Graphenalgorithmen I

Geschickt Programmieren für den ICPC-Wettbewerb

Felix Weissenberger

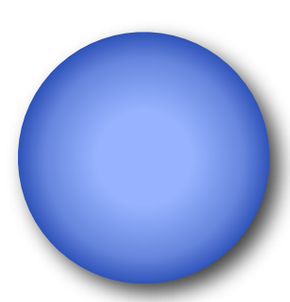
Inhalt

- Grundlagen zu Graphen
  - Begriffe
  - Darstellung von Graphen
- Graphenalgorithmen
  - Breitensuche
  - Tiefensuche
  - Topologisches Sortieren
  - Zusammenhang - Brücken - Artikulationspunkte
  - Euler-Touren

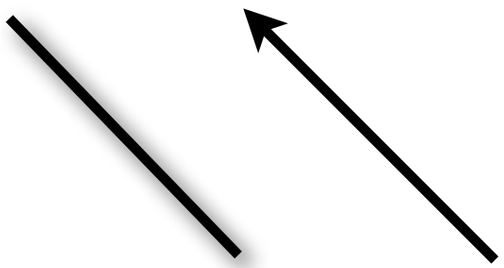
Graph

- Ein Graph G = (V, E) besteht aus:

einer Knotenmenge V (vertices)

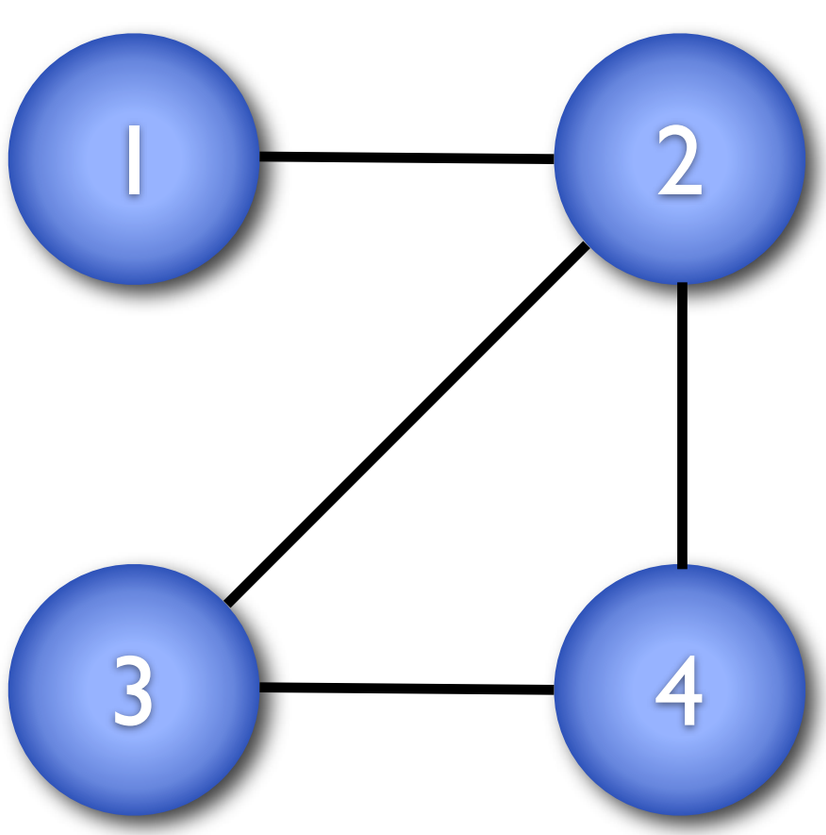


einer Kantenmenge E (edges)



Ungerichteter Graph

- Kanten repräsentiert durch Teilmengen
  $\{u, v\} \subseteq V$



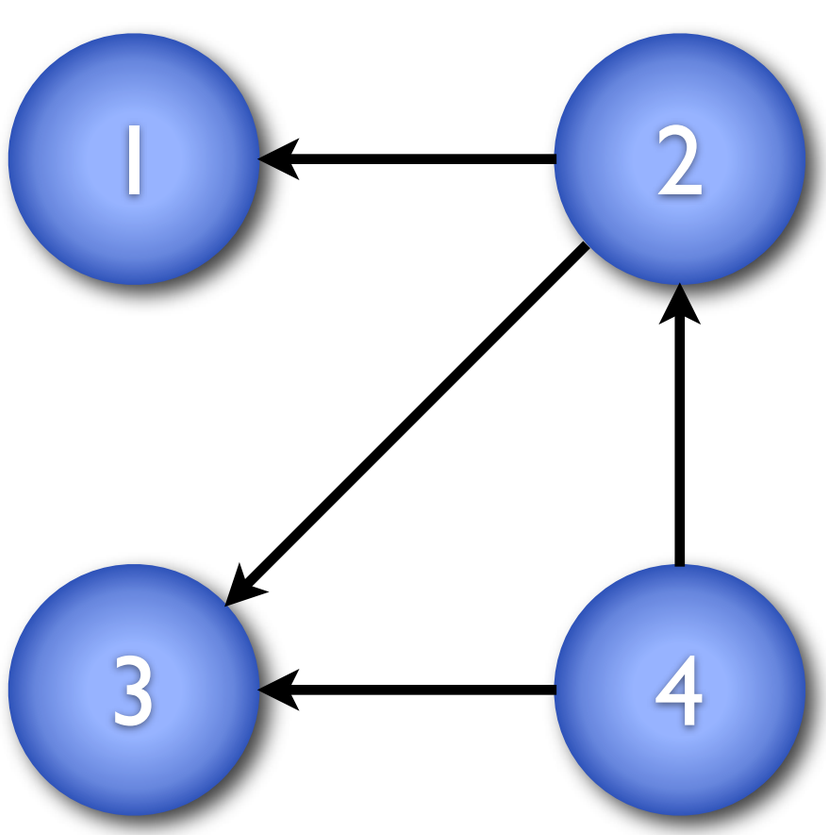
$G = (V, E)$

$V = \{1, 2, 3, 4\}$

$E = \{\{1, 2\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$

Gerichteter Graph

- Kanten repräsentiert durch Tupel
  $(u, v) \in V \times V$



$G = (V, E)$

$V = \{1, 2, 3, 4\}$

$E = \{(2, 1), (2, 3), (4, 2), (4, 3)\}$

Wege, Pfade, Touren

- eine Sequenz $v_0, e_1, v_1, ..., e_n, v_n$ mit $v_i \in V$ und $e_i = \{v_{i-1}, v_i\}$ oder $(v_{i-1}, v_i) \in E$ heißt Weg
- ein Weg mit paarweise verschiedenen Kanten heißt Tour
- ein Weg mit paarweise verschiedenen Knoten heißt Pfad
- gilt $v_0 = v_n$ heißen Pfade/Touren geschlossen

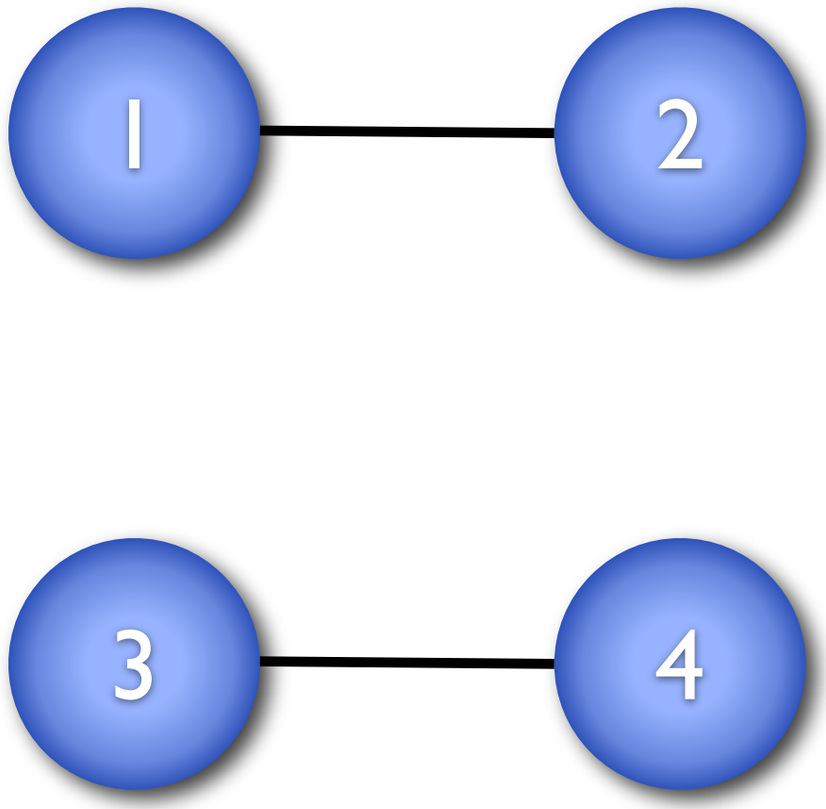
Nachbarschaft, Grad

- zwei Knoten, die über eine Kante miteinander verbunden sind heißen Nachbarn
- die Anzahl der Nachbarn eines Knoten ist sein Grad

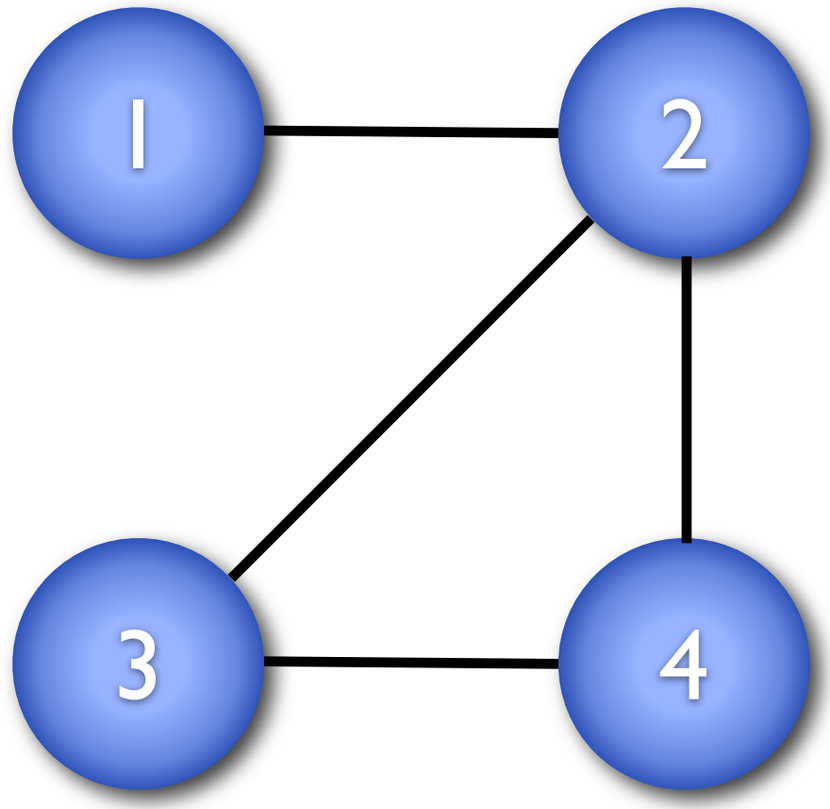
Zusammenhang

- gibt es einen Pfad von u nach v, dann ist v von u aus erreichbar
- ein Graph heißt zusammenhängend, wenn jeder Knoten von jedem anderen Knoten erreichbar ist
- eine Zusammenhangskomponente ist ein maximal zusammenhängender Teilgraph

Zusammenhang

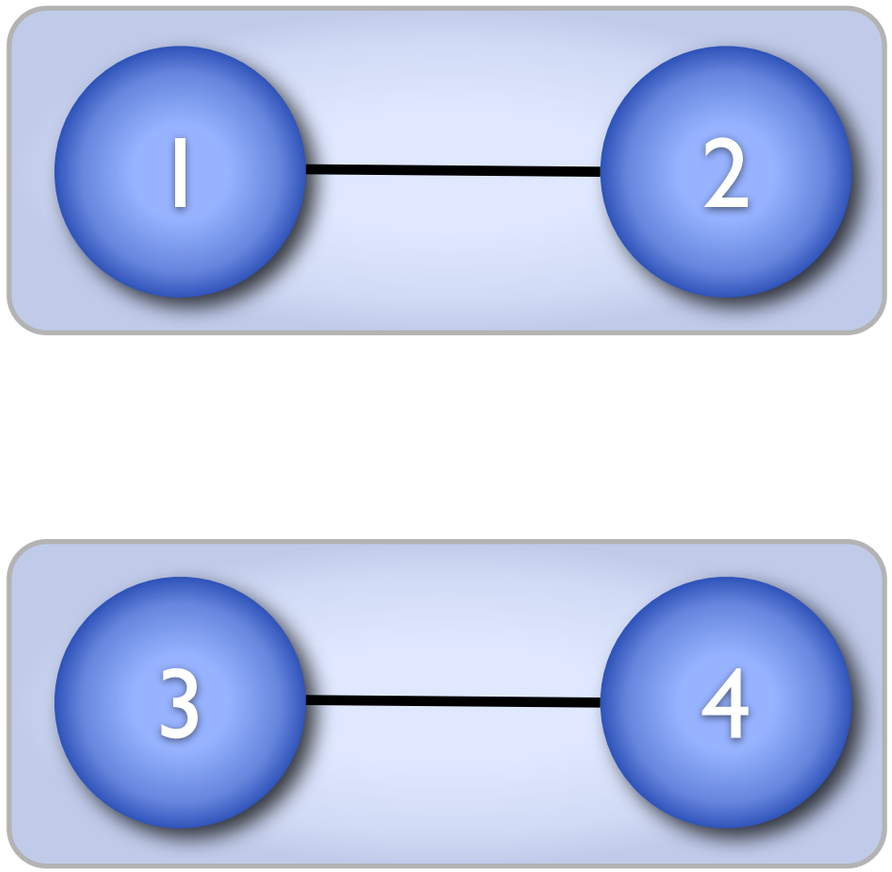


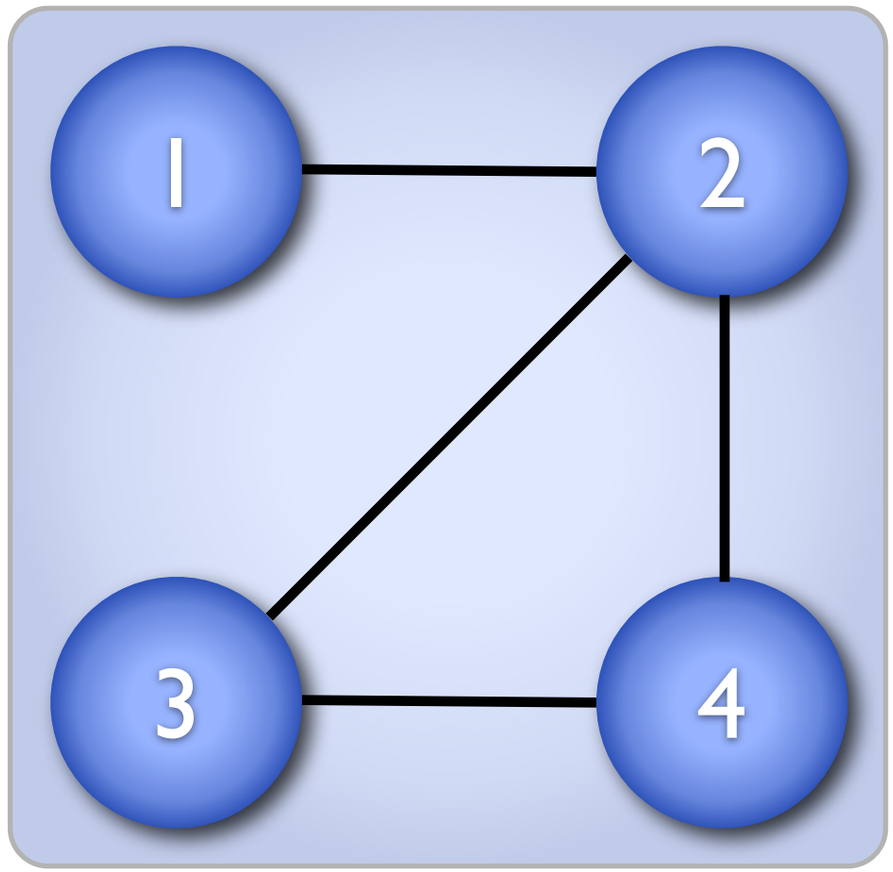
nicht zusammenhängend



zusammenhängend

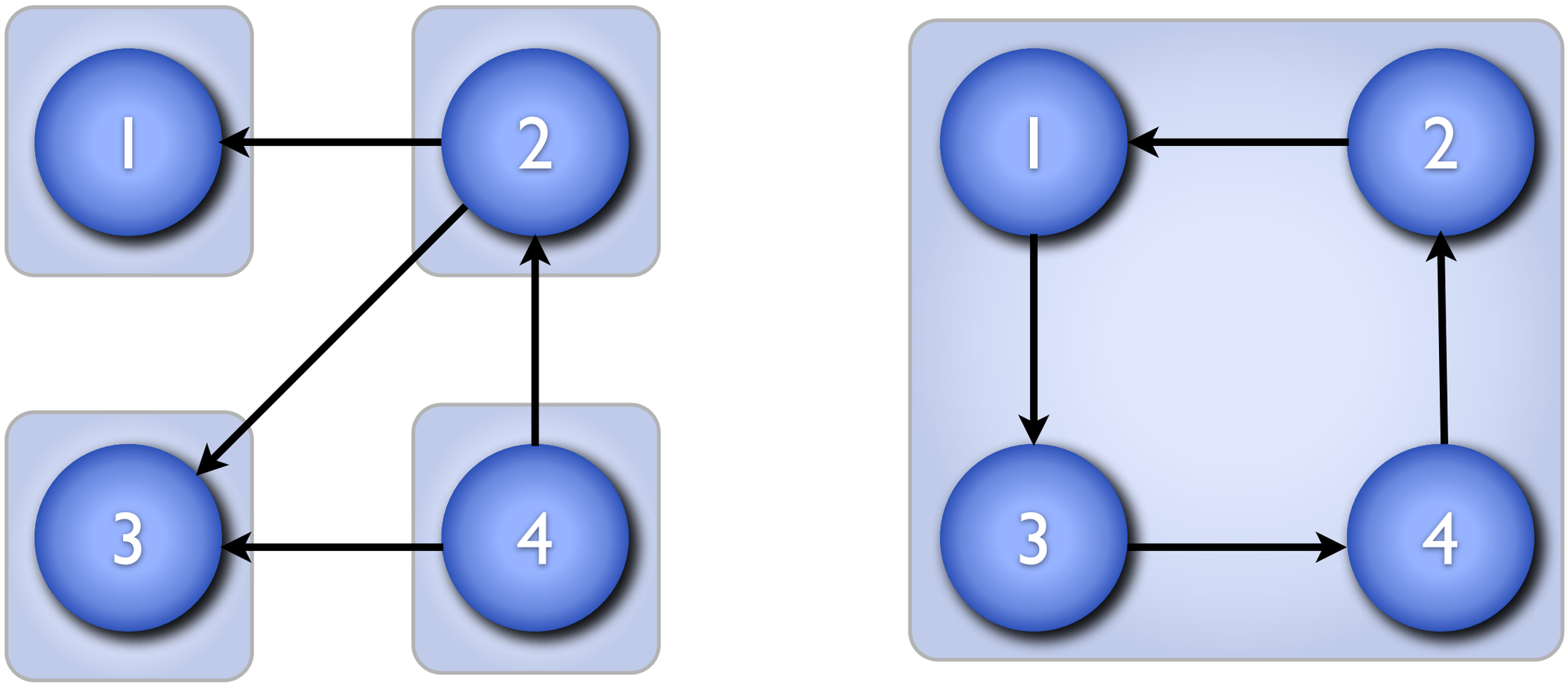
Zusammenhangs-komponenten CC´s





in ungerichteten Graphen

starke Zusammenhangs-komponenten sCC´s

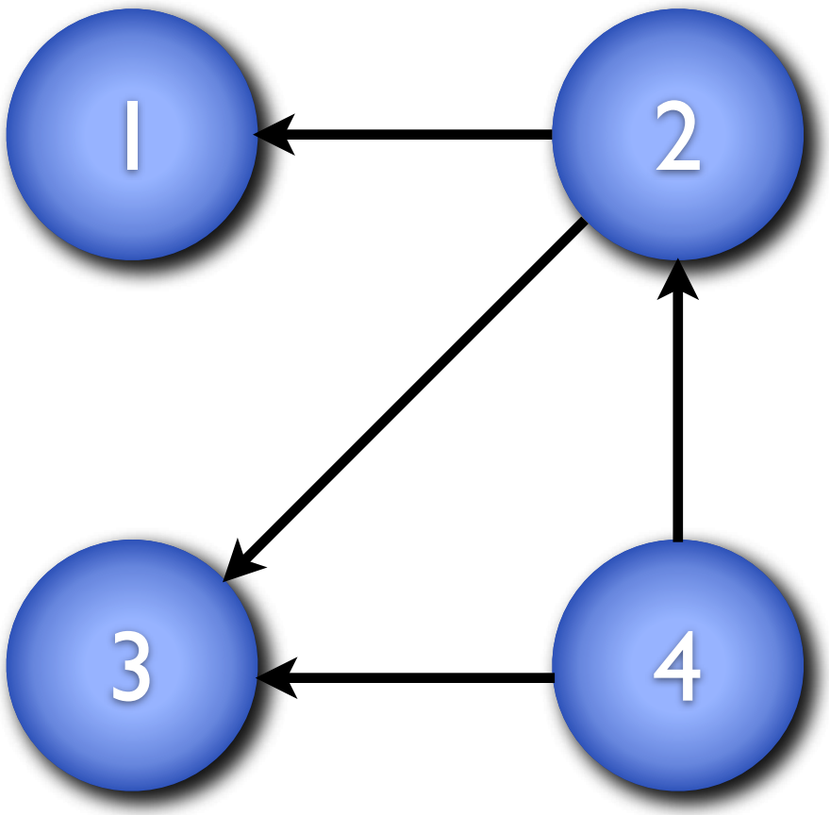


in gerichteten Graphen

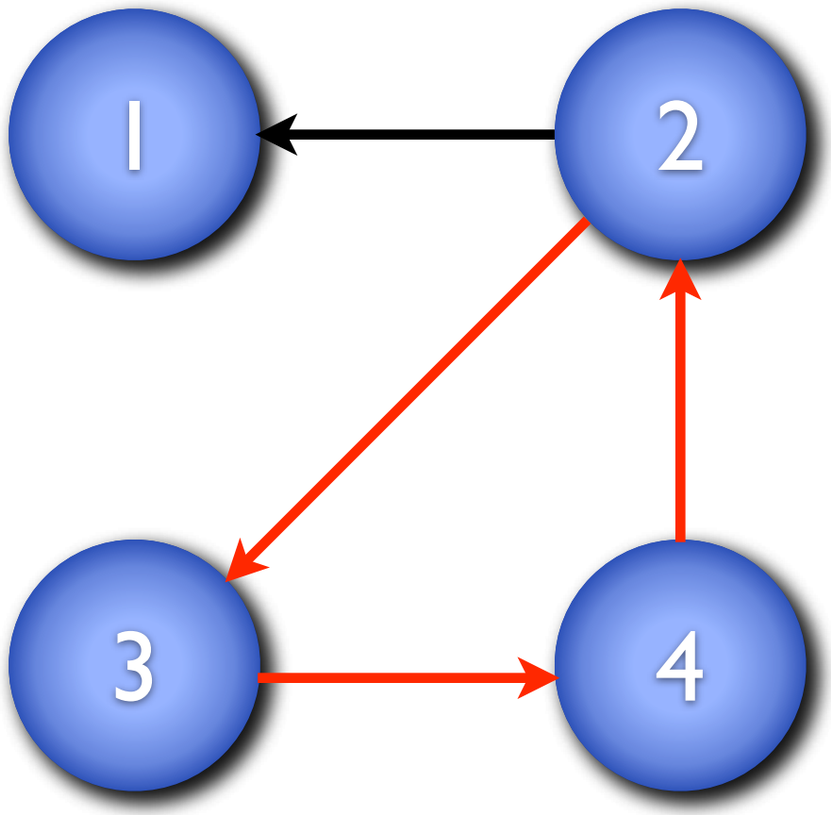
DAG

- DAG (directed acyclic graph) ist ein gerichteter azyklischer Graph, d.h. er enthält keinen gerichteten Kreis (Kreis ist geschlossener Pfad).

DAG

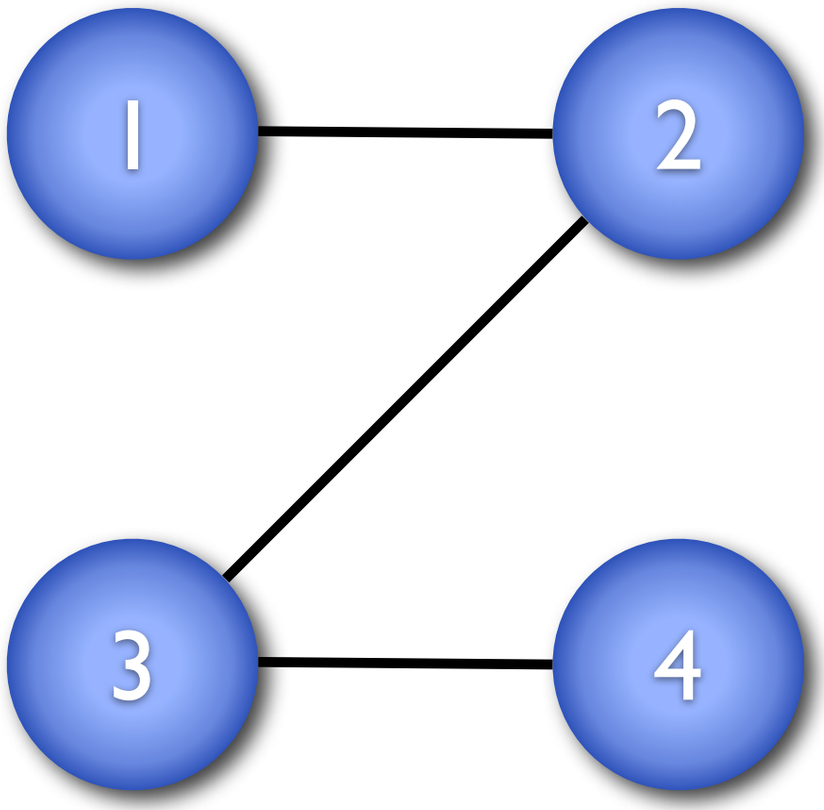


DAG

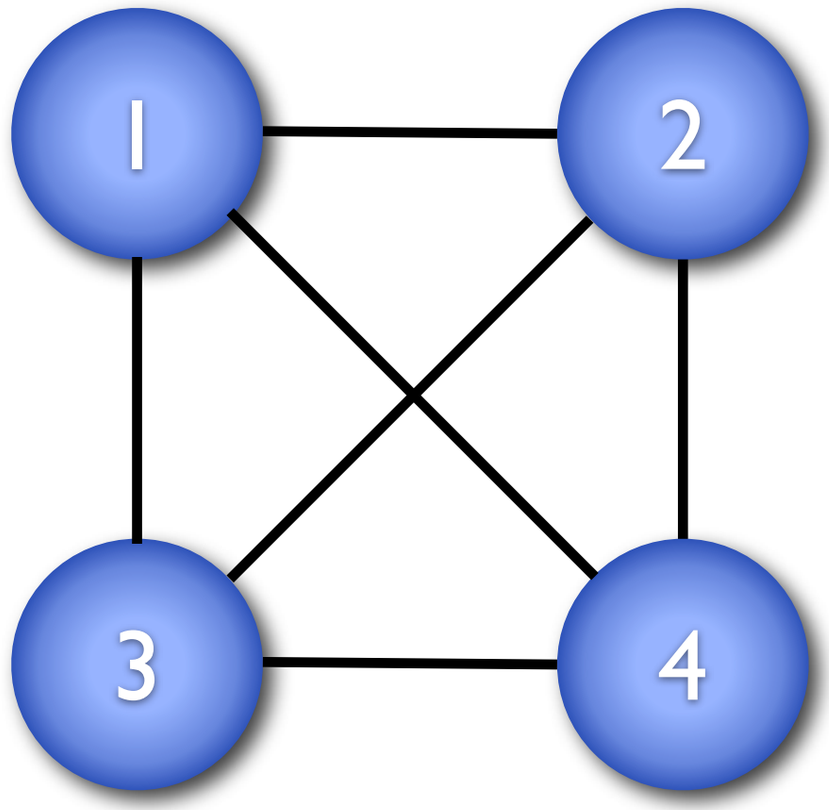


kein DAG

dünn- und dicht besetzte Graphen



dünn besetzter Graph
$|E| \in O(|V|)$



dicht besetzter Graph
$|E| \in O(|V|^2)$

Fragen?

Knoten auf optimalem Pfad

- gegeben ist ein ungerichteter, zusammenhängender Graph und zwei seiner Knoten x und y
- gesucht sind alle Knoten, die auf allen optimalen Pfaden (min. Anz. Kanten) von x nach y liegen

Knoten auf optimalem Pfad

- Eingabe:
  - 1. Zeile: Anz. Knoten N < 7500, Anz. Kanten M < 14000, x, y
    - dann M Zeilen Kanten a b
- Ausgabe:
  - alle gesuchten Knoten (aufsteigend sortiert)

Darstellung von Graphen

- häufige Anforderungen an eine Graph-Datenstruktur:

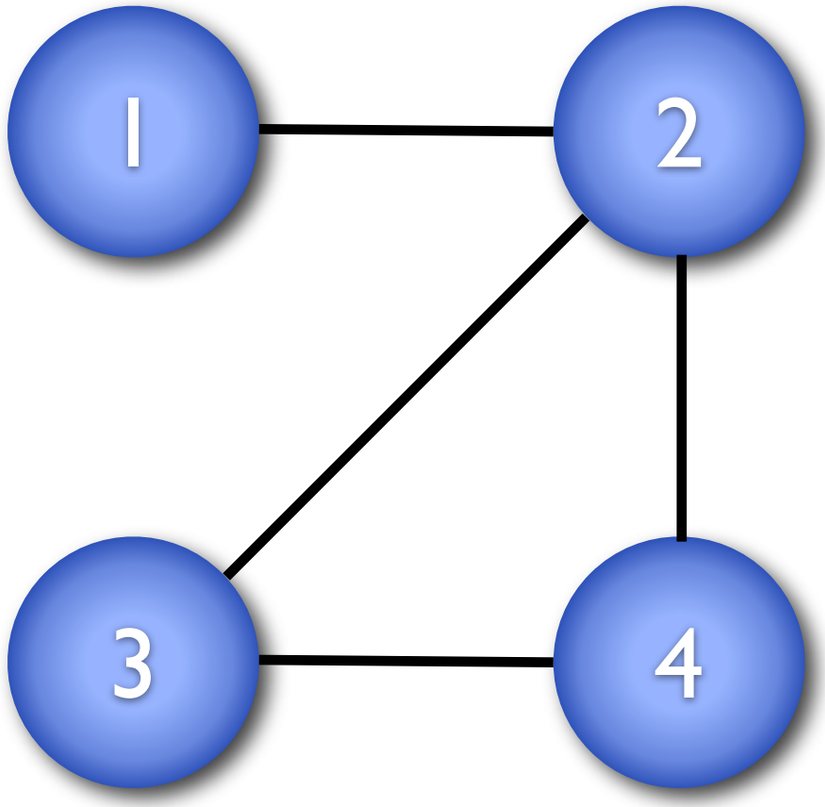
  Kante einfügen/ löschen

  Kante finden (Test ob Kante in Graph)

  Nachbarknoten finden

Adjazenzmatrix - Konzept

- |V| x |V|-Matrix A = (a_{ij}) mit
  a_{ij} = 1 falls (i, j) ∈ E, 0 sonst



	1	2	3	4
1	0	1	0	0
2	1	0	1	1
3	0	1	0	1
4	0	1	1	0

Adjazenzmatrix - Implementierung

- einlesen in Adjazenzmatrix

Adjazenzmatrix - Analyse

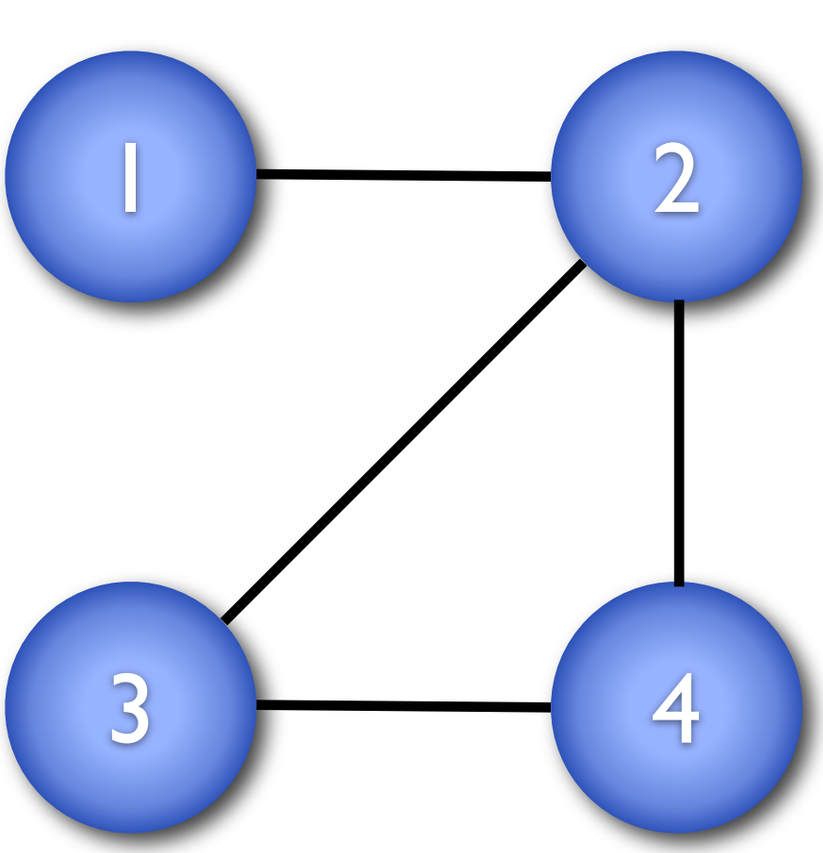
- Laufzeiten:
  - Einfügen/ Löschen
    O (1)
  - Finden/ Testen O (1)
  - Nachbarn finden
    O (|V|)
- Speicher:
  - O ($|V|^2$) bei gerichtetem Graph
  - O ($|V|^2$ / 2) bei ungerichtetem Graph

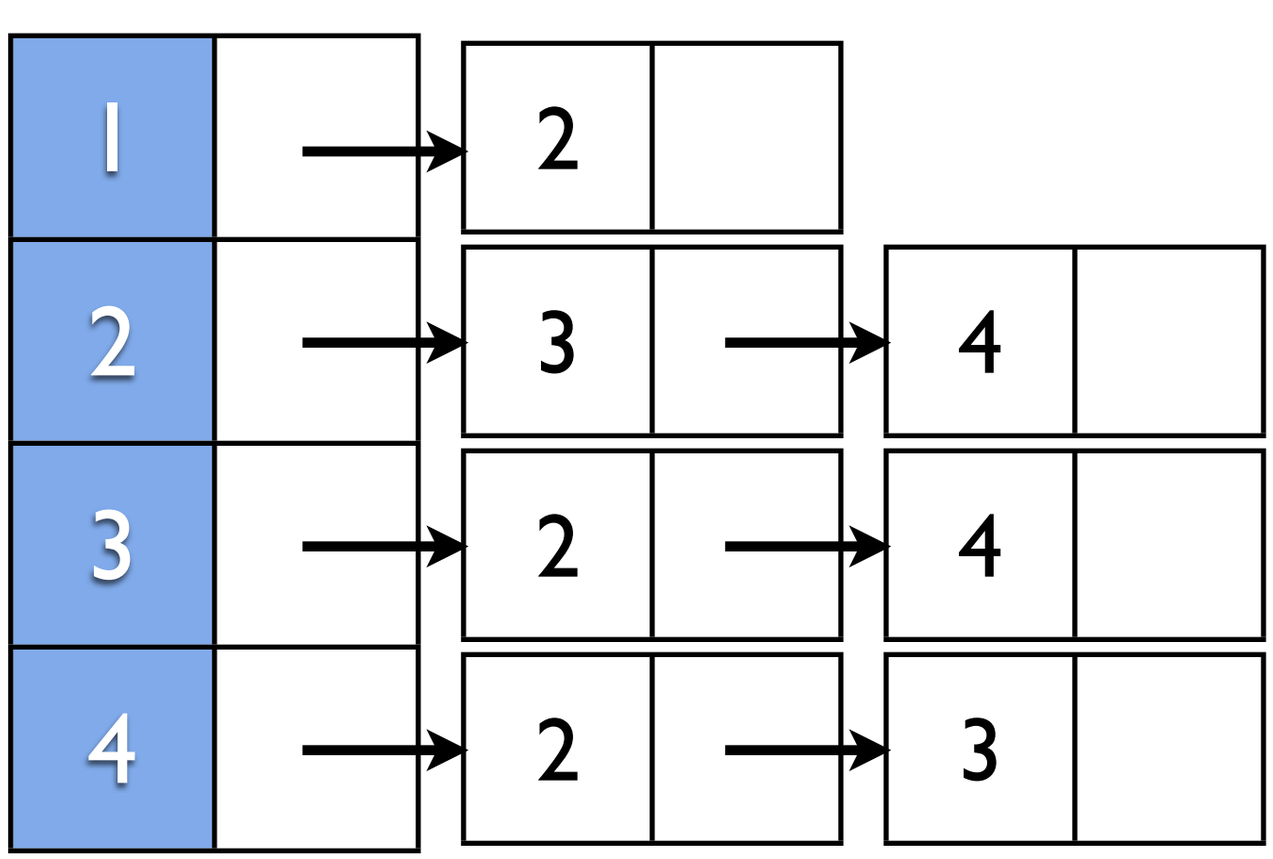
Adjazenzmatrix - Analyse

- Verwendung:
  - bei relativ kleinen Graphen wegen Einfachheit
  - bei dicht besetzten Graphen
  - wenn Test ob Kante in Graph effizient implementiert werden soll

Adjazenzlisten - Konzept

- Feld der Länge |V|, das für jeden Knoten eine Liste seiner Nachbarn enthält





Adjazenzlisten - Implementierung

- einlesen in Adjazenzlisten

Adjazenzlisten - Analyse

- Laufzeiten:
  - Einfügen O (1)
  - Löschen O (d)
  - Finden/ Testen O (d)
  - Nachbarn finden
    O (1)
  - mit d max. Grad
- Speicher:
  - O (|V| + |E|) bei gerichtetem Graph
  - O (|V| + 2|E|) bei ungerichtetem Graph

Adjazenzlisten - Analyse

- Verwendung:
  - bei dünn besetzten Graphen
  - wenn schnell alle Nachbarn eines Knotens gefunden werden müssen

Inzidenzlisten

- jeder Knoten hat eine Liste von Pointern auf seine inzidenten Kanten
- Kanteninformation kann leicht modifiziert werden, da für jede Kante nur eine Instanz existiert

Datenstruktur für Aufgabe

welche Datenstruktur brauchen wir für die Beispielaufgabe?

Fragen?

