low[v] = min({d[v]} ∪ {d[w] | es gibt Nachfolger u von v (einschließlich v), so dass (u, w) Rückwärtskante ist})

8.8. **Algorithmus** (Berechnung des low-array)

In DFS(G) wird DFS-visit(u) nur einmal aufgerufen. DFS-visit(u) wird in folgender Weise modifiziert:

Prozedur Mod-DFA-visit(u)

```
1.  color[u] := grau
2.  d[u] := time
3.  low[u] := time                          ; Initialisierung von low[u] auf d[u]
4.  time := time + 1
5.  for each v ∈ Adj[u] do
6.     if color[v] = weiß
7.        then π[u] := u
8.             Mod-DFS-visit(v)
9.             low[u] := min(low[u], low[v])
10.    if color[v] = grau and π[u] ≠ v       ; Test auf Rückwärtskante
11.       then low[u] := min(low[u], d[v])
12. color[u] := schwarz
13. f[u] := time
14. time := time + 1
```

Die Laufzeit des Algorithmus ist $\Theta(|V| + |E|)$.

8.9. **Algorithmus** (Bestimmung der Artikulationspunkte)

Prozedur Artikulationspunkte

1. Berechne das low-array für den Graphen G nach Algorithmus 8.8.
2. Teste in $G_\pi = (V, E_\pi)$ die Bedingungen (a) und (b):

   (a) Die Wurzel des Baums ist Artikulationspunkt ⇔ die Wurzel hat mindestens zwei Söhne

   (b) Ein Nichtwurzelknoten ist Artikulationspunkt ⇔ es gibt in G_π einen Sohn v von u, so dass $low[v] \geq d[u]$.

Die Korrektheit des Algorithmus ergibt sich aus Lemma 8.6.

8.10. **Beispiel** (Bestimmung von zweifachen Zusammenhangskomponenten)

Betrachte den Graphen von Abbildung 8.2. Der Graph hat drei zweifache Zusammenhangskomponenten. Offensichtlich ist C ein Artikulationspunkt.

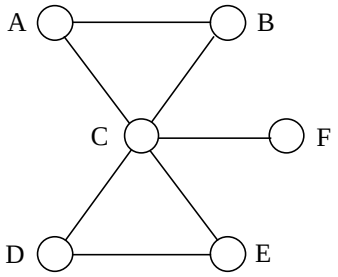


Abbildung 8.2

8.11. **Algorithmus** (DFS-visit-stack)

Der Algorithmus stellt eine Modifikation von DFS-visit dar, bei der die betrachteten Kanten in einem Stack verwaltet werden. Er wird nur im Zusammenhang mit dem Algorithmus zur Bestimmung des low-Array benützt (siehe 8.12).

Prozedur DFS-visit-stack(u)

```
1.   color[u] := grau
2.   d[u] := time
3.   time := time + 1
4.   for each v ∈ Adj[u]
5.a     do   speichere (u, v) auf Stack, falls sie noch nicht dort ist
5.b          if color[v] = weiß
6.           then π[v] := u
7.                DFS-visit-stack(v)
8.a               if low[v] = d[u]
8.b                  then gib Stack bis einschließlich (u, v) als eine zweifache Zusammenhangs-
                          komponente aus
8.c               if low[v] = d[v]
8.d                  then gib Kellerspitze mit (u, v) als eine zweifache Zusammenhangskomponente
aus
9.   color[u] := schwarz
10.  f[u] := time
11.  time := time + 1
```

8.12. **Algorithmus** (Zweifache Zusammenhangskomponenten)

Prozedur Zweifache-Zusammenhangskomponenten

1. Führe Prozedur Mod-DFA-visit(u) (Berechnung des low-Array) durch.
2. Führe Prozedur DFS-visit-stack(u) in derselben Durchlaufreihenfolge wie in 1. durch.

8.13. **Satz** (Korrektheit von Zweifache-Zusammenhangskomponenten)

Die Prozedur Zweifache-Zusammenhangskomponenten ist korrekt.

9. Kürzeste Wege I: Floyd-Warshall-Algorithmus

9.1. **Definition** (Gewichteter, gerichteter Graph)

(a) Sei G = (V, E) ein gerichteter Graph, d.h. $E \subseteq V \times V$, und eine Kante $(v, w) \in E$ sieht wie in Abbildung 9.1 dargestellt aus. Es ist also $(v, w) \neq (w, v)$. Ist $(v, w) \in E$, dann ist $v \neq w$, d.h. es gibt keine Schleifen.

