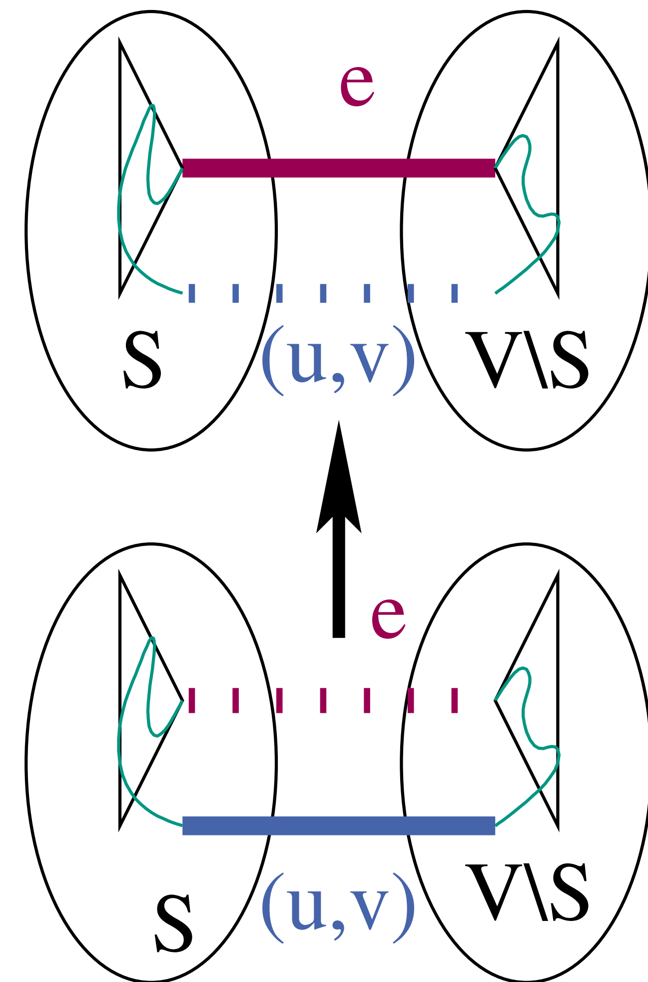
MST-Kanten auswählen und verwerfen

Die Schnitteigenschaft (Cut Property)

Für beliebige Teilmenge $S \subset V$ betrachte die Schnittkanten
$C = \{\{u,v\} \in E : u \in S, v \in V \setminus S\}$

> Die leichteste Kante in C kann in einem MST verwendet werden.

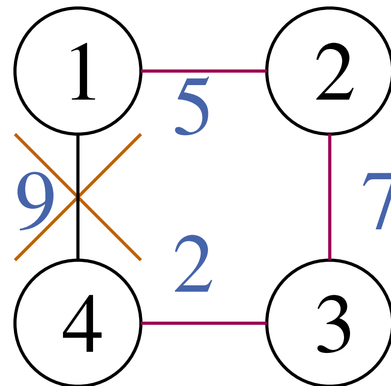
Beweis: Betrachte MST T'.
Fall $e \in T'$: Beweis fertig.
Sonst: $T' \cup \{e\}$ enthält Kreis K.
Betrachte eine Kante $\{u,v\} \in C \cap K \neq e$.
Dann ist $T = T' \setminus \{\{u,v\}\} \cup \{e\}$ ein Spannbaum, der nicht schwerer ist.
Denn: $c(e) \leq c(\{u,v\})$



MST-Kanten auswählen und verwerfen

Die Kreiseigenschaft (Cycle Property)

Die schwerste Kante auf einem Kreis wird nicht für einen MST benötigt.



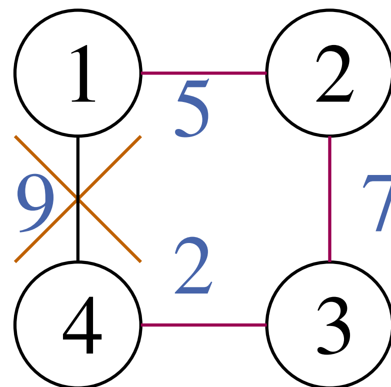
MST-Kanten auswählen und verwerfen

Die Kreiseigenschaft (Cycle Property)

Die schwerste Kante auf einem Kreis wird nicht für einen MST benötigt.