Graphen „Durchsuchen“

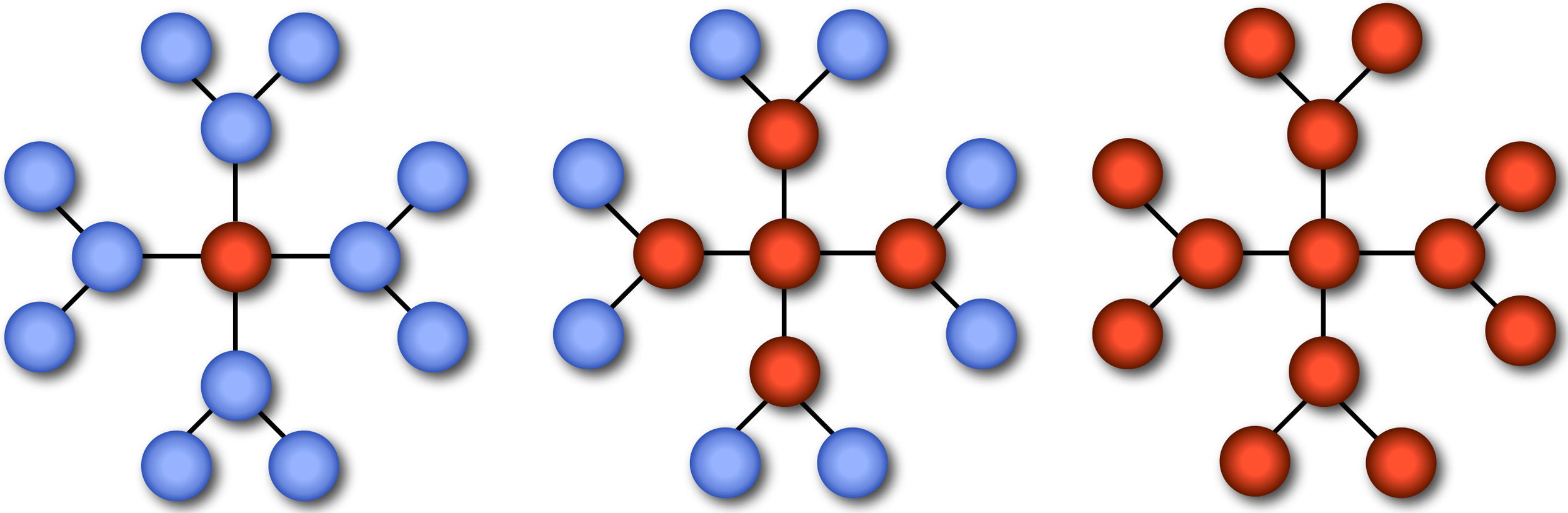
- auf systematische Weise den Kanten folgen und die Knoten des Graphen besuchen

- Ziel: Eigenschaften über die Struktur des Graphen aufdecken

- Breiten- und Tiefensuche sind Grundlage vieler Graphenalgorithmen

Breitensuche - Konzept

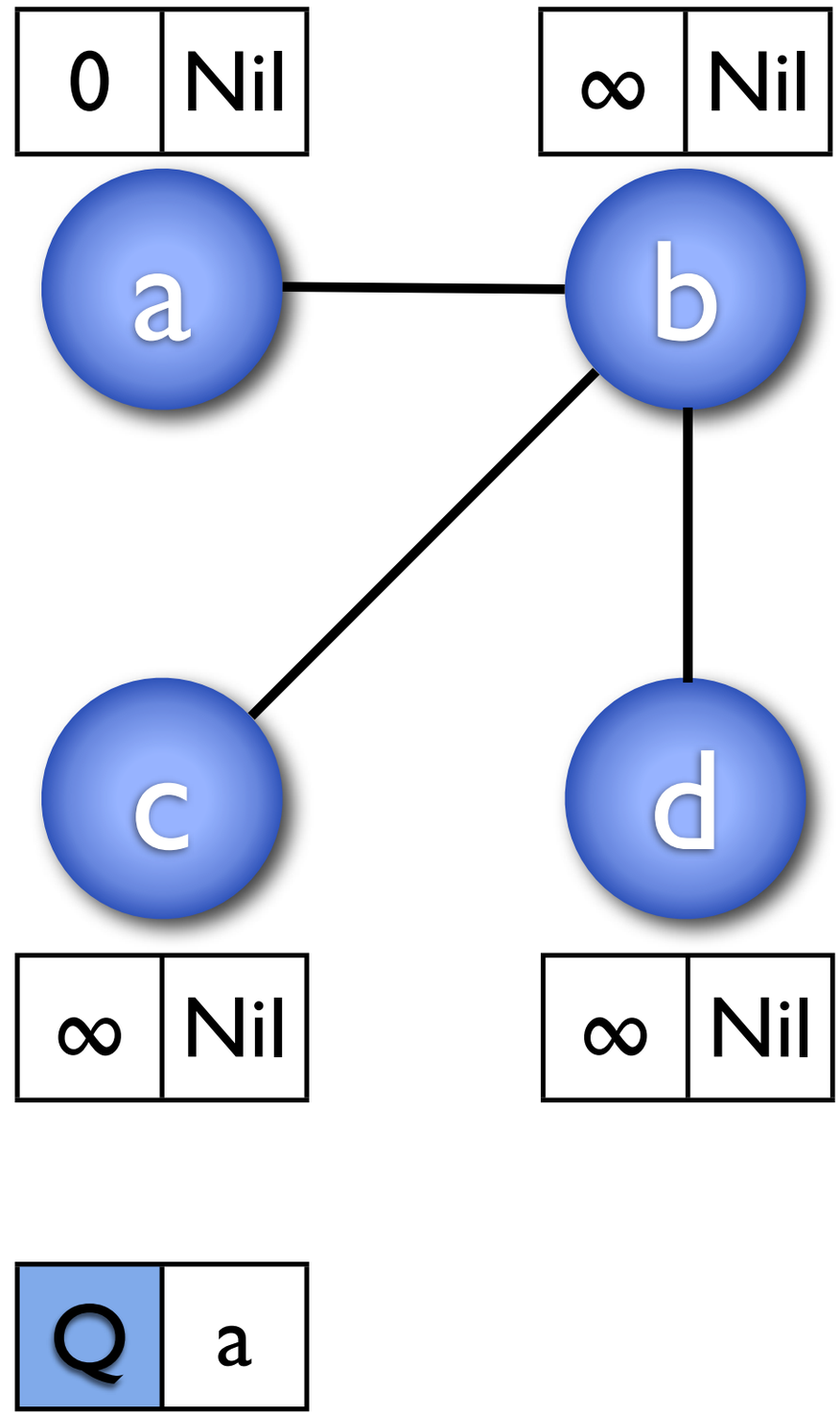
- besuche alle Knoten, die den Abstand (Anz. Kanten) k vom Startknoten haben, bevor irgendein Knoten mit Abstand k+1 besucht wird



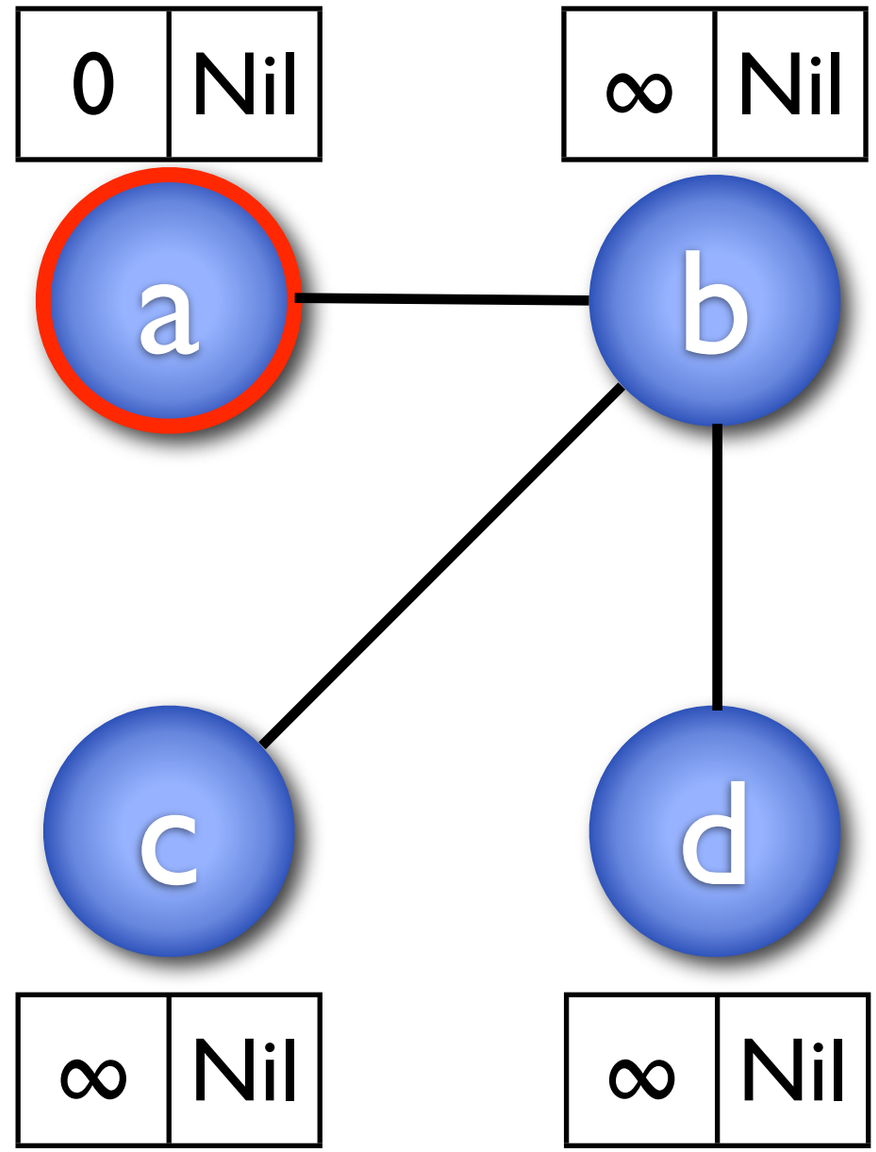
Breitensuche (BFS) - Implementierung

- Eingabegraph G = (V, E) (in Adjazenzlisten-Darstellung) und Startknoten s
- zusätzliche Datenstrukturen:
  - Vorgänger-Feld `pred[]`
  - Abstand zum Startknoten `d[]`
  - FIFO - Queue `Q`

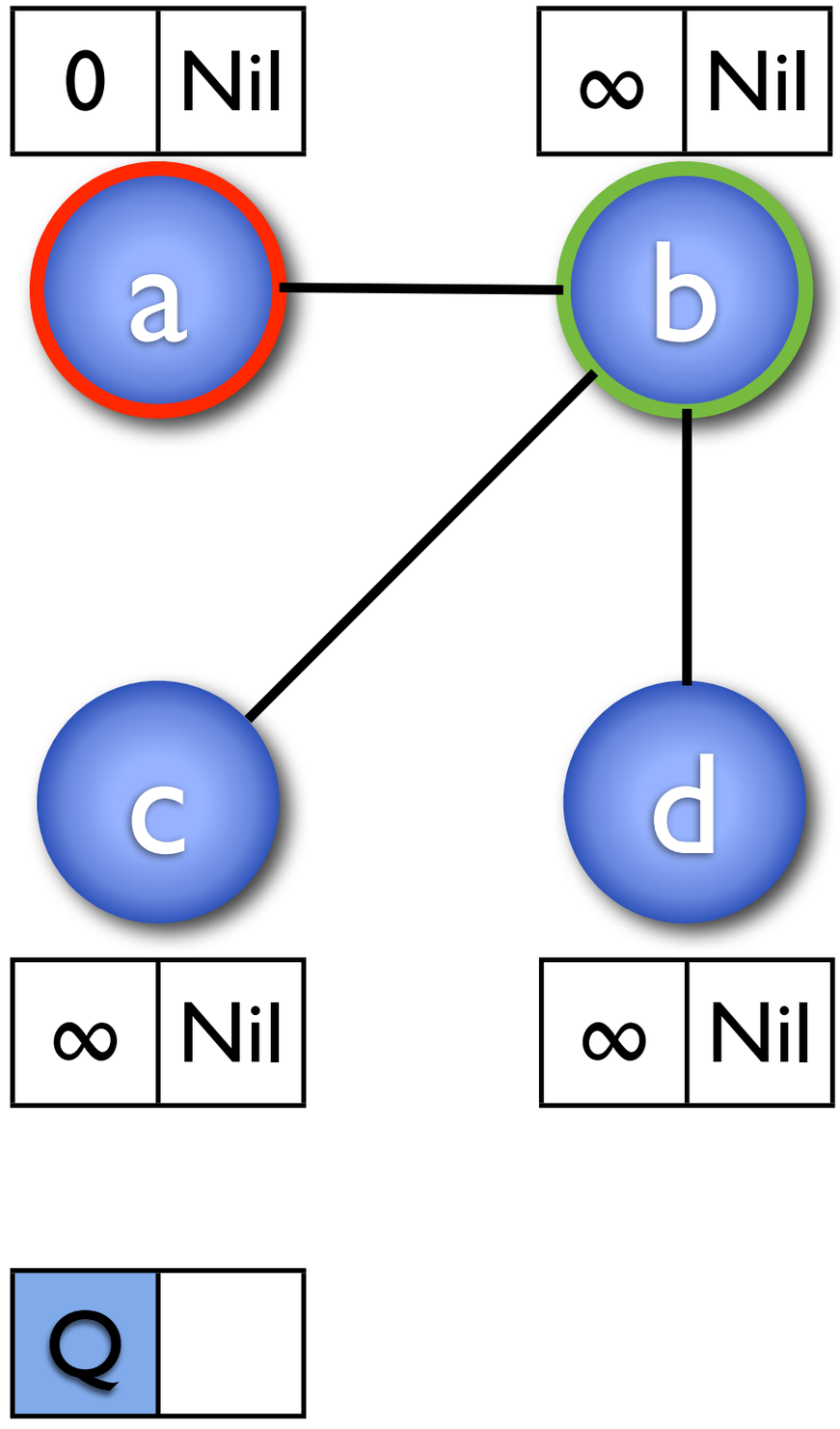
```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
         then d[v] ← d[u] + 1
              pred[v] ← u
              Enqueue(Q, v)
```



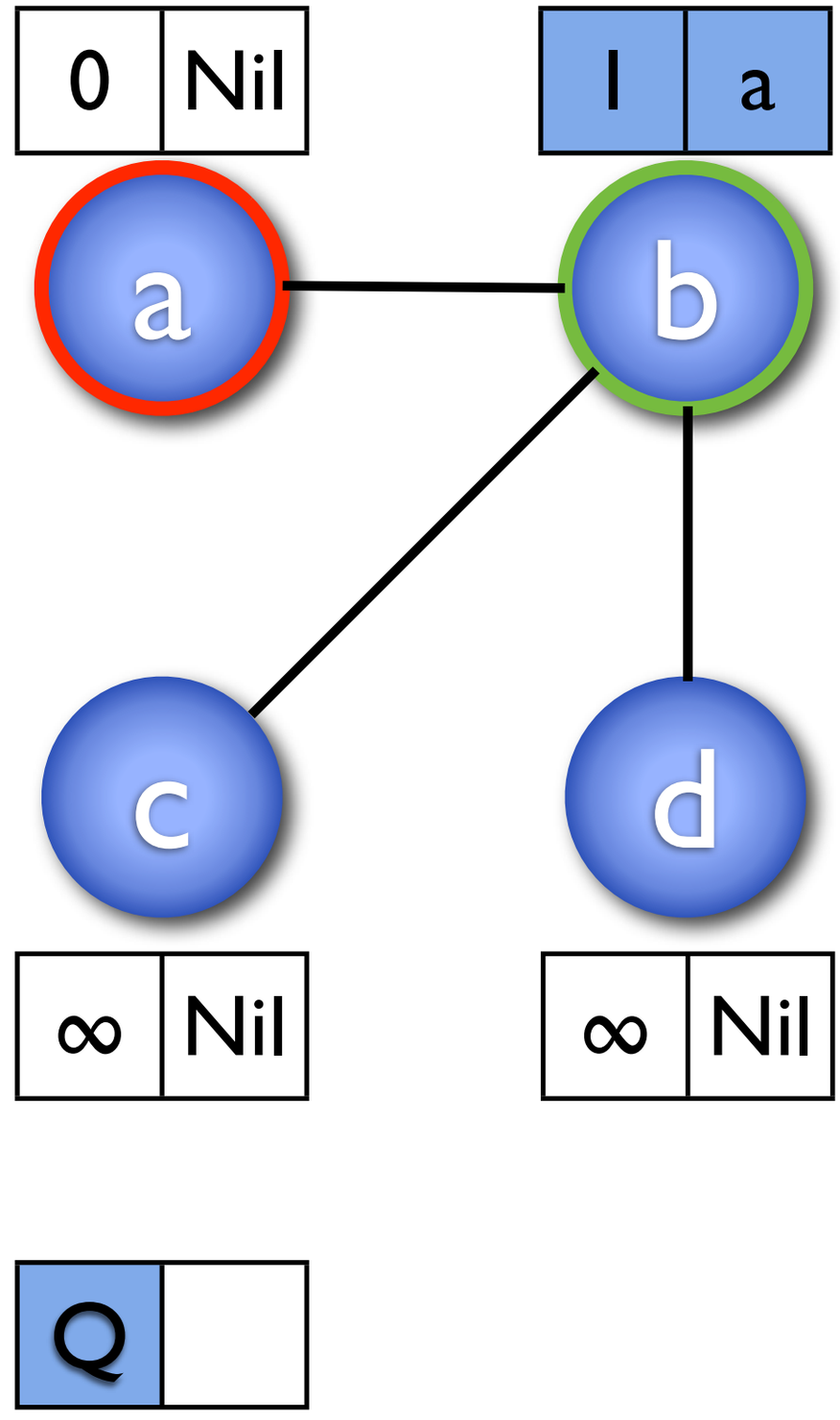
```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
         then d[v] ← d[u] + 1
              pred[v] ← u
              Enqueue(Q, v)
```



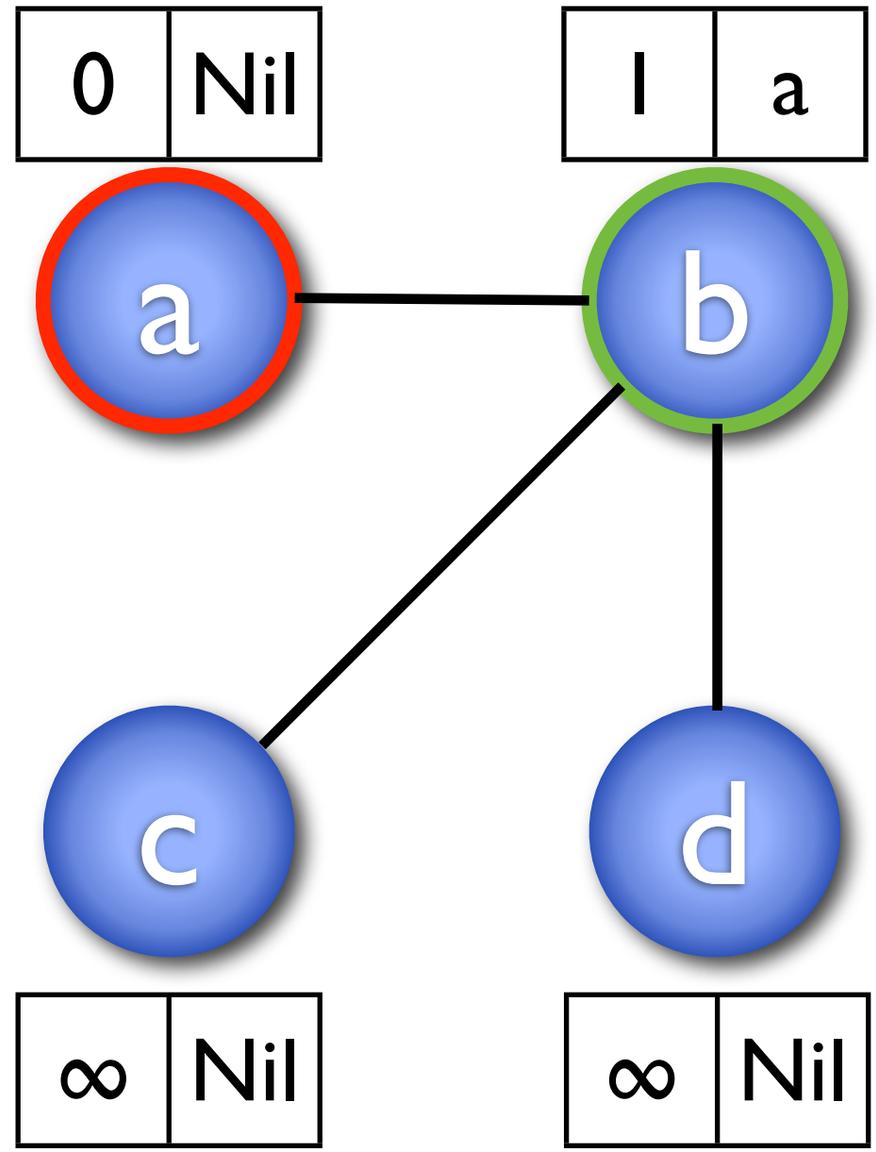
```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
         then d[v] ← d[u] + 1
              pred[v] ← u
              Enqueue(Q, v)
```



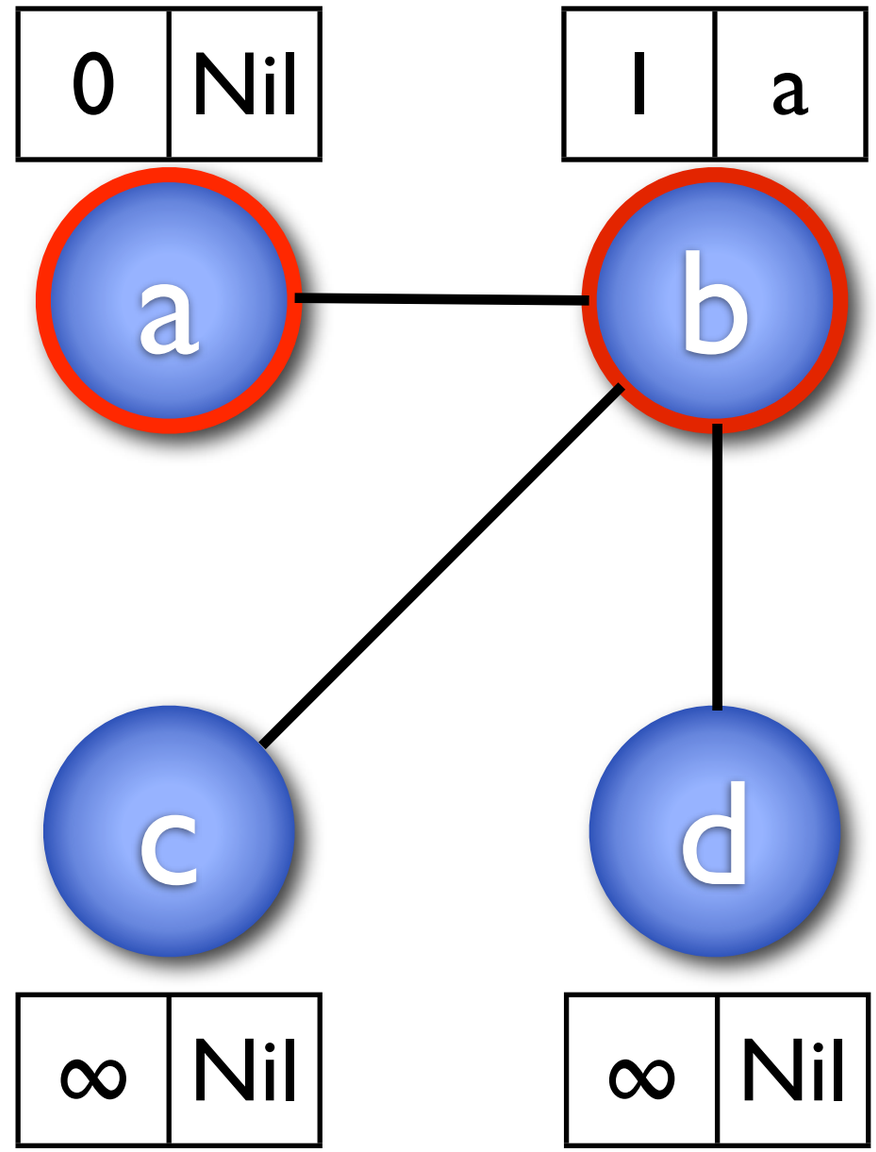
```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
           then d[v] ← d[u] + 1
                pred[v] ← u
                Enqueue(Q, v)
```



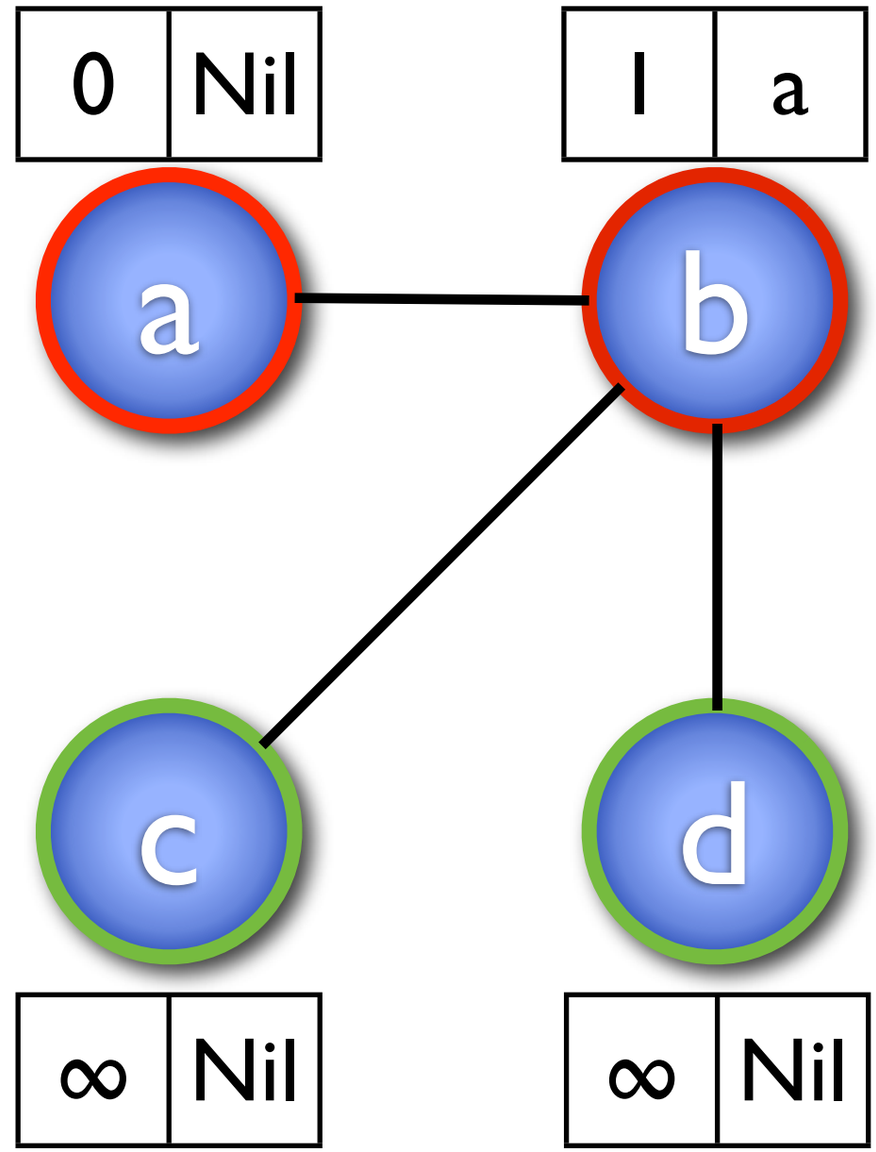
```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
          then d[v] ← d[u] + 1
               pred[v] ← u
               Enqueue(Q, v)
```



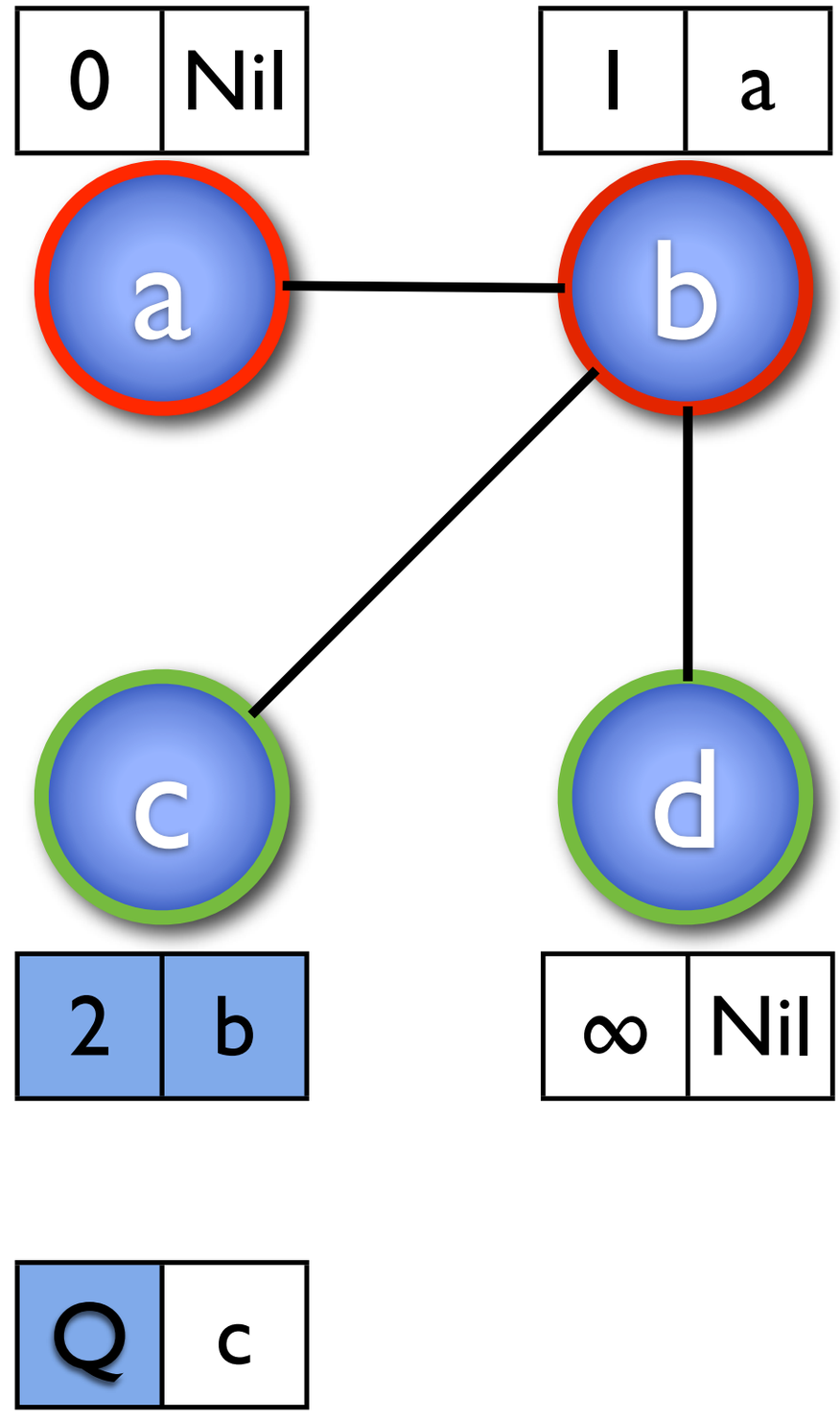
```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
         then d[v] ← d[u] + 1
              pred[v] ← u
              Enqueue(Q, v)
```



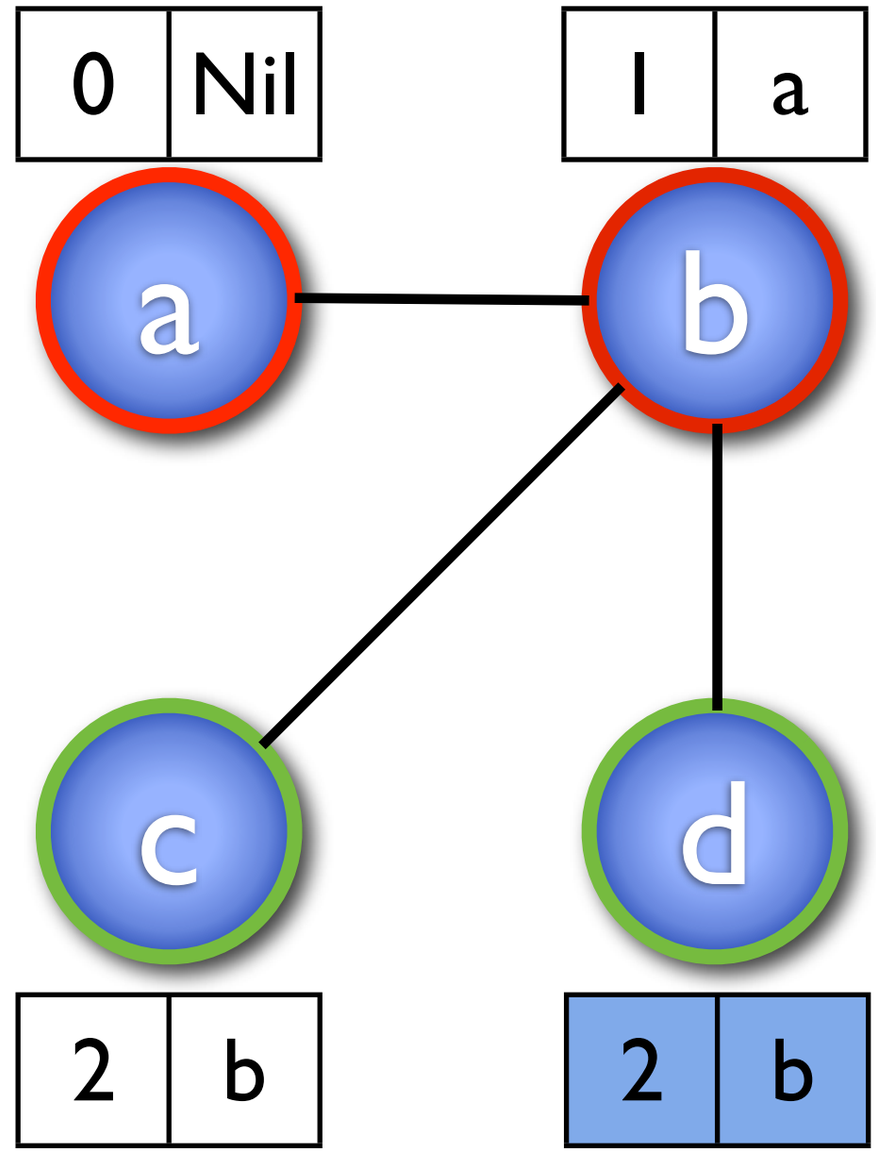
```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
         then d[v] ← d[u] + 1
              pred[v] ← u
              Enqueue(Q, v)
```



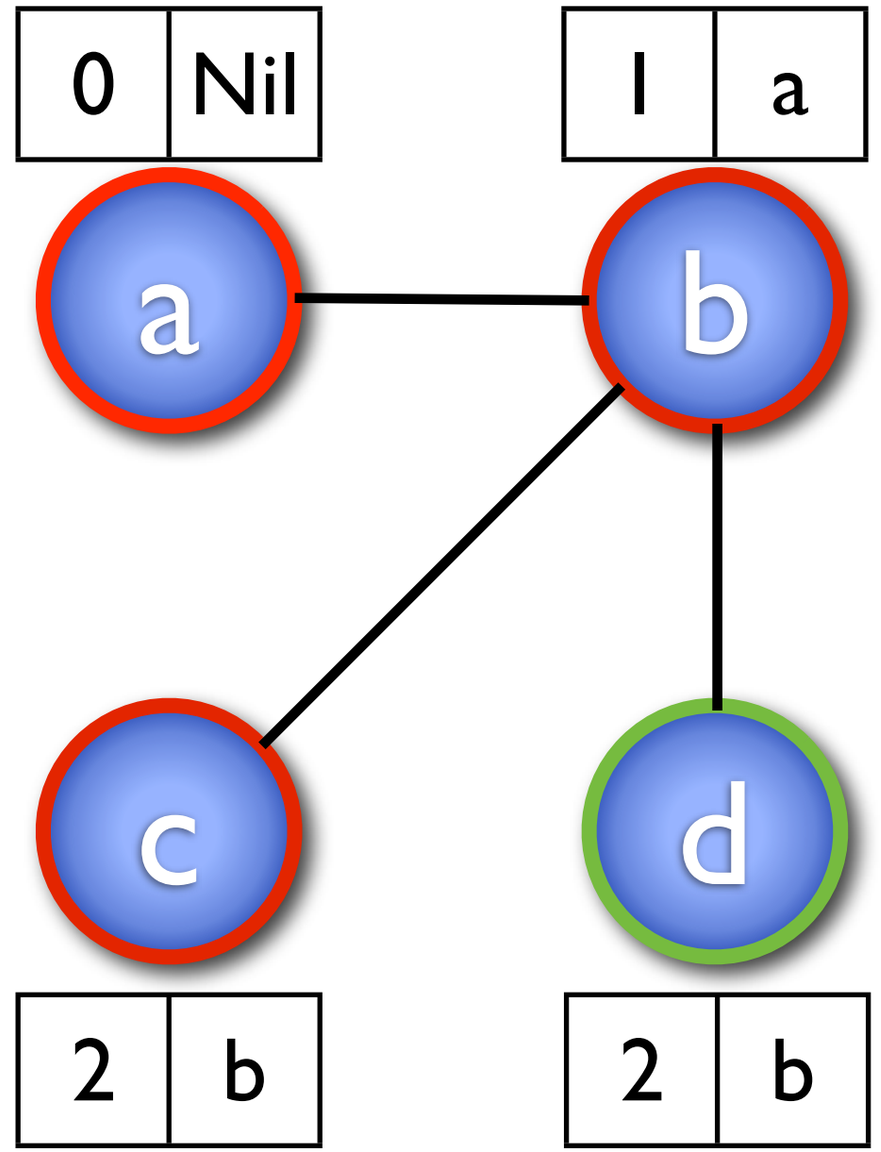
```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
           then d[v] ← d[u] + 1
                pred[v] ← u
                Enqueue(Q, v)
```



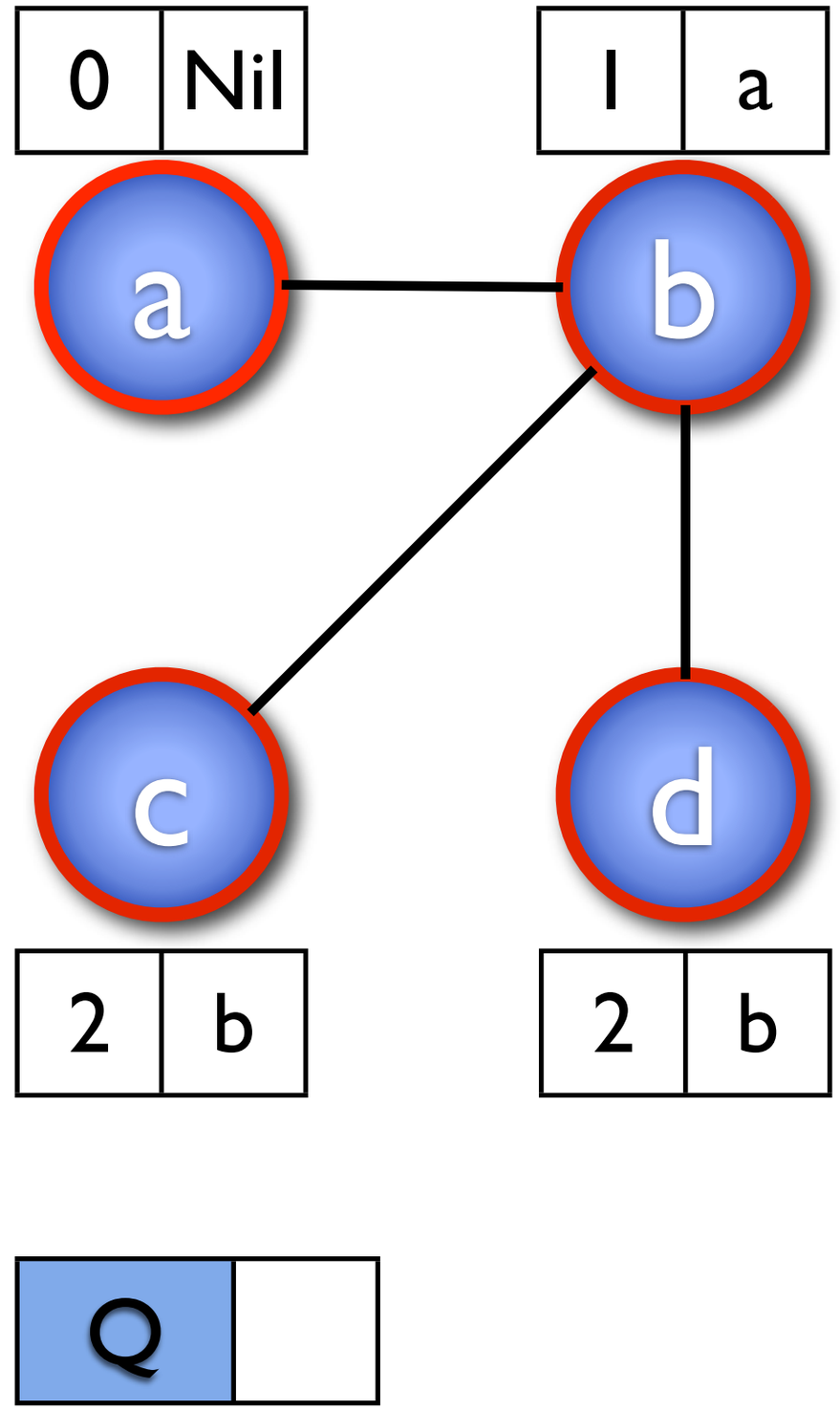
```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
           then d[v] ← d[u] + 1
                pred[v] ← u
                Enqueue(Q, v)
```



```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
         then d[v] ← d[u] + 1
              pred[v] ← u
              Enqueue(Q, v)
```



```
BFS (G,s)
for alle Knoten u ∈ V[G] - {s}
  do d[u] ← ∞
     pred[u] ← Nil
d[s] ← 0
pred[s] ← Nil
Q ← ∅
Enqueue(Q, s)
while Q ≠ ∅
  do u ← Dequeue(Q)
    for alle v ∈ Adj[u]
      do if d[v] = ∞
         then d[v] ← d[u] + 1
              pred[v] ← u
              Enqueue(Q, v)
```

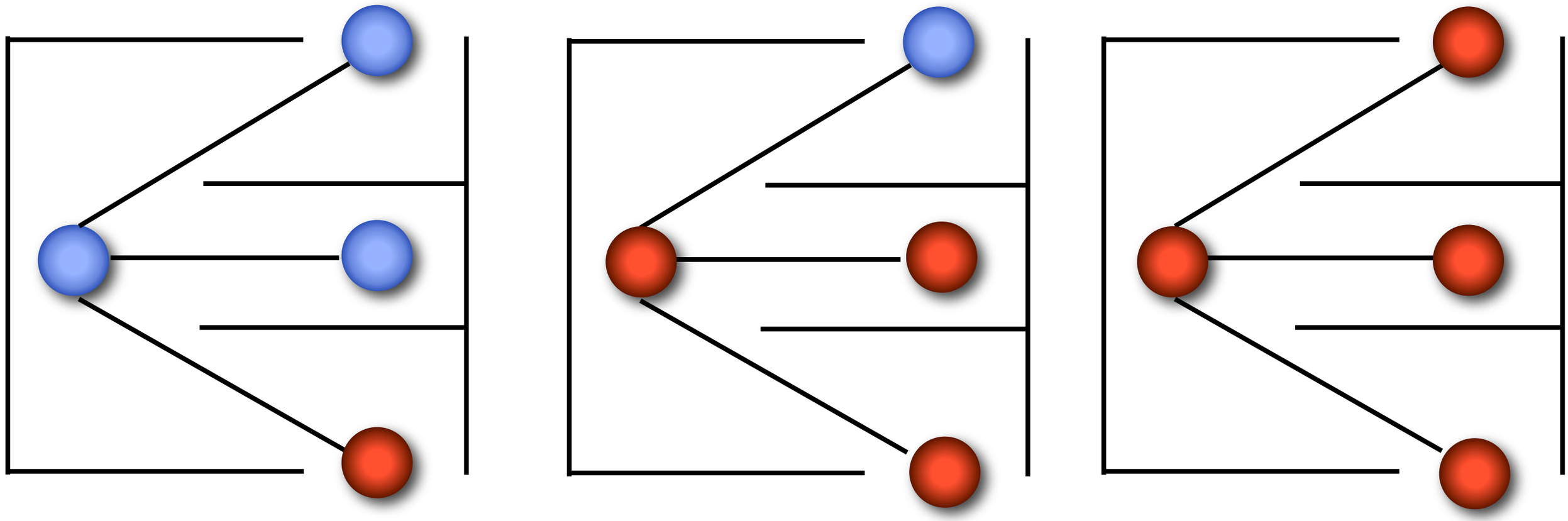


Breitensuche - Analyse

- Laufzeit: O (|V| + |E|)
- Ergebnis:
  - alle vom Startknoten erreichbaren Knoten
    v `d[v]` $\neq \infty$
  - Kürzeste Pfade (Anz. Kanten) jedes Knoten **v** vom Startknoten `d[v]`
  - Vorgängerteilgraph über `pred[v]`