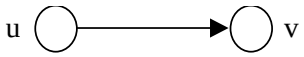


Abbildung 9.1

Eine Funktion $d: E \rightarrow \mathbb{R}$ heißt *Gewichtsfunktion*.

(b) Ein *Weg* in G ist eine Folge $(v_1, v_2, ..., v_n)$ von Knoten, wobei für alle i mit $1 \leq i \leq n-1$ gilt: $(v_i, v_{i+1}) \in E$.

Die *Länge* eines Weges ist definiert durch

$$Lg(v_1, v_2, ..., v_n) = \sum_{i=1}^{n-1} d(v_i, v_{i+1})$$

Ein *Kreis* ist ein Weg mit $v_n = v_1$. Ein *einfacher Kreis* ist ein Kreis mit $v_i \neq v_j$ für alle $i, j \in \{1, ..., n\}$.

(c) Die Distanz zweier Knoten $v, w \in V$ ist definiert durch:

$$Dist(v, w) = min\{Lg(W) \mid W \text{ ist ein Weg von } v \text{ nach } w\}$$

9.2. **Algorithmus** (Floyd-Warshall-Algorithmus)

Der Algorithmus bestimmt einen kürzesten Weg von jedem Knoten u zu jedem Knoten v. Er wurde von Floyd 1962 publiziert.

Prozedur Kürzeste-Wege

Eingabe: G = (V, E) mit der Knotennummerierung V = {1, ..., |V|} und der Gewichtsfunktion $d: E \rightarrow \mathbb{R}$. G besitze keine Kreise negativer Länge.
Ausgabe: A[1..|V|, 1..|V|], mit A[i, j] = Länge eines kürzesten Weges von i nach j.

```
1.  for each (i, j) ∈ E do A[i, j] := d[i, j]
2.  for each (i, j) ∉ E, i ≠ j, do A[i, j] := ∞
3.  for each i do A[i, i] := 0
4.  for k = 1 to |V| do
5.     for i = 1 to |V| do
6.        for j = 1 to |V| do
7.           if A[i, k] + A[k, j] < A[i, j]
8.              then A[i, j] := A[i, k] + A[k, j]
```

Die Laufzeit des Algorithmus ist $O(|V|^3)$.

9.3. Satz (Korrektheit des Floyd-Warshall-Algorithmus)

Der Floyd-Warshall-Algorithmus ermittelt die kürzesten Wege in einem gerichteten, gewichteten Graphen G.

10. Kürzeste Wege II: Dijkstras Algorithmus (Dijkstra 1959)

10.1. Algorithmus (Dijkstra-Algorithmus)

Prozedur Kürzeste-Wege

Eingabe: Ein gewichteter Graph G = (V, E) in Adjazenzlistendarstellung mit Gewichten $d[i, j] \geq 0$ für alle $(i, j) \in E$. Die Gewichte sind bei den Kanten (i, j) gespeichert.

Ausgabe: $D = D[1..|V|]$ mit $D[i] = Dist(1, i)$ für alle $i \in V$.

```
1. S := {1}          ; S ist die Menge der Knoten, zu denen ein kürzester Weg gefunden worden ist
2. D[1] := 0
3. for i = 2 to n do D[i] := ∞
4. for each i ∈ Adj[1] do D[i] := d[1, i]
5. for i = 1 to n – 1
6.    wähle Knoten w ∈ V – S, so dass D[w] minimal ist      ; am Anfang wird die kürzeste
                                                              Kante gewählt
7.    S := S ∪ {w}
8.    for each v ∈ (Adj[w] ∩ V–S)                  ; es werden nur Knoten nicht in S betrachtet
9.       do D[v] := min{D[v], D[w] + d[w, v]}      ; prüfe, ob es über w einen kürzeren Weg gibt
```

10.2. Satz (Korrektheit des Dijkstra-Algorithmus)

Am Ende von Dijkstras Algorithmus ist $D[u] = Dist(1, u)$ für alle $u \in V$.

10.3. Definition (Datenstrukturen für Mengen)

(a) Darstellung einer Menge M als Boolesches Array:

array M[1..|V|] of boolean
$M[i] = true \Leftrightarrow i \in M$

Operationen auf M:

$Insert_M(i)$	Einfügen: O(1)
$Find_M(i)$	Suchen: O(1)
$Delete_M(i)$	Löschen: O(1)

(b) Die Elemente in M sind durch eine Funktion, gegeben durch das Array D, angeordnet. Dann können die folgenden Operationen definiert werden:

Min_M	Minimum (bezüglich D) finden: O(n)
Delete-Min_M	Minimum löschen: O(1)

10.4. Folgerung

Stellt man in Dijkstras Algorithmus die Menge V – S durch direkte Adressen dar, d.h.