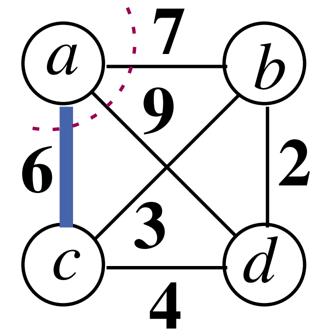
Beweis.

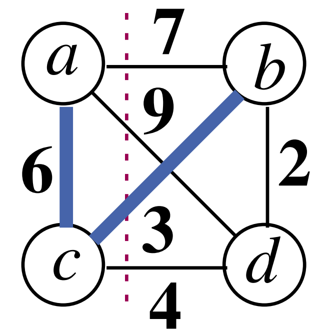
Angenommen, MST T' benutzt die schwerste Kante e' auf Kreis C.
Wähle $e \in C$ mit $e \notin T'$.
Es gilt $c(e) \leq c(e')$.
Dann ist $T = T' \setminus \{e'\} \cup \{e\}$ auch ein MST.

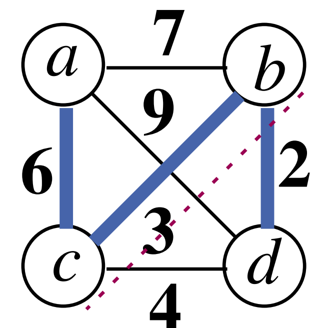


Der Jarník-Prim-Algorithmus

[Jarník 1930, Prim 1957]







Idee: Lasse einen Baum wachsen

$T := \emptyset$

$S := \{s\}$ for arbitrary start node s

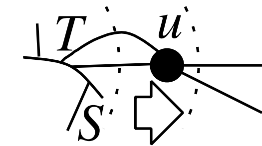
repeat $n - 1$ times

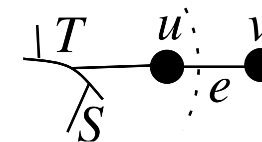
find (u, v) fulfilling the cut property for S

$S := S \cup \{v\}$

$T := T \cup \{(u, v)\}$

```
Function jpMST : Set of Edge          // weitgehend analog zu Dijkstra
    pick any s ∈ V
    d = {∞,...,∞}; parent[s]:= s; d[s] := 0;   Q.insert(s)
    while Q ≠ ∅ do
        u := Q.deleteMin
        d[u]:= 0
        // scan u
        foreach edge e = (u,v) ∈ E do
            if c(e) < d[v] then                          // relax
                d[v]:= c(e)
                parent[v] := u                            // update tree
                if v ∈ Q then Q.decreaseKey(v)
                else Q.insert(v)
    return {(v,parent[v]) : v ∈ V \ {s}}
```





- $d[u] = 0 \Leftrightarrow u \in S$
- $\Rightarrow$ am Ende jeder **while**-Iteration: endliches $d[v] > 0$ speichert **Gewicht der leichtesten Kante von $v \notin S$ über den Schnitt**

Analyse

Praktisch identisch zu Dijkstra

- $O(m+n)$ Zeit ausserhalb der PQ
- $n \times$ deleteMin (Zeit $O(n \log n)$)
- $O(m) \times$ decreaseKey

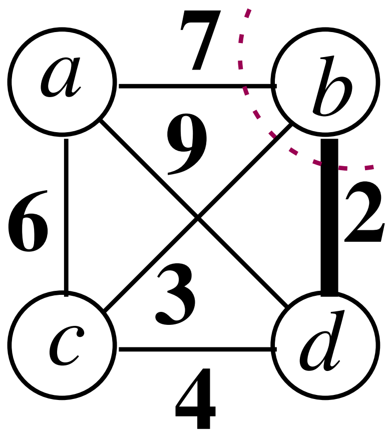
$\rightsquigarrow$ $O((m+n)\log n)$ mit binären Heaps