Tiefensuche - Konzept

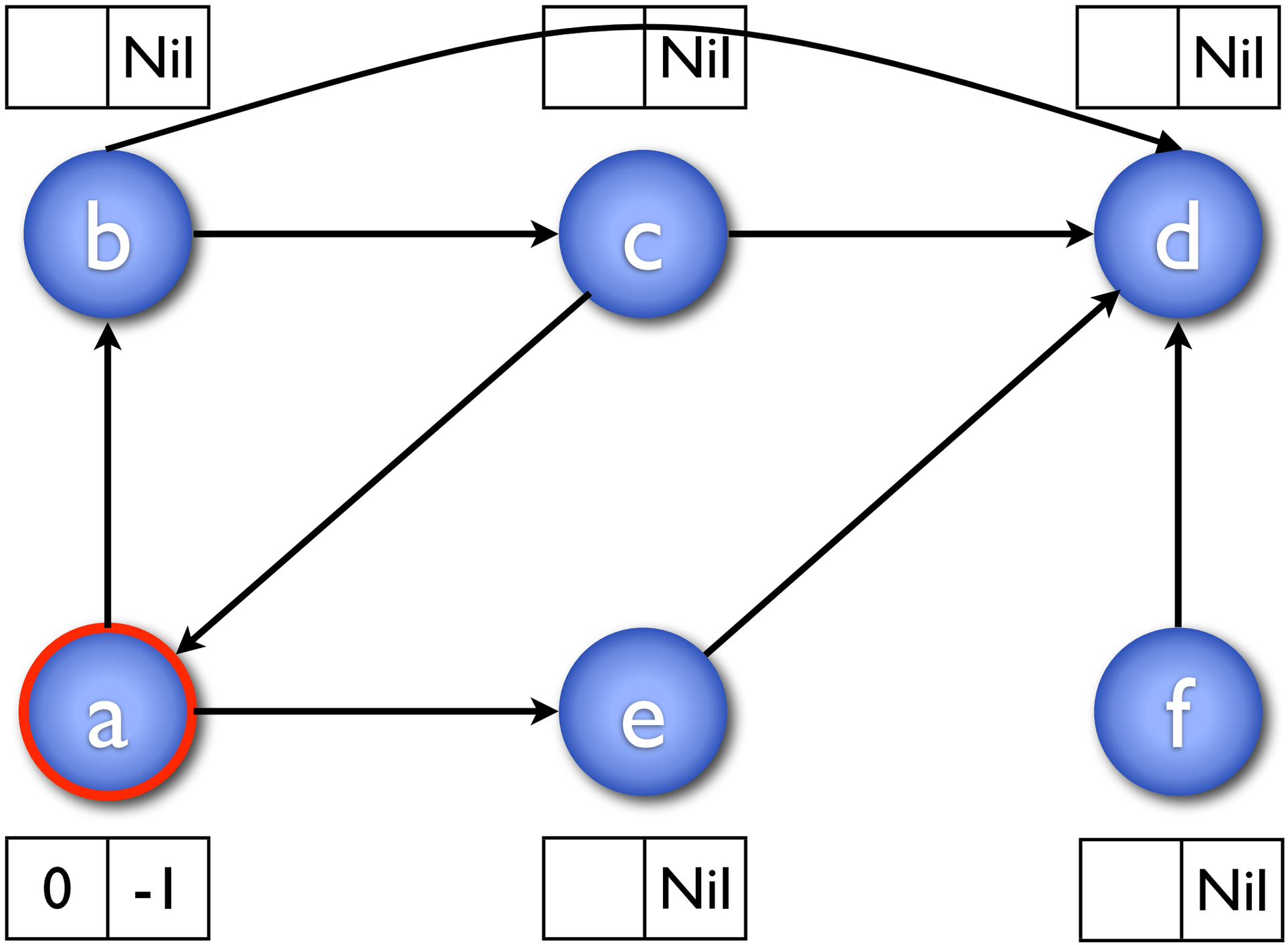
- gehe so tief einen Pfad in den Graph hinein, bis es nicht mehr weiter geht, erst dann drehe um

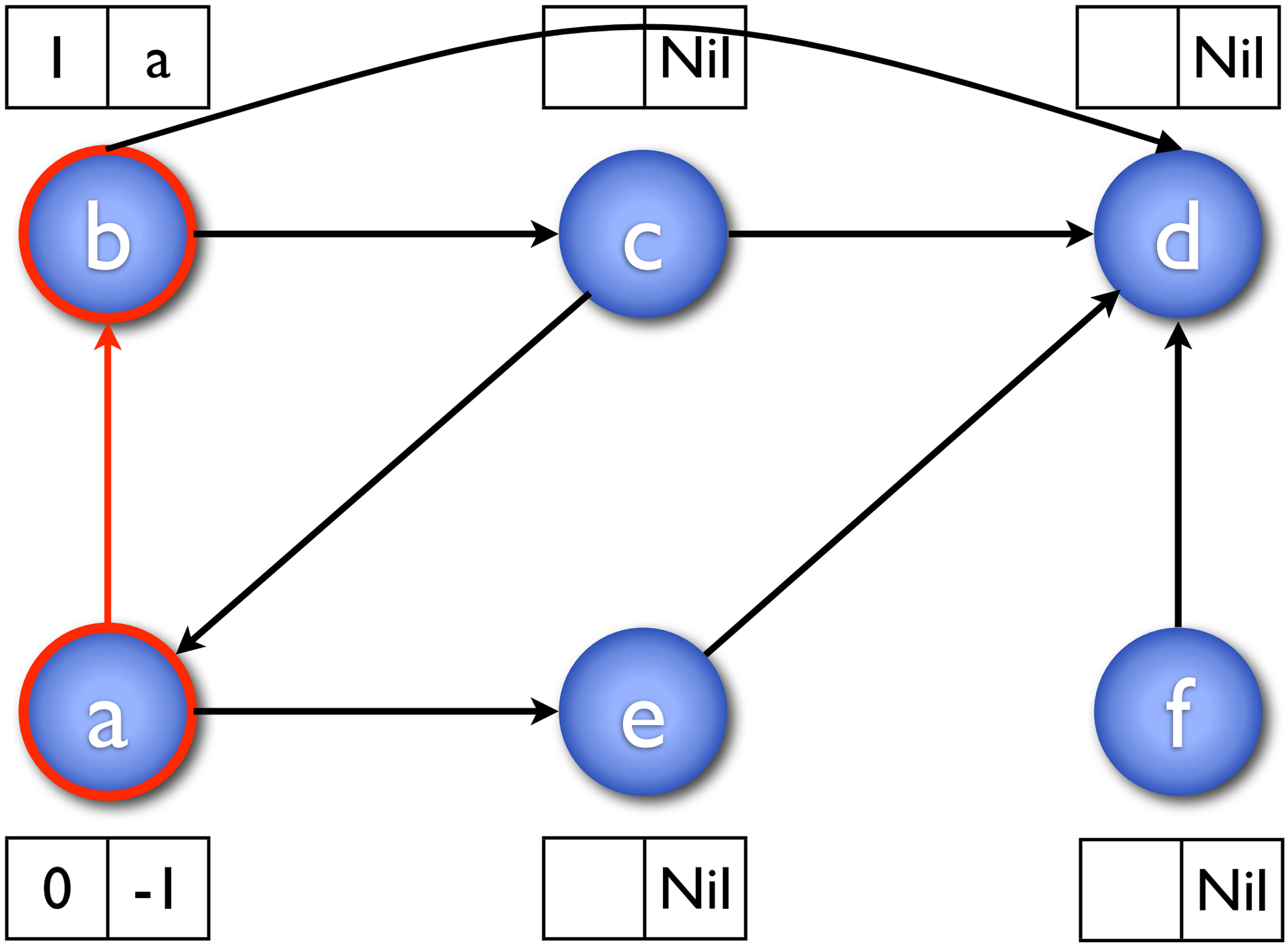


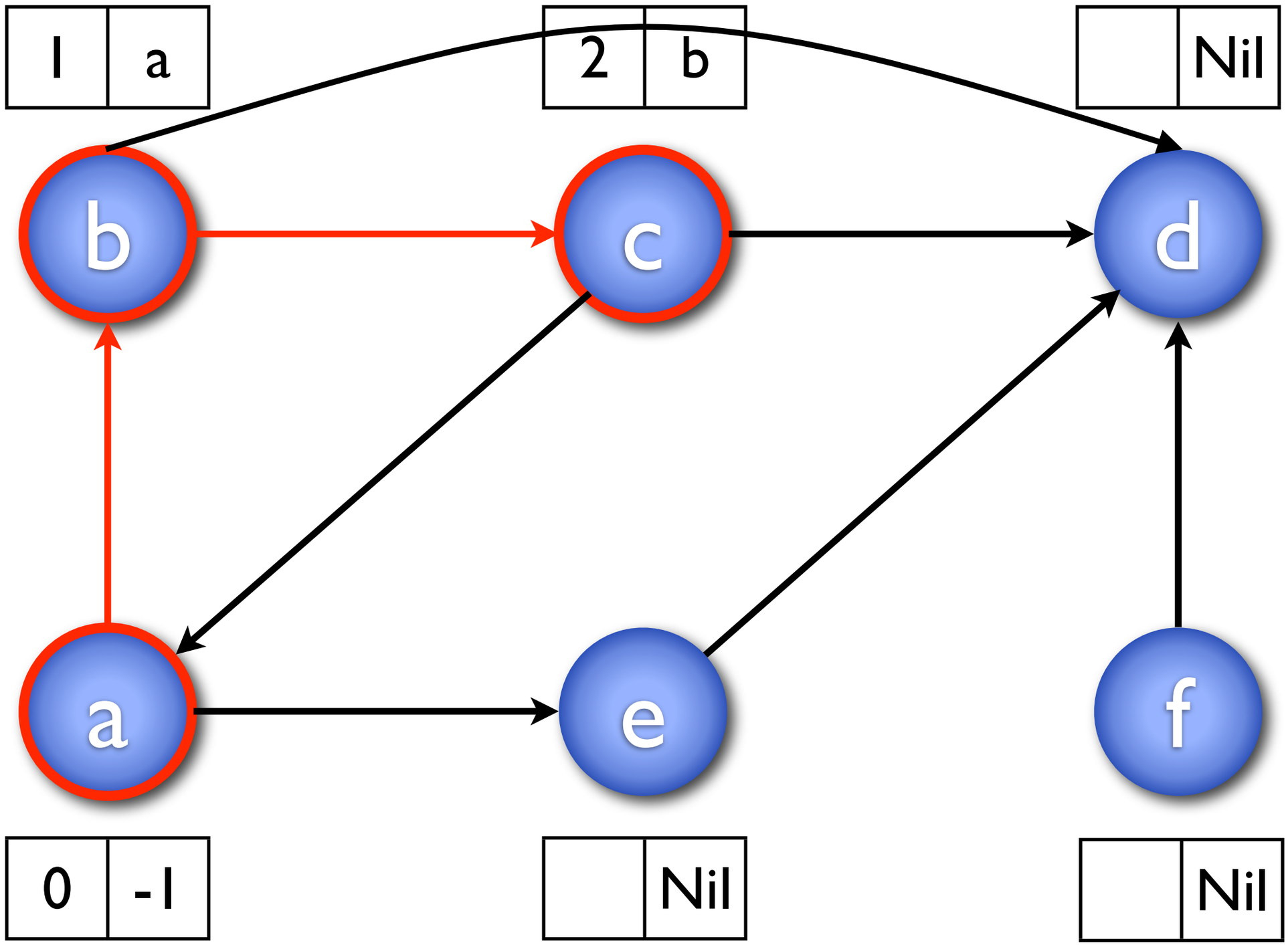
Tiefensuche (DFS) - Implementierung

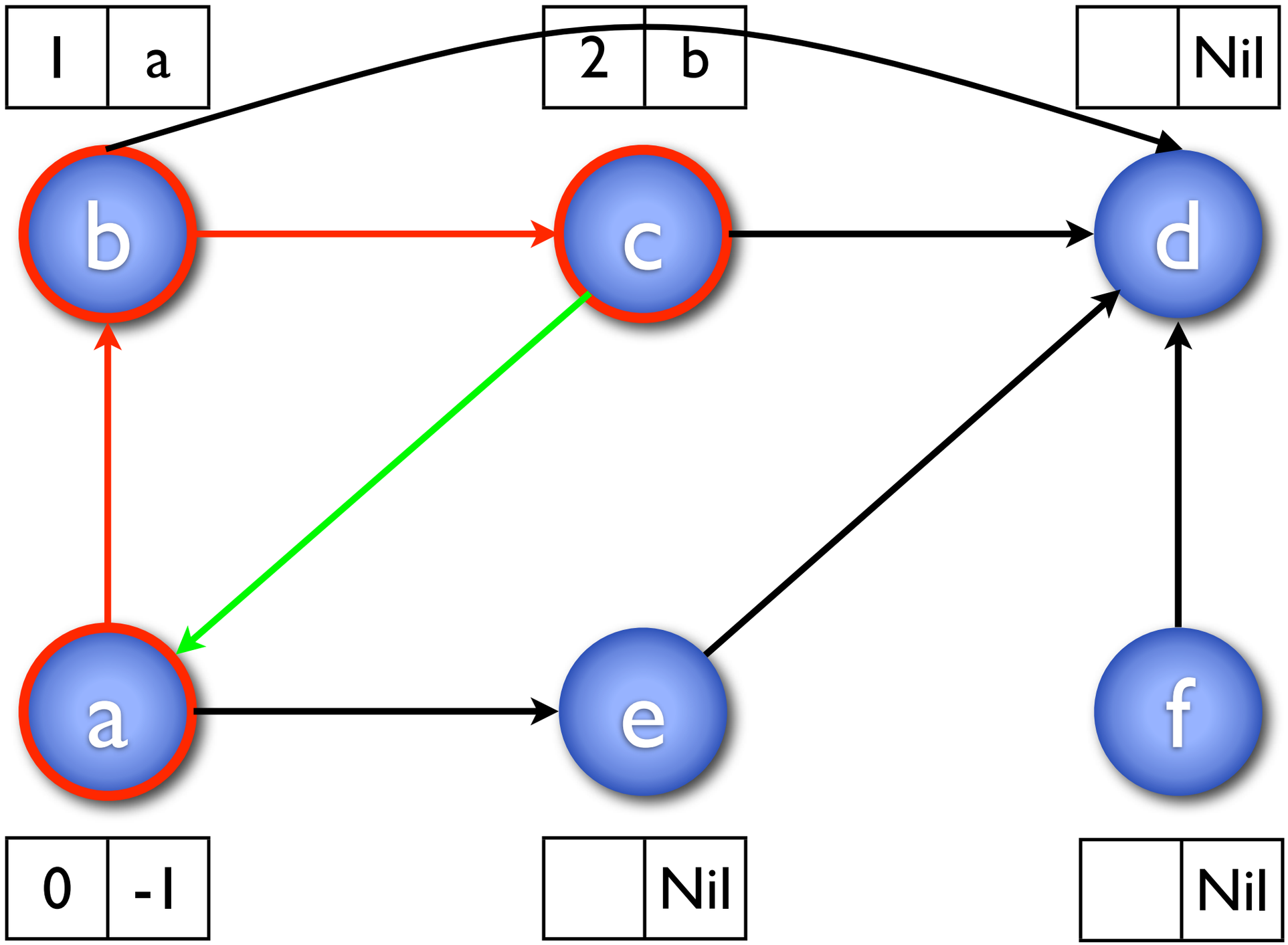
- Eingabegraph G = (V, E) (in Adjazenzlisten-Darstellung)
- zusätzliche Datenstrukturen:
  - Vorgänger-Feld `pred[]`
  - Besucht-Feld `visited[]`
  - Entdeckungszeit `d[]`, Endzeit `f[]`

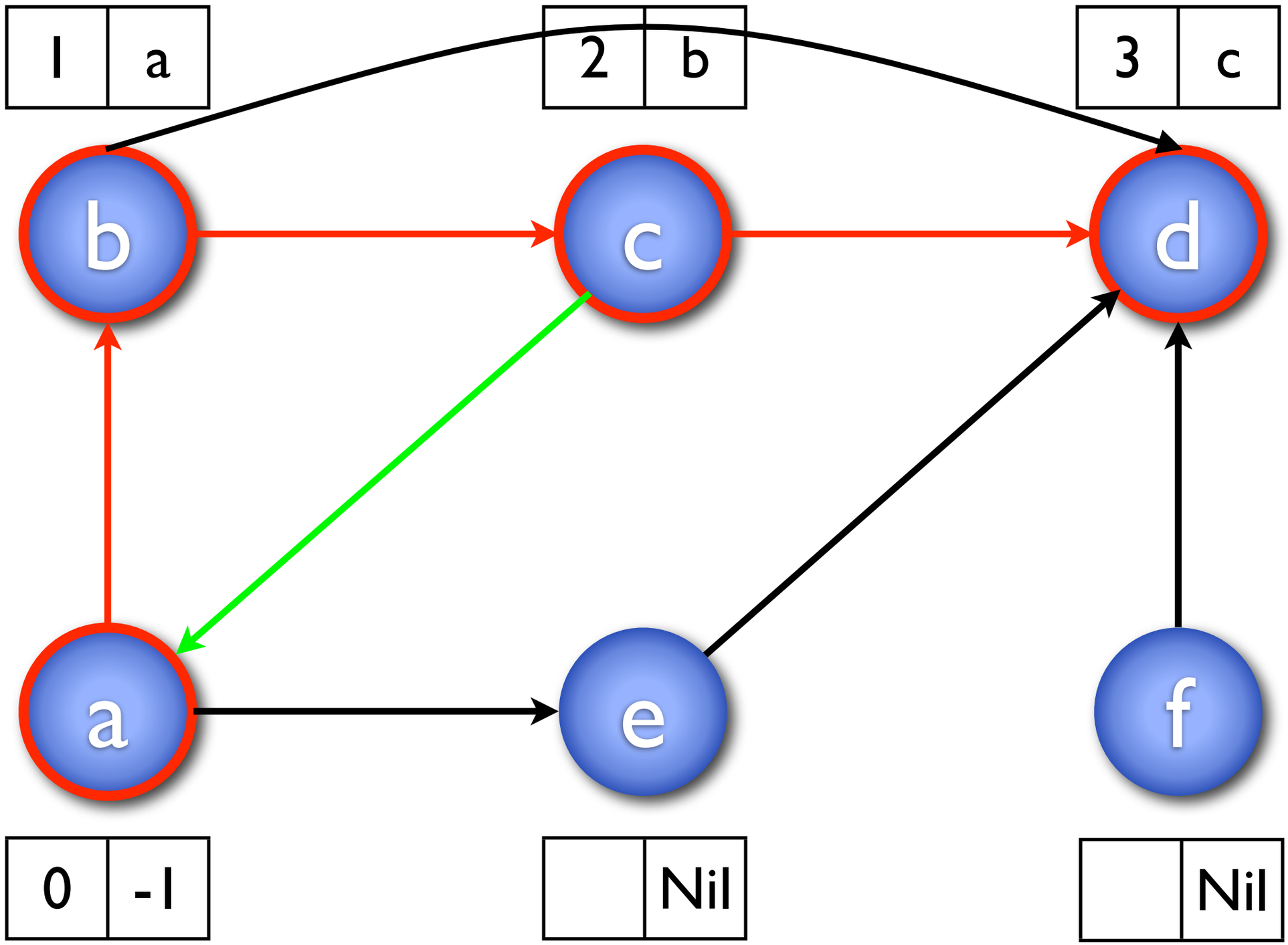
```
DFS(G)
for alle Knoten u ∈ V[G]          // Initialisierung
  do visited[u] ← false
     pred[u] ← Nil
zeit ← 0
for alle Knoten u ∈ V[G]          // für alle unbesuchten
  do if visited[u] = false        //  Knoten rufe
    then pred[u] ← -1             //   DFS-Visit auf
         DFS-Visit(u)


DFS-Visit(u)
visited[u] ← true
d[u] ← time
time ← time + 1
for alle v ∈ Adj[u]
  do if not visited[v]
     then pred[v] ← u
          DFS-Visit(v)
f[u] ← time
time ← time + 1
```