V – S = V–S[1..|V|] of boolean

dann hat der Algorithmus eine Laufzeit von $O(n^2)$.

10.5. **Definition** (Datenstruktur Heap)

Ein Heap ist ein „fast vollständiger“ binärer Baum, dessen Knoten mit Funktionswerten versehen sind. Die Knoten sind so angeordnet, dass die Funktionswerte von den Blättern zu der Wurzel immer kleiner werden. Genauer formuliert: Für alle Knoten u, v eines Heap gilt: Ist u Vorgänger von v, dann ist Wert(u) ≤ Wert(v).

Beispiel

Der in Abbildung 10.1 dargestellte Baum ist ein Heap. Die Werte der Knoten seien durch einen Array D gegeben. Die Beschriftung des Knotens i hat die Form i/D[i].

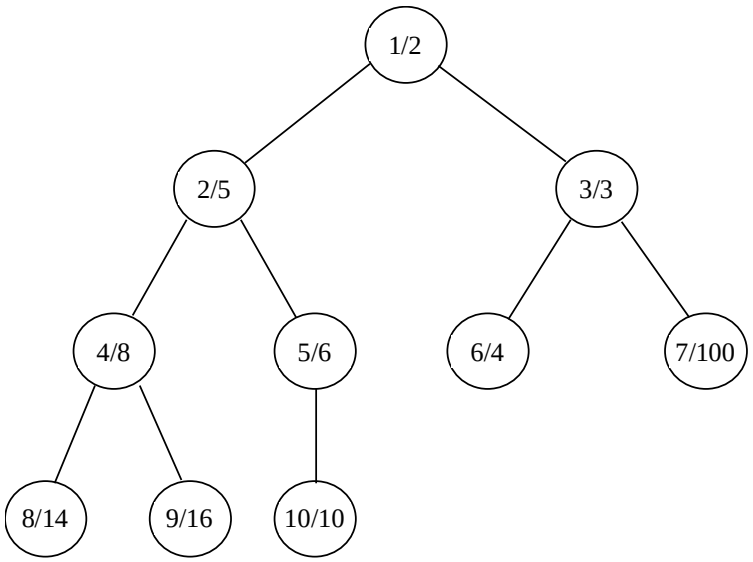


Abbildung 10.1

Operationen auf einem Heap:

(a) Minimum finden: Zeit O(1)

(b) Minimum löschen:

1. Entferne letztes Blatt.
2. Überschreibe die Wurzel mit dem letzten Blatt
3. Lasse die neue Wurzel an die richtige Stelle „sickern“.

Setzt man für einen Vertauschungsschritt den Zeitbedarf O(1) an, so ist die Gesamtzeit für die Operation *Minimum löschen* O(log n), wobei n die Zahl der Elemente im Heap ist.

(c) Element einfügen

1. Setze das neue Element auf das nächste freie Blatt.
2. Lasse das neue Element im Baum aufsteigen bis die Heap-Eigenschaft wieder hergestellt ist.

10.6. **Beispiel** (Datenstruktur Heap)

Die Verwendung eines Heap zur Darstellung der Menge V – S wird am Beispiel des Graphen von Abbildung 10.2 dargestellt.

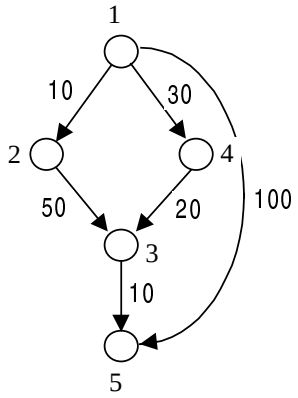


Abbildung 10.2

Zu Beginn des Algorithmus von Dijkstra ist S = {1}, V – S = {2, 3, 4, 5}, D[2] = 10, D[3] = ∞, D[4] = 30, D[5] = 100. Es gibt zwei alternative Darstellungen der Menge V – S als Heap.

Im nächsten Schritt ist S = {1, 2}, V – S = {3, 4, 5}, D[3] = 60, D[4] = 30, D[5] = 100. Bezogen auf den Heap muss also die Operation *Minimum löschen* durchgeführt werden.

