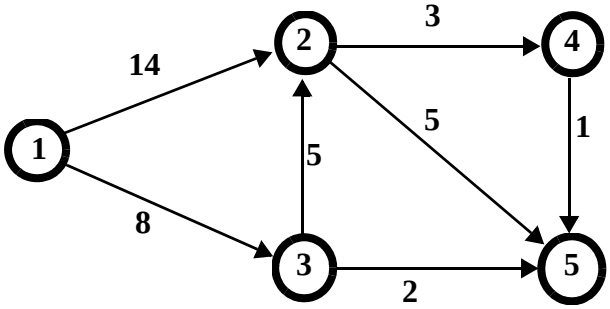
8.13 Algorithmus von Dijkstra für kürzeste Wege



Gerichteter Graph mit positiv (>=**0**) bewerteten Kanten:
Knoten **K** = { 1, 2, 3, 4, 5, 6 }
Kanten **V** = {(1,2),(1,3), (3,2), (2,5),(2,4), (4,5), (3,5) }

isolierter Knoten

Von einem Startknoten **s**=**1** aus werden die kürzesten Wege zu <u>allen</u> anderen Knoten berechnet.

Wegevektor **w(k)** := <u>geordnete</u> Knotenfolge von **s** nach **k** **minimaler** Länge

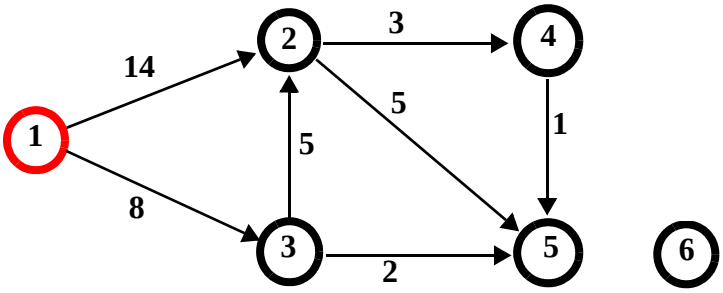
Länge **d(k)** des Weges **w(k)** zum Knoten **k** := Summe aller Kantenwerte des Weges **w(k)** von **s** nach **k**

<u>Start:</u> d(**s**)=d(**1**)=**0**, d(**2**)=d(**3**)=d(**4**)=d(**5**)=d(**6**)=∞

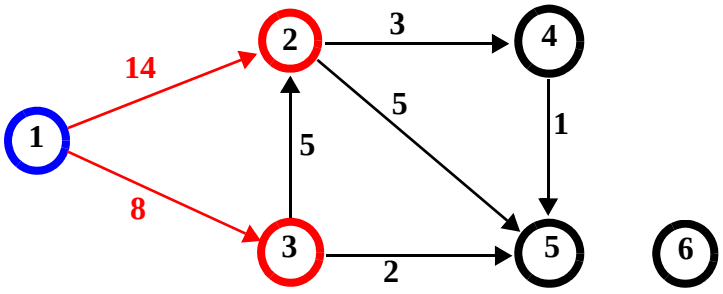
Disjunkte Unterteil. der Knoten: **K**=**E**∪**G**∪**U**, **E**∩**G**=∅, **E**∩**U**=∅, **G**∩**U**=∅

E	Erkundete Knoten	**E**=∅
G	Grenzknoten	**G**=∅
U	Unerkundete Knoten	**U**=**K**={**1**,**2**,**3**,**4**,**5**,**6**}

Kanten zwischen erkundeten Knoten:
Kanten von erkundeten nach Grenzknoten:
Kanten von Grenz-/unerkundeten nach unerkundeten Knoten



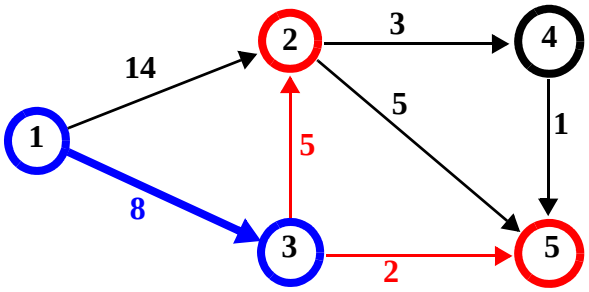
<u>1. Schritt:</u>
Startknoten **s** = **1** mit **d**(**s**)=0 wird "Grenzknoten":
E=∅ , **G**={**1**}, **U**={**2**,**3**,**4**,**5**,**6**}, **w**(**1**)=(**1**)