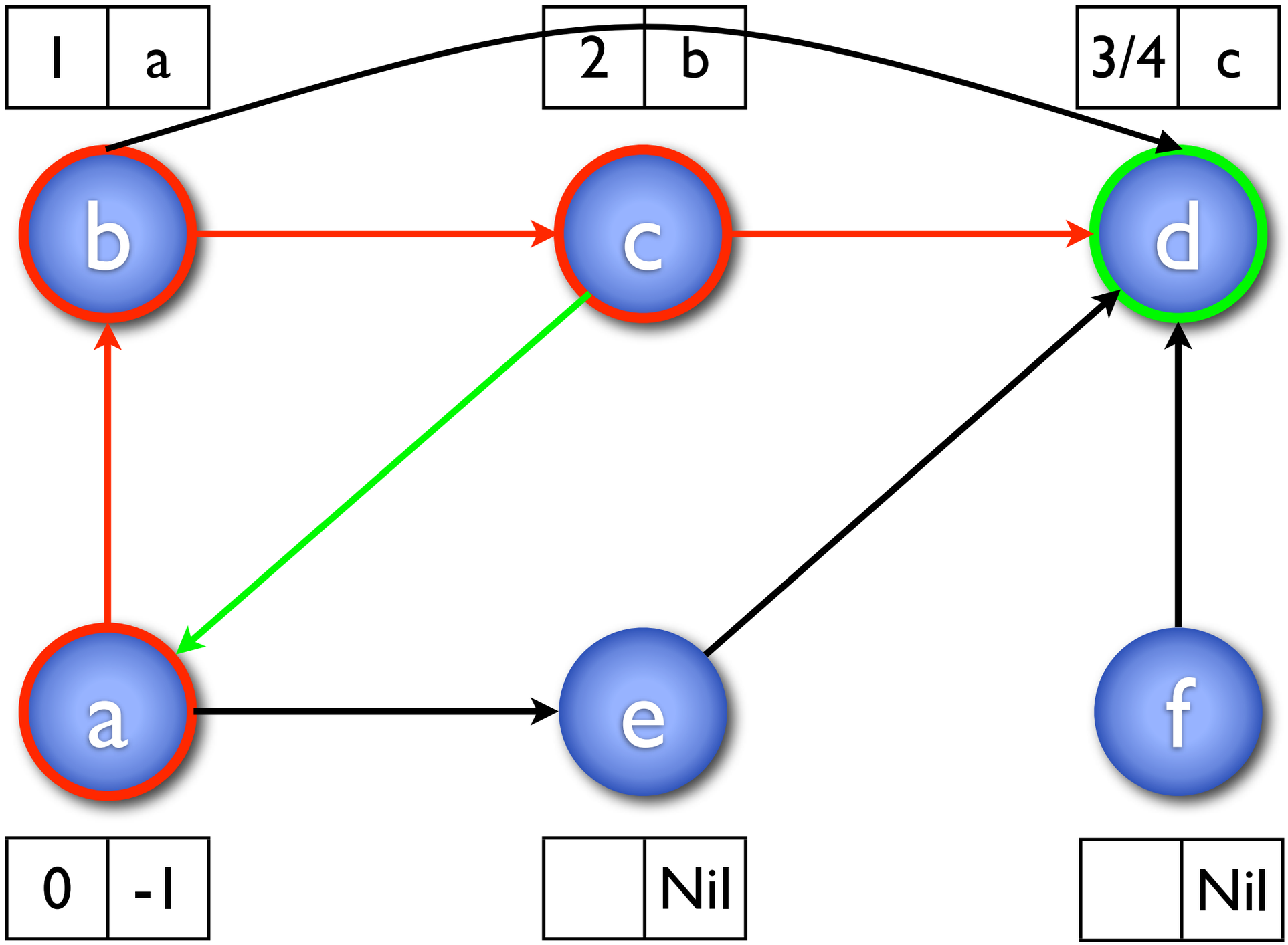

Nil
Nil
Nil
b
c
d
a
e
f
0
-1
Nil
Nil

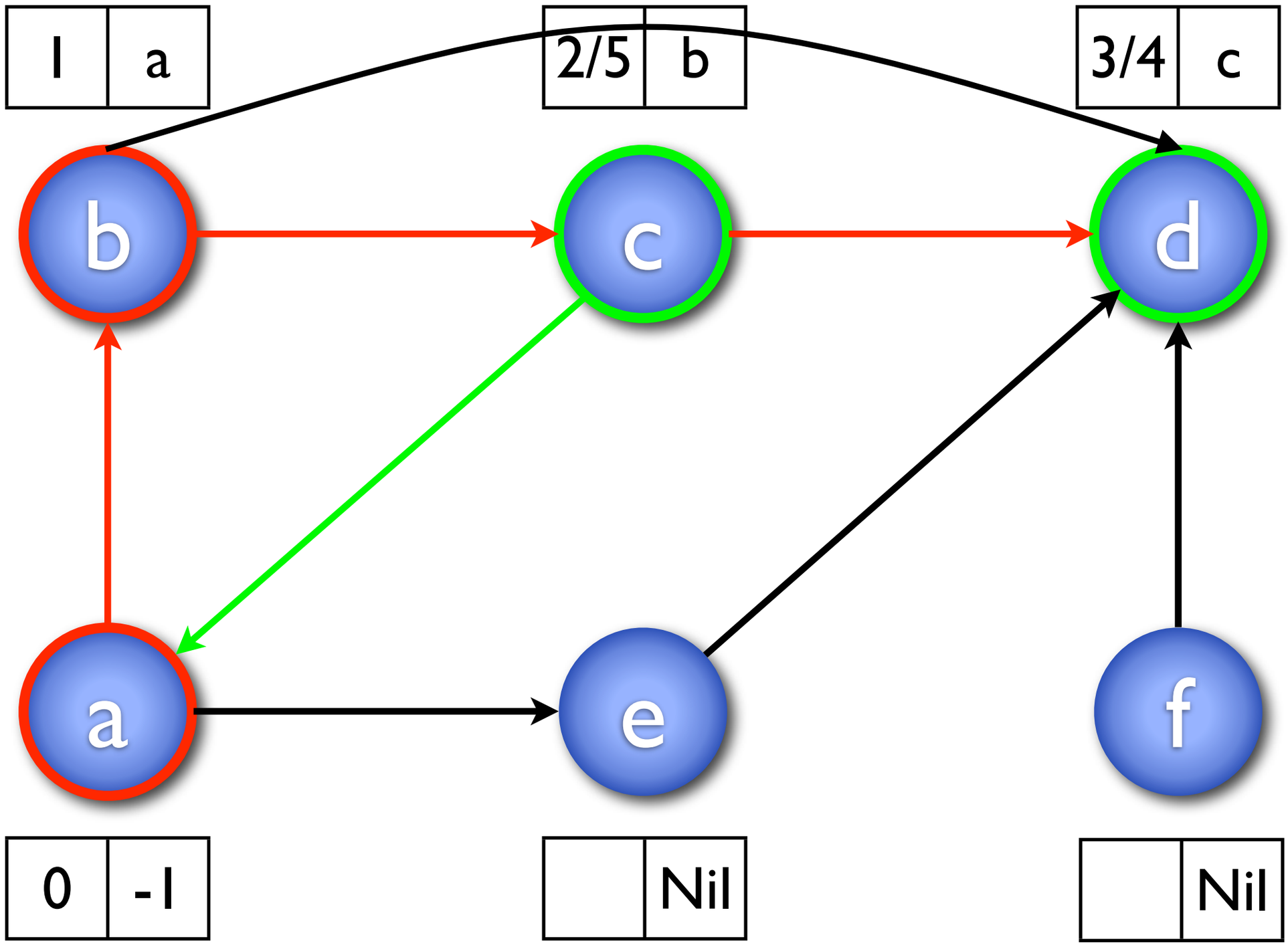

I
a
Nil
Nil
b
c
d
a
e
f
0
-I
Nil
Nil

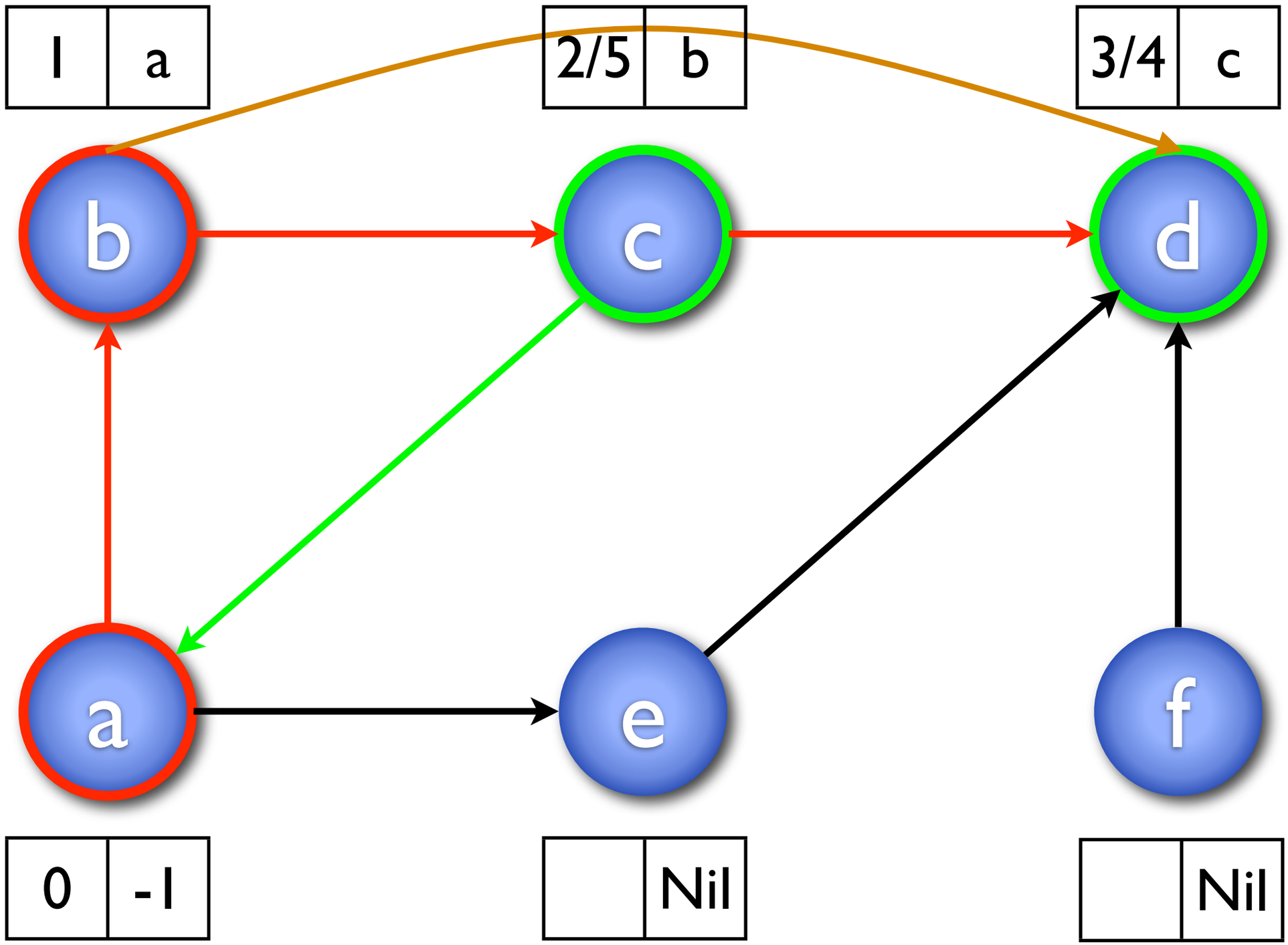

1
a
2
b
Nil
b
c
d
a
e
f
0
-1
Nil
Nil

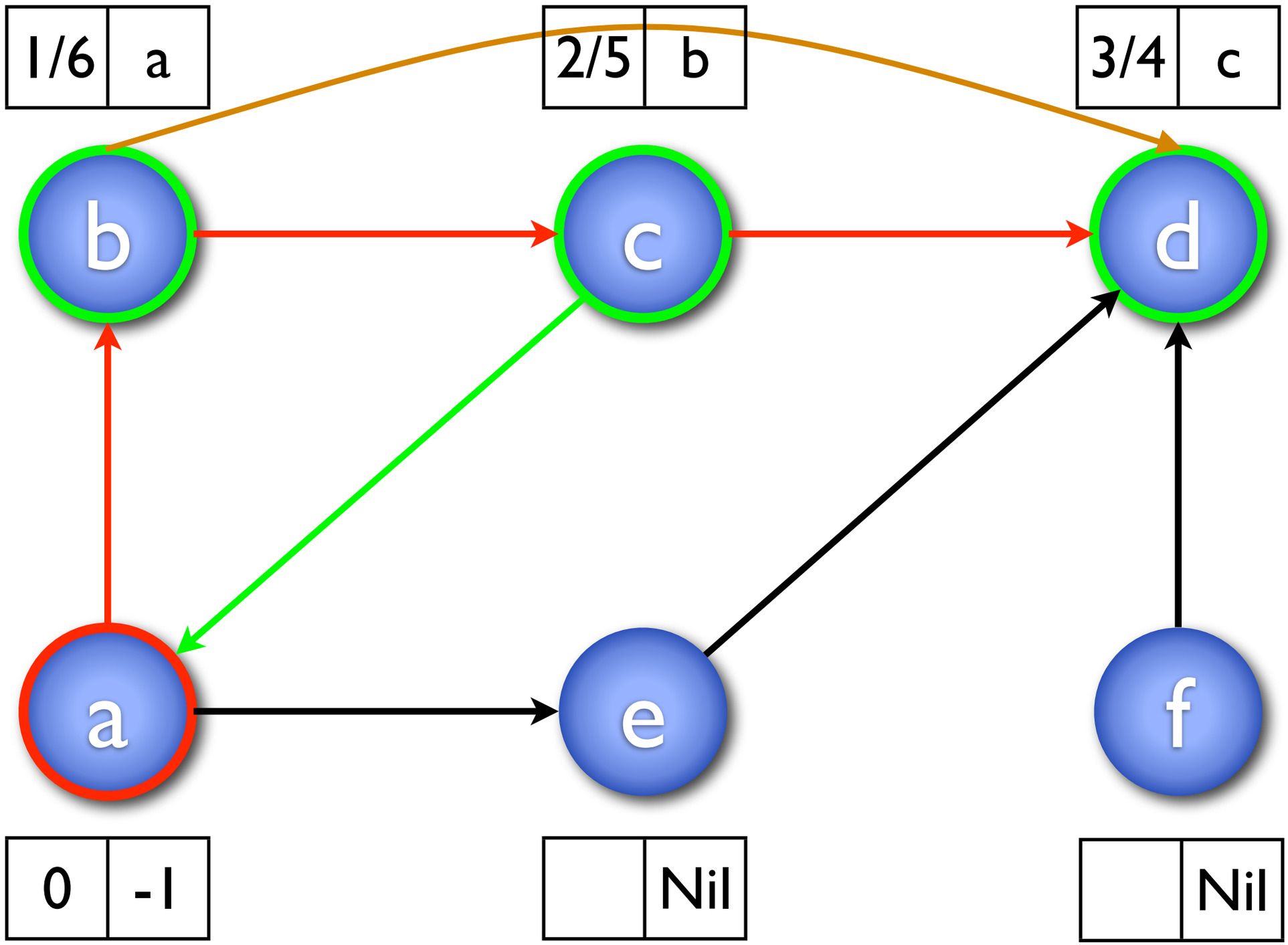

1
a
2
b
Nil
b
c
d
a
e
f
0
-1
Nil
Nil

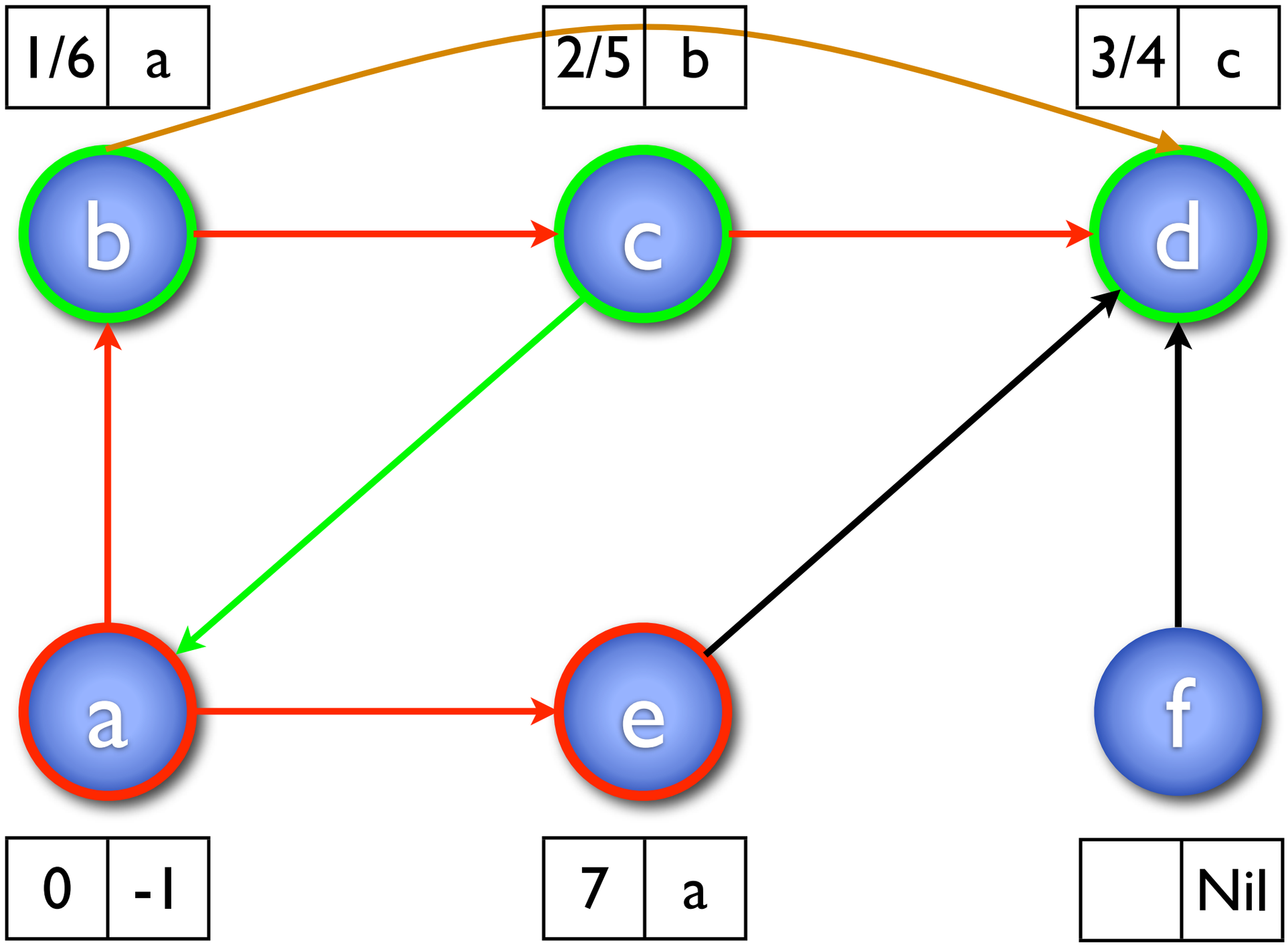

1
a
2
b
3
c
b
c
d
a
e
f
0
-1
Nil
Nil

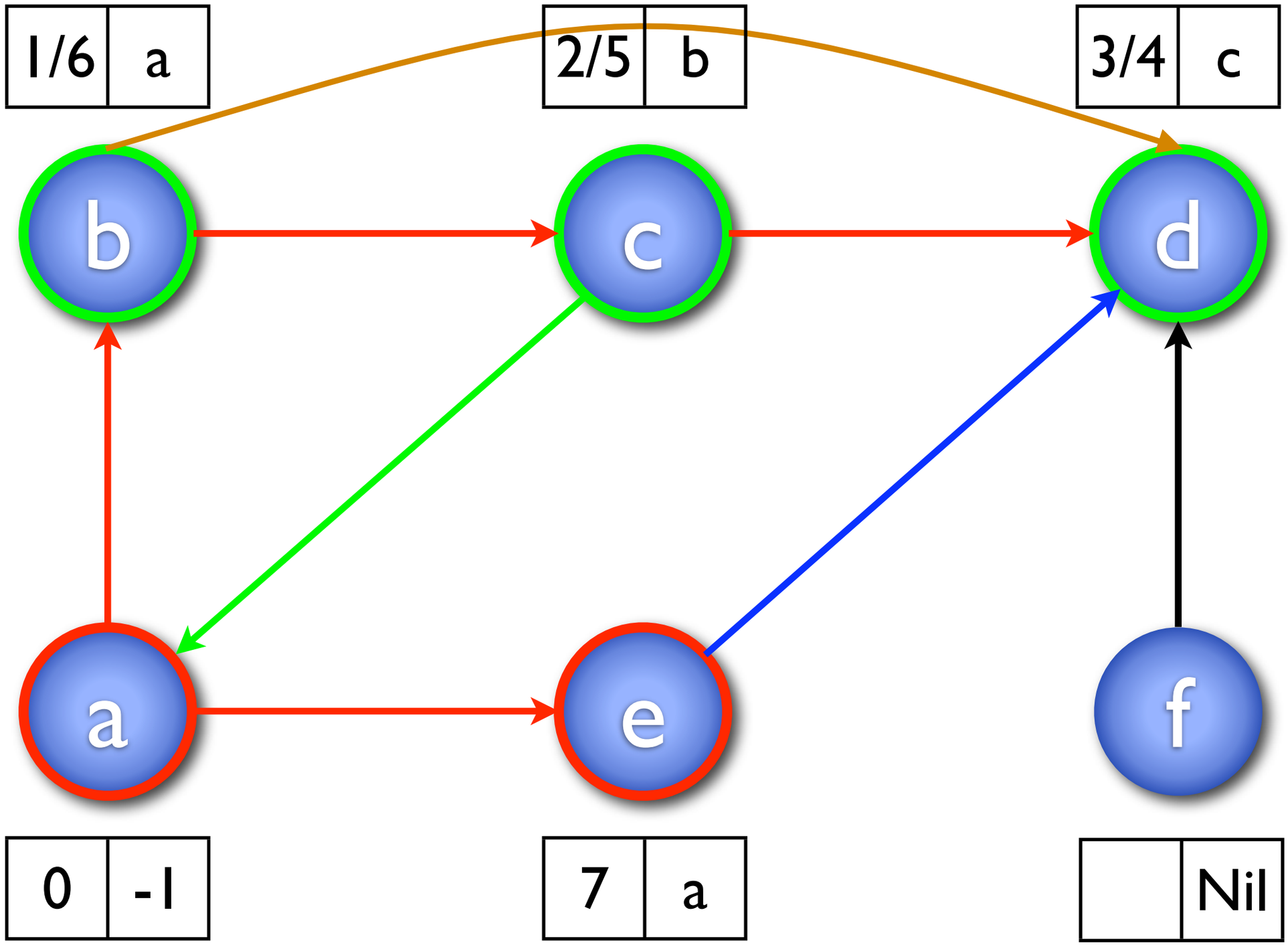

1
a
2
b
3/4
c
b
c
d
a
e
f
0
-1
Nil
Nil

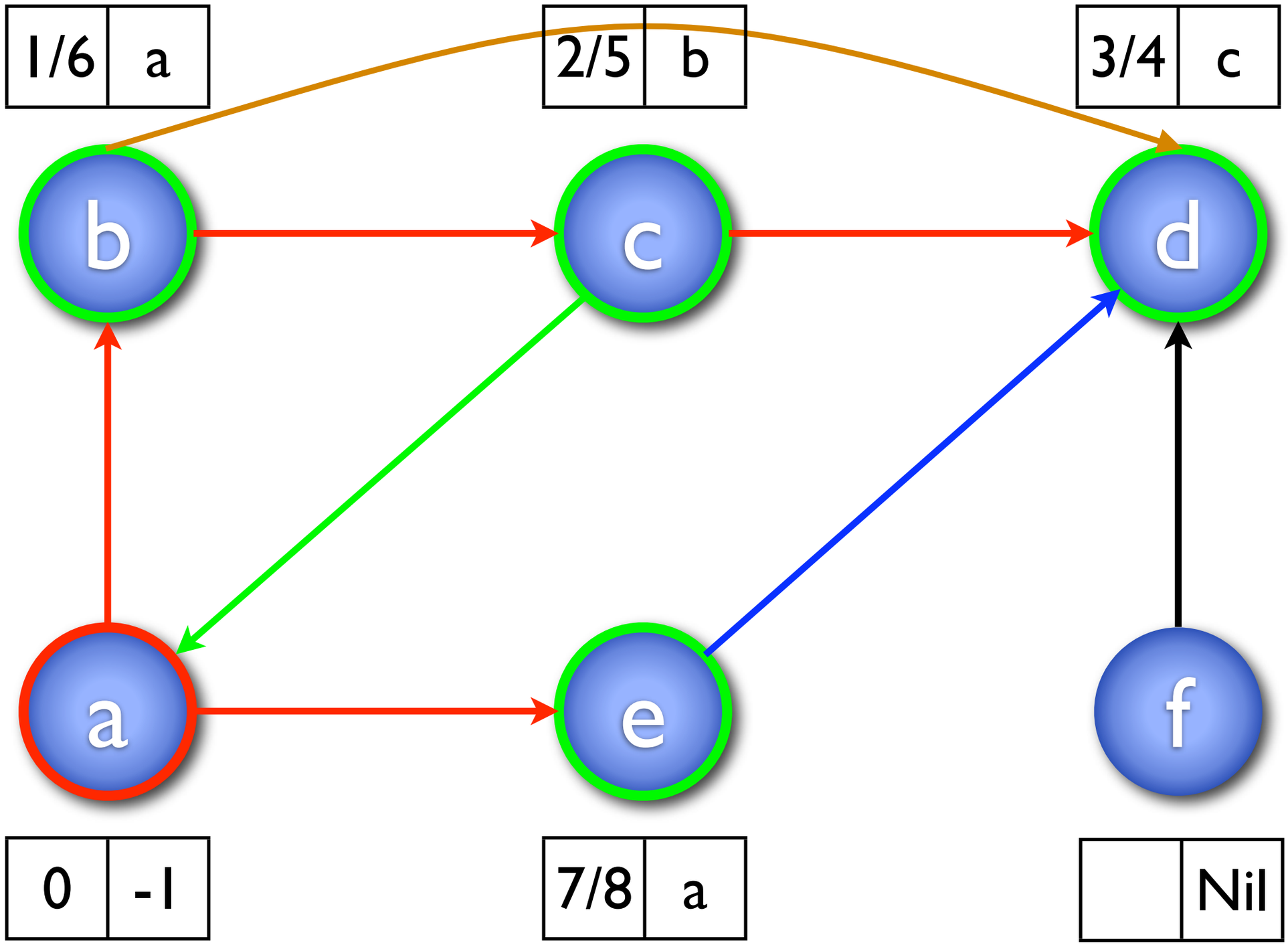

1
a
2/5
b
3/4
c
b
c
d
a
e
f
0
-1
Nil
Nil

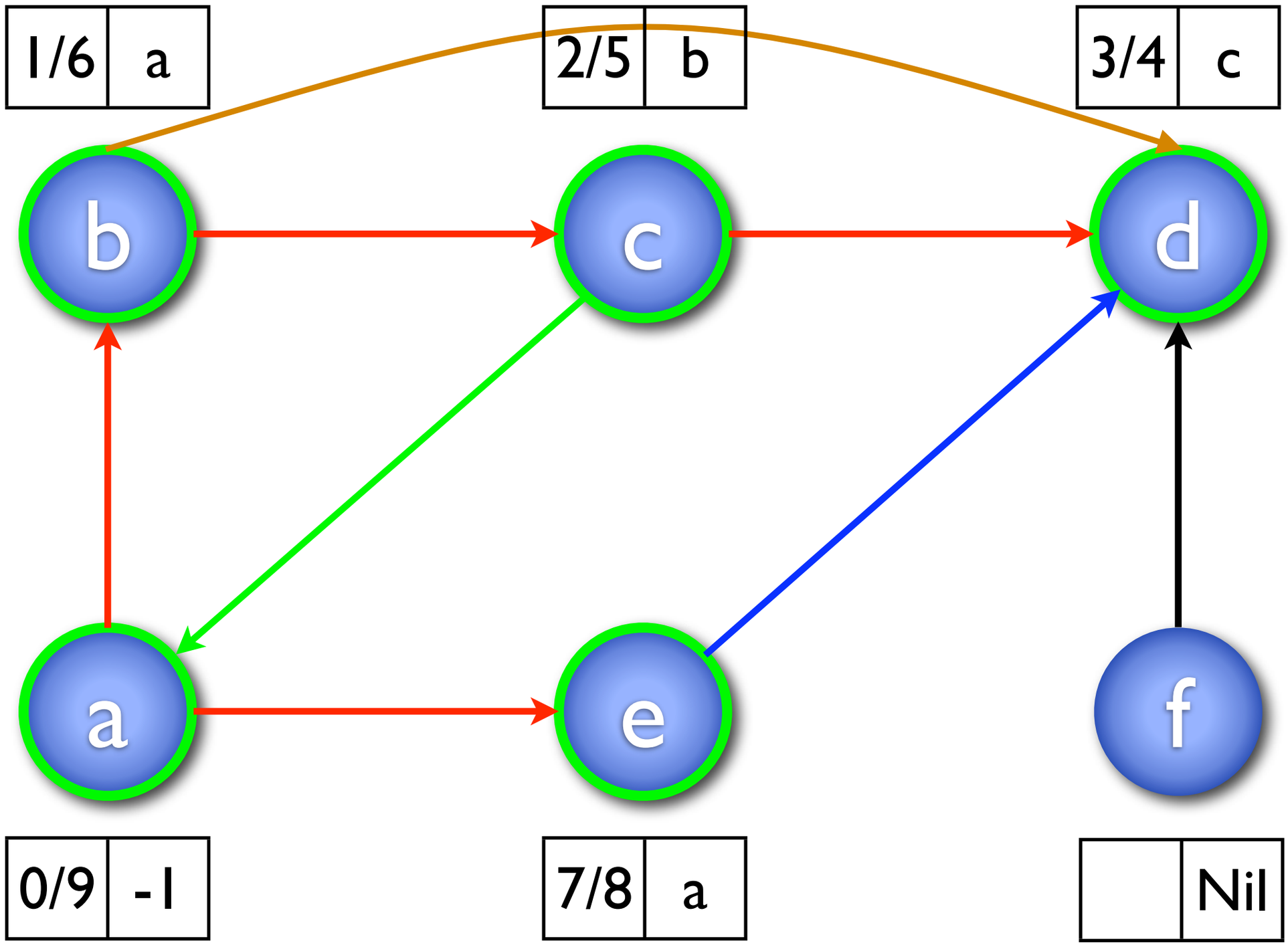

1
a
2/5
b
3/4
c
b
c
d
a
e
f
0
-1
Nil
Nil

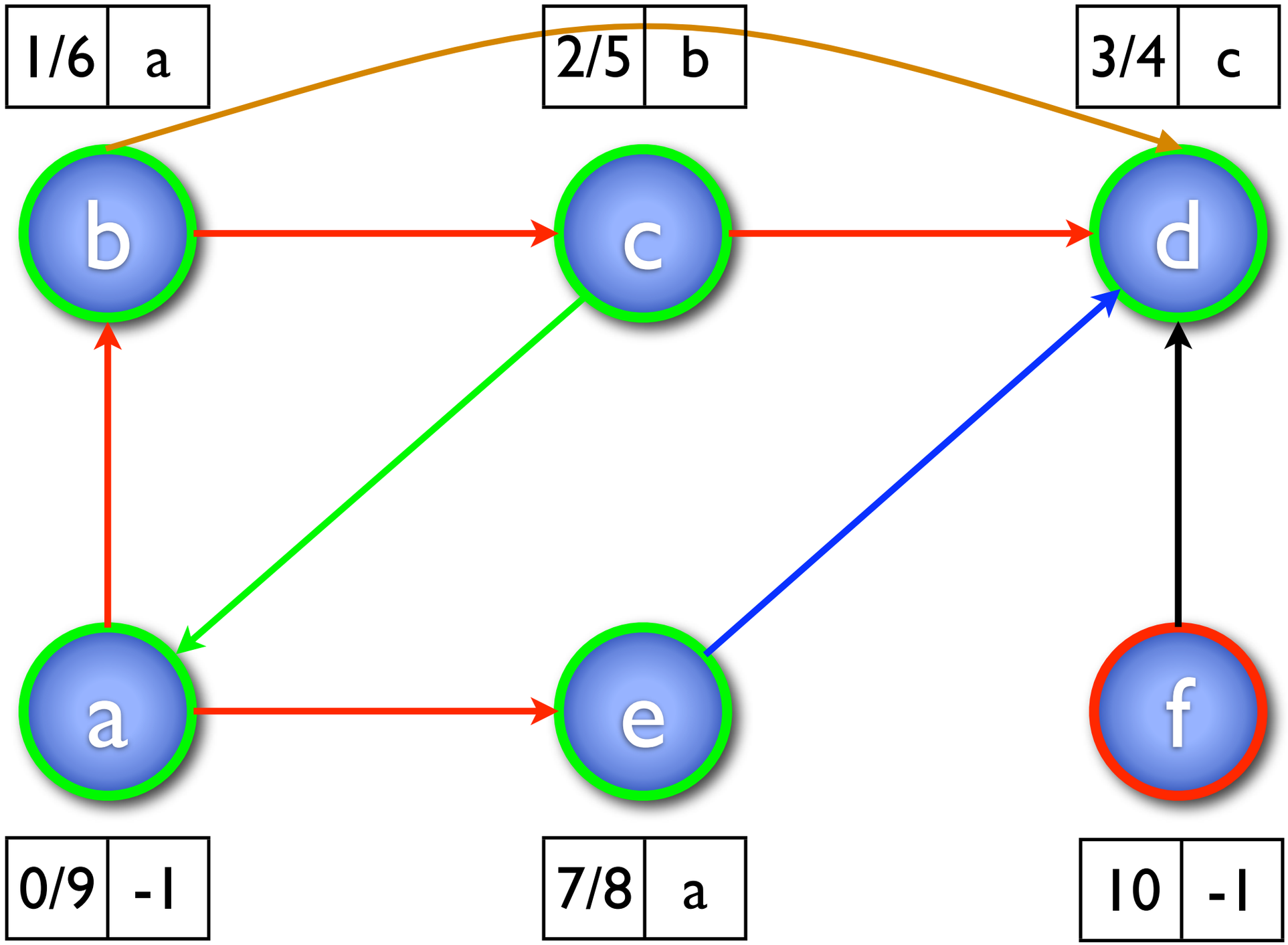

1/6
a
2/5
b
3/4
c
b
c
d
a
e
f
0
-1
Nil
Nil

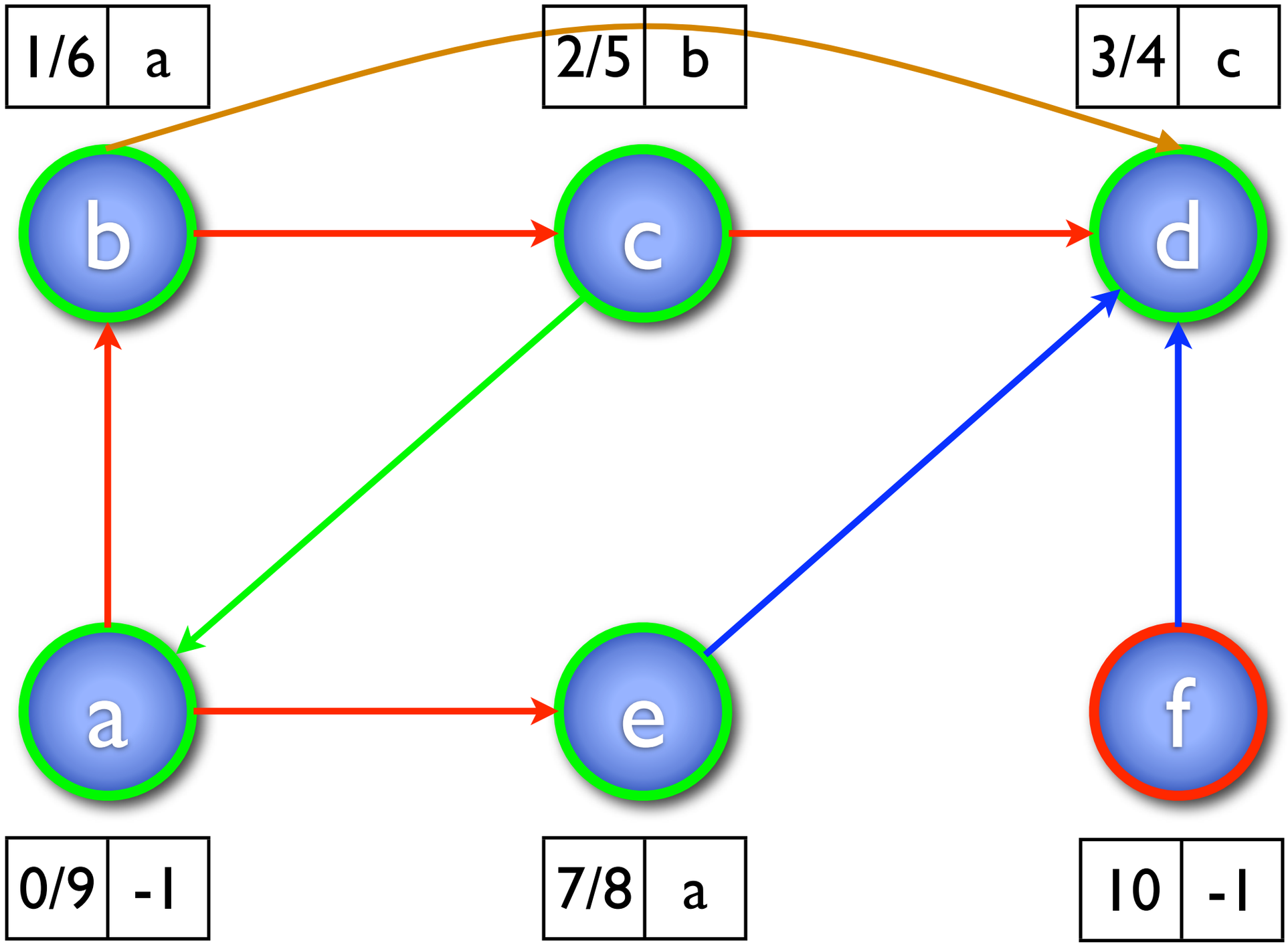

1/6
a
2/5
b
3/4
c
b
c
d
a
e
f
0
-1
7
a
Nil

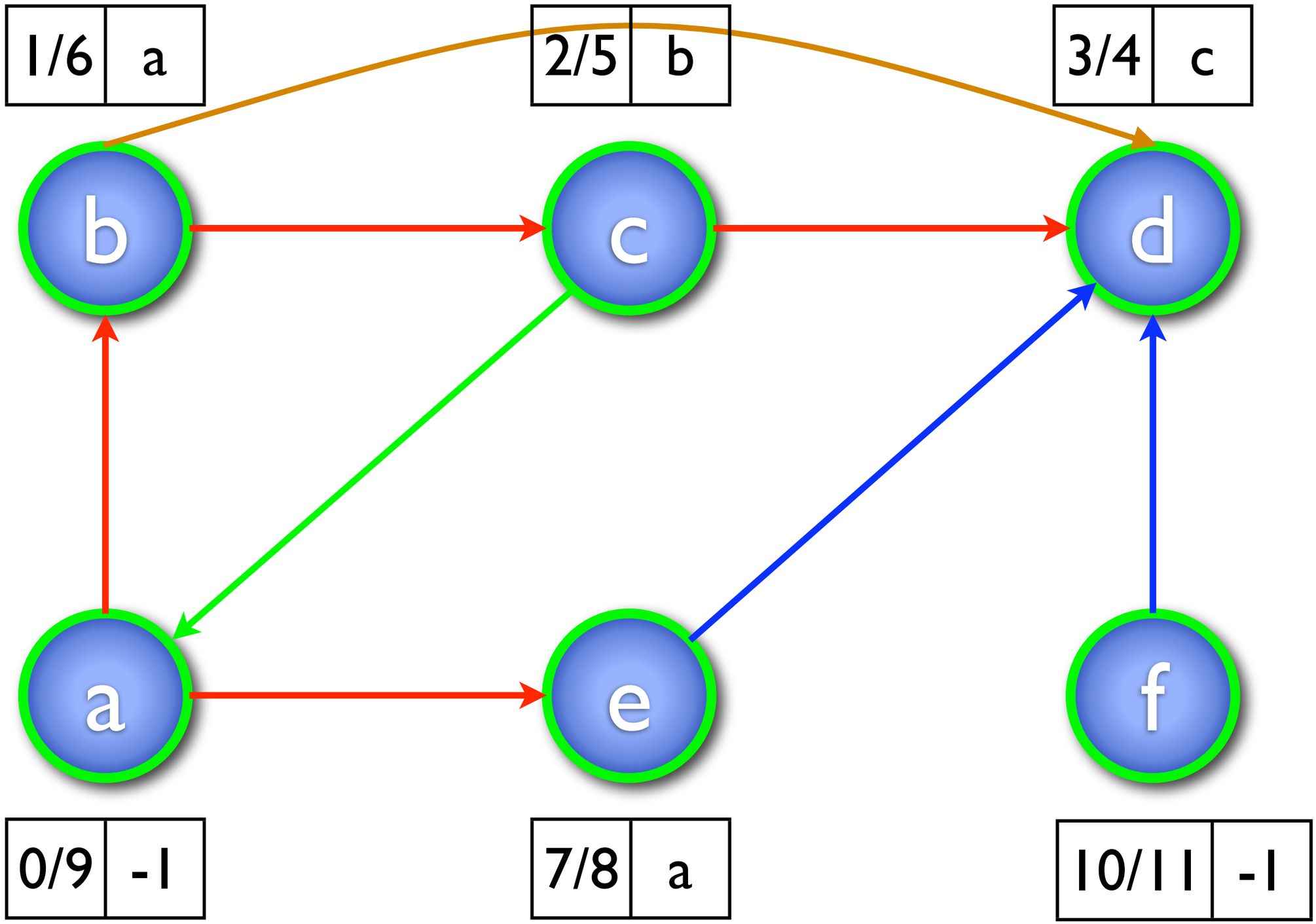

1/6
a
2/5
b
3/4
c
b
c
d
a
e
f
0
-1
7
a
Nil


1/6
a
2/5
b
3/4
c
b
c
d
a
e
f
0
-1
7/8
a
Nil


1/6
a
2/5
b
3/4
c
b
c
d
a
e
f
0/9
-1
7/8
a
Nil


1/6
a
2/5
b
3/4
c
b
c
d
a
e
f
0/9
-1
7/8
a
10
-1


1/6
a
2/5
b
3/4
c
b
c
d
a
e
f
0/9
-1
7/8
a
10
-1

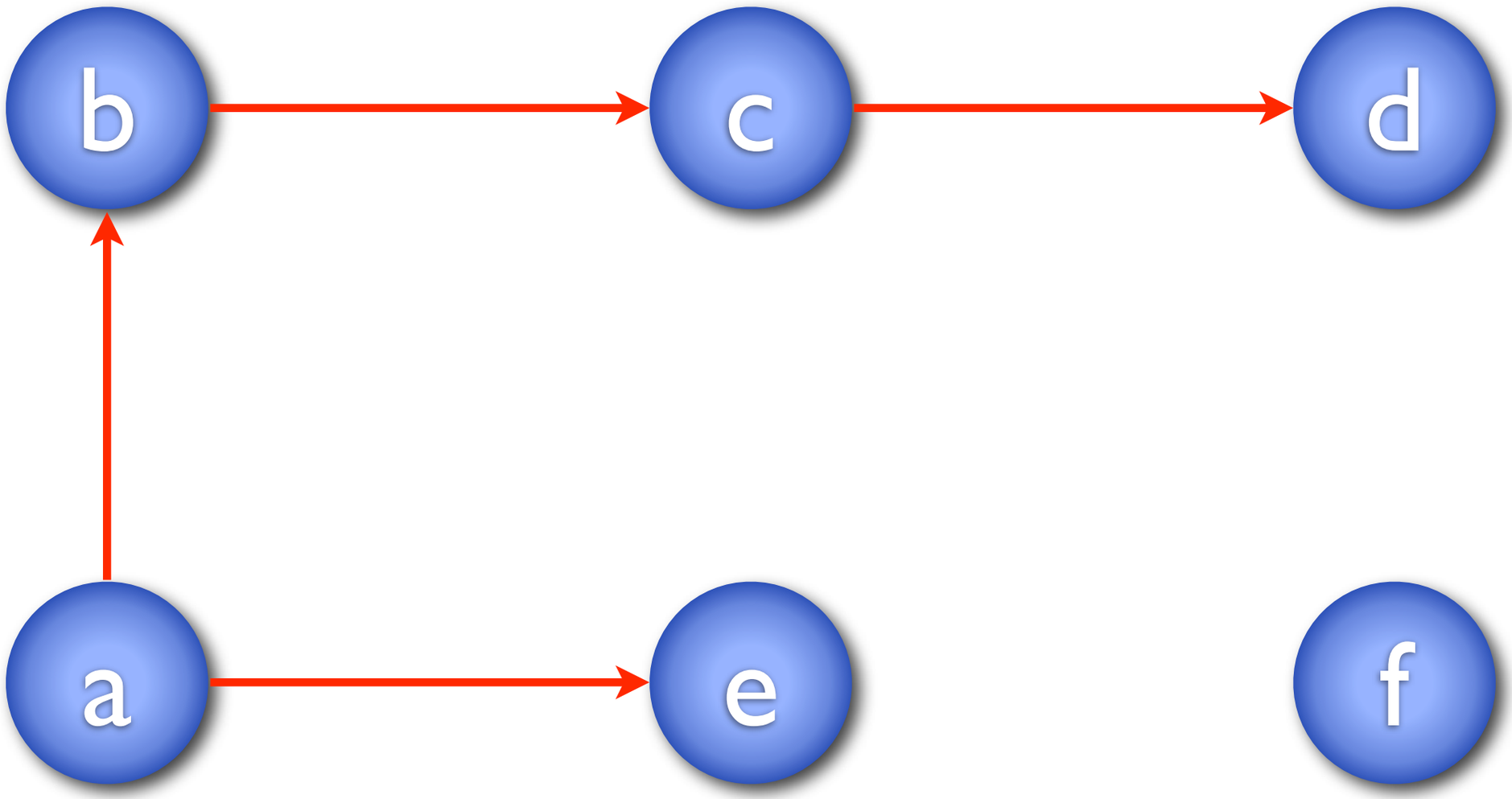

1/6
a
2/5
b
3/4
c
b
c
d
a
e
f
0/9
-1
7/8
a
10/11
-1

Tiefensuche - Analyse

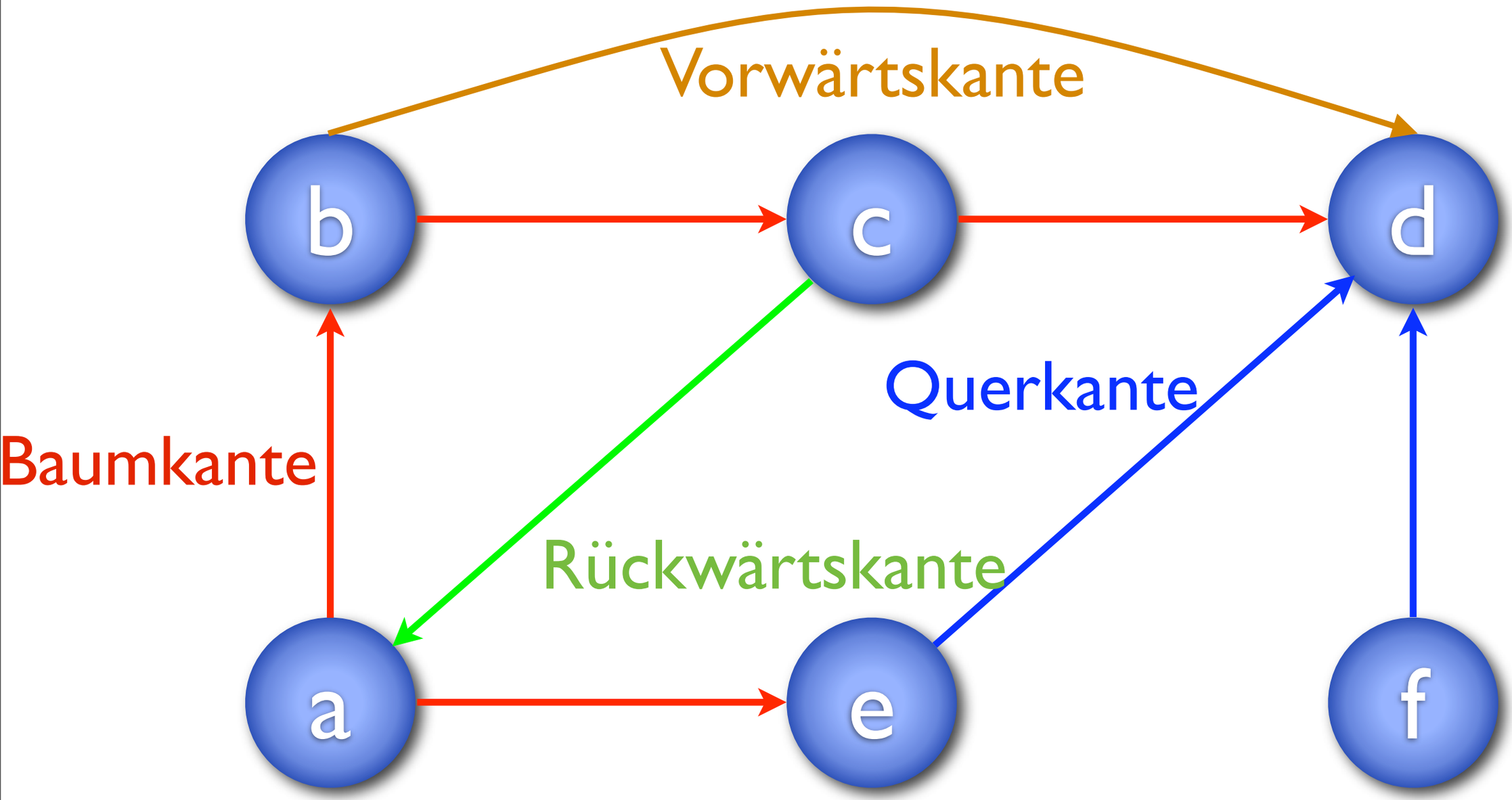
- Laufzeit: O(|V| + |E|)
- Ergebnis:
  - Tiefensuchwald `pred[]`

    Wurzel eines Tiefensuchbaums sind Startknoten von DFS-Visit
  - Entdeckungs- und Endzeiten `d[], f[]`
  - Klassifizierung der Kanten

Tiefensuchwald



Klassifizierung der Kanten



Klassifizierung der Kanten

- Baum- Vorwärtskante **(u, v):** `d[u] < d[v] ∧ f[u] > f[v]`

- Rückwärtskante **(u, v):** `d[u] > d[v] ∧ f[u] < f[v]`

- Querkante **(u, v):** `d[u] > d[v] ∧ f[u] > f[v]`

Topologische Sortierung

- eine Topologische Sortierung eines DAG G ist eine lineare Anordnung aller seiner Knoten mit der Eigenschaft, dass u in der Anordnung vor v liegt, falls es in G eine Kante (u, v) gibt

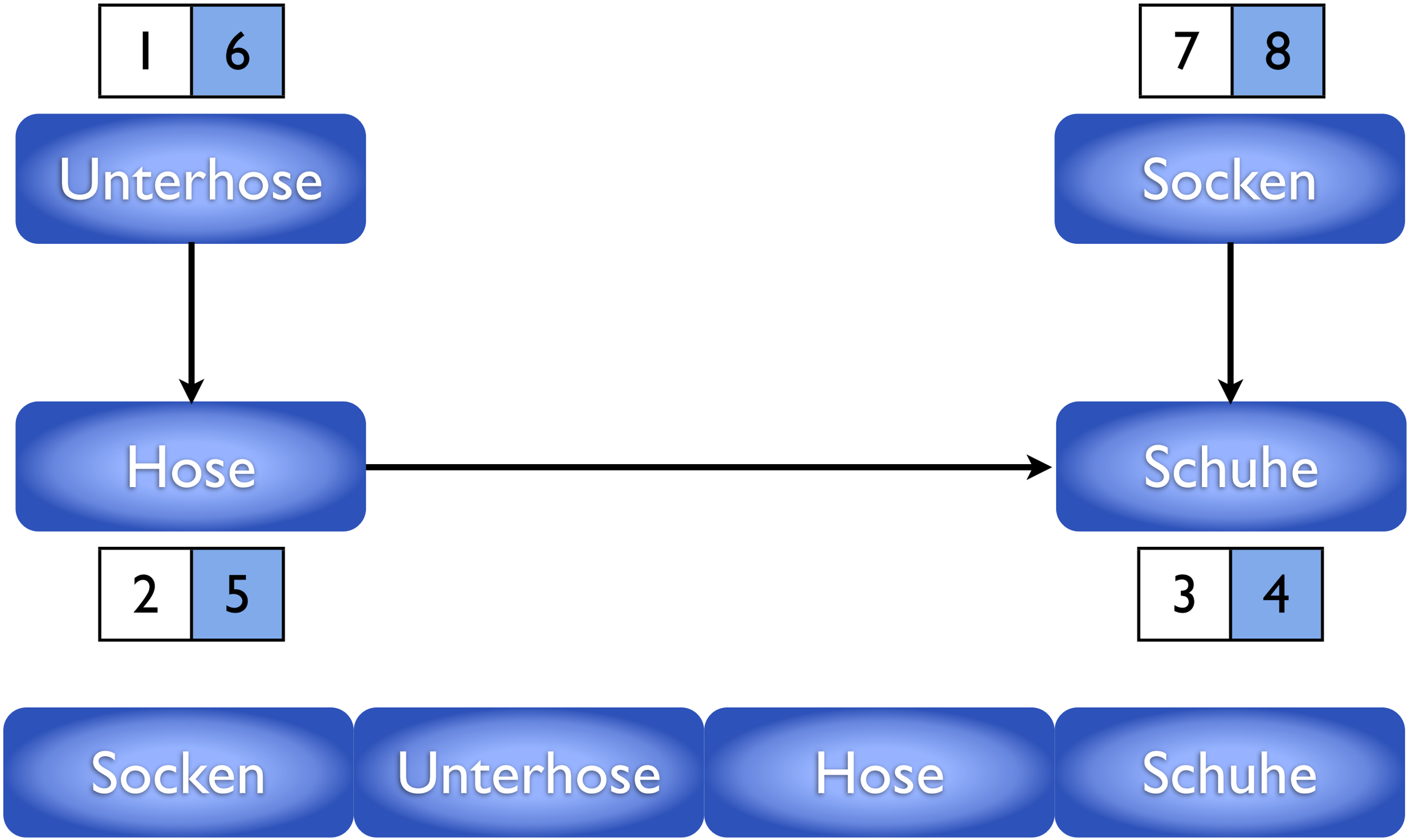
Topologisches Sortieren - Konzept

- führe DFS auf G durch und füge Knoten bei ihrer Fertigstellung vorne in eine verkettete Liste ein

Topologisches Sortieren - Implementierung

- Eingabegraph DAG G = (V, E) (in Adjazenzlisten-Darstellung)
- Datenstrukturen für DFS
- verkettete Liste $\llcorner$

```
Topo(G)
for alle Knoten u ∈ V[G]
  do visited[u] ← false
     pred[u] ← Nil
zeit ← 0
for alle Knoten u ∈ V[G]
  do if visited[u] = false
    then pred[u] ← -1
         DFS-Visit(u)

Topo-Visit(u)
visited[u] ← true
d[u] ← time
time ← time + 1
for alle v ∈ Adj[u]
  do if not visited[v]
     then pred[v] ← u
          DFS-Visit(v)
f[u] ← time
time ← time + 1
Add(L,u)
```


1
6
Unterhose
7
8
Socken
Hose
Schuhe
2
5
3
4
Socken
Unterhose
Hose
Schuhe

Topologisches Sortieren - Analyse

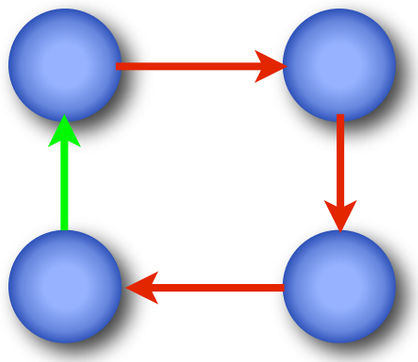
- Laufzeit: O(|V| + |E|) (DFS)
- Ergebnis: topologische Sortierung
- Alternativer Algorithmus: Eingangskanten zählen, Knoten mit Eingangsgrad 0 entfernen

starke Zusammenhangs-komponenten sCC´s

- Finde starke Zusammenhangskomponenten in einem gerichteten Graphen

sCC´s - Konzept

- Erkennung der sCC´s mit Hilfe der Klassifizierung der Kanten also DFS:
  - Rückwärtskanten erzeugen sCC´s



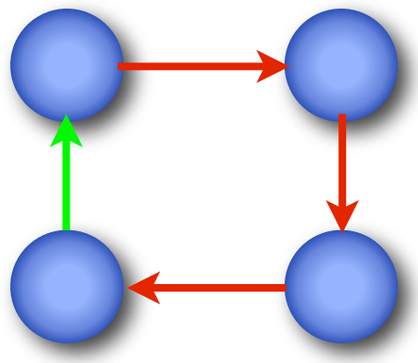
- Problem: Unterscheidung von Rück- und Querkanten

sCC´s - Konzept

- finde min. Entdeckungszeit von Knoten u, die von v über eine beliebige Folge von Baum- und eine Rückwärtskante erreichbar sind:

sind:

`lowlink` **von v**



sCC´s Konzept

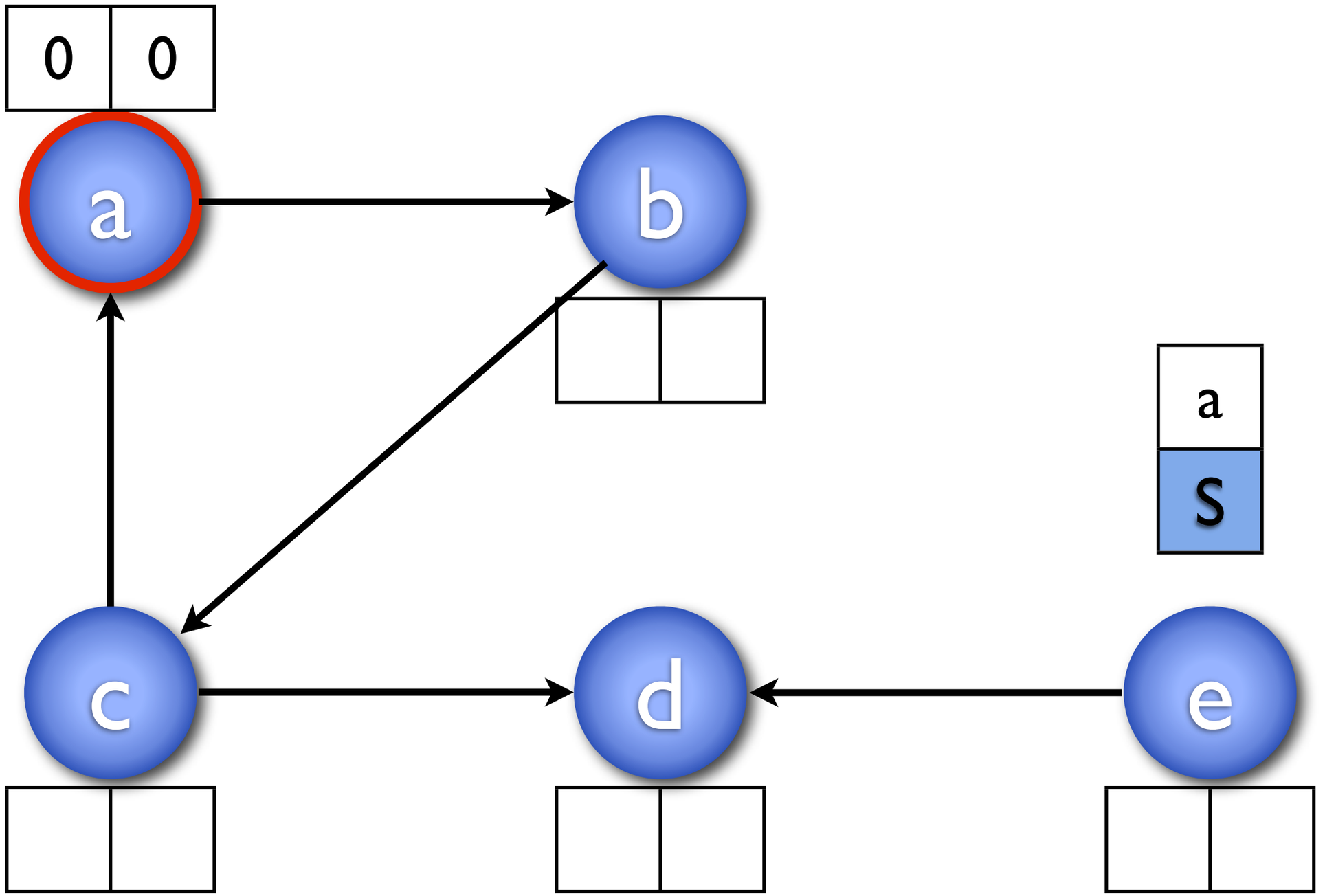
- `lowlink[u]` = Min von
  - `d[u]`
  - min. `d[v]` aller Rückwärtskanten (u, v)
  - min. `lowlink[v]` aller Baumkanten (u, v)
- ist lowlink eines Knotens u gleich seiner Entdeckungszeit, dann ist u die Wurzel einer Zusammenhangskomponente

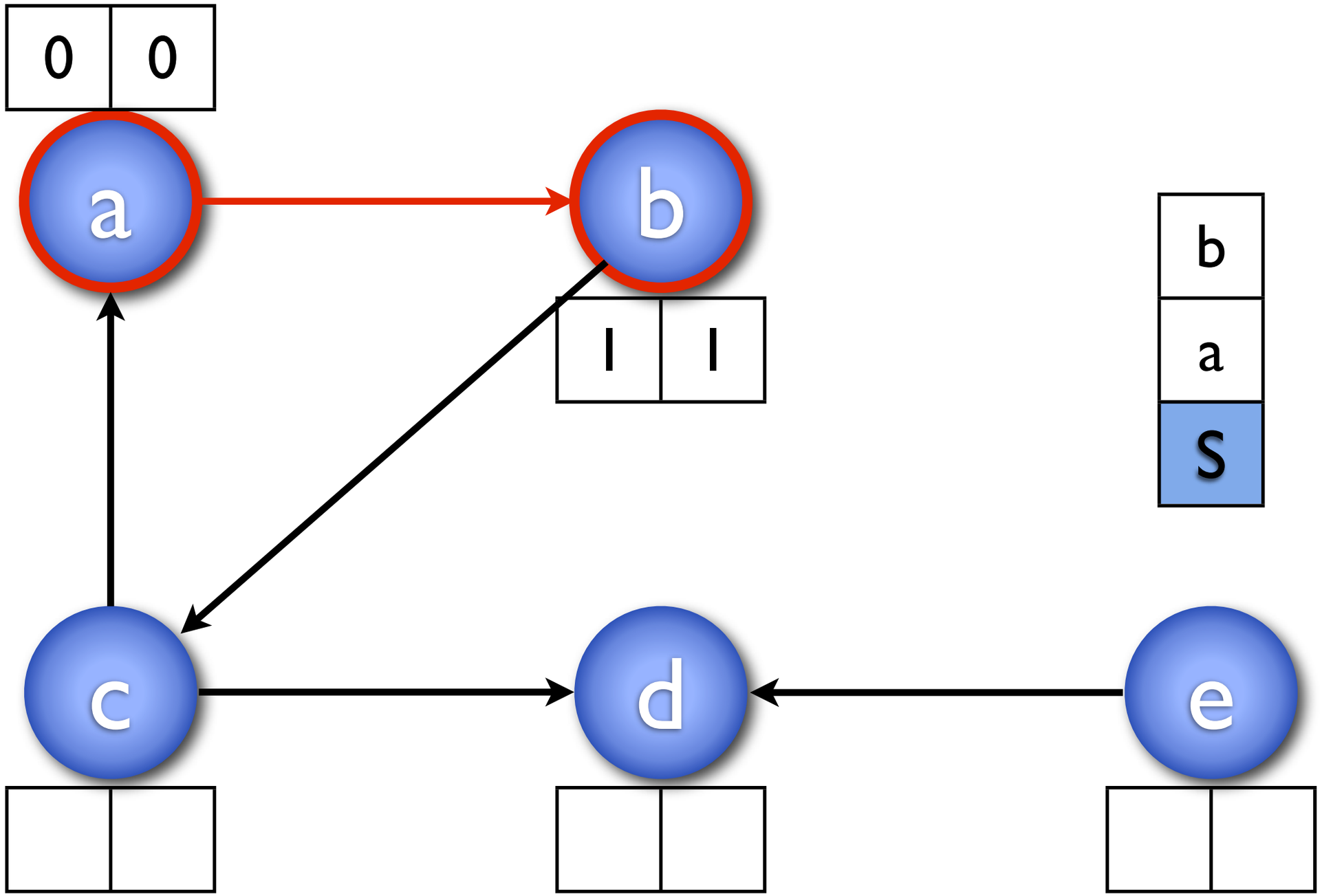
sCC´s - Implementierung

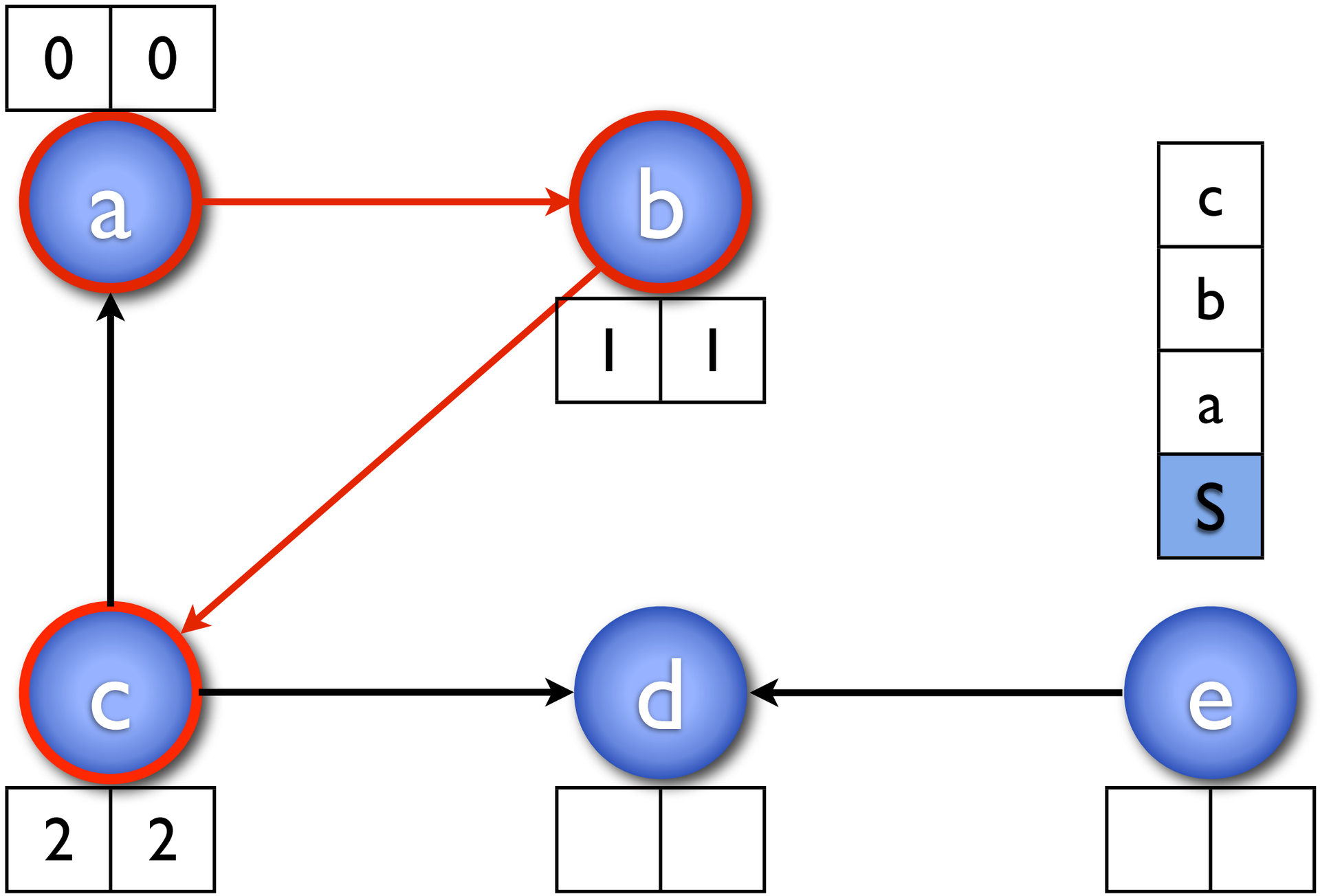
- Stack `S` zur Unterscheidung von Rückwärts- und Querkanten
- Feld zu Speicherung von lowlink von v: `lowlink[v]`
- Feld zur Speicherung der sCC´s: `sCC[]`