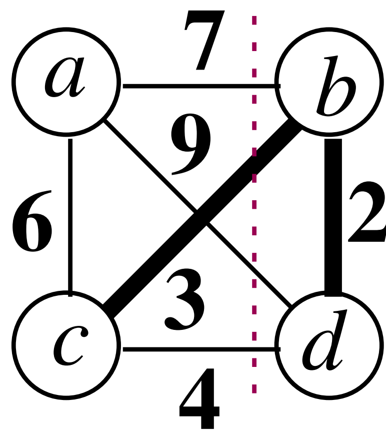
$\rightsquigarrow$ $O(m + n\log n)$ mit Fibonacci Heaps

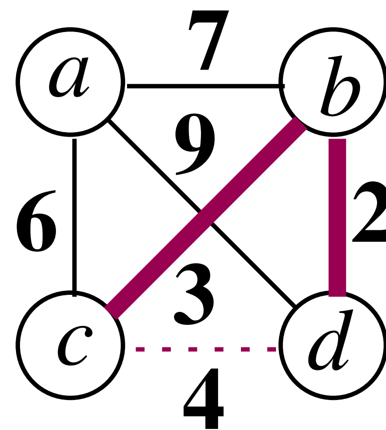
Wichtigster Unterschied: monotone PQs reichen nicht   Warum?

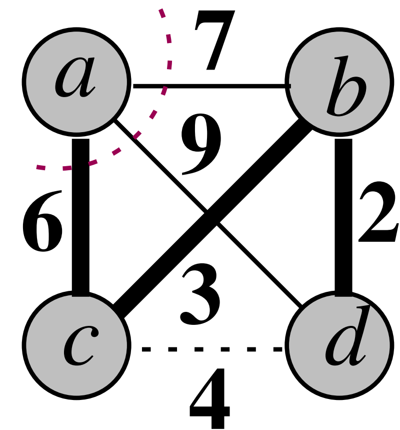
Kruskals Algorithmus [1956]

$T := \emptyset$ // subforest of the MST
foreach $(u, v) \in E$ in ascending order of weight **do**
  if u and v are in different subtrees of (V, T) **then**
    $T := T \cup \{(u, v)\}$ // Join two subtrees
return T









Kruskals Algorithmus – Korrektheit

```
T := ∅                                                    // subforest of the MST
foreach (u, v) ∈ E in ascending order of weight do
    if u and v are in different subtrees of (V, T) then
        T := T ∪ {(u, v)}                                 // Join two subtrees
return T
```

Fall u, v in verschiedenen Teilbäumen: benutze Schnitteigenschaft

$\Longrightarrow (u, v)$ ist leichteste Kante im $\text{cut}(\text{Komponente}(u), V \setminus \text{Komponente}(u))$

$\Longrightarrow (u, v) \in$ MST

Sonst: benutze Kreiseigenschaft

$\Longrightarrow (u, v)$ ist schwerste Kante im Kreis $\langle u, v, v\text{–}u\text{-Pfad in } T\rangle$

$\Longrightarrow (u, v) \notin$ MST $\blacksquare$

Union-Find Datenstruktur

$\Longrightarrow$ Wir brauchen eine effiziente Methode für
“Verbindet die Kante verschiedene Teilbäume?”

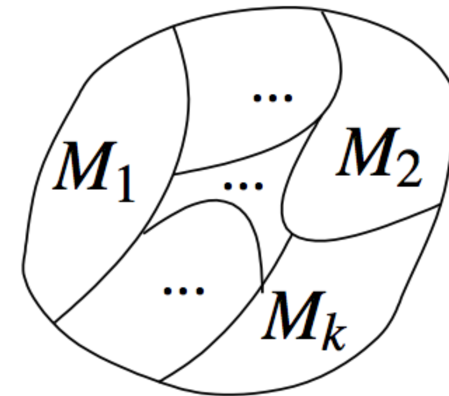
Union-Find Datenstruktur

$\Longrightarrow$ Wir brauchen eine effiziente Methode für "Verbindet die Kante verschiedene Teilbäume?"