<u>2. Schritt:</u>
d(**1**)=0=**min**(d(k)), k∈**G**,
d.h. Knoten **1** "erkundet":

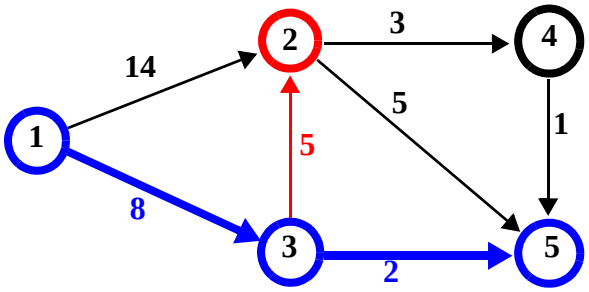
E={**1**} , **G**={**2**,**3**}
U={**4**,**5**,**6**}, **w**(**1**)=(**1**)
w(**2**)=(**1**,**2**), **w**(**3**)=(**1**,**3**)
d(**2**)=14, **d**(**3**)=8

Algorithmus von Dijkstra für kürzeste Wege



6

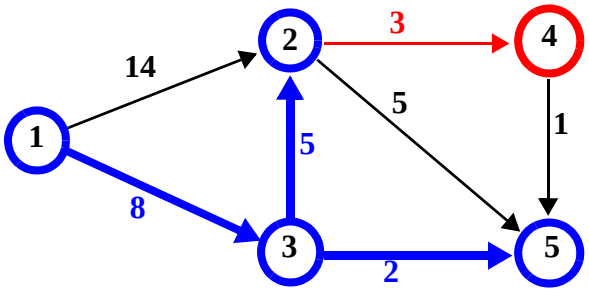
3. Schritt:
d(3)=8=**min**(d(k)), k∈**G**,
d.h. Knoten **3** "erkundet":
E={**1**,**3**}, **G**={**2**,**5**}, **U**={**4**,**6**}
w(**3**)=(**1**,**3**), **d**(**3**)=8
verändert: **w**(**2**)=(**1**,**3**,**2**)
verändert: **d**(**2**)=13
w(**5**)=(**1**,**3**,**5**), d(**5**)=10



6

4. Schritt:
d(**5**)=10=**min**(d(k)), k∈**G**,
d.h. Knoten **5** "erkundet":

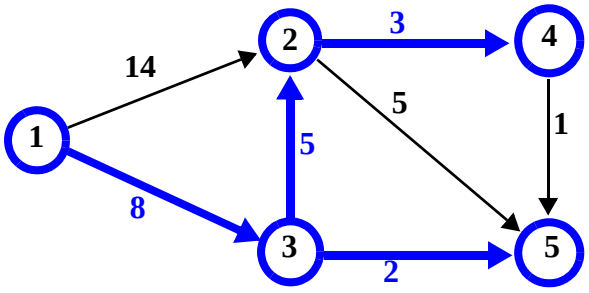
E={**1**,**3**,**5**}, **G**={**2**}, **U**={**4**,**6**}
w(**5**)=(**1**,**3**,**5**), **d**(**5**)=10
w(**2**)=(**1**,**3**,**2**), **d**(**2**)=13



6

5. Schritt:
d(**2**)=13=**min**(d(k)), k∈**G**,
d.h. Knoten **2** "erkundet":

E={**1**,**2**,**3**,**5**}, **G**={**4**}, **U**={**6**}
w(**2**)=(**1**,**3**,**2**), **d**(**2**)=13
w(**4**)=(**1**,**3**,**2**,**4**), **d**(**4**)=16



6

6. Schritt:
d(**4**)=16=**min**(d(k)), k∈**G**,
d.h. Knoten **4** "erkundet":
E={**1**,**2**,**3**,**4**,**5**}, **G**=∅, **U**={**6**}
w(**4**)=(**1**,**3**,**2**,**4**), **d**(**4**)=16