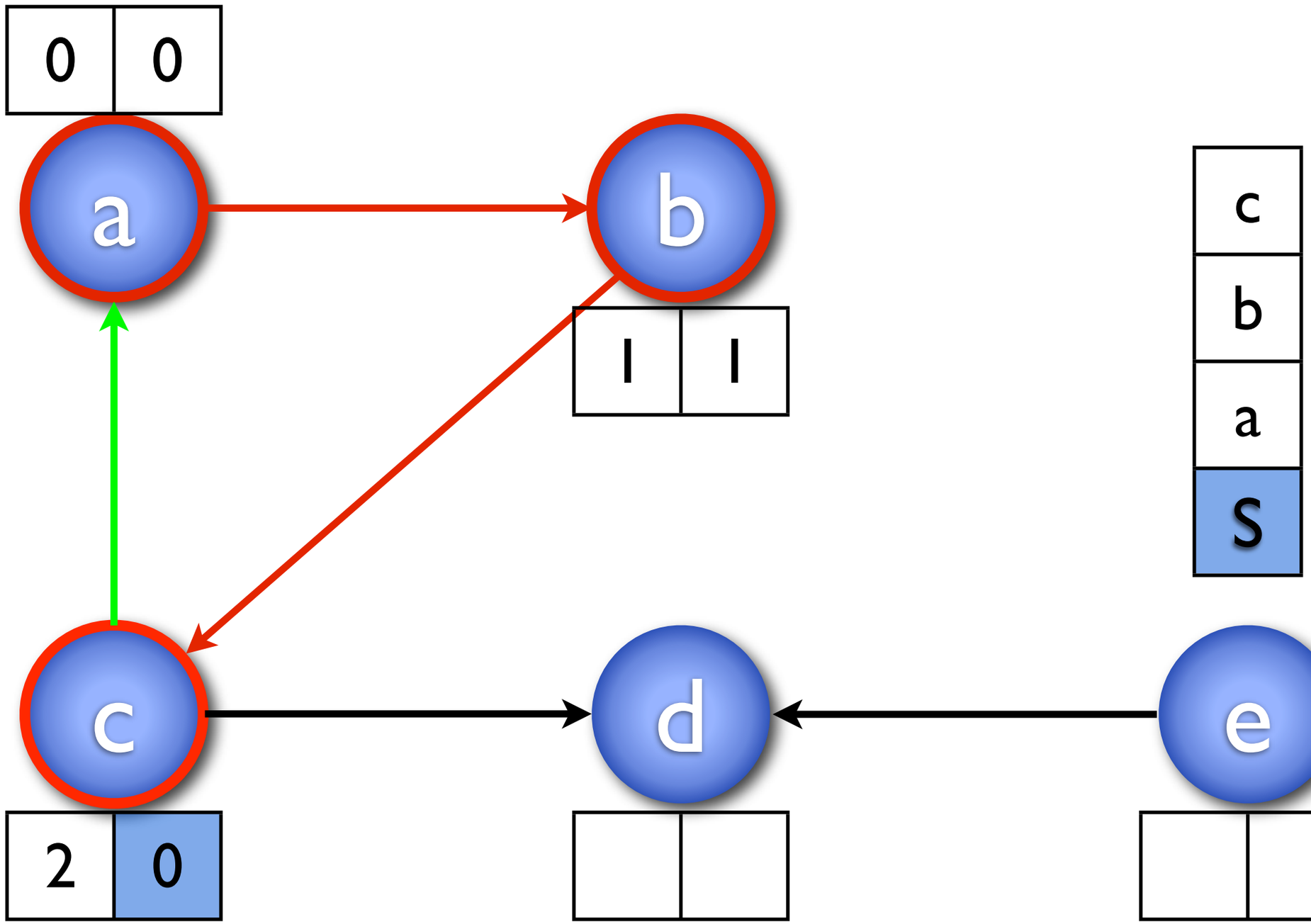
```
Tarjan(G)
for alle Knoten u ∈ V[G] // Inititalisierung
  do visited[u] ← false
     sCC[u] ← u
zeit ← 0
for alle Knoten u ∈ V[G]
  do if visited[u] = false
    then Tarjan-Visit(u)
```

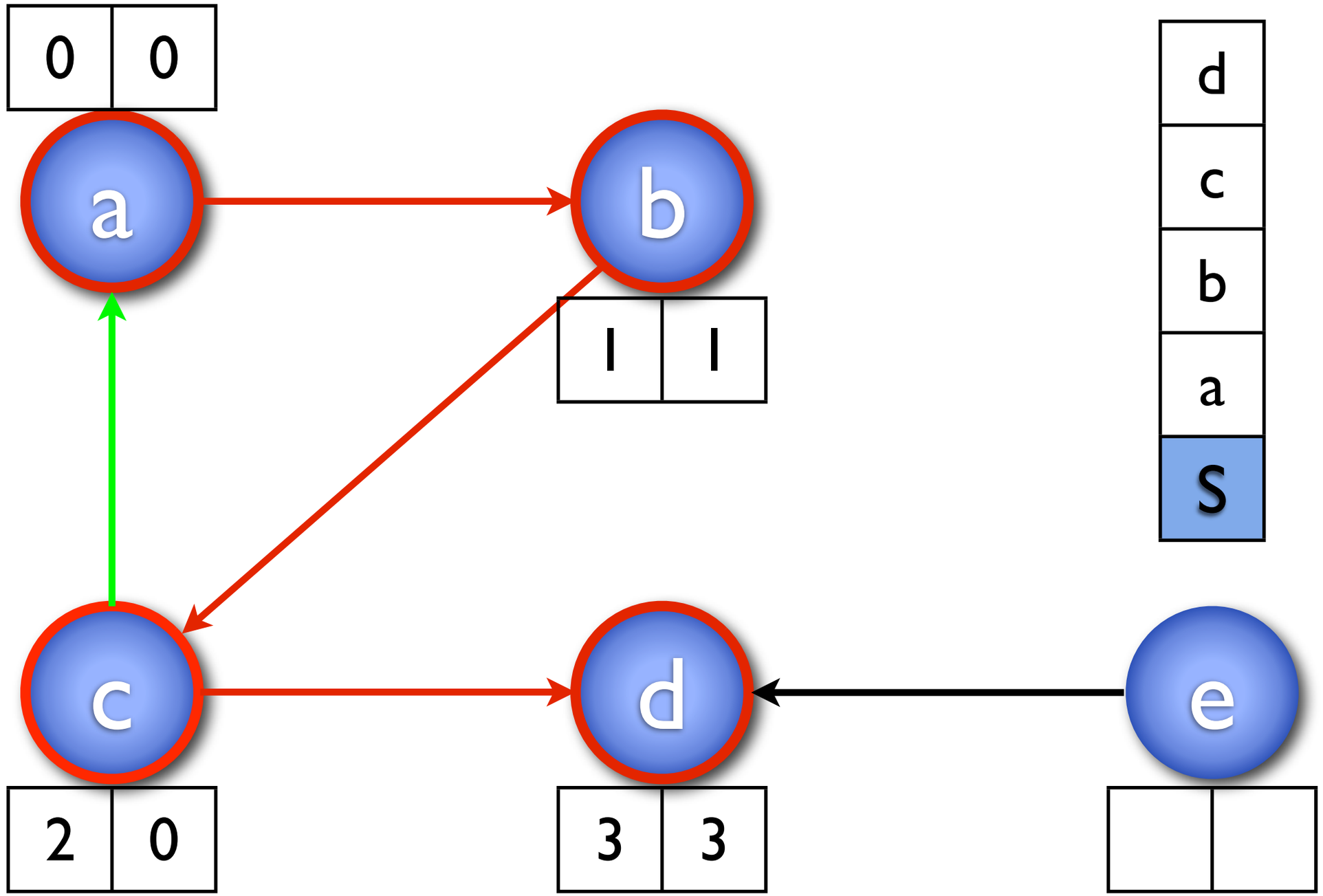
```
Tarjan-Visit(u)
visited[u] ← true
d[u] ← time
lowlink[u] ← time
time ← time + 1
Push(S, u) // aktive Knoten
for alle v ∈ Adj[u]
  do if not visited[v]
       then Tarjan-Visit(v) // Baumkante (u,v)
            lowlink[u] ← min(lowlink[u], lowlink[v])
     else if inStack(v) //Rückwärts-(Vorwärts)kante
     then lowlink[u] ← min(lowlink[u], d[v])
if (lowlink[u] = d[u]) // Wurzel einer sCC
   then sCC[u] ← u)
        repeat v ← Pop(S)
               sCC[v] ← u
        until (v = u)
```

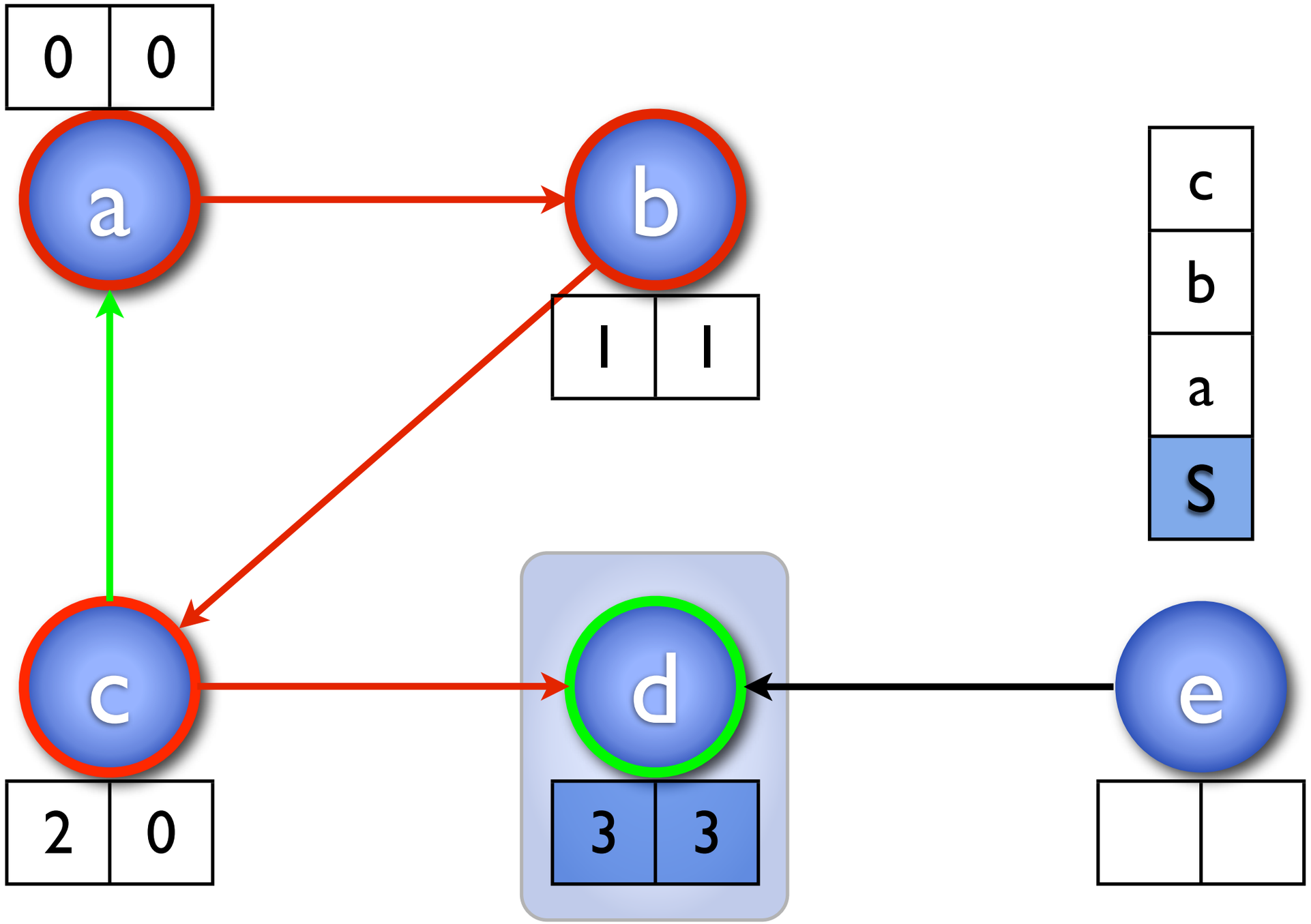

0
0
a
b
c
d
e
a
S

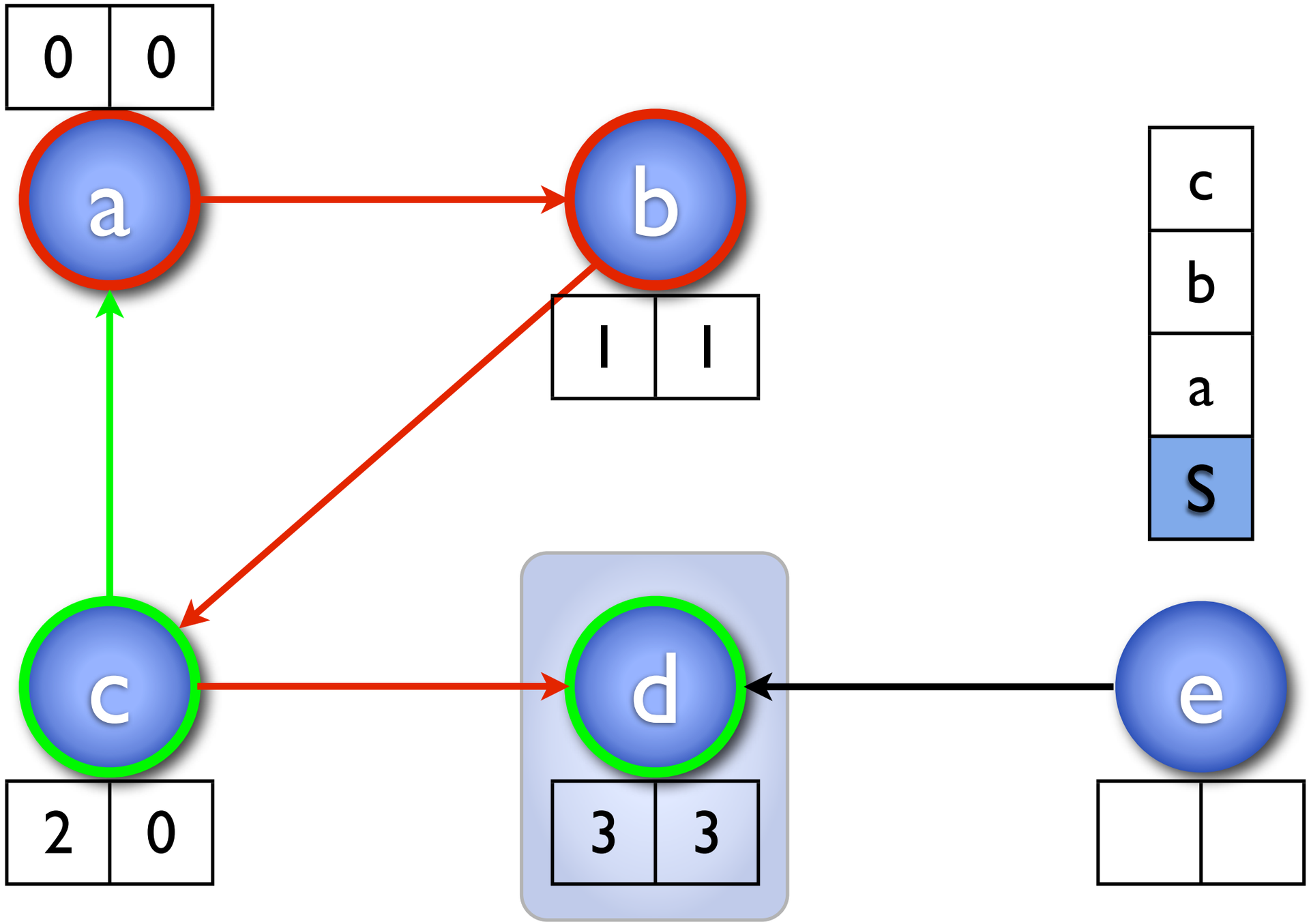

0
0
a
b
1
1
b
a
S
c
d
e

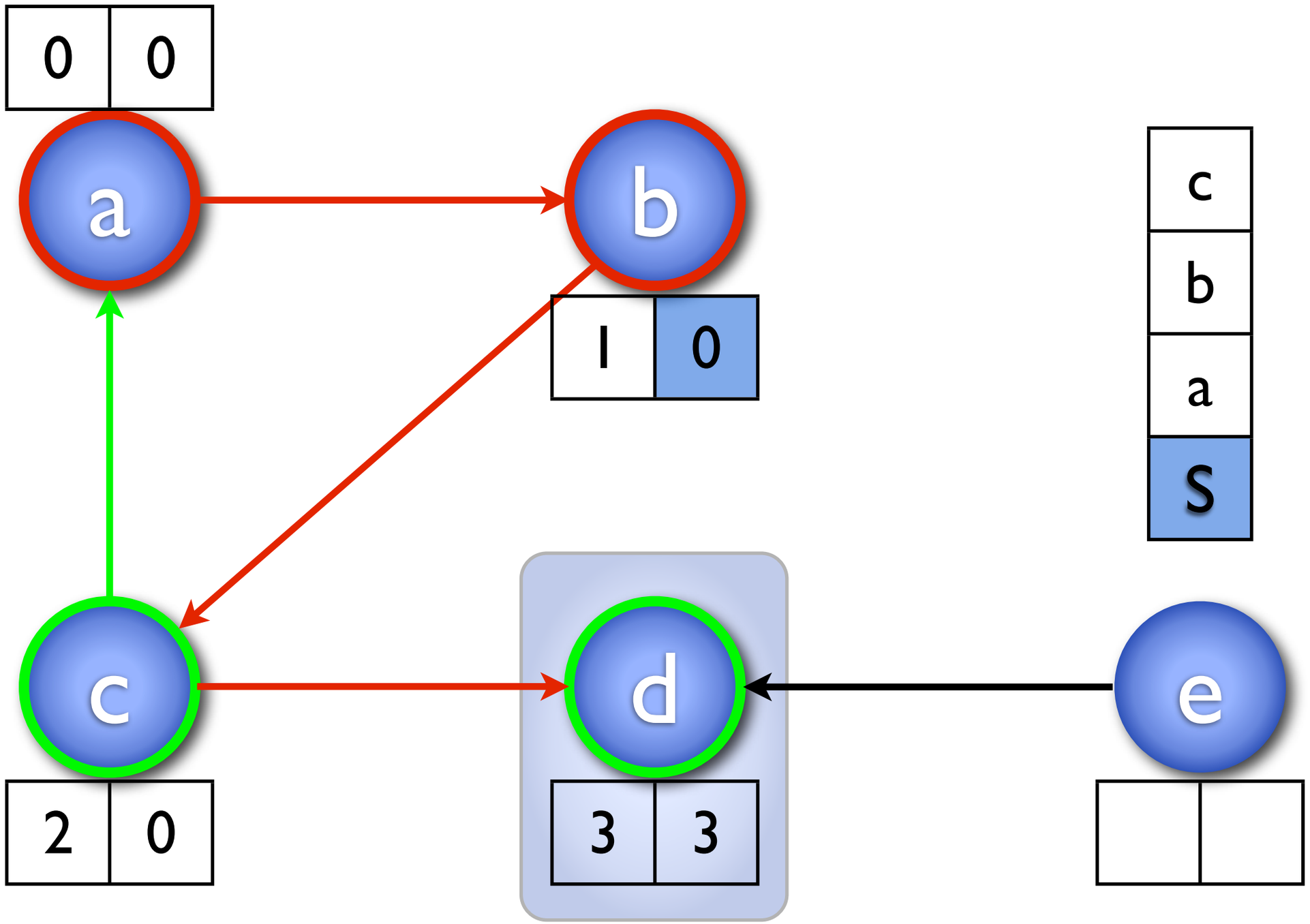

0
0
a
b
1
1
c
b
a
S
c
d
e
2
2

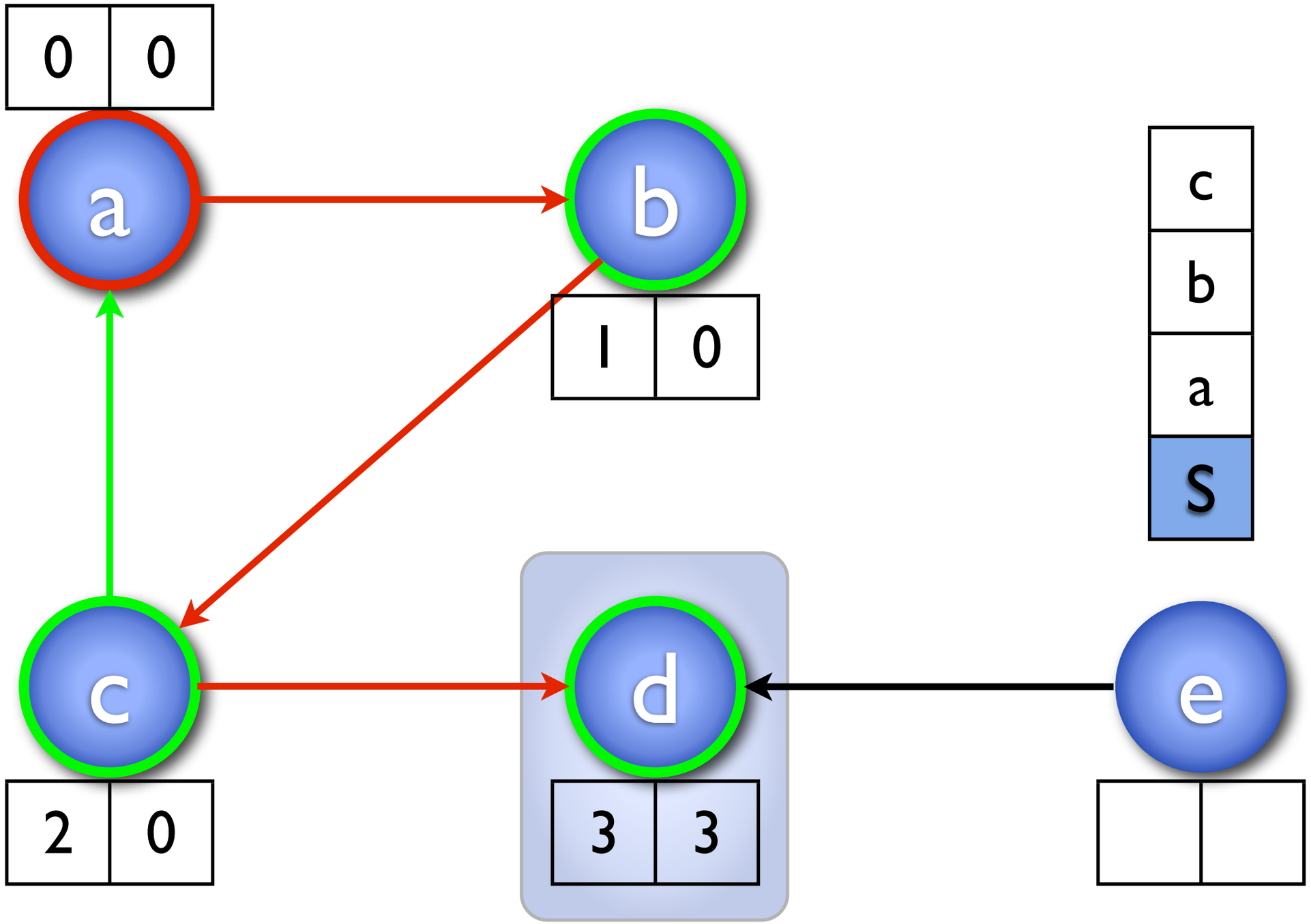

0
0
a
b
1
1
c
b
a
S
c
d
e
2
0

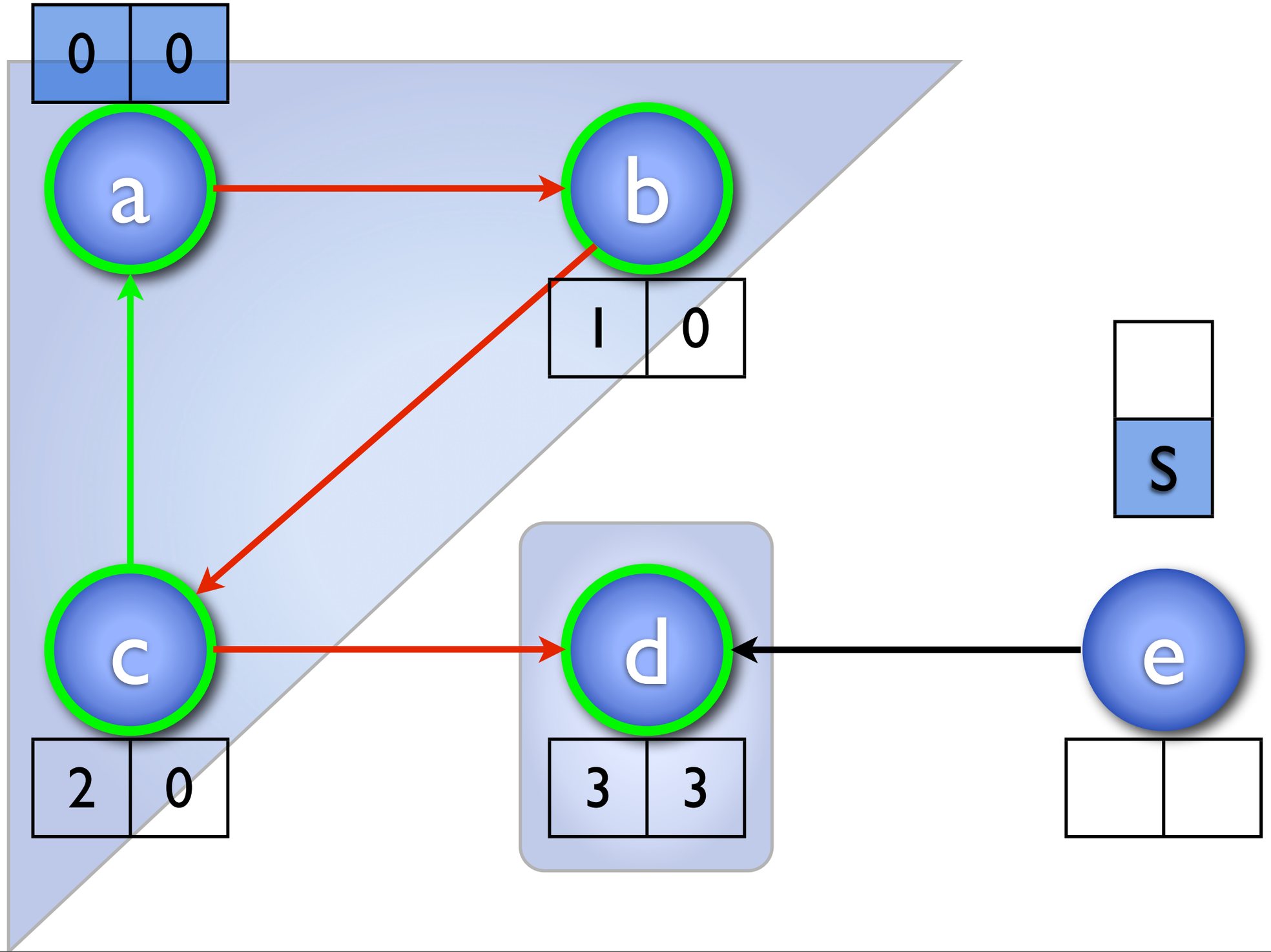

0
0
a
b
1
1
d
c
b
a
S
c
d
e
2
0
3
3

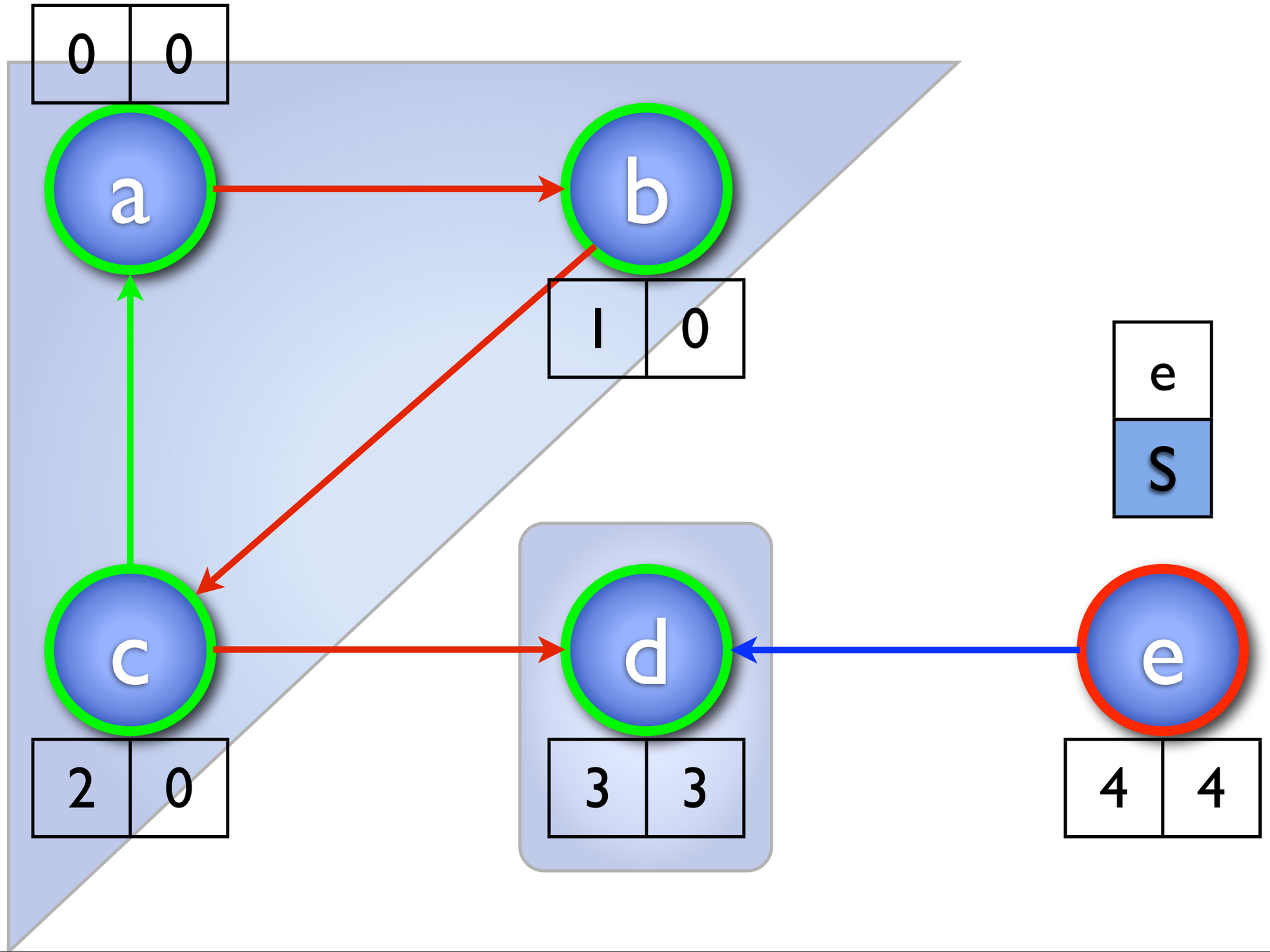

0
0
a
b
1
1
c
b
a
S
c
d
e
2
0
3
3

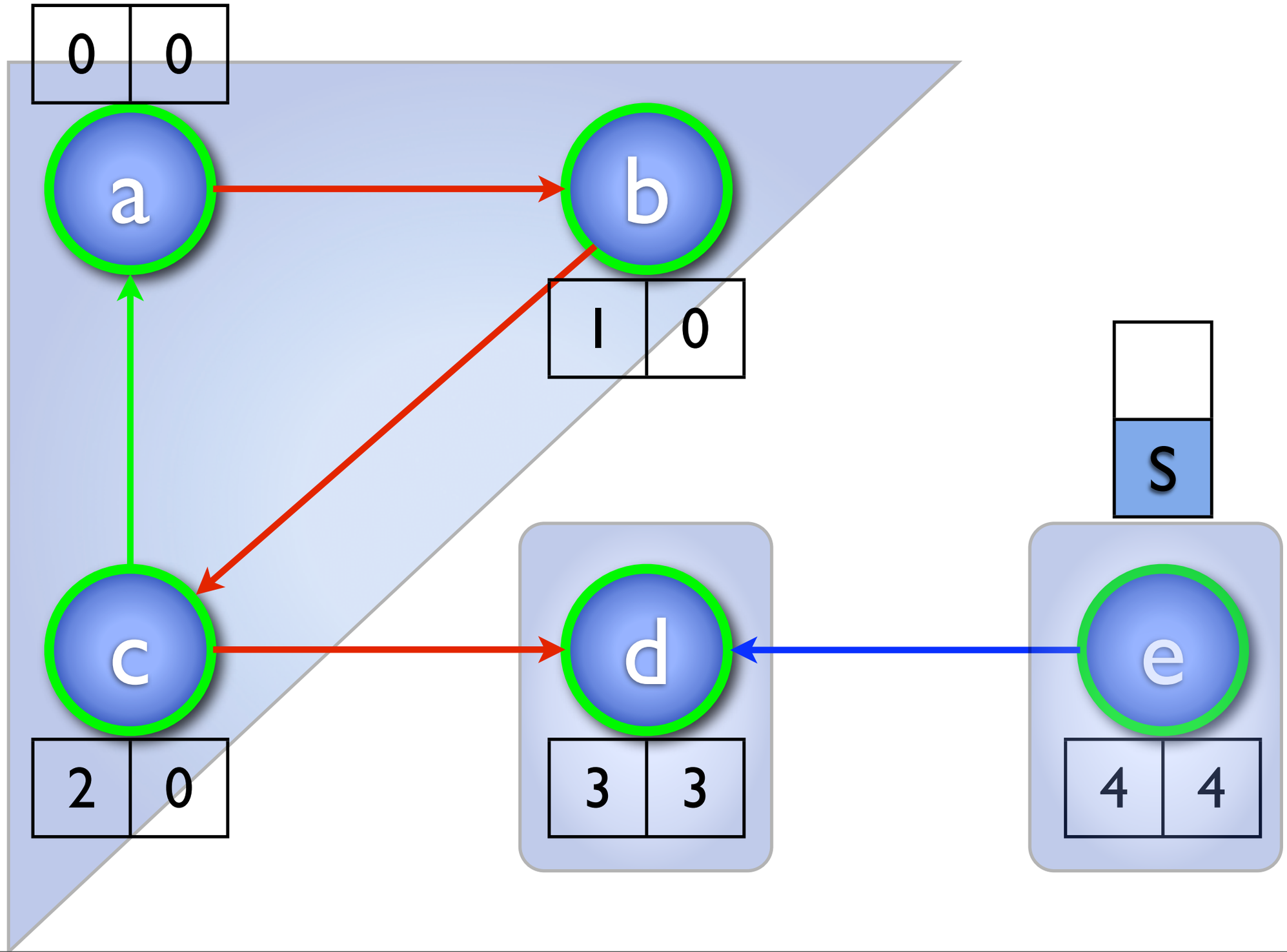

0
0
a
b
1
1
c
b
a
S
c
d
e
2
0
3
3


0
0
a
b
1
0
c
b
a
S
c
d
e
2
0
3
3


0
0
a
b
1
0
c
b
a
S
c
d
e
2
0
3
3


0
0
a
b
1
0
S
c
d
e
2
0
3
3


0
0
a
b
1
0
e
S
c
d
e
2
0
3
3
4
4

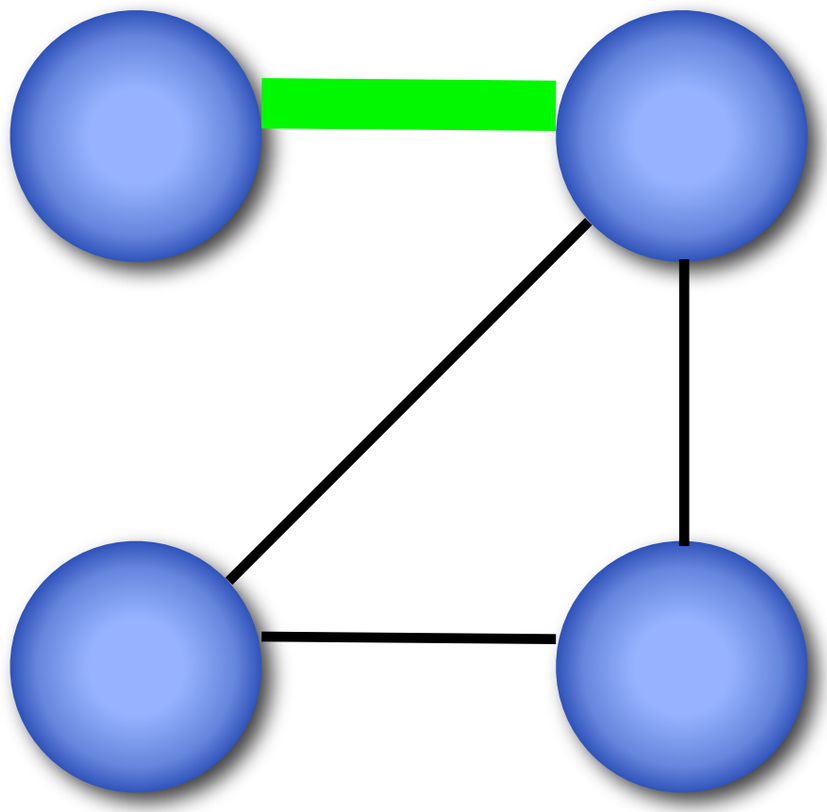

0
0
a
b
1
0
S
c
d
e
2
0
3
3
4
4

sCC´s - Analyse

- Laufzeit: O(|V| + |E|) (DFS)
- Ergebnis: u, v in einer sCC gdw.
  `sCC[u] = sCC[v]`
- altenativer Algorithmus: zweimal BFS mit transponiertem Graph

Brücken

- eine Brücke (kritische Kante) in einem Graph ist eine Kante, deren Entfernen die Zahl der Zusammenhangskomponenten erhöht



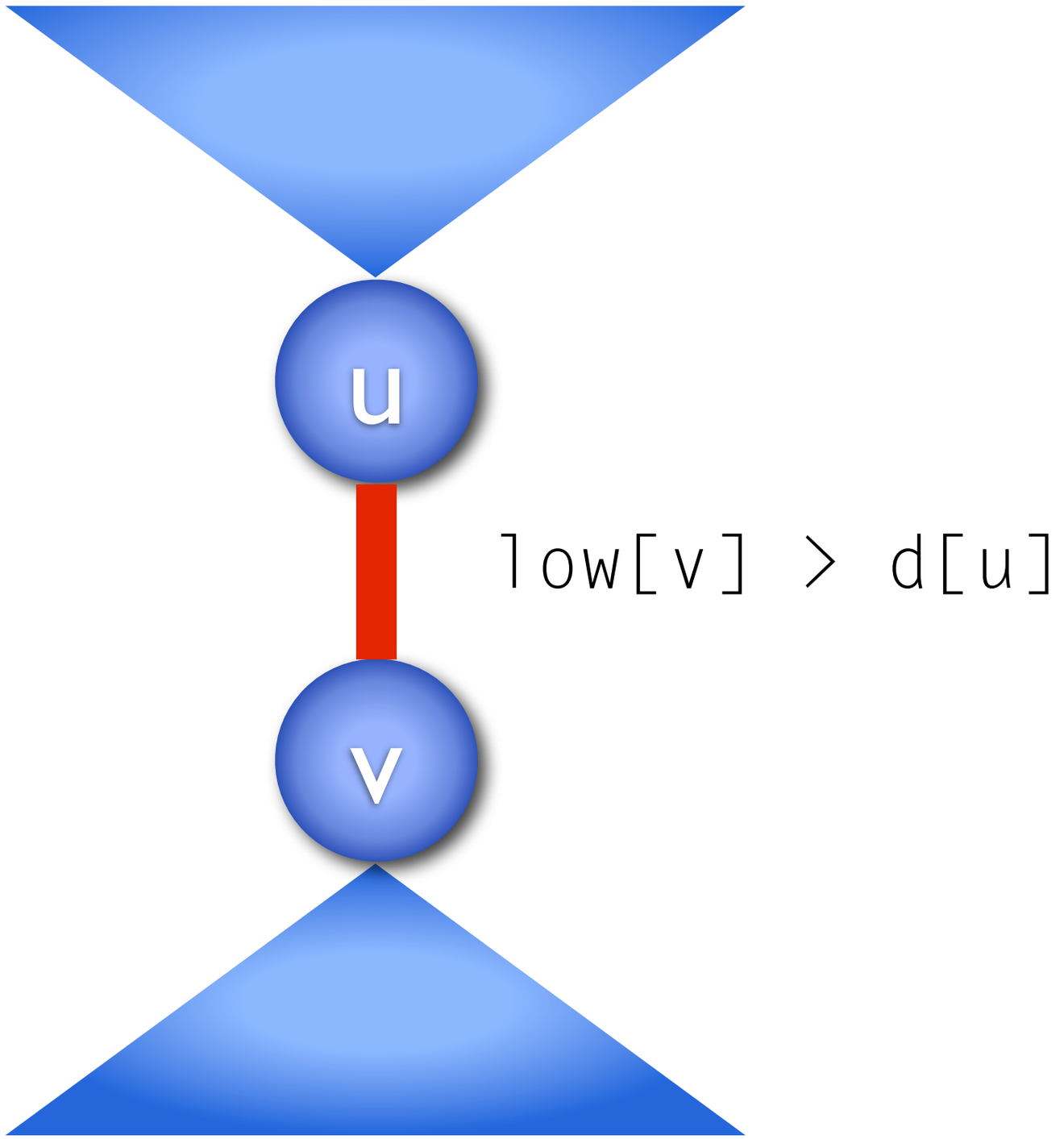
Brücken - Konzept

- führe DFS auf G durch und berechne aktuelle Suchtiefe `d[]`

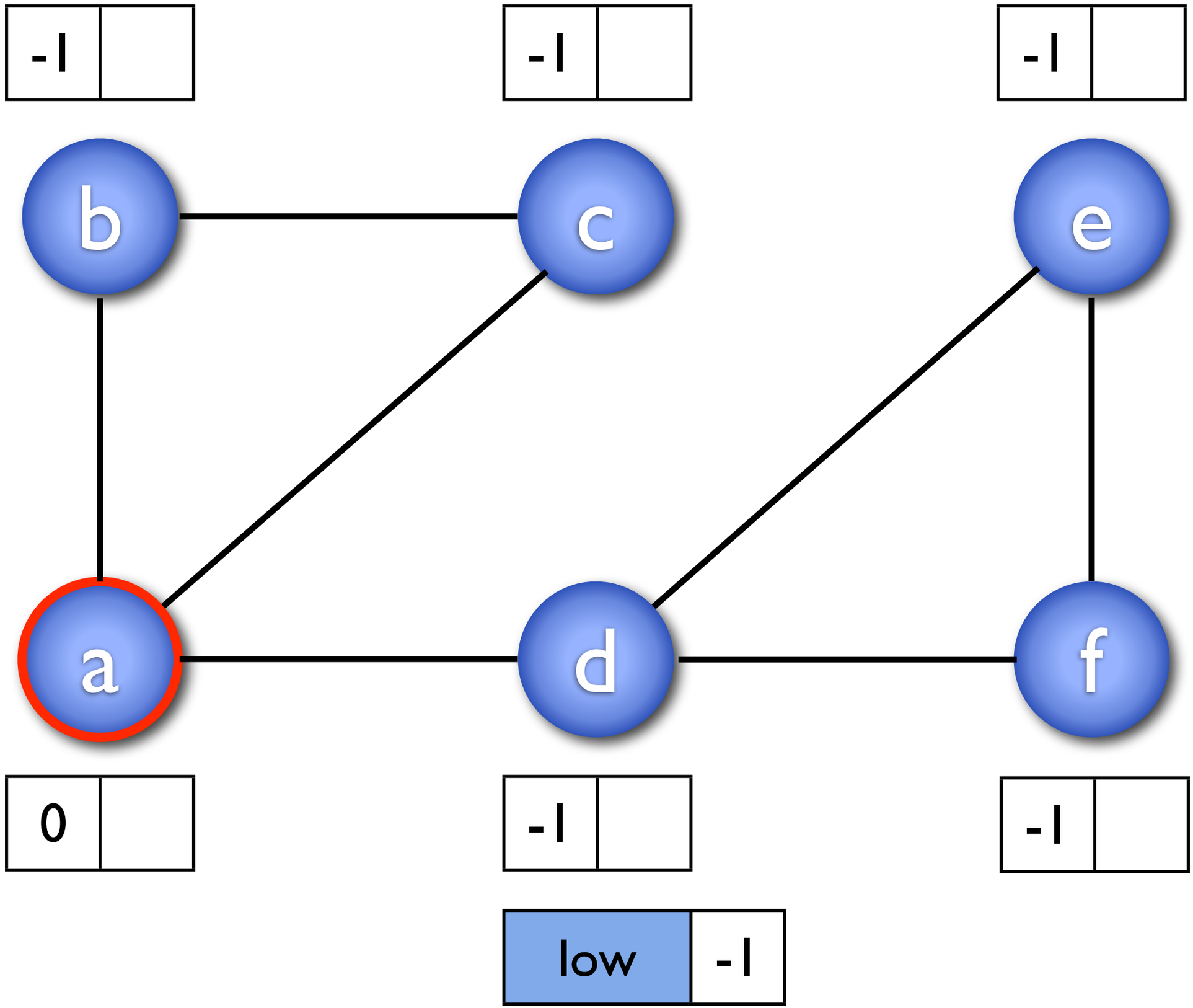
- min. Suchtiefe von Knoten u, die von v über eine beliebige Folge von Baum- und eine Rückwärtskante erreichbar sind:
  `low` **von v**