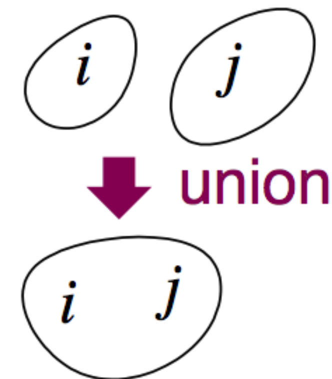
Verwalte Partition der Menge $1..n$,
d. h., Mengen (Blöcke) $M_1, \ldots, M_k$ mit

$$M_1 \cup \cdots \cup M_k = 1..n,$$

$$\forall i \neq j : M_i \cap M_j = \emptyset$$



Class UnionFind($n : \mathbb{N}$)
Procedure union($i, j : 1..n$)
join the blocks containing i and j
to a single block.
Function find($i : 1..n$) : $1..n$
return a unique identifier
for the block containing i.



Union-Find Datenstruktur – Erste Version

Class UnionFind($n : \mathbb{N}$)

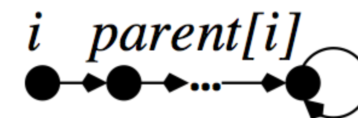
parent=$\langle 1, 2, \ldots, n \rangle$: **Array** $[1..n]$ **of** $1..n$

invariant parent-refs lead to unique Partition-Reps

Function find($i : 1..n$) : $1..n$

if parent$[i] = i$ **then return** i

else return find(parent$[i]$)



Union-Find Datenstruktur – Erste Version

Class UnionFind($n : \mathbb{N}$)

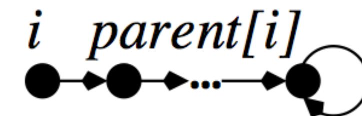
parent=$\langle 1, 2, \ldots, n \rangle$: **Array** $[1..n]$ **of** $1..n$

invariant parent-refs lead to unique Partition-Reps

Function find($i : 1..n$) : $1..n$

if parent[i] $= i$ **then return** i

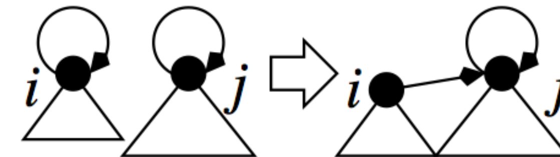
else return find(parent[i])



Procedure link($i, j : 1..n$)

assert i and j are representatives of different blocks

parent[i] := j



Procedure union($i, j : 1..n$)

if find(i) $\neq$ find(j) **then** link(find(i), find(j))

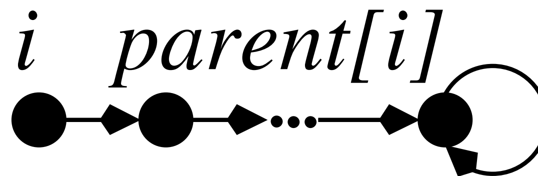
Union-Find Datenstruktur – Erste Version

Analyse

- $+$: union braucht konstante Zeit
- $-$: find braucht Zeit $\Theta(n)$ im schlechtesten Fall!

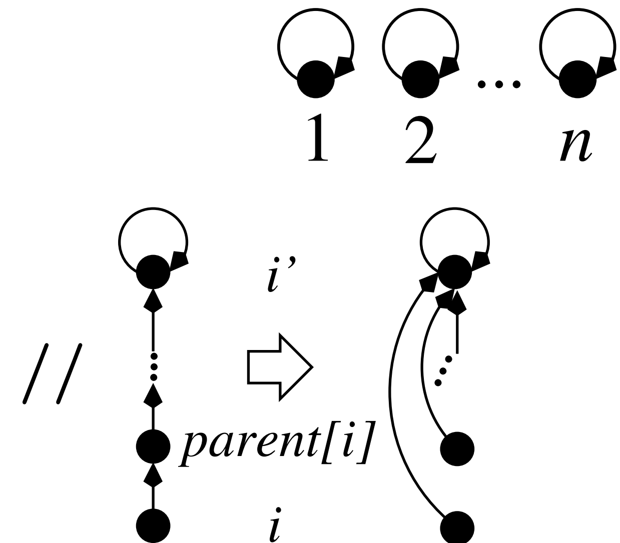
Zu langsam!