(d) Operation Decrease(x, k):

1. Finde Element x im Heap über direkte Adressen
2. Erniedrige Wert von x auf k
3. Lasse das Element im Heap aufsteigen, falls die Heap-Eigenschaft verletzt ist. Modifiziere dabei das Adressen-Array in entsprechender Weise.

Der Zeitbedarf für Decrease(x, k) ist O(log n), falls eine Vertauschung in Zeit O(1) durchführbar ist. n ist die Größe des Heap.

10.7. **Implementierung** (Datenstruktur Heap)

Ein Heap wird durch die Arrays *A[1..n]*, *A'[1..n]* und *D[1..n]* und eine Variable *heapsize* implementiert. In A werden die Elemente des Heap gespeichert, in A' ihre Adressen und in D die Funktionswerte. heapsize gibt die aktuelle Größe des Heap an, d.h. die Anzahl der enthaltenen Elemente. Abbildung 10.3 illustriert die Implementierung an einem Beispielheap.

Auf- und Absteigen im Baum auf der Basis der Array-Darstellung.

Für den Knoten i werden die Operationen Vater(i), Linker-Sohn(i), Rechter-Sohn(i), Sicker(i) und Steig(i) definiert.

Vater(i) = A[$\lfloor$A'[i]/2$\rfloor$]. Zeitbedarf: O(1).

Linker-Sohn(i) = 2·A'[i]

Rechter-Sohn(i) = 2·A'[i] + 1

Sicker(i)

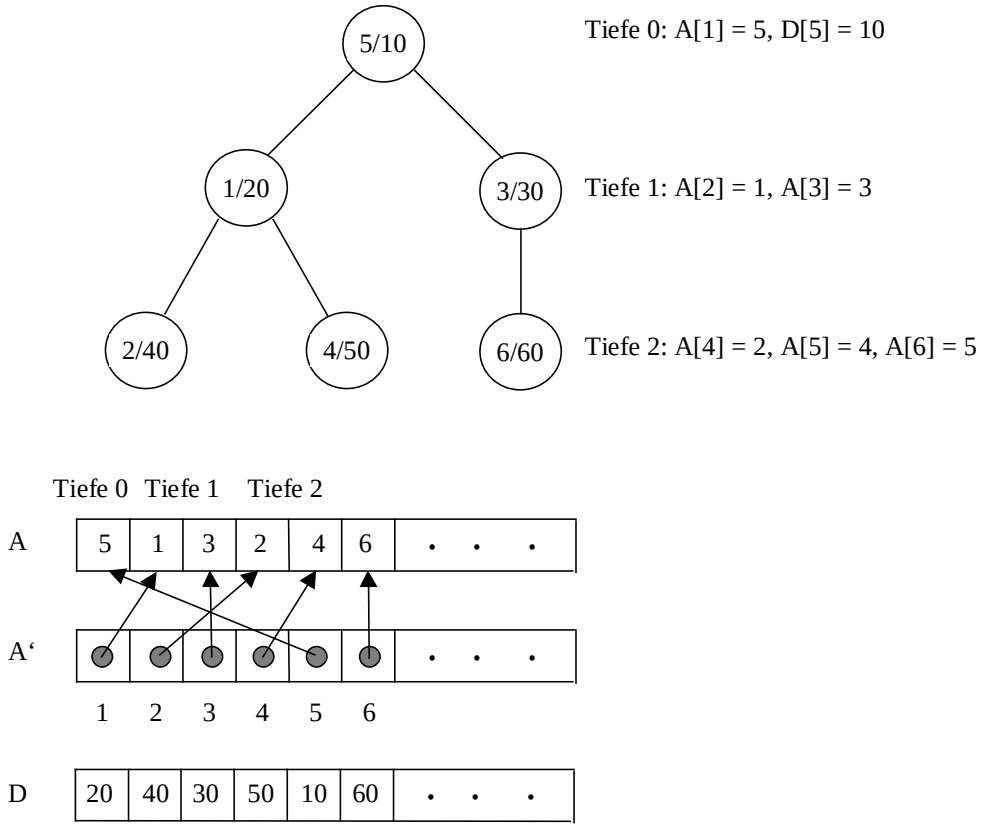


Abbildung 10.3

```
1.  j := 0
2.  if D[A[i]] > min{D[A[2i]], D[A[2i+1]]}
3.      then    k := Index von min{D[A[2i]], D[A[2i+1]]}
4.              Vertausche A[i] und A[k]
5.              A‘[A[k]] := k
6.              A‘[A[i]] := i
7.              return k
```

Der Zeitbedarf für Sicker(i) ist O(1), da der Algorithmus keine Schleife enthält. Ist $k \neq 0$, dann ist k der neue Index des alten A[i].