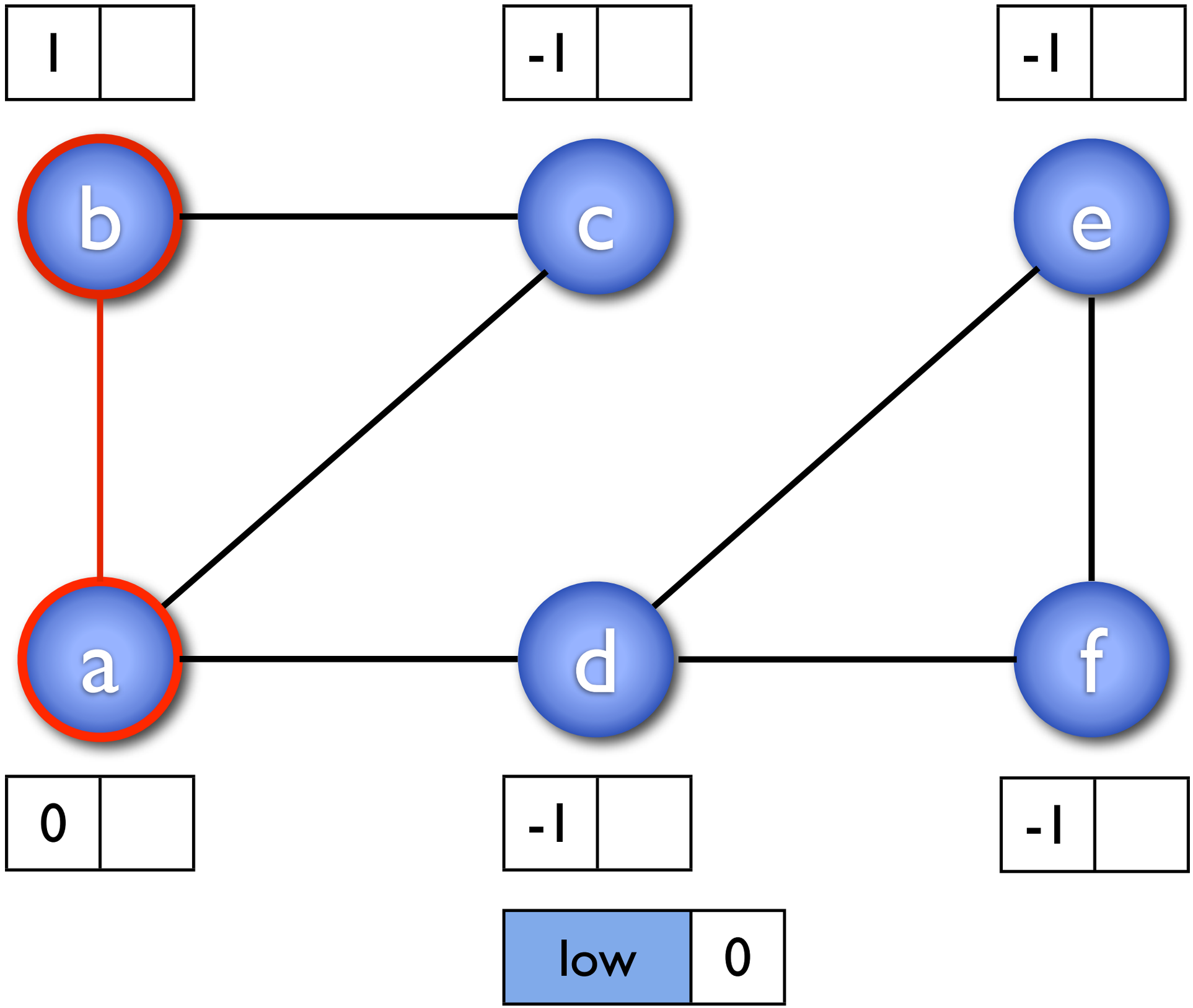
Brücken Konzept

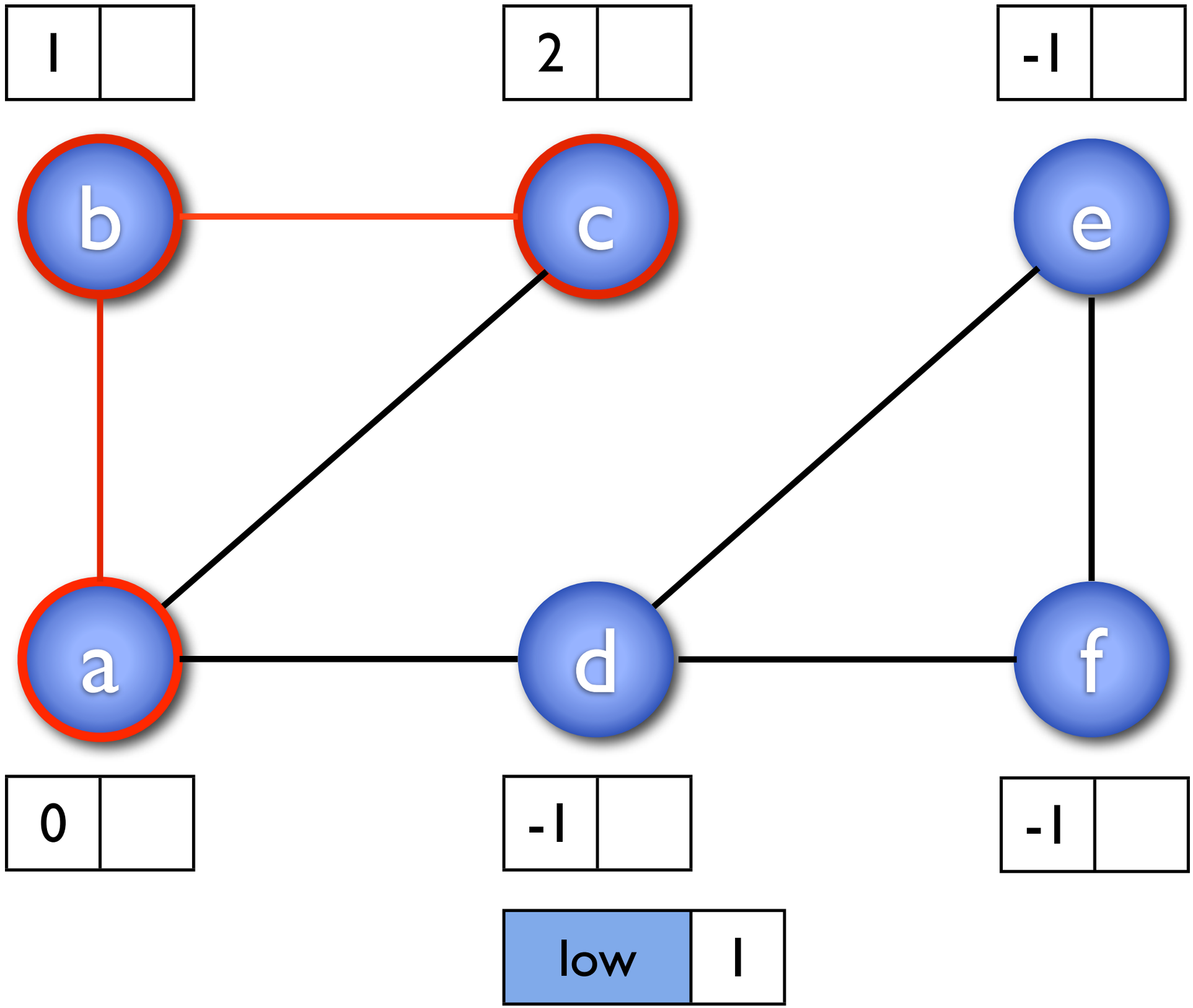
- eine Baumkante e = (u, v) ist eine Brücke, **falls** `low[v] > d[u]`
- gibt es keine andere Möglichkeit von u nach v zu gelangen außer e zu benutzen?
  ja? e ist Brücke

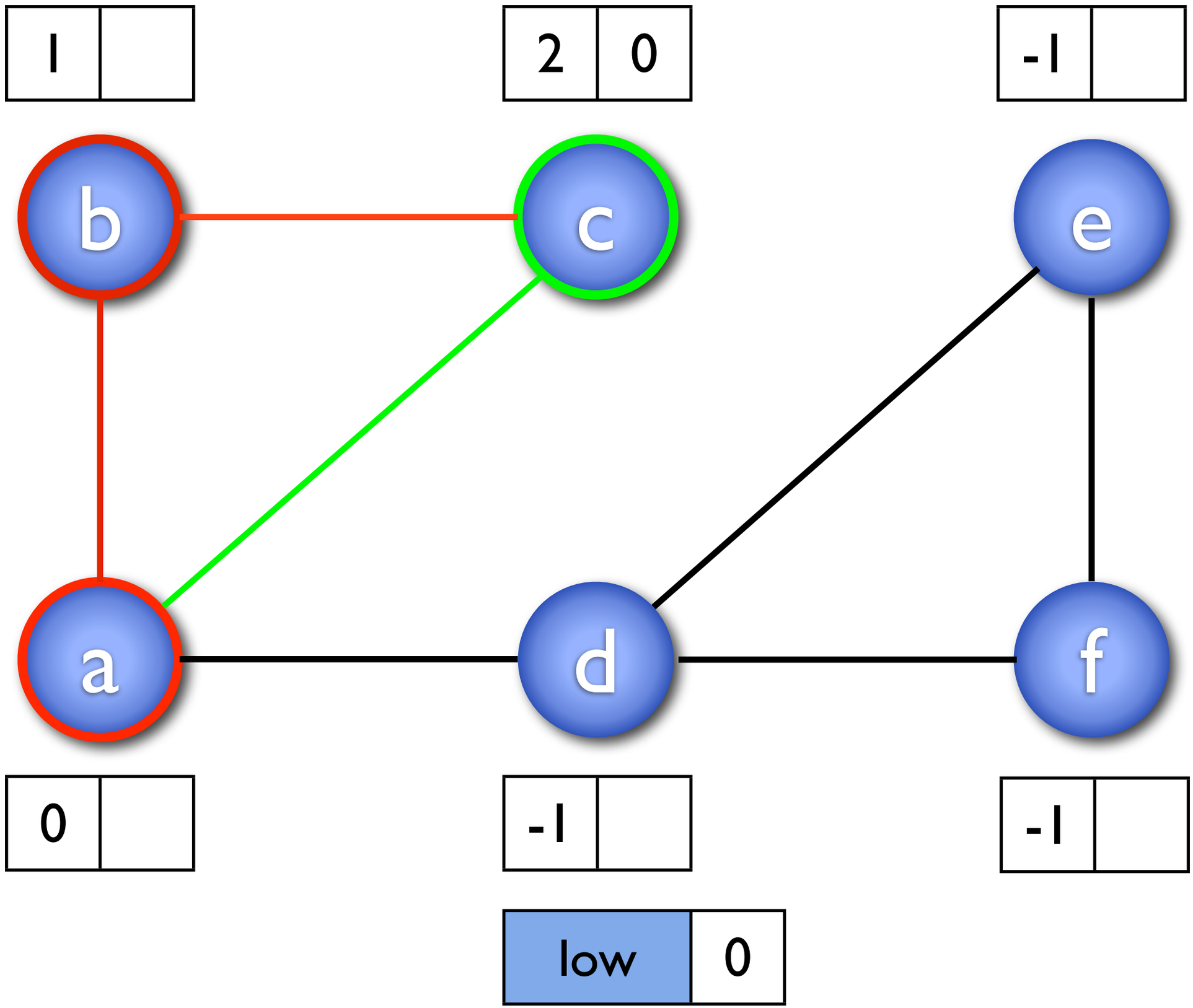

u
v
low[v] > d[u]

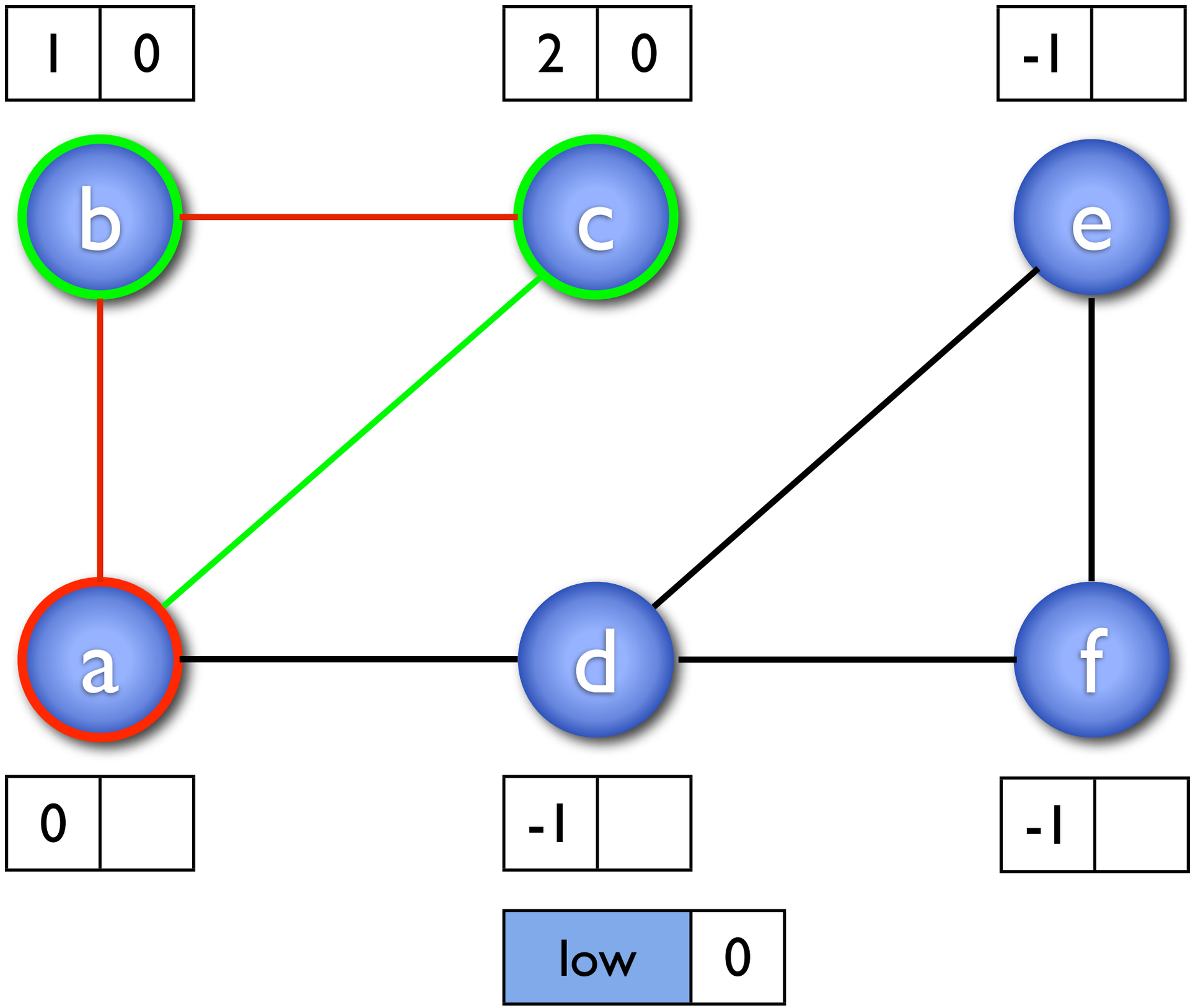
```
Bridges(G)
for alle Knoten u ∈ V[G]  // Initialisierung
  do d[u] ← -1
for alle Knoten u ∈ V[G]
  do if d[u] = -1
    then Bridges-Visit(u, 0, -1)

Bridges-Visit(from, cd, parent)
low ← d[u]
d[u] ← cd
for alle v ∈ Adj[u] mit v ≠ parent
  do if d[v] = -1
     then i ← Bridges-Visit(v, cd+1, u)
          if i > cd then Brücke(u,v)
          low ← Min(low, i)
     else low ← Min(low, d[v])
return low
```

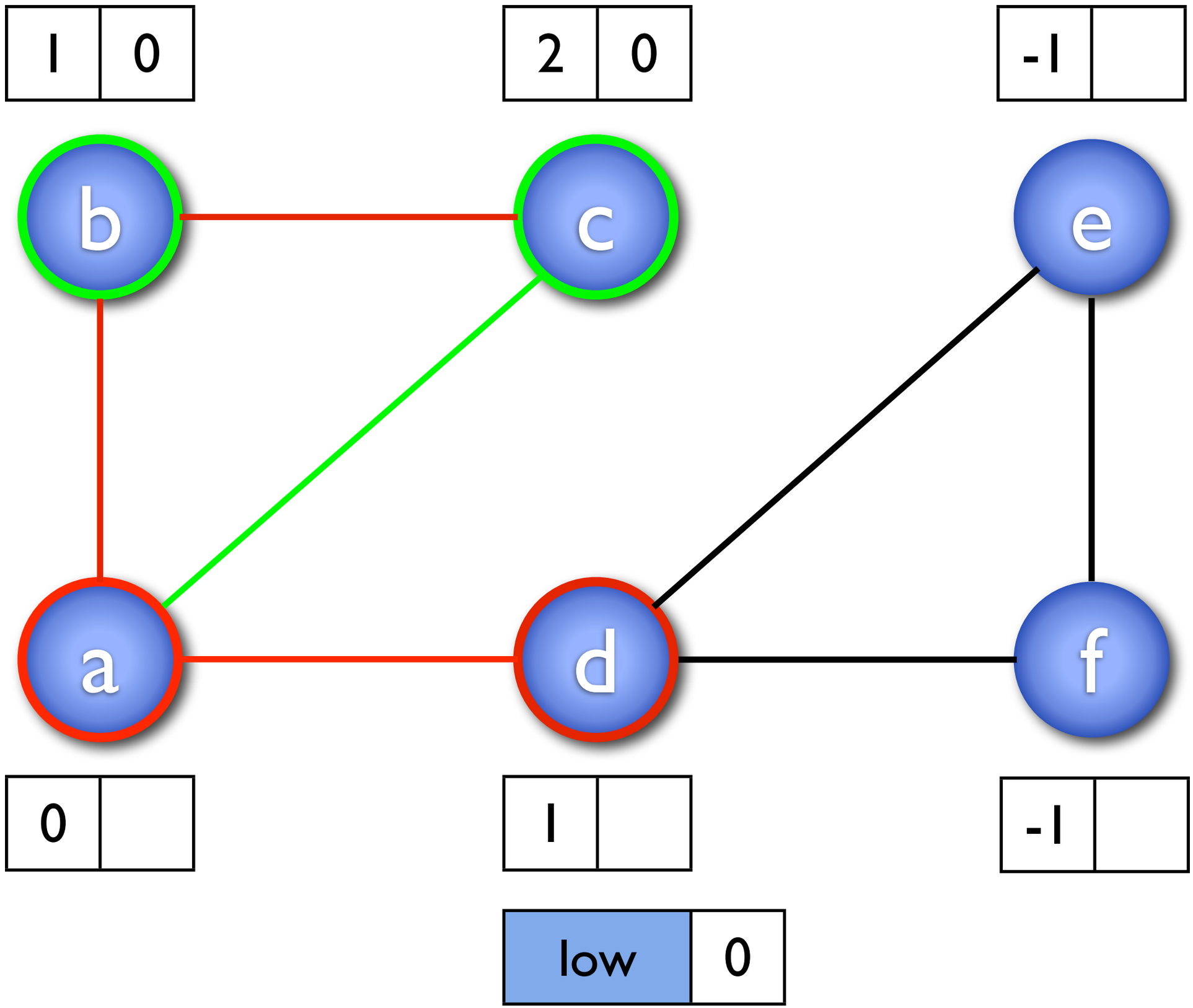

-1
-1
-1
b
c
e
a
d
f
0
-1
-1
low
-1

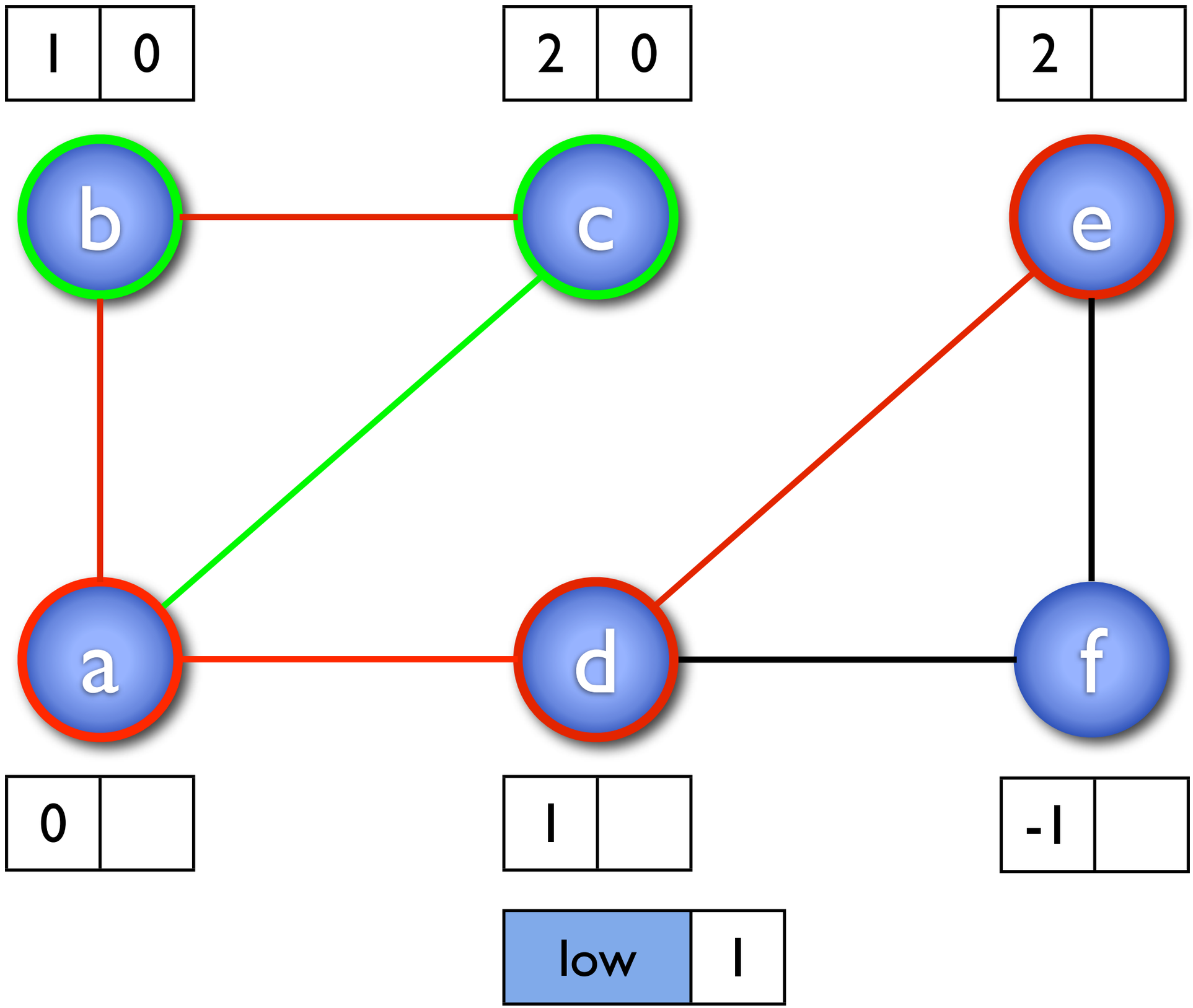

1
-1
-1
b
c
e
a
d
f
0
-1
-1
low
0

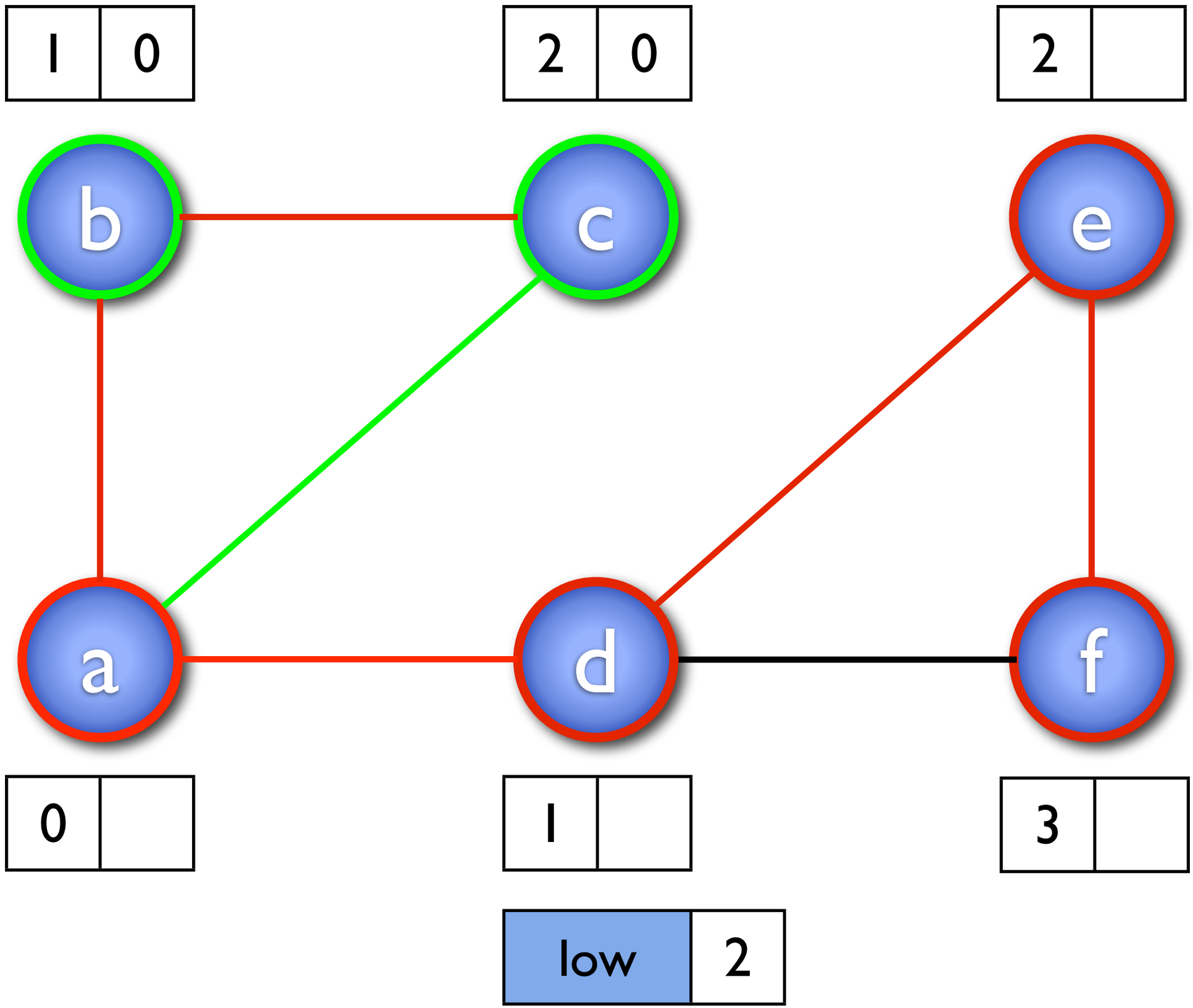

1
2
-1
b
c
e
a
d
f
0
-1
-1
low
1

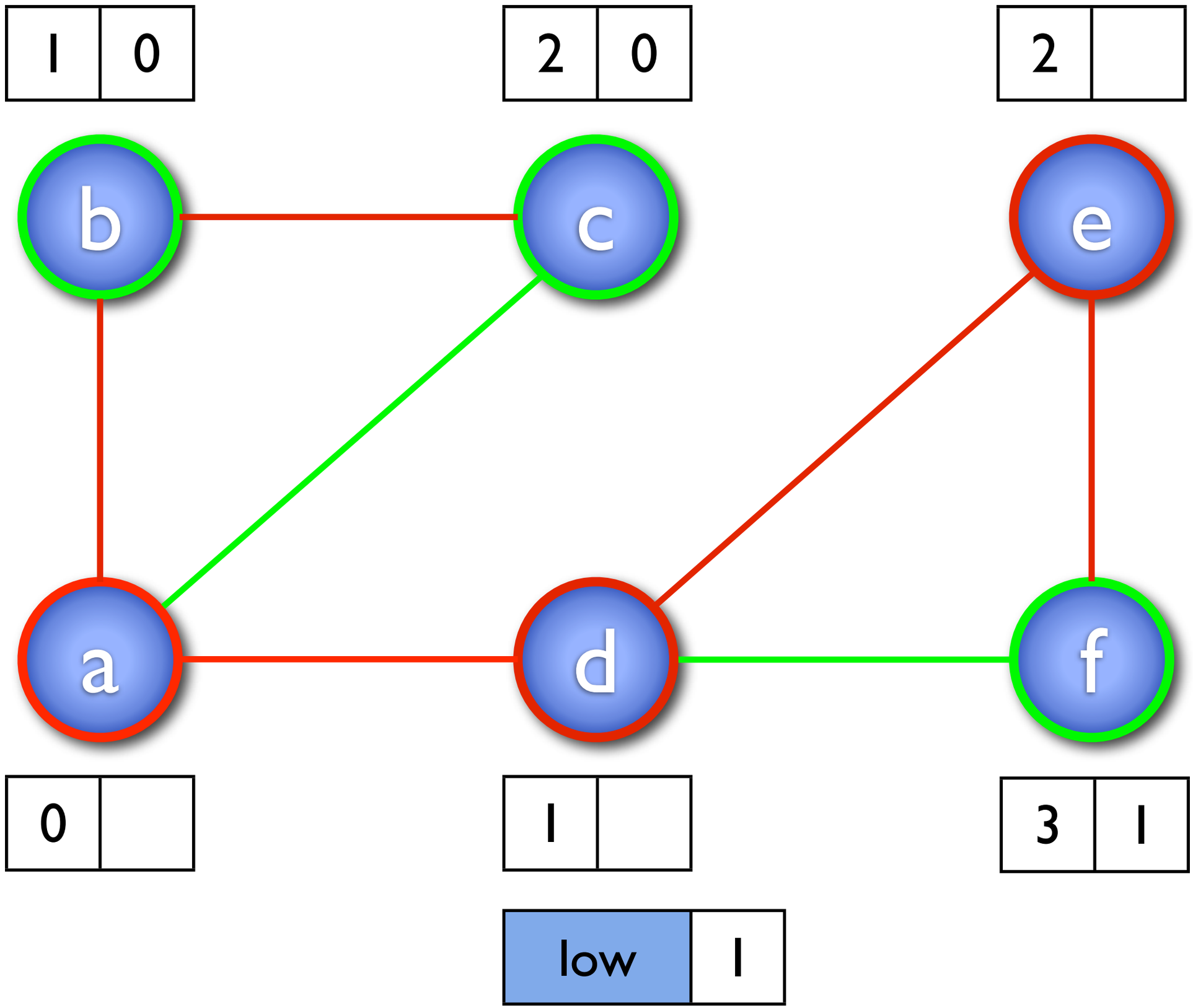

1
2
0
-1
b
c
e
a
d
f
0
-1
-1
low
0

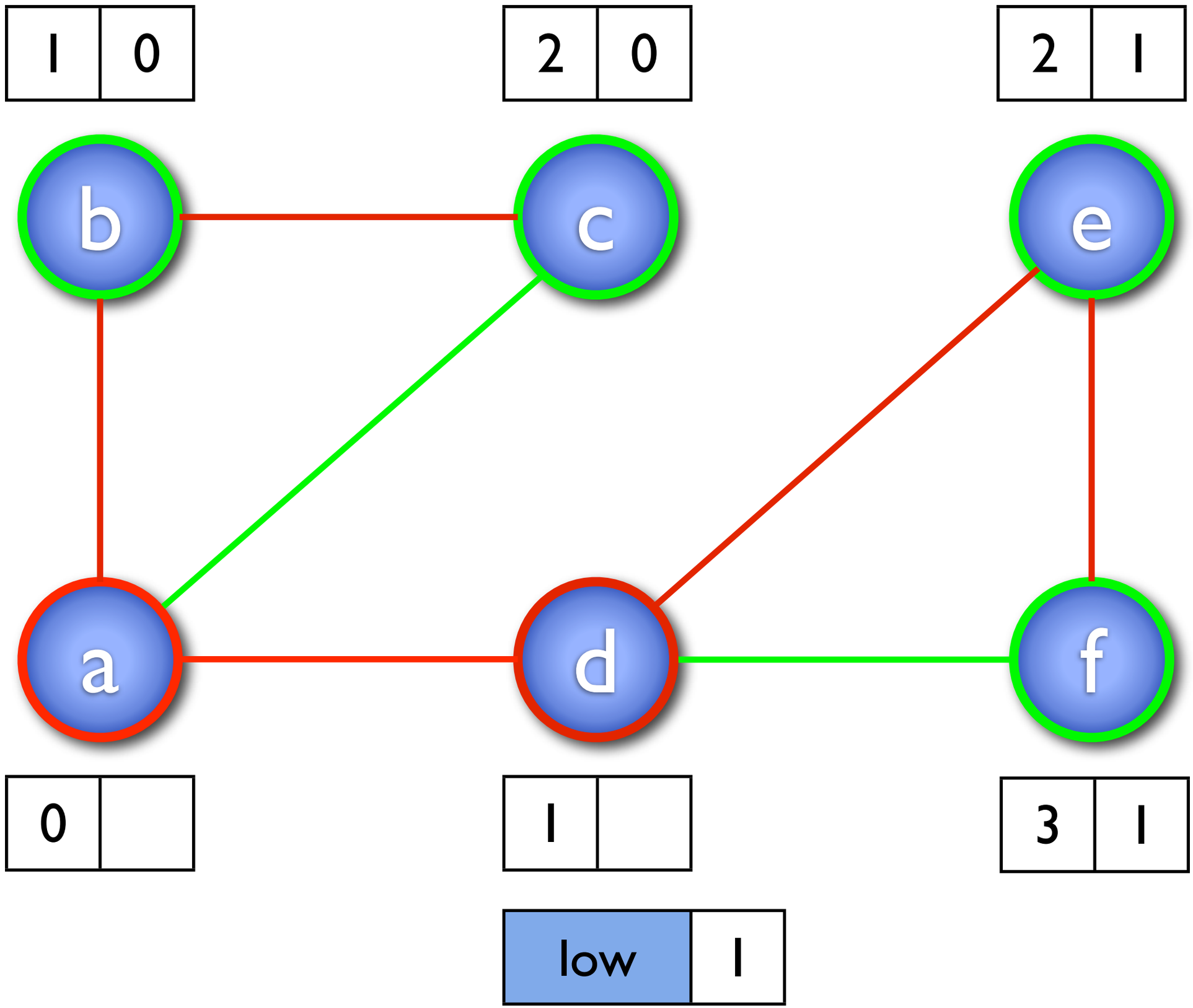

1
0
2
0
-1
b
c
e
a
d
f
0
-1
-1
low
0

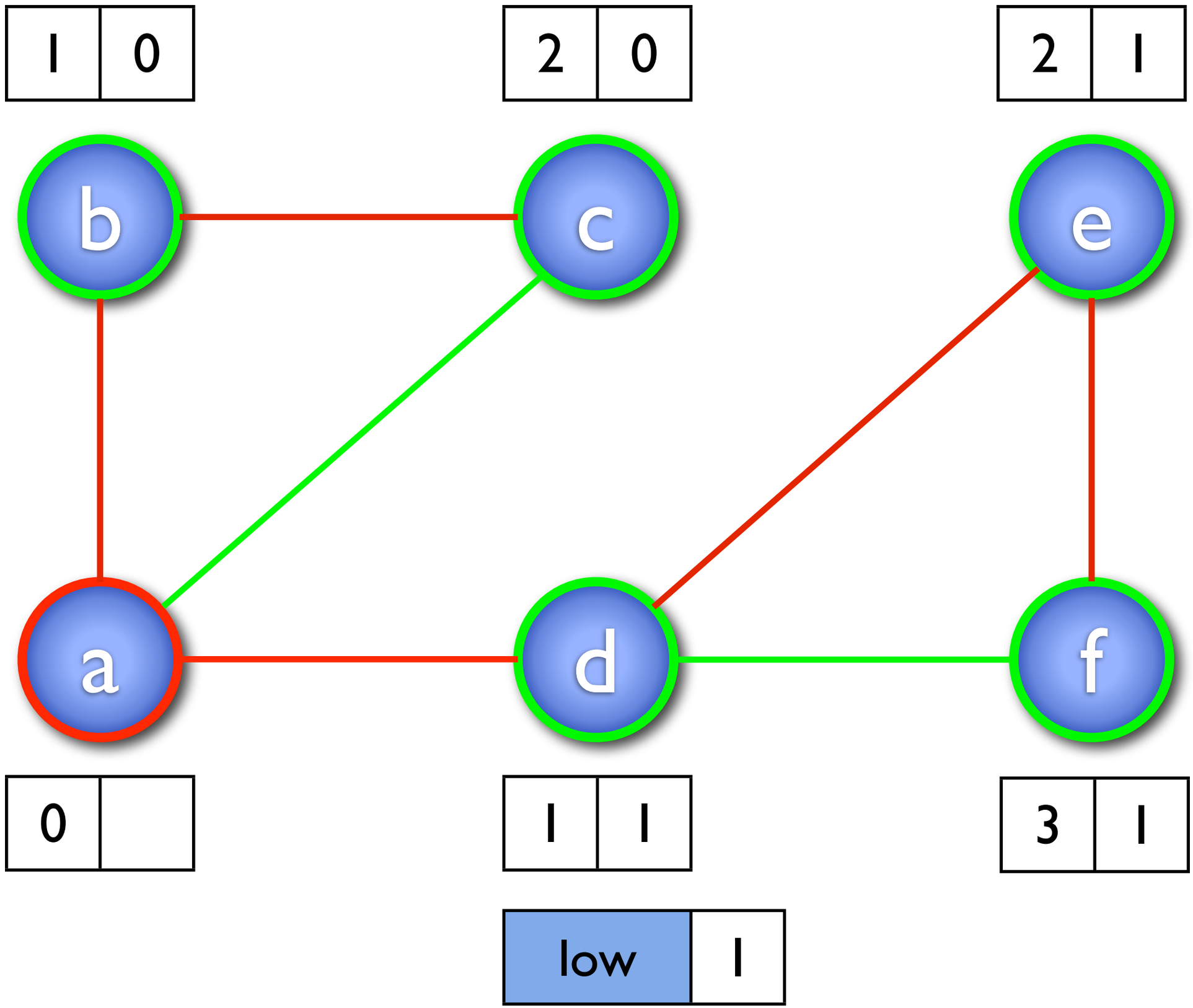

1
0
2
0
-1
b
c
e
a
d
f
0
1
-1
low
0

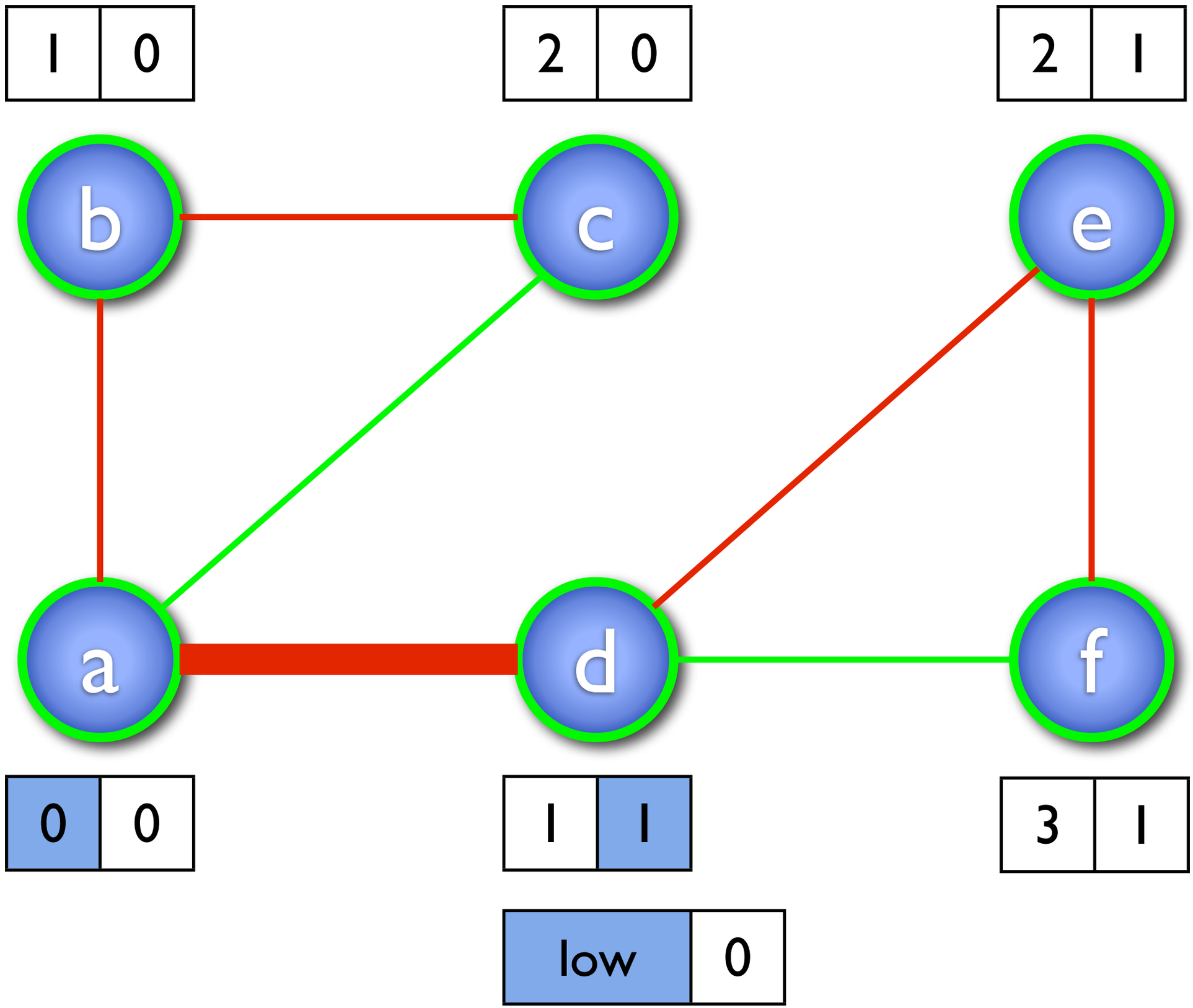

1
0
2
0
2
b
c
e
a
d
f
0
1
-1
low
1


1
0
2
0
2
b
c
e
a
d
f
0
1
3
low
2


1
0
2
0
2
b
c
e
a
d
f
0
1
3
1
low
1


1
0
2
0
2
1
b
c
e
a
d
f
0
1
3
1
low
1


1
0
2
0
2
1
b
c
e
a
d
f
0
1
1
3
1
low
1

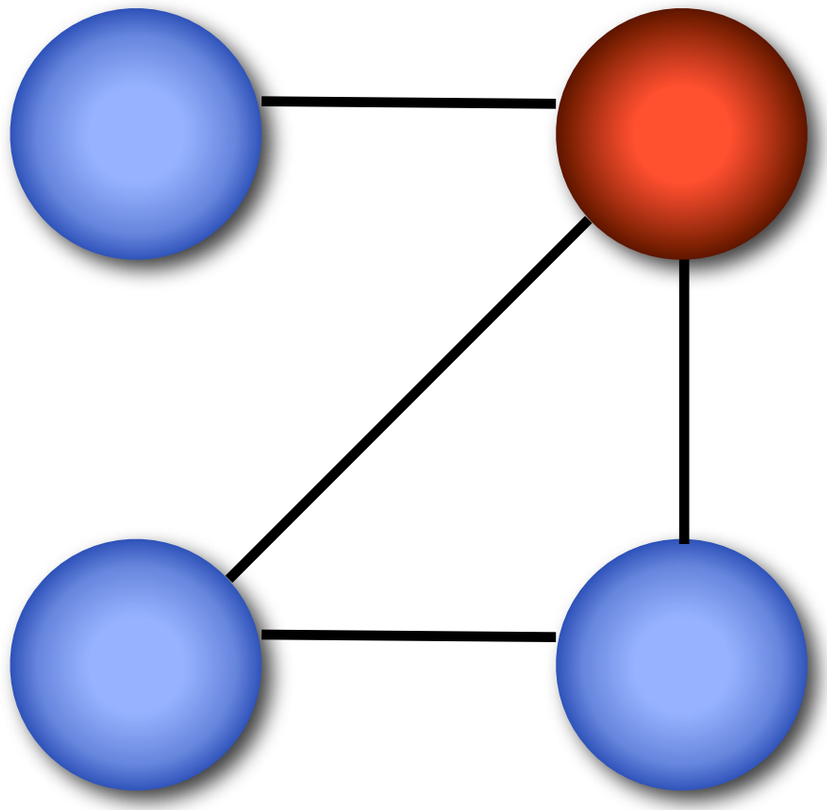

1
0
2
0
2
1
b
c
e
a
d
f
0
0
1
1
3
1
low
0

Brücken - Analyse

- Laufzeit O(|V| + |E|) (DFS)
- Ergebnis: alle Brücken des Graphen

Artikulationspunkte

- eine Artikulationspunkt (kritischer Knoten) in einem Graph ist ein Knoten, dessen Entfernen die Zahl der Zusammenhangskomponenten erhöht