Idee: find-Pfade kurz halten



Pfadkompression

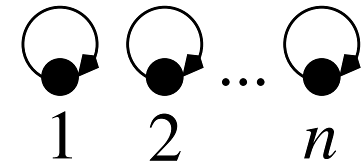
```
Class UnionFind(n : ℕ)
    parent=⟨1,2,...,n⟩  : Array [1..n] of 1..n

    Function find(i : 1..n) : 1..n
        if parent[i] = i then return i
        else i' := find(parent[i])
            parent[i] := i'
            return i'
```



Union by Rank

Class UnionFind($n : \mathbb{N}$)

parent=$\langle 1, 2, \ldots, n \rangle$: **Array** [1..n] **of** 1..n

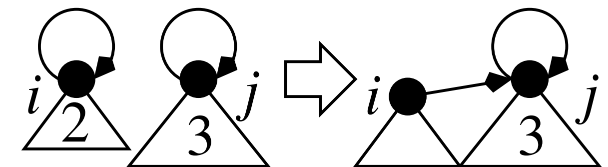


rank=$\langle 0, \ldots, 0 \rangle$: **Array** [1..n] **of** 0..log n

Procedure link(i, j : 1..n)

assert i and j are representatives of different blocks

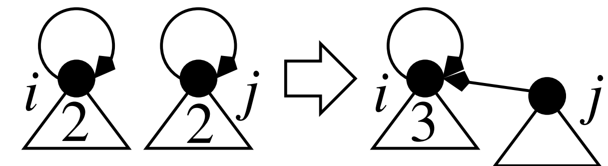
if rank[i] < rank[j] **then** parent[i] := j



else

parent[j] := i

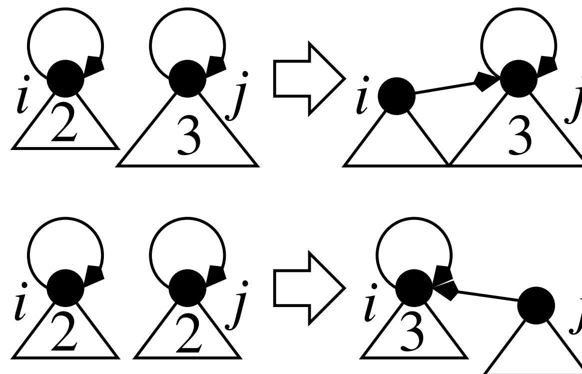
if rank[i] = rank[j] **then** rank[i]++



Analyse – nur Union by rank

invariant Der Pfad zum Repr. x hat Länge höchstens rank[x]
invariant x ist Repr. $\Rightarrow$ x's Menge hat Größe mindestens $2^{\text{rank}[x]}$

Korollar: find braucht Zeit $O(\log n)$



Analyse – nur Pfadkompression

Satz: find braucht Zeit $O(\log n)$ (amortisiert)
Beweis: im Buch

Analyse – Pfadkompression + Union by rank

Satz: $m\times$ find $+$ $n\times$ link brauchen Zeit $\mathrm{O}(m\alpha_T(m,n))$ mit

$$\alpha_T(m,n) = \min\{i \geq 1 : A(i, \lceil m/n \rceil) \geq \log n\}$$

und

$$\begin{aligned} A(1,j) &= 2^j && \text{for } j \geq 1, \\ A(i,1) &= A(i-1,2) && \text{for } i \geq 2, \\ A(i,j) &= A(i-1, A(i,j-1)) && \text{for } i \geq 2 \text{ and } j \geq 2. \end{aligned}$$

Beweis: [Tarjan 1975, Seidel Sharir 2005]
A ist die Ackermannfunktion und α_T die inverse Ackermannfunktion.
$\alpha_T(m,n) = \omega(1)$ aber ≤ 5 für alle physikalisch denkbaren n, m.