Steig(i)

```
1.  j := 0
2.  if D[A[i]] < D[A[Vater(i)]]
3.      then    j := Vater(i)
4.              Vertausche A[i] und A[Vater(i)]
5.              Passe A‘ an
6.              return j
```

Der Zeitbedarf für Steig(i) ist O(1), da der Algorithmus keine Schleife enthält.

10.8. **Algorithmus** (Dijkstra-Algorithmus mit Heap)

Sei A = A[1..n] ein Heap. Zur Bearbeitung eines Heap werden folgende Operationen benötigt:

Min_A (Minimum finden) Zeitbedarf: O(1)

$Deletemin_A$	(Minimum löschen)	Zeitbedarf: O(log n)
$Insert_A(x)$	(Einfügen von x)	Zeitbedarf: O(log n)
$Decrease_A(x, k)$	(Vermindern von x auf den Wert k)	Zeitbedarf: O(log n)

Im Dijkstra-Algorithmus mit Heap wird die Menge V – {s} (im Folgenden geschrieben: V–S) als Heap dargestellt. Die Werte der Elemente des Heap sind durch das Array D[1..|V|] gegeben. Es werden folgende Arrays benötigt:

D = D[1..|V|] - Werte
(V–S)‘ = (V–S)‘[1..|V|] - direkte Adressen
S = S[1..|V|] array of boolean

1. S : = {1} ; eigentlich Initialisierung des Arrays S
2. D[1] := 0
3. for i = 2 to n do D[i] := ∞
4. for each i ∈ Adj[1] do D[i] := d[1, i]
5. Initialisiere V–S mit den Elementen 2, ..., |V| als Heap bezüglich der Werte in D
6. Setze (V–S)‘
7. for i = 1 to n–1 do
8. w := Min_{V_S} ; in jedem Durchlauf wird ein Element entfernt
9. S := S ∪ {w} ; Ändern des Array S
10. $Deletemin_{V\_S}$; mit Anpassen von (V–S)‘
11. for each v ∈ Adj[w] mit S[v] = 0 ; v ∉ S
12. do D[v] := min{D[v], D[w] + d[w, v]}
13. $Decrease_{V\_S}$ (w, D[v]) ; mit Anpassen von (V–S)‘

10.9. Satz (Zeitbedarf des Dijkstra-Algorithmus mit Heap)

Der Dijkstra-Algorithmus mit Heap benötigt bei Eingabe von G = (V, E) die Zeit O(|E|·log |V|), falls |E| ≥ |V|.

11. Minimale Spannbäume: Kruskals Algorithmus

11.1. Definition (Minimaler Spannbaum)

Sei G = (V, E) ein ungerichteter, zusammenhängender Graph.

(a) Ein Teilgraph H = (V, F) ist ein *Spannbaum* von G genau dann, wenn gilt:

(i) H = (V, F) ist zusammenhängend.
(ii) Für alle e ∈ F ist H – {e} = (V, F – {e}) nicht mehr zusammenhängend.

(b) Ist c: E → IR eine Kostenfunktion, dann ist H = (V, F) ein *minimaler Spannbaum* genau dann, wenn gilt:

(i) H ist ein Spannbaum
(ii) Kosten(H) = $\sum_{e \in F} c(e)$ ≤ Kosten(K) für alle Spannbäume K.

11.2. **Algorithmus** (Minimaler Spannbaum, Kruskal 1956)

Eingabe: Ungerichteter Graph G = (V, E), V = $\{v_1, ..., v_n\}$, mit Kostenfunktion c.
Ausgabe: T – die Menge der kanten eines minimalen Spannbaums von G.

1. T := ∅
2. Kanten in priority queue H nach den Kosten eintragen
3. P := $\{\{v_1\}, ..., \{v_n\}\}$
4. while |P| > 1 do ; läuft so lange P ≠ $\{\{v_1\}, ..., \{v_n\}\}$, maximal |E| mal, mindestens (|V| – 1) mal
5. (v, w) := Min_H ; es wird getestet, ob v und w in verschiedenen Mengen von P vorkommen
6. $Deletemin_H$
7. if U,W ∈ P and U ≠ W and v ∈ U and w ∈ W
8. then P := (P – {U, W}) ∪ {U ∪ W}
9. T := T ∪ {(v, w)}

11.3. **Definition** (Aufspannender Wald)