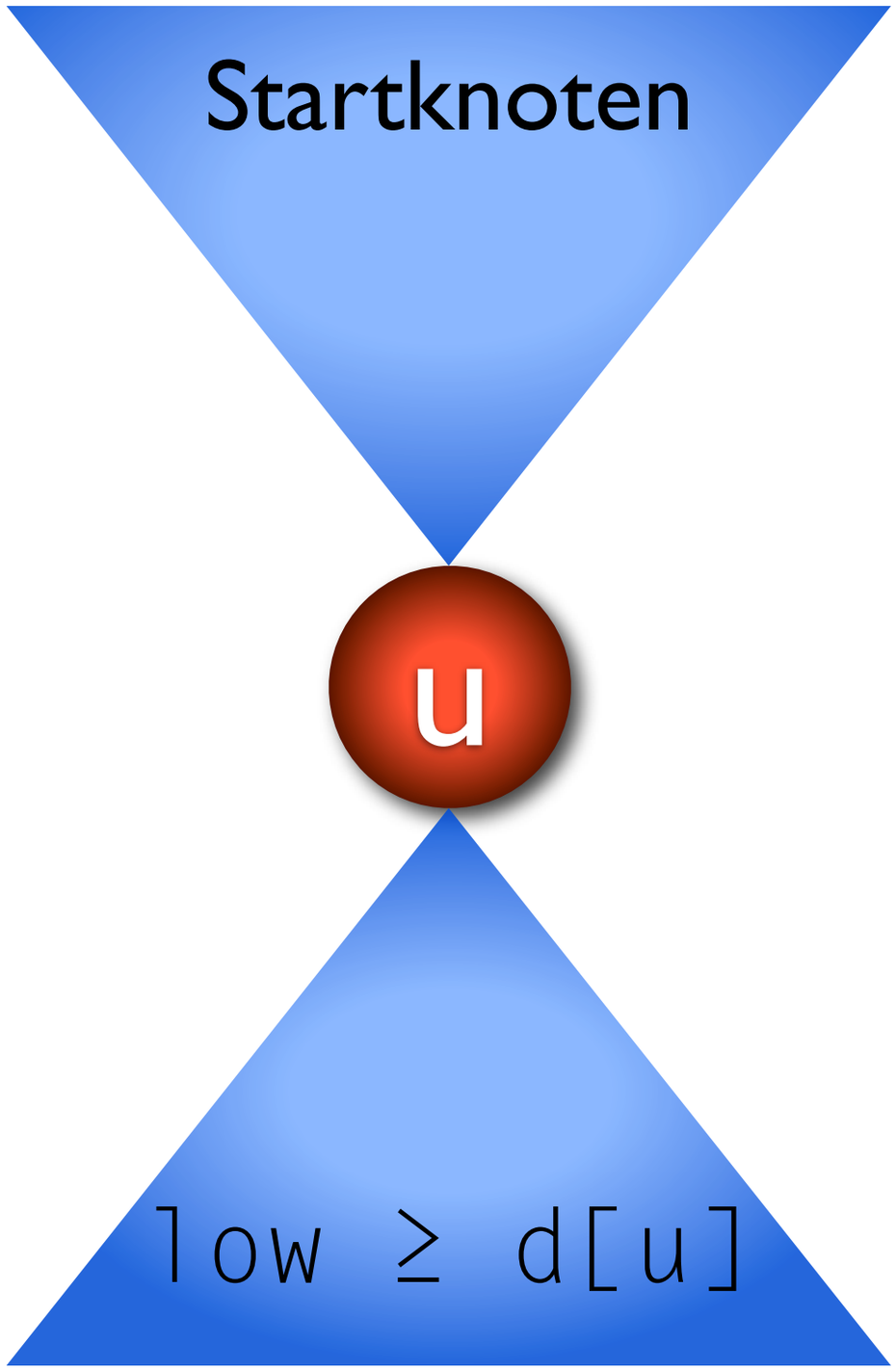
Artikulationspunkte - Konzept

- führe DFS auf G durch und berechne aktuelle Suchtiefe `d[]`

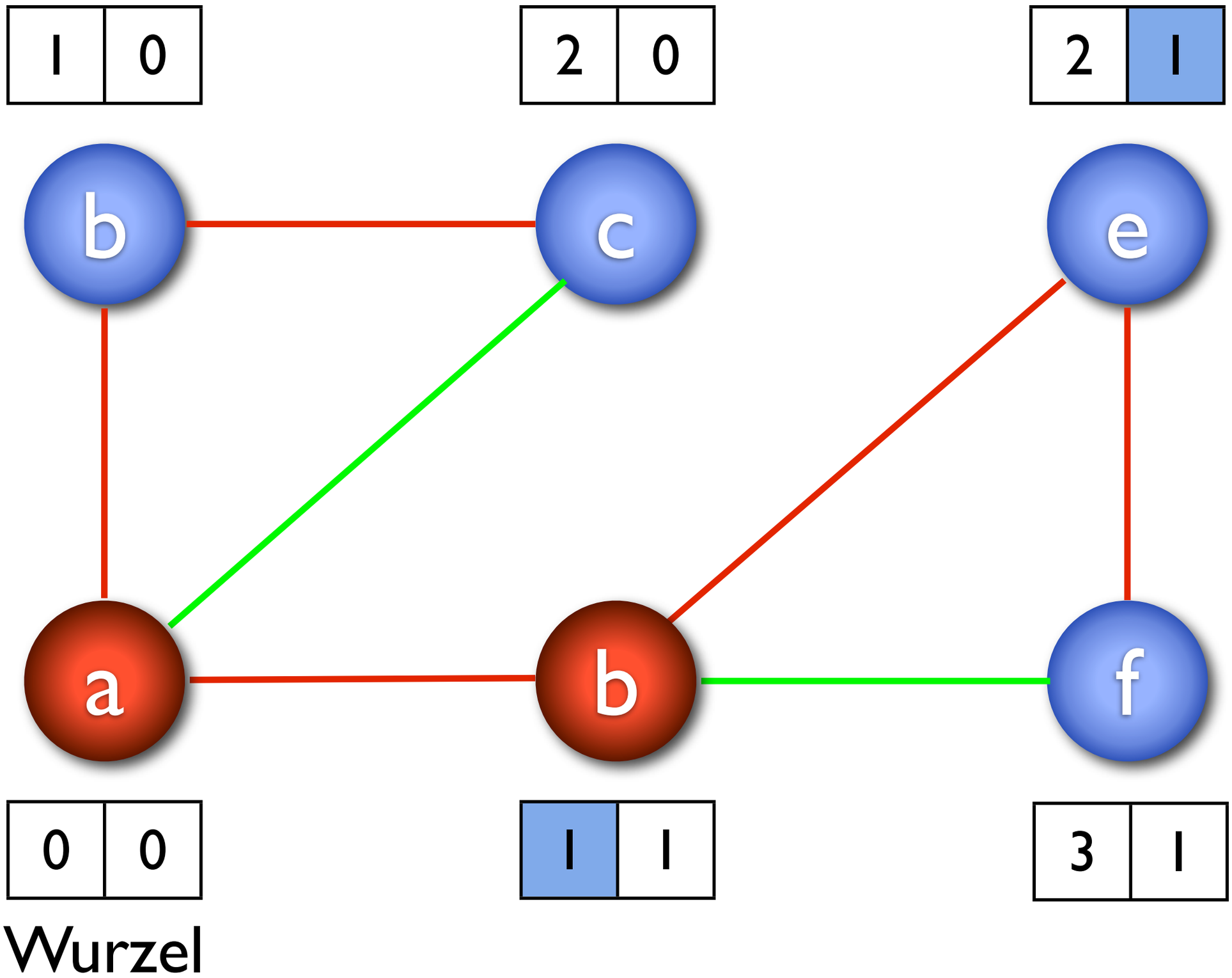
- min. Suchtiefe von Knoten u, die von v über eine beliebige Folge von Baum- und eine Rückwärtskante erreichbar sind:
  `low` **von v**

Artikulationspunkte (AP) - Konzept

- Wurzel eines Tiefensuchbaums ist AP, falls sie mehr als ein Kind hat
- jeder andere Knoten u ist AP, falls er ein Kind v hat mit `low[v] ≥ d[u]`
- hat u ein Kind, das mit keinem Knoten, der vor v besucht wurde verbunden ist?
- ja? u ist AP


Startknoten
u
low ≥ d[u]

```
APoints(G)
for alle Knoten u ∈ V[G]
  do d[u] ← -1
for alle Knoten u ∈ V[G]
  do if d[u] = -1
    then APoints(u, 0, -1)

APoints-Visit(from, cd, parent)
low ← d[u]
d[u] ← cd
children ← 0
ap ← false
for alle v ∈ Adj[u] mit v ≠ parent
  do if d[v] = -1
      then children ← children + 1
           i ← APoints-Visit(v, cd+1, u)
           if i ≥ cd then ap ← true
           low ← Min(low, i)
      else low ← Min(low, d[v])
if (ap ∧ cd > 0) ∨ (cd = 0 ∧ children > 1)
then Artikulationspunkt u
return low
```

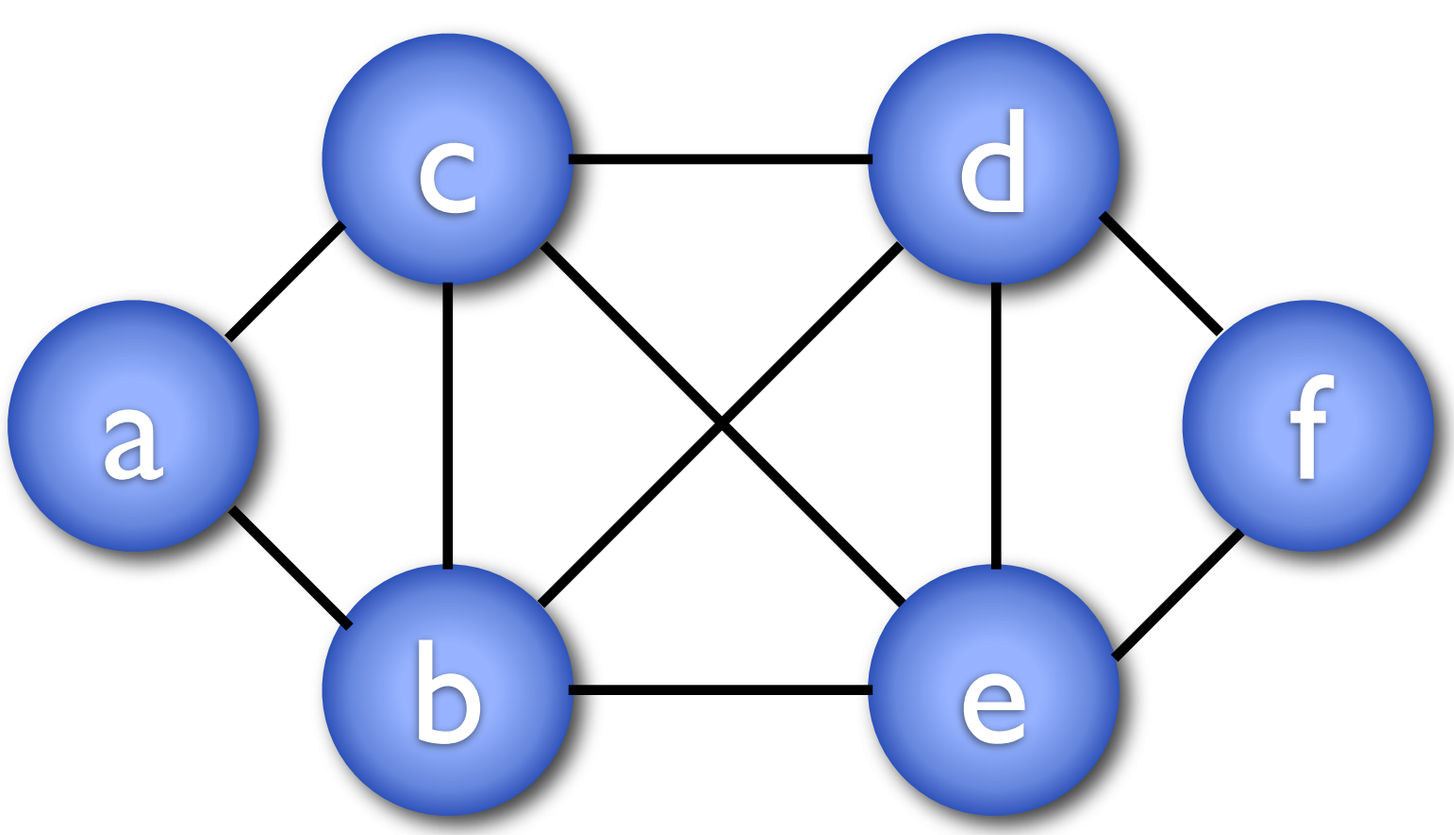

1 0
2 0
2 1
b
c
e
a
b
f
0 0
1 1
3 1
Wurzel

Artikulationspunkte - Analyse

- Laufzeit O(|V| + |E|)
- Ergebnis: alle Artikulationspunkte des Graphen

Euler-Touren

- eine Euler-Tour in einem Graphen, ist eine Tour, die jede Kante besucht

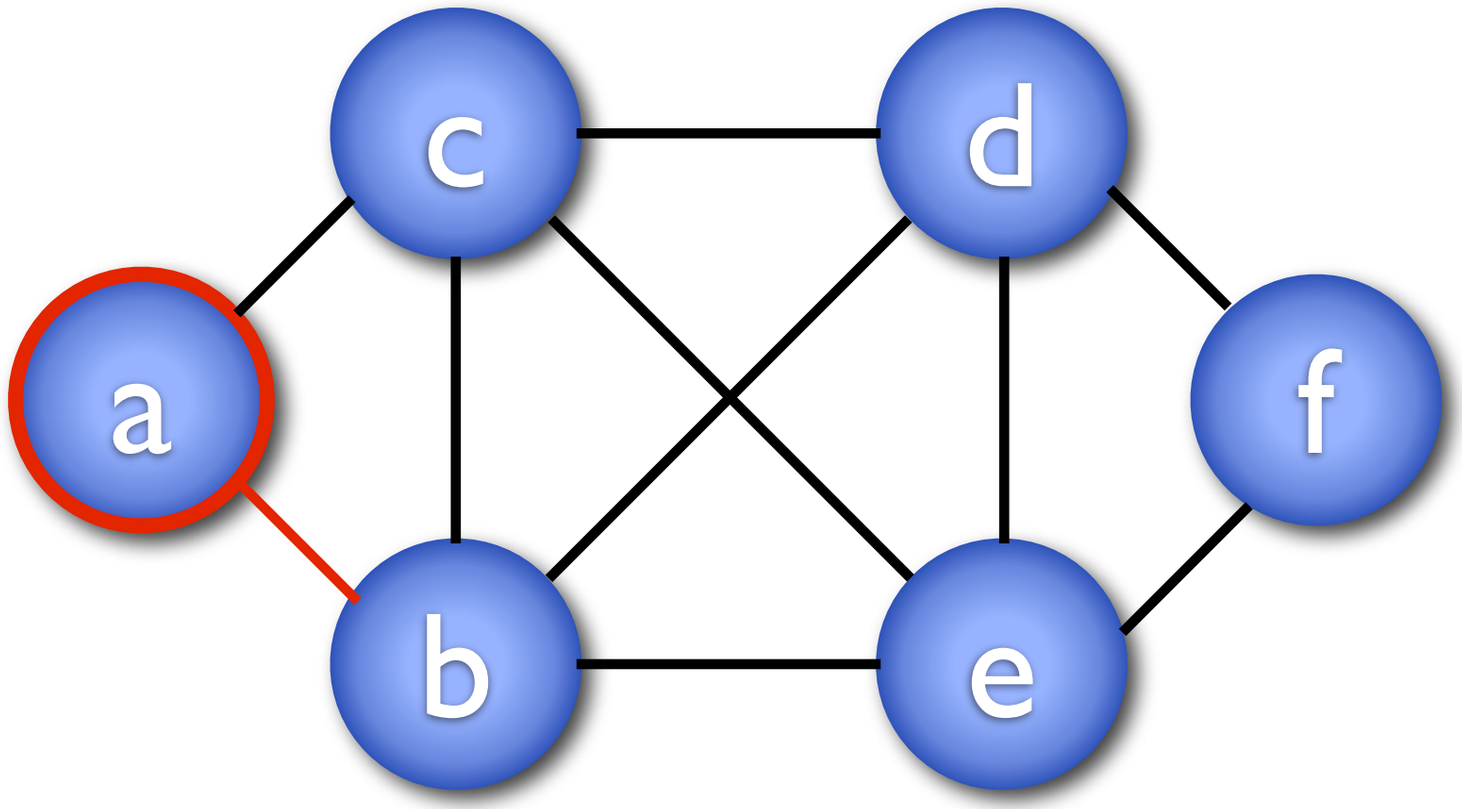


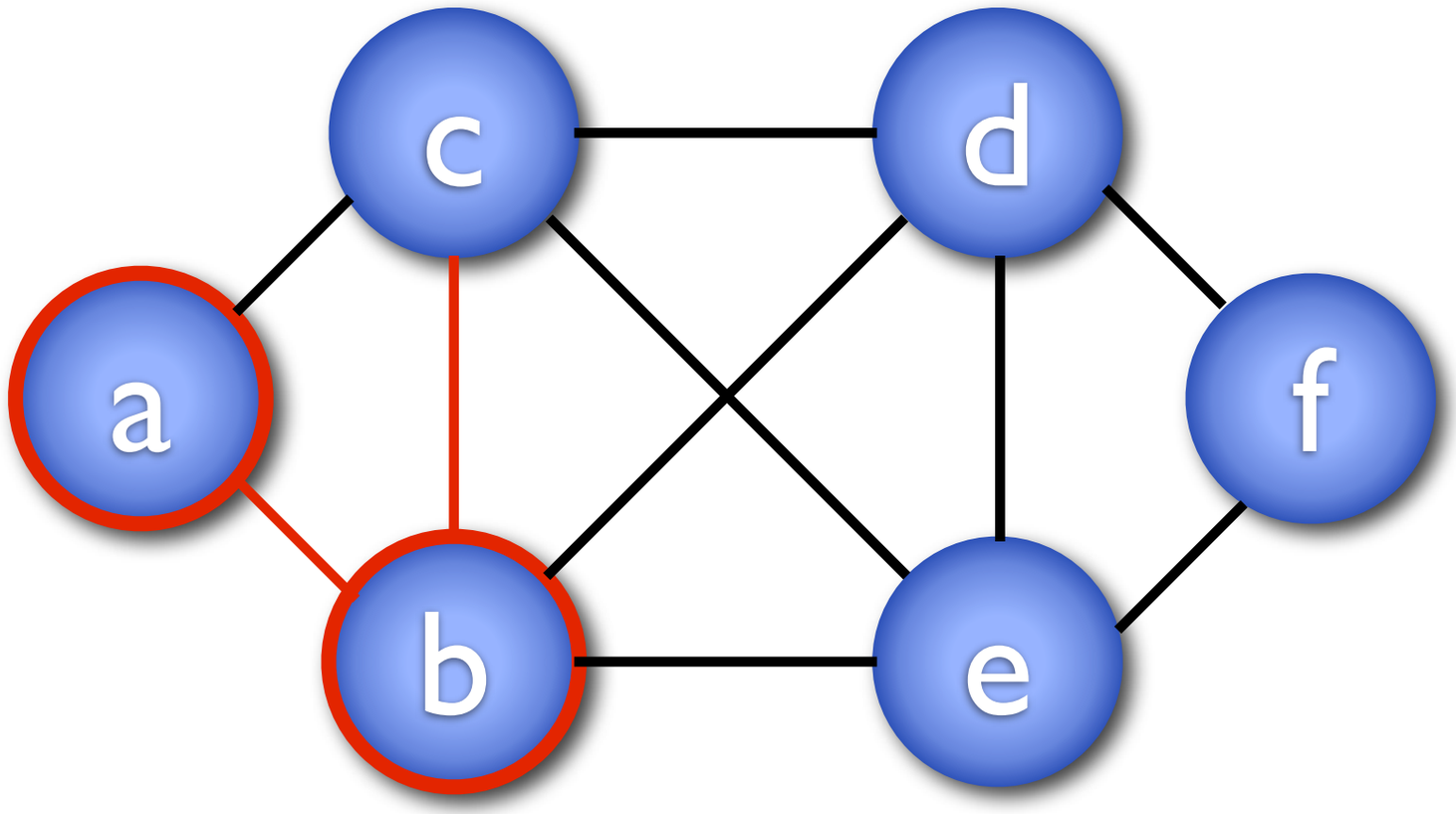
Euler-Tour

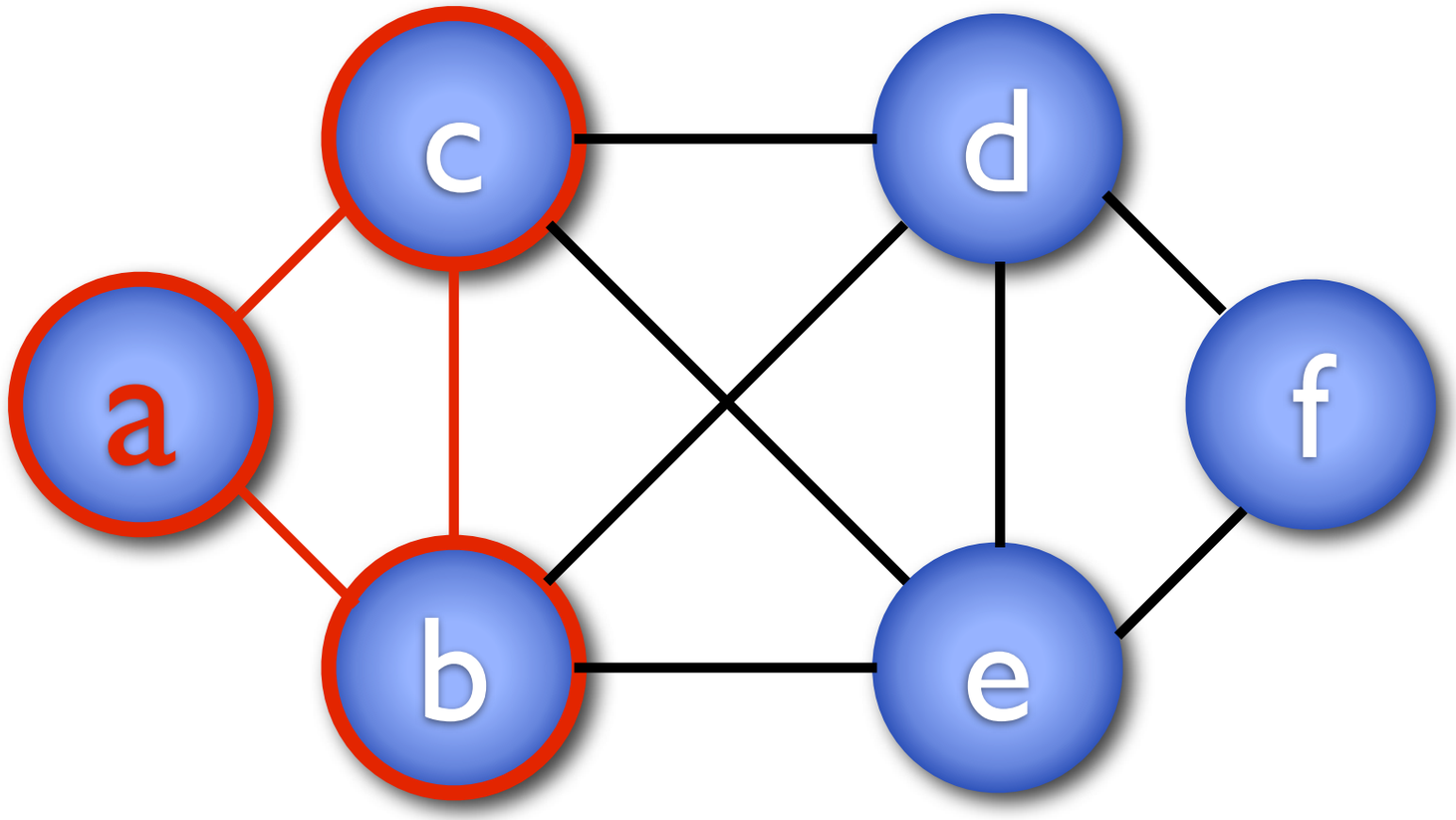
- notwendige und hinreichende Bedingungen
  - geschlossene Euler-Tour: zusammenhängend und alle Knoten haben gerade Grade
  - offene Euler-Tour: zusammenhängend und alle Knoten außer zwei mit ungeradem Grad haben gerade Grade

Euler-Tour - Konzept

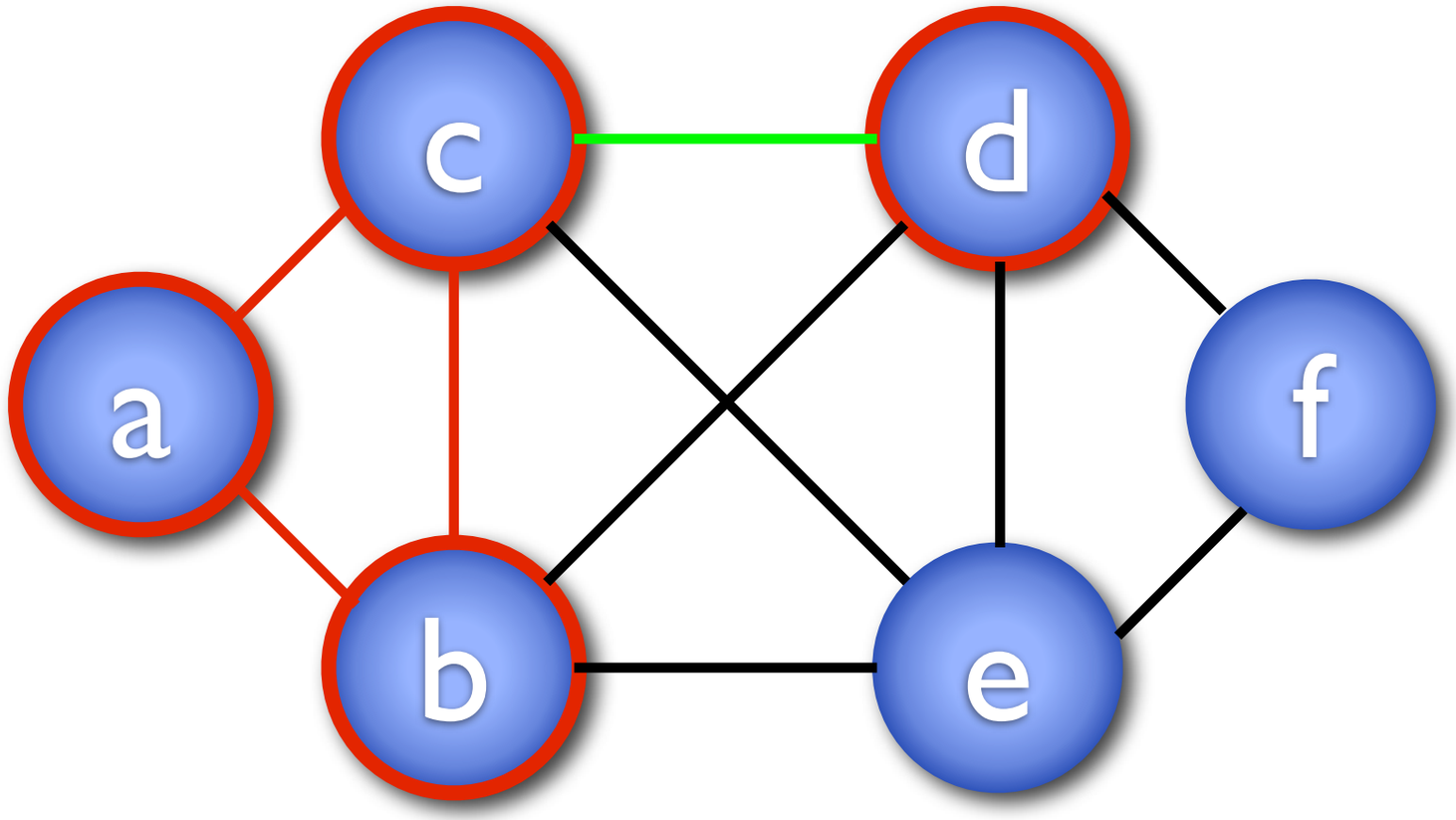
- finde eine geschlossene Euler-Tour:
- DFS von u bis man über Pfad p wieder zu u gelangt
- falls noch unbesuchte Kanten, finde Euler-Tour p´ von u´ (auf p und u´ inzident zu unbesuchter Kante)
- füge p´ in p ein


