) \in E$ **then**
            $f(u, v) \leftarrow f(u, v) + c_f(p)$
        **else**
            $f(v, u) \leftarrow f(u, v) - c_f(p)$

# Edmonds-Karp Algorithmus

Wähle in der Ford-Fulkerson-Methode zum Finden eines Pfades in $G_f$ jeweils einen Erweiterungspfad kürzester Länge (z.B. durch Breitensuche).

**Theorem**

*Wenn der Edmonds-Karp Algorithmus auf ein ganzzahliges Flussnetzwerk $G = (V, E)$ mit Quelle $s$ und Senke $t$ angewendet wird, dann ist die Gesamtanzahl der durch den Algorithmus angewendete Flusserhöhungen in $\mathcal{O}(|V| \cdot |E|)$*
*$\Rightarrow$ Gesamte asymptotische Laufzeit: $\mathcal{O}(|V| \cdot |E|^2)$*

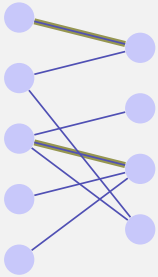
[Ohne Beweis]

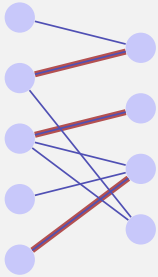
# Anwendung: Maximales bipartites Matching

Gegeben: bipartiter ungerichteter Graph $G = (V, E)$.

Matching $M$: $M \subseteq E$ so dass $|\{m \in M : v \in m\}| \leq 1$ für alle $v \in V$.

Maximales Matching $M$: Matching $M$, so dass $|M| \geq |M'|$ für jedes Matching $M'$.





# 3. Programmieraufgabe

# Aufgabe 10.3: Union-Find

- Input: *union*-Operationen, gefolgt von Anfragen, ob sich zwei Elemente im selben Set befinden.
- Output: Für jede Anfrage, beantworte ob sich die Elemente im selben Set befinden.
- Stelle sicher, dass der Code für die nächste Aufgabe wiederverwendet werden kann.

# Aufgabe 10.4: Kruskals MST-Algorithmus

- Kanten müssen sortiert werden.

# Aufgabe 10.4: Kruskals MST-Algorithmus

- Kanten müssen sortiert werden.
- Erstelle eine *Edge*-Klasse, die Vergleichsoperator implementiert.
- Dann verwende *std::sort*.

# Fragen?