6

G=∅ –> **Abbruch**

Ergebnis:
Gerüst kürzester Wege

w(**1**)=(**1**),	**d**(**1**)= 0
w(**2**)=(**1**,**3**,**2**),	**d**(**2**)=13
w(**3**)=(**1**,**3**),	**d**(**3**)= 8
w(**4**)=(**1**,**3**,**2**,**4**),	**d**(**4**)=16
w(**5**)=(**1**,**3**,**5**),	**d**(**5**)=10
w(**6**)=∅,	**d**(**6**)=∞

Algorithmus von Dijkstra für kürzeste Wege

Gerüst kürzester Wege - Algorithmus von Dijkstra

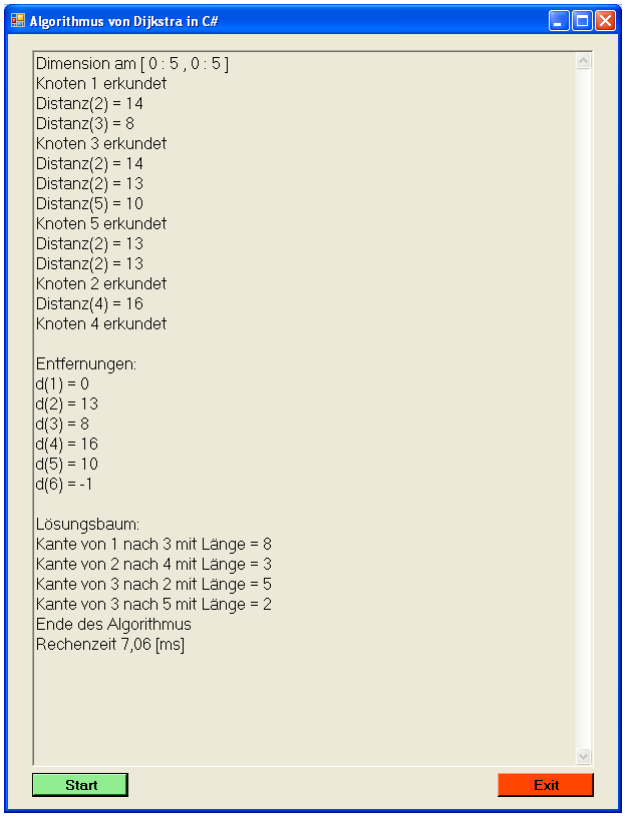
Das Verfahren von **Dijkstra** gilt als klassisches und wohl am häufigsten verwendetes Verfahren in der Angewandten Informatik. Dabei wird von einem **Startknoten** aus zu allen Knoten des Graphen das **Gerüst kürzester Wege** ermittelt. Der eigentliche **kürzeste Weg** vom Start- zu einem Zielknoten muß danach aus dem **Kantengerüst** bestimmt werden. Der Algorithmus von Dijkstra wird am Beispiel von 6 Knoten eines gerichteten Graphen erläutert. Der Knoten 6 ist ein **isolierter Knoten**, der über keine Kante erreicht werden kann. **Knoten 1** wird als **Startknoten** definiert.