c
d
a
f
b
e

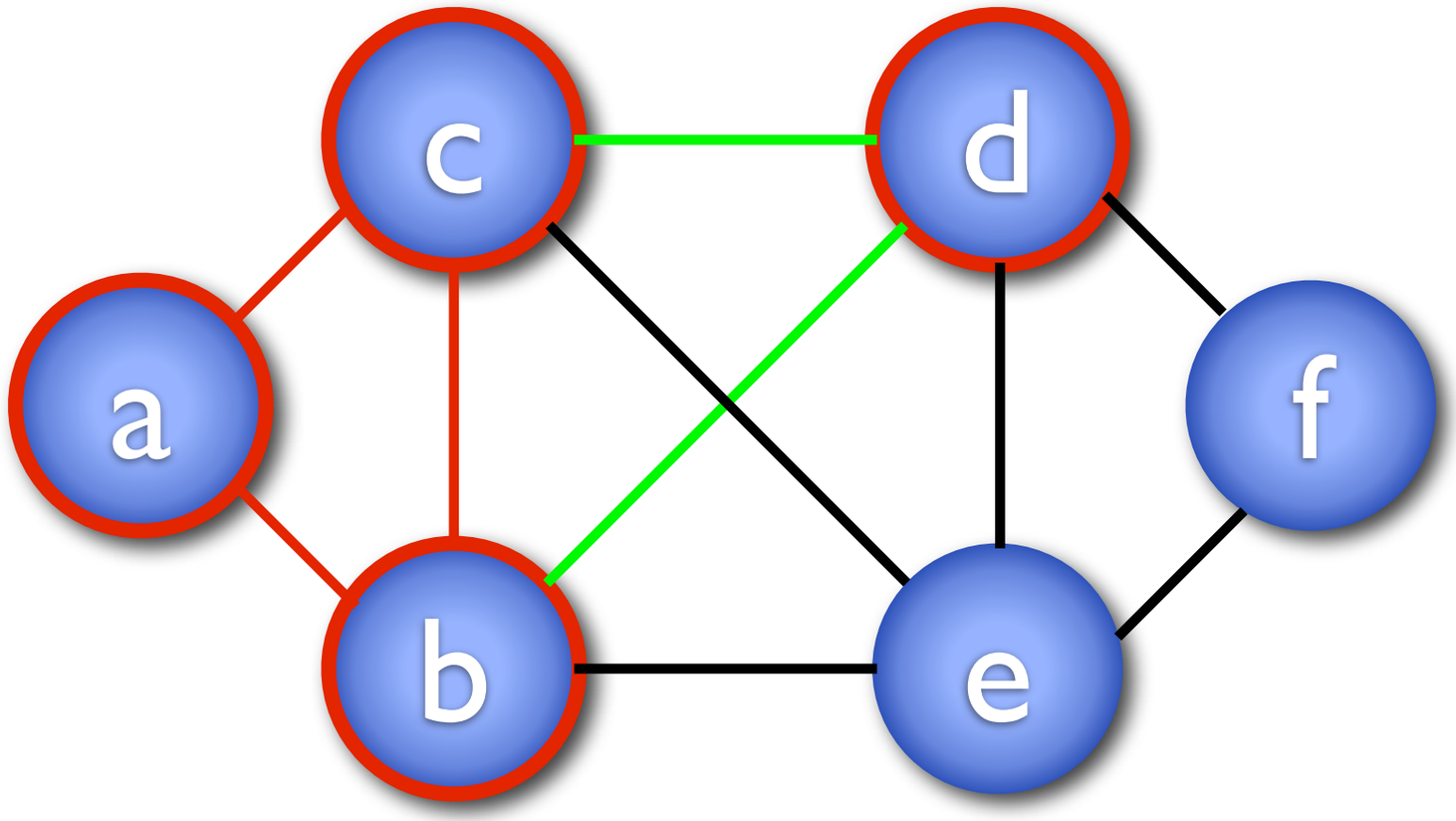

c
d
a
f
b
e



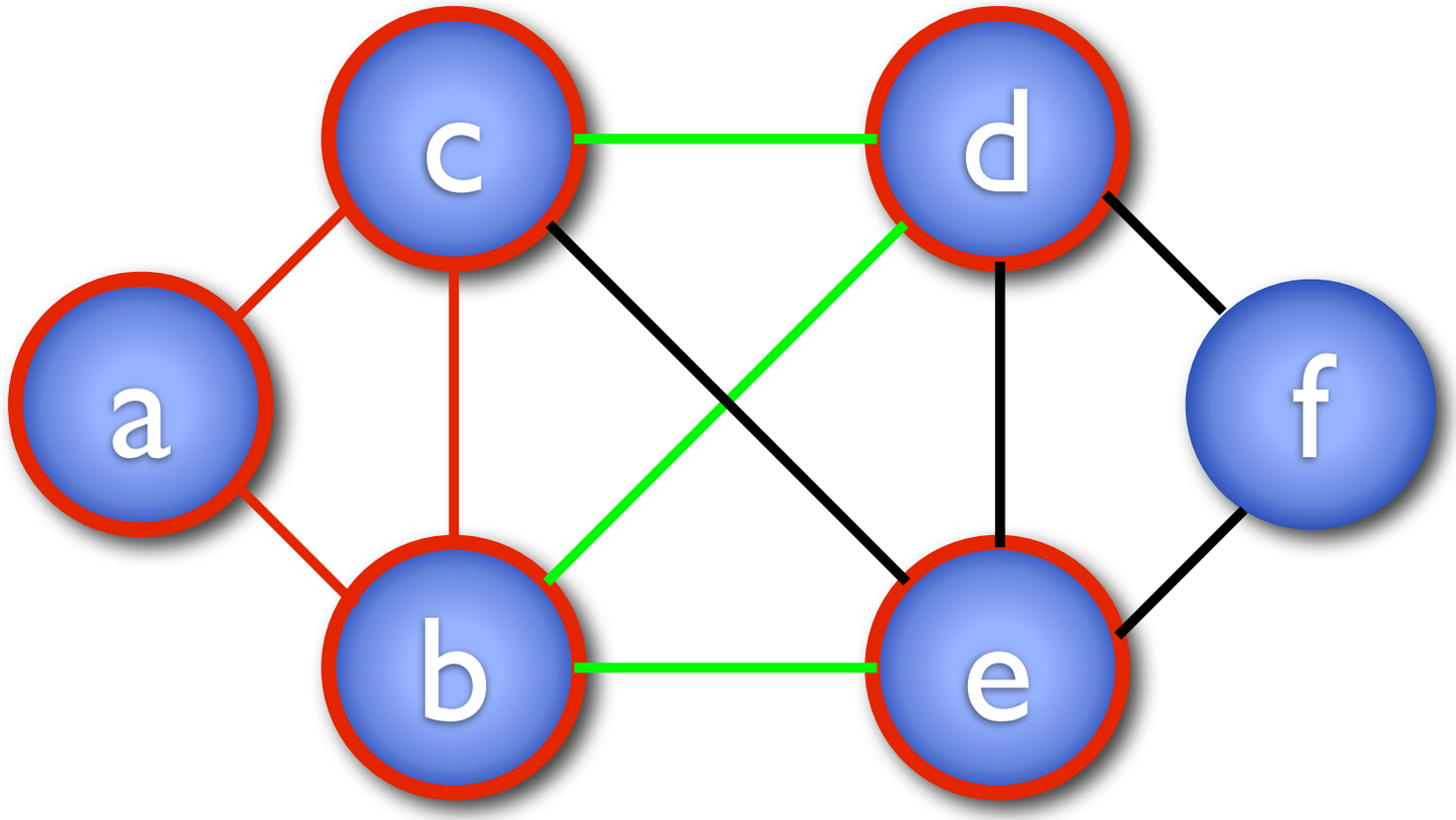
a



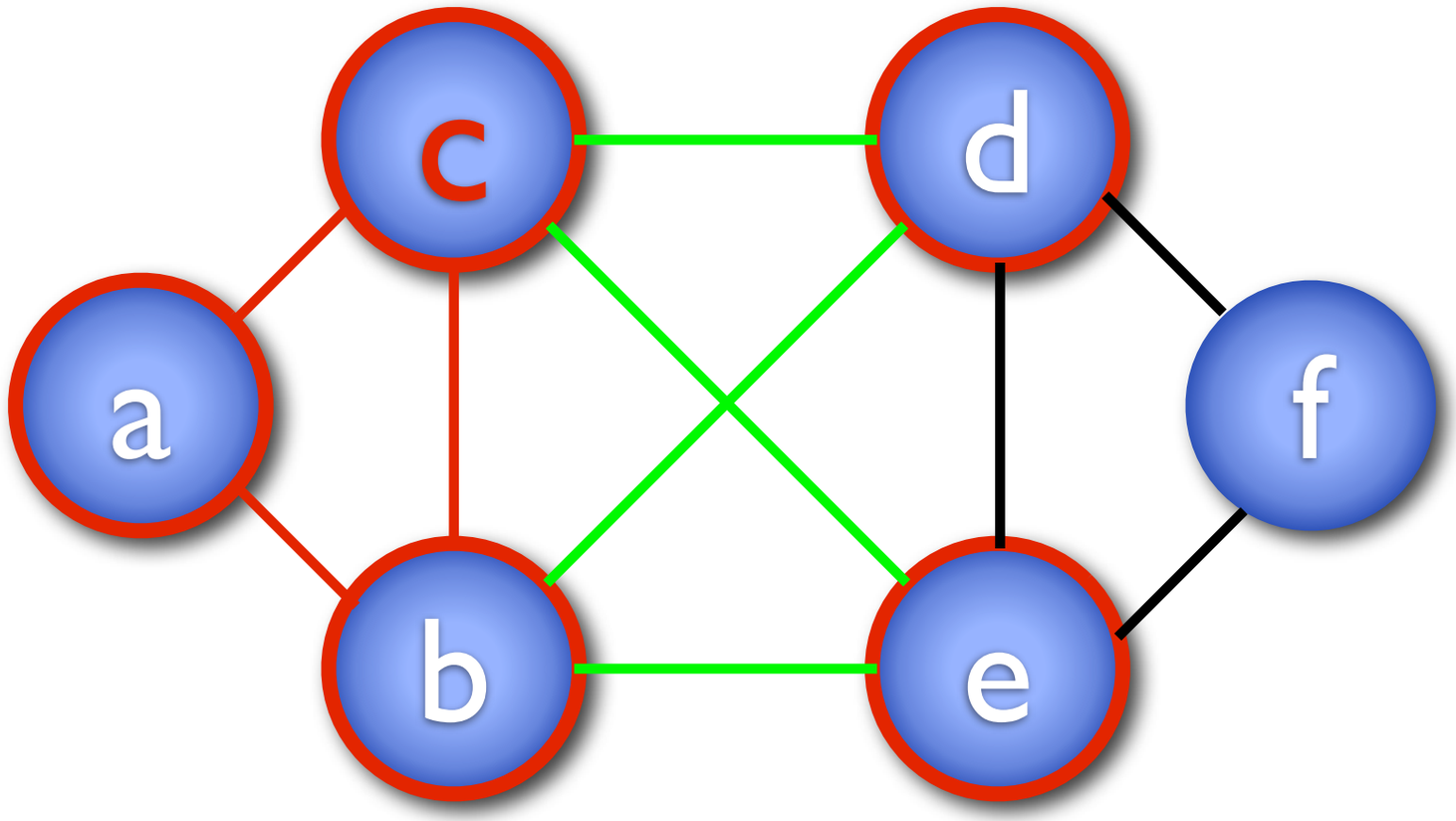
a



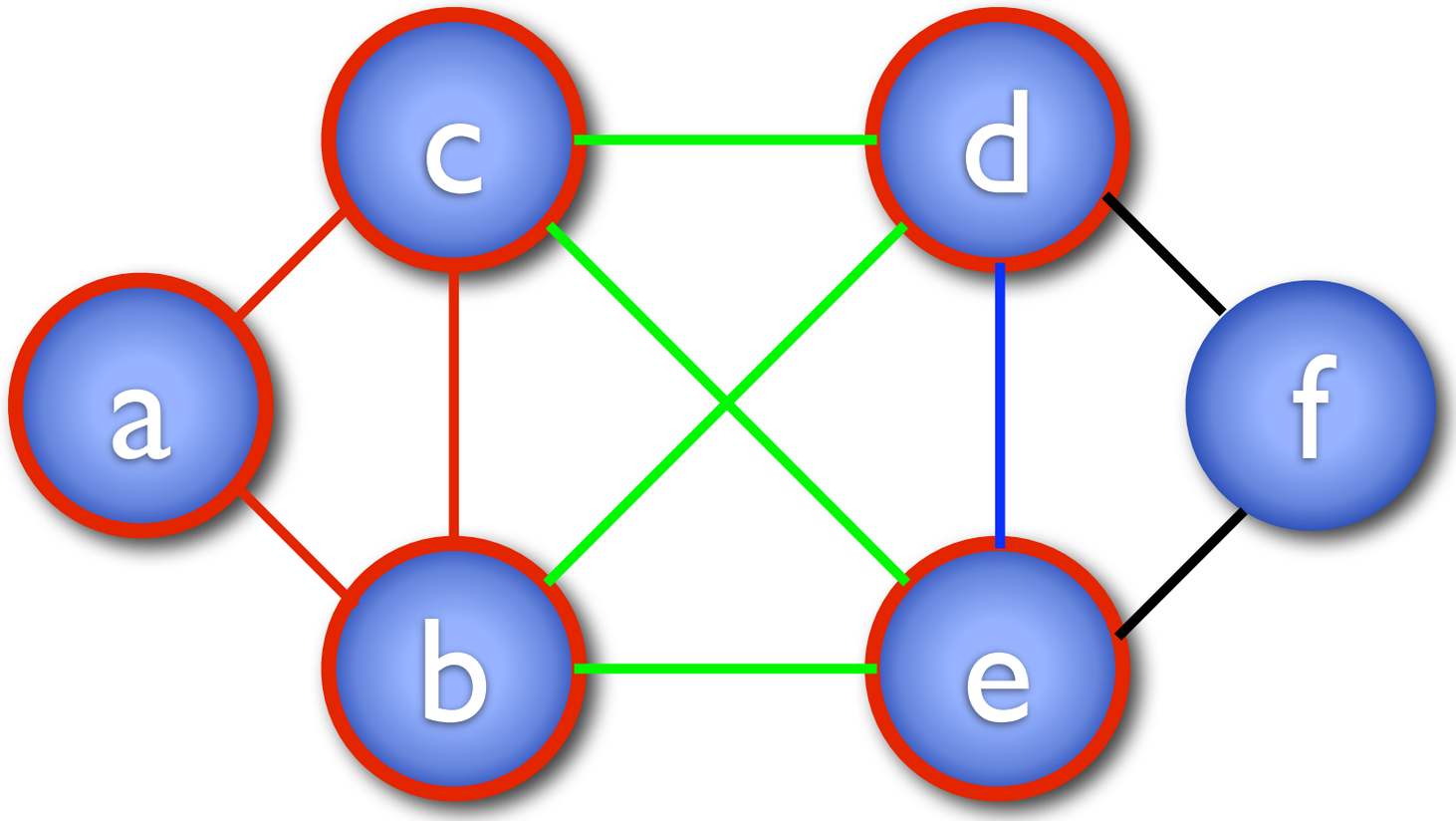
a



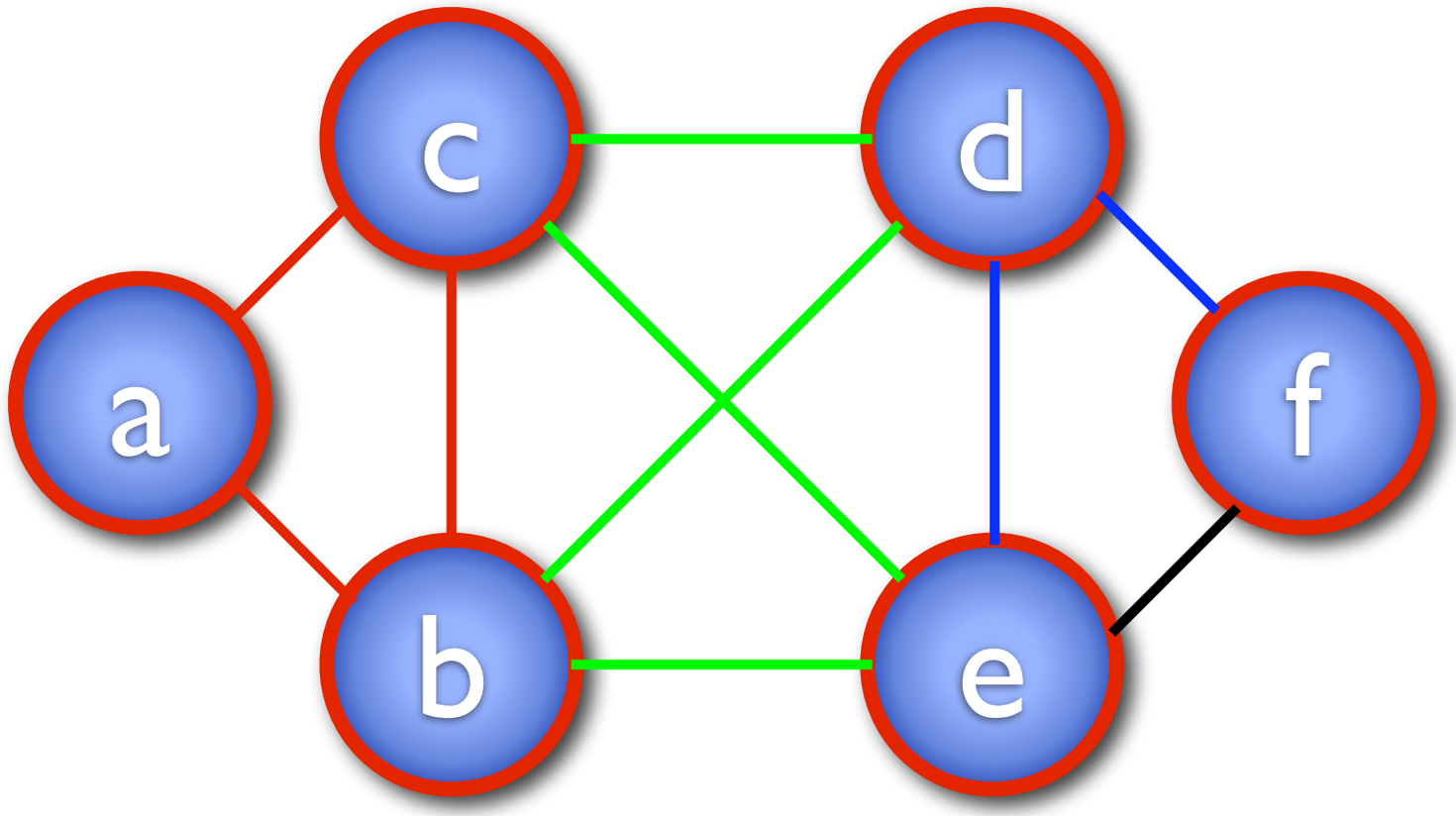
a



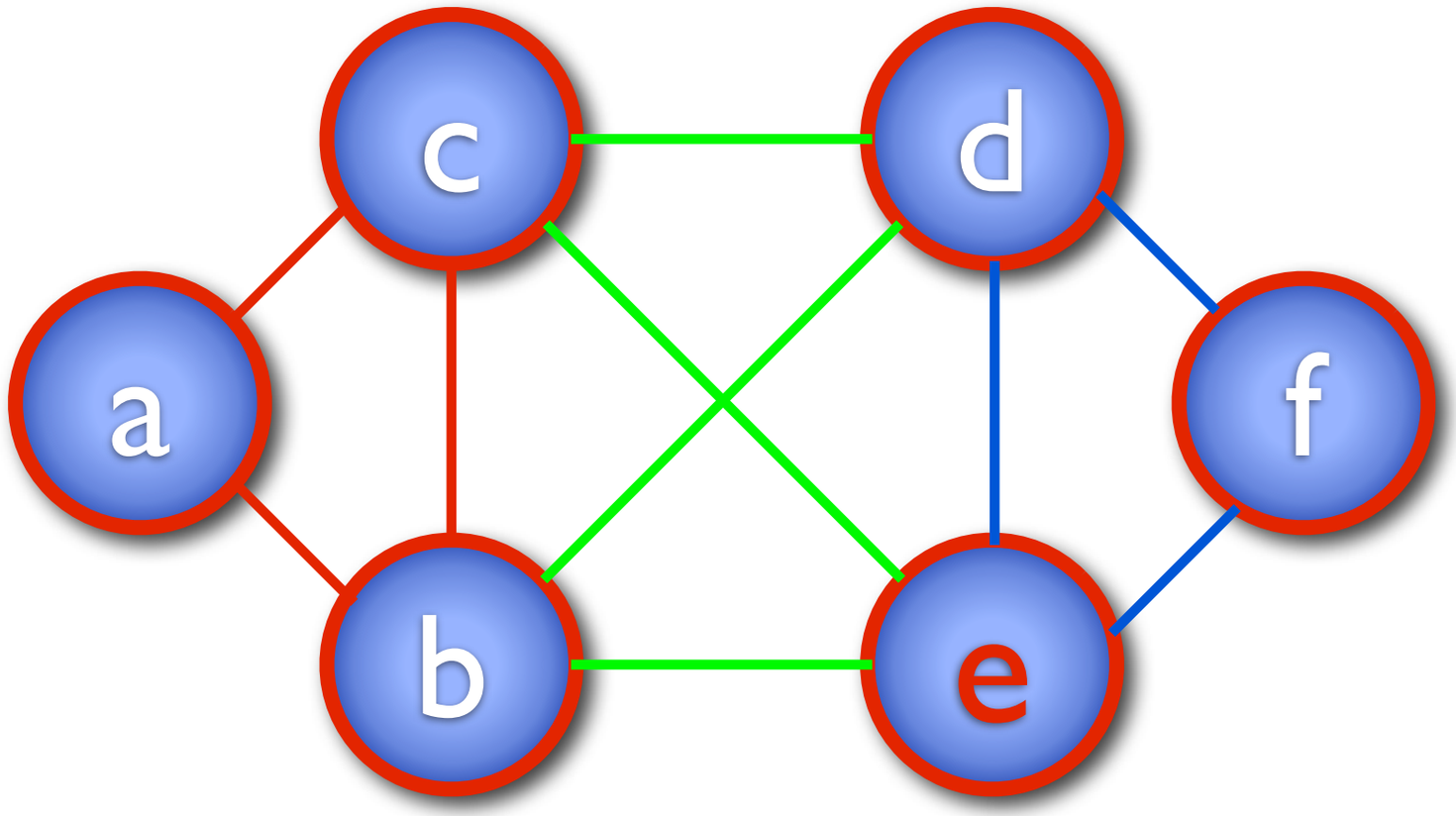
a c



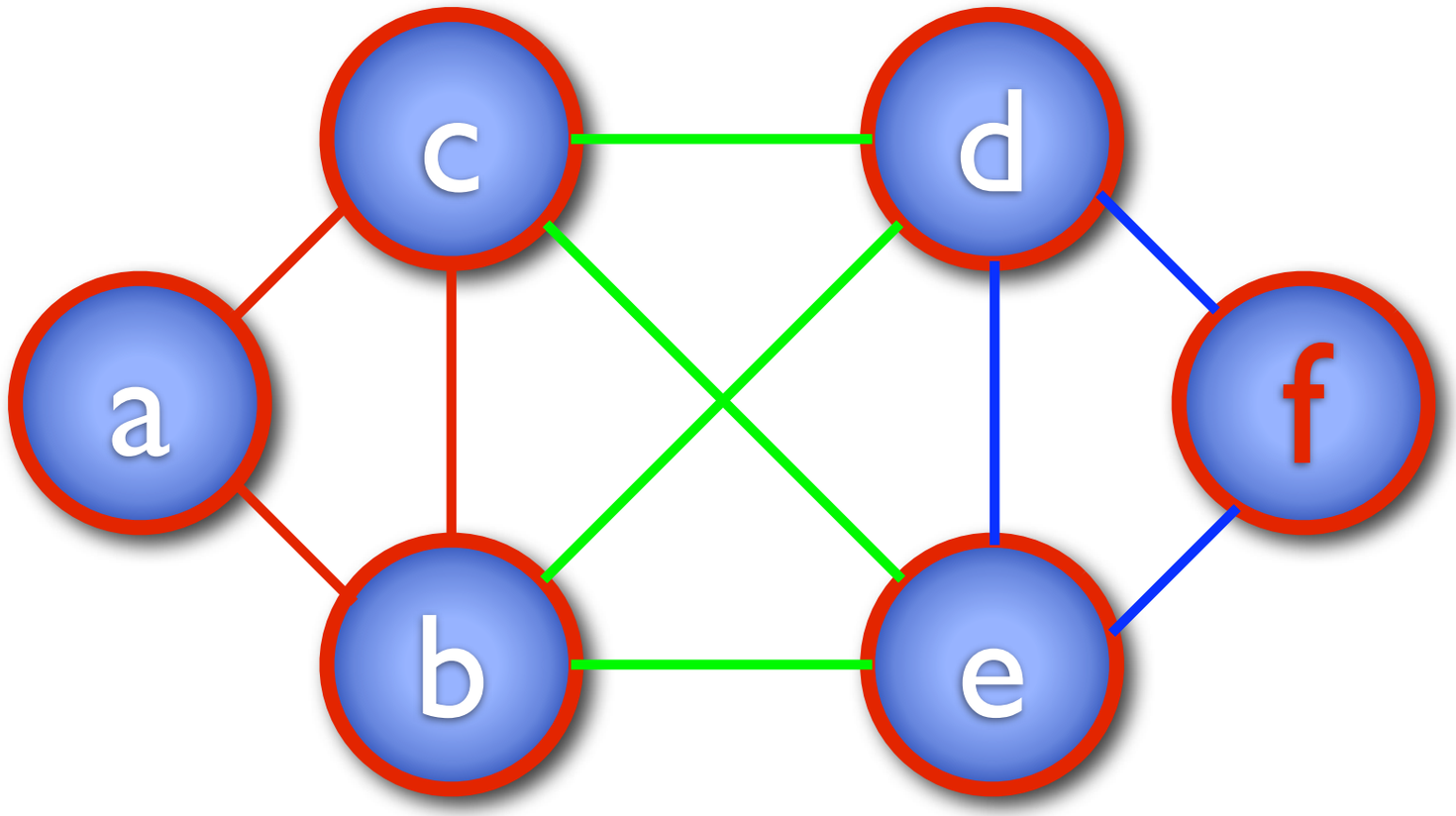
a c



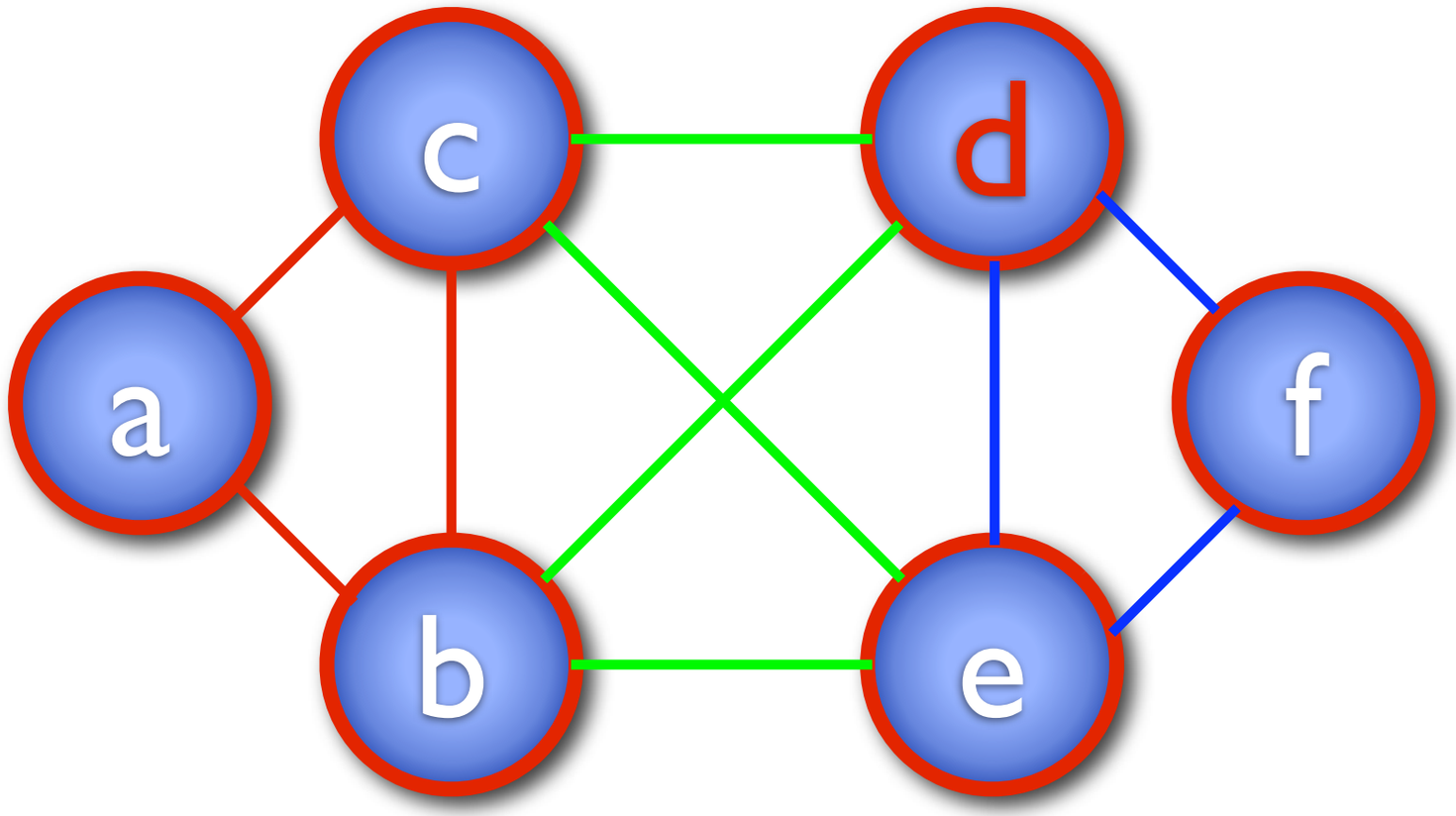
a c



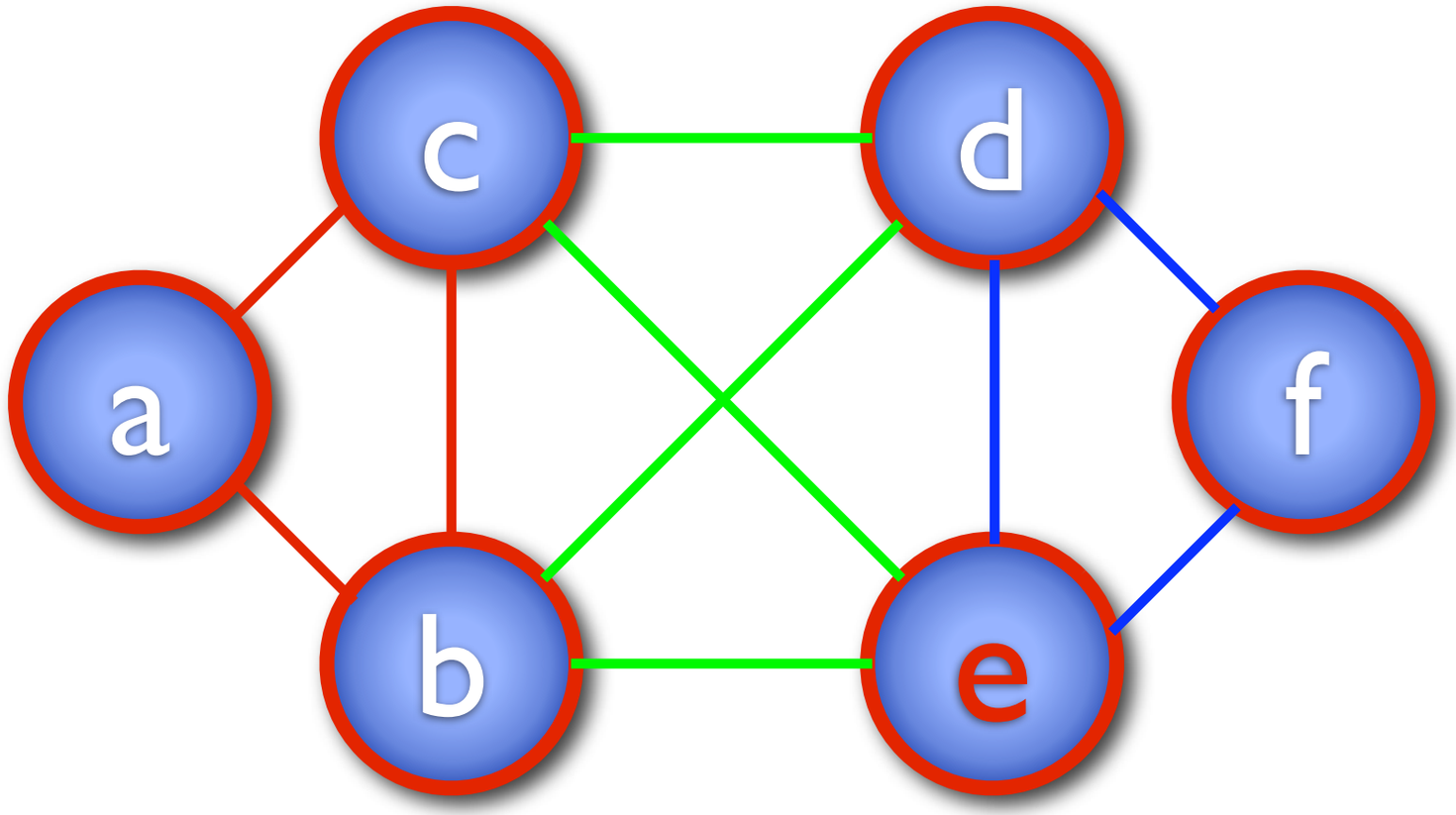
a c e



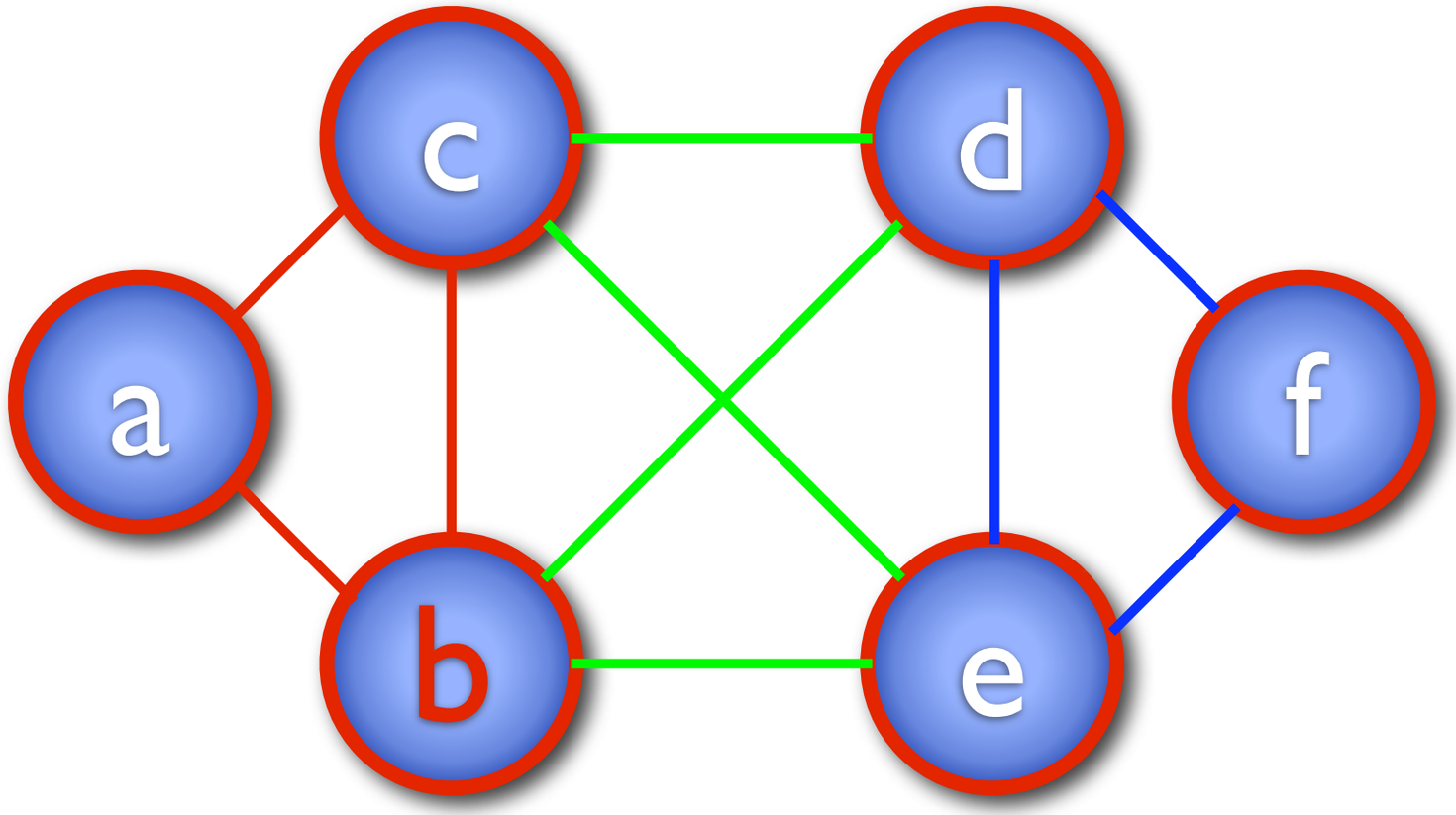
a c e f



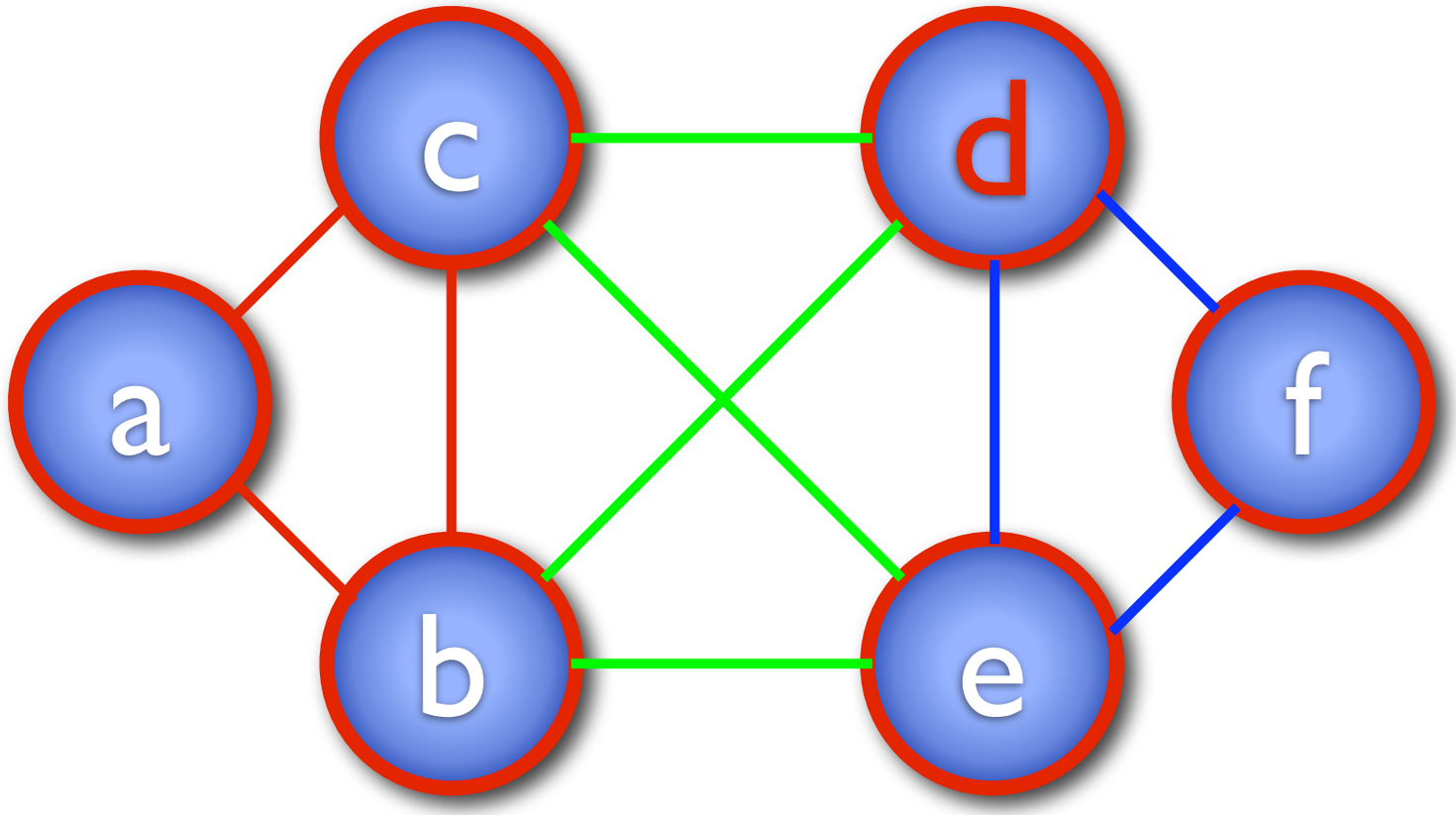
a c e f d



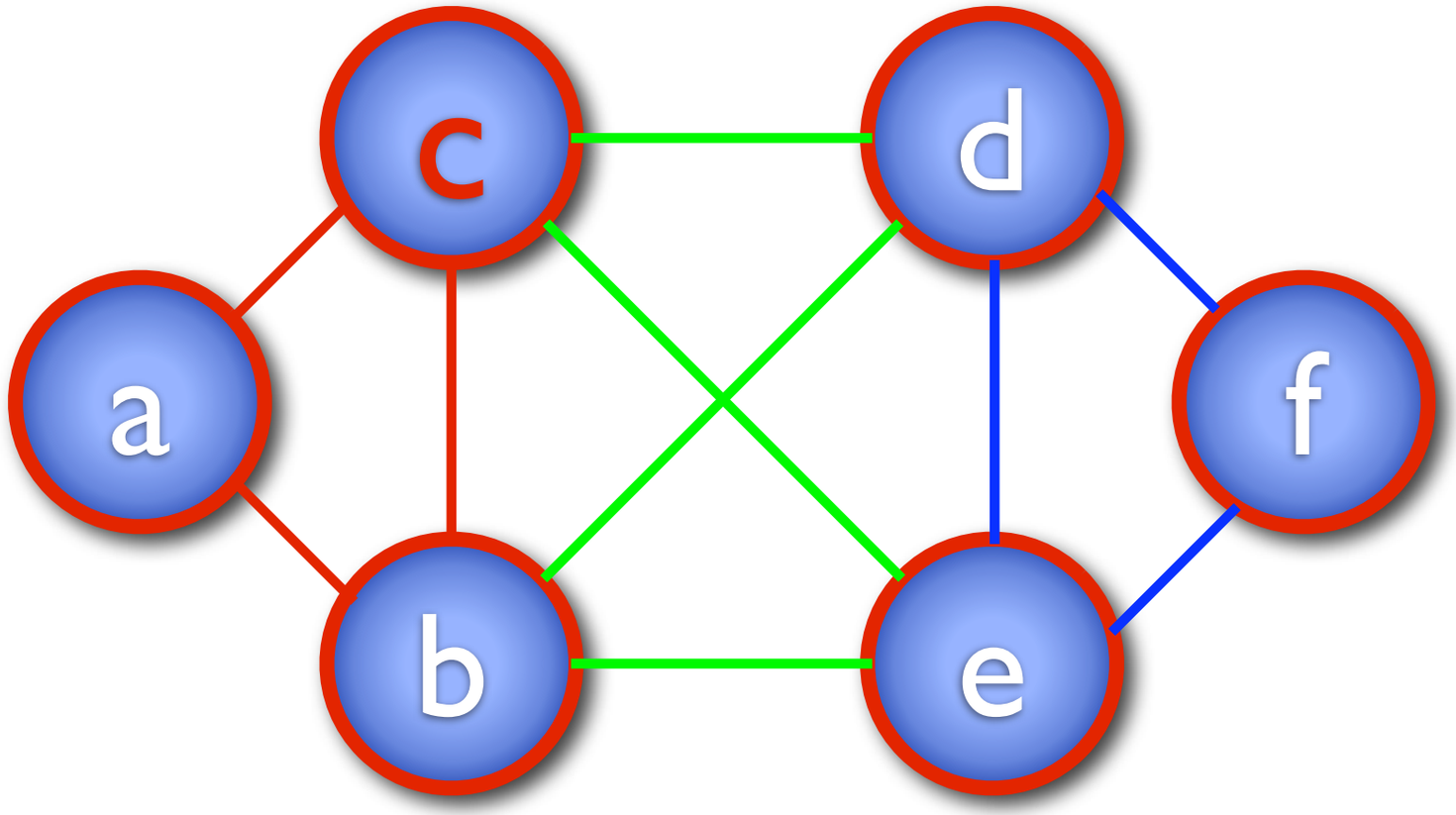
a c e f d e



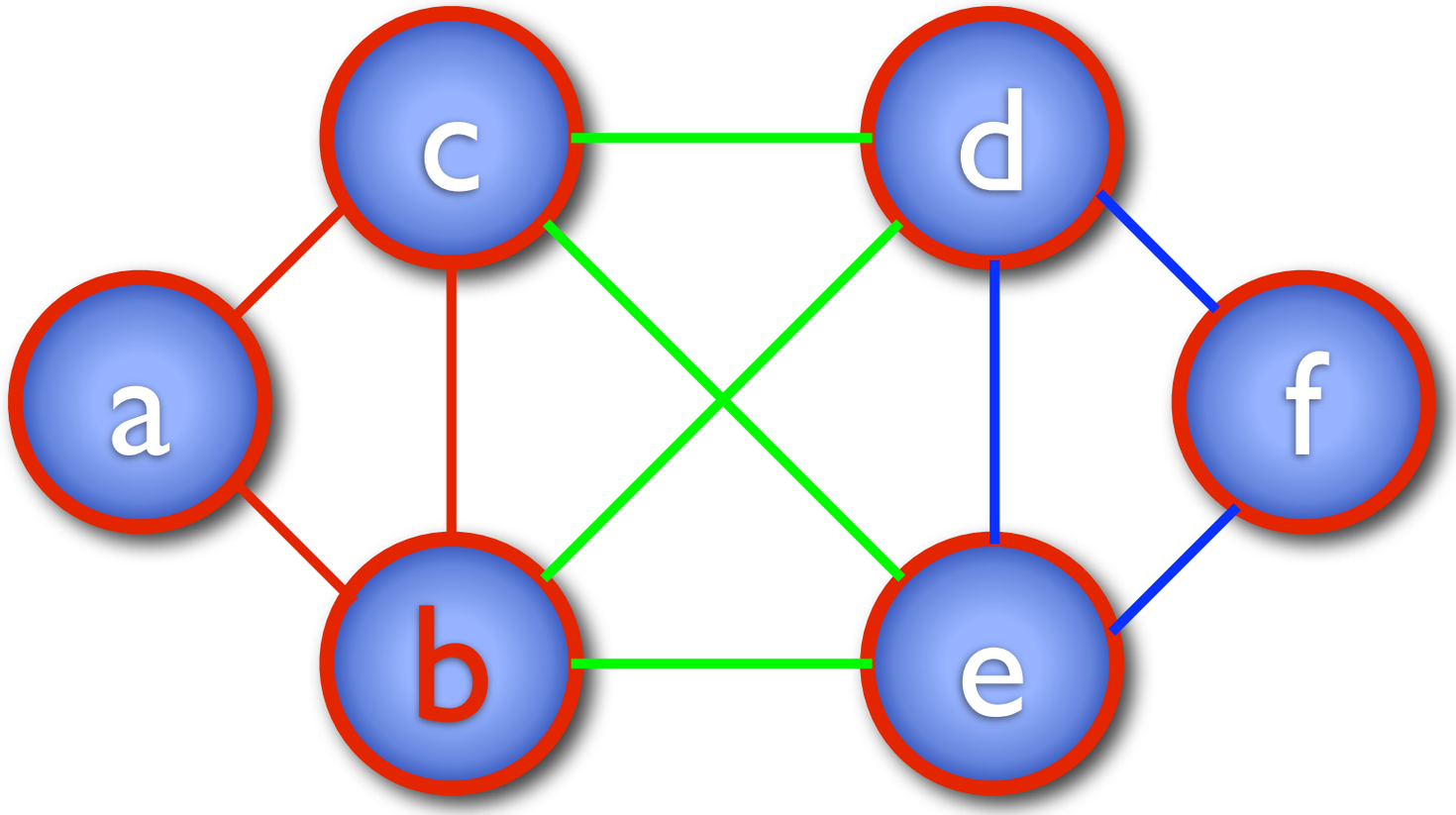
a c e f d e b



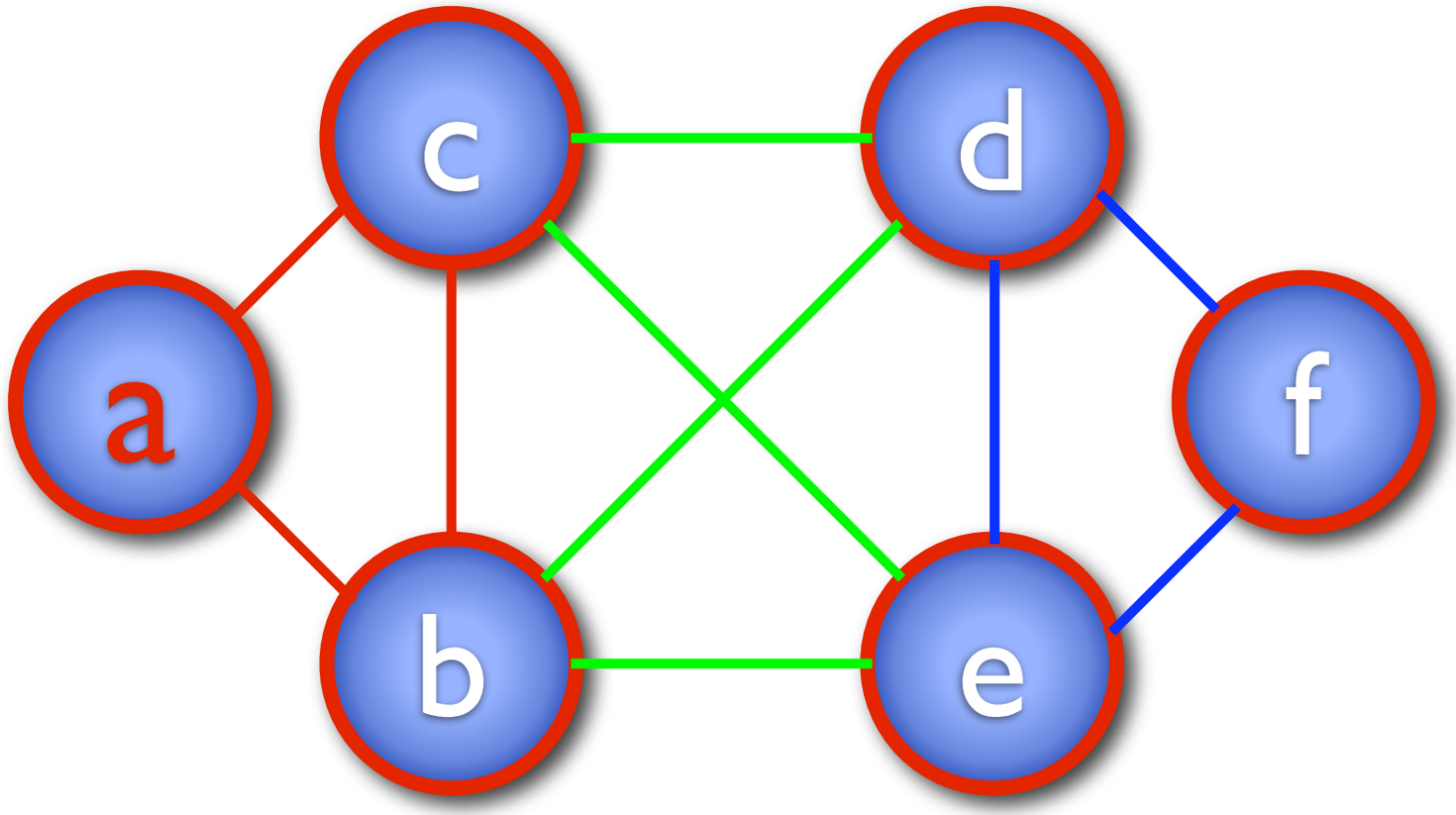
a c e f d e b d



a c e f d e b d c



a c e f d e b d c b



a c e f d e b d c b a

```
Euler(u)
for alle (u,v) ∈ E
  do if not visited(u,v)
     then visited(u,v) ← true
          Euler(v)
u zu Euler-Tour
```

Laufzeit: O(|V| + |E|) (DFS)

Fragen?

zurück zur Aufgabe

- Idee:
  - BFS von x liefert `d[0][]`
    BFS von y liefert `d[1][]`
  - ein Knoten u liegt auf einem optimalen Pfad, falls
    `d[0][u] + d[1][u] = dx[0][y]`
    und `d[0][u]` einzigartig