Sei G = (V, E) ein ungerichteter Graph. Ein *aufspannender Wald* von G ist eine Menge von Bäumen (d.h. kreisfreien zusammenhängenden Graphen)

$$SW = \{(V_1, T_1) , (V_2, T_2), ..., (V_m, T_m)\}$$

mit folgenden Eigenschaften:

- $V_i \subseteq V$ für alle i = 1, ..., m
- $V_i \subseteq V_j = \emptyset$ für alle i ≠ j
- $\bigcup_{i=1}^{m} V_i = V$, d.h. $\{V_1, ..., V_m\}$ ist Partition von V
- $T_i \subseteq E$, genauer: $T_i \subseteq \{(v, w) \mid v, w \in V_i\}$. $T_i = \emptyset$ ist möglich, dann ist aber $V_i = \{v\}$.

11.4. **Satz** (Minimaler Spannbaum)

Sei G = (V, E) ein zusammenhängender Graph mit Kostenfunktion c: E → IR. Sei SW = $\{(V_1, T_1), ..., (V_k, T_k)\}$ ein aufspannender Wald von G mit k > 1. Sei $T = \bigcup_{i=1}^{k} T_i$ und e = {v, w} eine Kante von minimalen Kosten, mit e ∈ E – T und v ∈ V_i, w ∈ V_j für i ≠ j. Dann gibt es einen Spannbaum SB von G, der T ∪ {e} als Teilmenge seiner Kanten enthält, so dass gilt:

c(SB) = min{c(SB') | SB' ist Spannbaum, dessen Kanten eine Obermenge von T bilden}

11.5. **Satz** (Korrektheit des Kruskal-Algorithmus)

Der Algorithmus von Kruskal zur Ermittlung minimaler Spannbäume ist korrekt.

Laufzeit des Kruskal-Algorithmus

1. T := ∅ O(1)
2. Kanten in priority queue H nach den Kosten eintragen O(|E|) (Heap-Aufbau in Linearzeit)

3.	P := $\{\{v_1\}, ..., \{v_n\}\}$	O(|V|)
4.	while |P| > 1 do	O(|V|)
5.	(v, w) := Min_H	} O(|E|·log|E|)
6.	$Deletemin_H$	
7.	if U,W ∈ P and U ≠ W and v ∈ U and w ∈ W	?
8.	then P := (P – {U, W}) ∪ {U ∪ W}	?
9.	T := T ∪ {(v, w)}	O(|V|) insgesamt

Der gesamte Zeitbedarf ergibt sich folgendermaßen:

O(|E|·log|E|)
\+ 2|E| mal feststellen, in welcher Menge der Partition ein Knoten liegt
\+ (|V| – 1) mal zwei Mengen der Partition vereinigen.

Darstellung von P als Array

Die Partition P = $\{V_1, ..., V_m\}$ von V wird als Array P = P[1..n] mit n = |V| definiert durch

$$P[v] = i \Leftrightarrow v \in V_i$$

Operationen auf P:

$Find_P$(v): return P[v] ; P[v] ist die Bezeichnung der Menge, in der v liegt
Zeitbedarf: O(1)

$Union_P$(i, j): for v = 1 to n do ; i, j sind Bezeichnungen von Mengen
if P[v] = i then P[v] := j
Zeitbedarf: O(n)

Mit der Datenstruktur Array für die Partition P hat der Kruskal-Algorithmus die Laufzeit

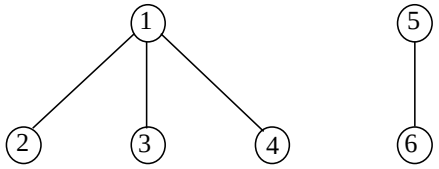
O(|E|·log|E|) ; |E| mal Minimum löschen
\+ O(|E|) ; ≤ 2|E| mal Find(v) ausführen
\+ $O(|V|^2)$; (|V| – 1) mal Union(i, j) ausführen

also insgesamt $O(|E|\cdot\log|E| + |V|^2)$.

Darstellung von P als Menge von Bäumen

Sei P = {{1, 2, 3, 4}, {5, 6}}. Dann gibt es im Prinzip die beiden in Abbildung 11.1 dargestellten Möglichkeiten, die Mengen in P als Bäume zu repräsentieren.

1. Möglichkeit



2. Möglichkeit

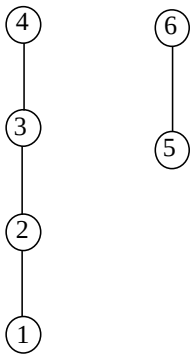


Abbildung 11.1