Der Algorithmus von Dijkstra ist in C# (Projekt in .NET 4.0) implementiert und befindet sich im Verzeichnis **Dijkstra1** bzw. gepackt in **Dijkstra1.rar**.

```
private double[,] am; // Adjazenzmatrix
private double[,] wm; // Wegematrix, entspricht Adjazenzmatrix der kuerzesten Wege von s aus
private double[]  d;  // Distanzvektor, entspricht akkumlierter Entfernung Knoten i zu Startknoten
private bool[]    m;  // Markierungsvektor (true: unerkundeter Knoten; false: bereits erkundet)

private int       n;  // Anzahl Knoten
private int       s;  // Startknoten

n = 6; // Anzahl Knoten ist 6
s = 0; // Startknoten festlegen

// Instantiierung Matrizen und Vektoren
am = new double[n,n];
wm = new double[n,n];
d  = new double[n];
m  = new bool[n];

// Initialisierung Matrizen und Vektoren
for (int i=0; i<n; i++) {
    m[i] = true;               // Anfangs alle Knoten als unerkundet markieren
    d[i] = -1;                 // Entfernung zum Startknoten ist unendlich (-1)
    for (int j=0; j<n; j++) {
        am[i,j] = -1;          // keine Verbindung zwischen Knonten i und j
        wm[i,j] = -1;
    }
}

// Kanten eintragen (Beispieldaten)
am[0,1] = 14;
am[0,2] = 8;
am[1,3] = 3;
am[1,4] = 5;
am[2,1] = 5;
am[2,4] = 2;
am[3,4] = 1;
// Entfernung zum Startknoten ist 0
d[s] = 0;

// Dijkstra Algorithmus iterativ abarbeiten
private void dijkstra() {
  while (knoten()) {
     for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            if (i != j && am[i,j] >= 0 && m[i]==false && m[j]) {
                // fuer j existiert noch keine Entfernung zum Startknoten
                if (d[j] < 0) {
                    d[j] = d[i] + am[i,j];
                }
                else {
                    // nach j existiert kuerzester Weg ueber i
                    if (d[j] > (d[i] + am[i,j]))
                        d[j] = (d[i] + am[i,j]);
                    // bisherige Grenzkanten zuruecksetzen
                    for (int k=0; k<n; k++)
                        if (wm[k,j] >= 0) wm[k,j] = -1;
                }
                // Grenzkante setzen
                wm[i,j] = am[i,j];
                textBox1.AppendText(String.Format("Distanz({0}) = {1}\r\n", j+1, d[j]));
            }
        }
     }
  }
}
```

Algorithmus von Dijkstra für kürzeste Wege

```
// existiert noch wöhlbarer Knoten?
private bool knoten() {
   int min = -1;
   for (int i=0; i<n; i++) {
       if (m[i] && d[i] >= 0)
           if (min < 0 || d[i] < d[min])
              min = i;
   }
   if (min >= 0) {
      m[min] = false;
      textBox1.AppendText(String.Format("Knoten {0} erkundet\r\n", min+1));
      return true;
   }
   return false;
}
```



Die vorliegenden Realisierungen des Dijkstra - Algorithmus zur Findung kürzester Wege innerhalb eines gerichteten Graphen in der Programmiersprache ist in der Programmiersprache CSharp realisiert worden. Dabei liegen die Graphen sowohl fest eingebettet im Programm als auch in einer Access - Datenbanken vor.

Lösungen

Die Vorgehensweise des Dijkstra-Algorithmus ist wie folgt zu beschreiben:

Nach Festlegung eines Startknotens werden alle angrenzenden Knoten erkundet und die Distanz bis dahin gemerkt. Der kürzeste Weg zu einem Knoten wird weiterverfolgt, wenn es zu den anderen noch andere Wege gibt um sie zu erreichen. Ist ein kürzester Weg gefunden wird die endgültige Entfernung eingetragen. Ist für einen Knoten noch ein anderer Weg möglich um den Knoten zu erreichen, werden die anderen Wege geprüft und wiederum der kürzeste Weg eingetragen.

Es wurden 3 Lösungen erstellt.

Lösung 1 ‚Dijkstra1' arbeitet mit einer Adjazenzmatrix (am) für die Entfernungen zwischen den Knoten, einer Wegematrix (wm) für die Ergebnisse der beteiligten Kanten an dem Weg zur kürzesten Entfernungen zwischen den Knoten und dem Startknoten, Distanzvektor (d) für die kürzesten Entfernungen zwischen den Knoten und dem Startknoten und einem Markierungsvektor (m) zur Markierung schon erkundeter Knoten und Auswahl des nächsten Knoten, der zu erkunden ist.

Lösung 2 ‚DijkstraDB' arbeitet ohne die Matrizen und Vektoren sondern in Zusammenarbeit mit der Datenbank ‚Graph.mdb'. Diese wurde von den schon bestehenden Beispielen in Visual Basic übernommen. Darin werden die Kanten, Knoten und Wege dynamisch gespeichert und entsprechend ausgewertet. Der Abarbeitungsalgorithmus sieht hier natürlich anders aus, arbeitet jedoch nach dem oben beschriebenen Prinzip.

Unterschiede in der Oberflächendarstellung liegen in der Verwendungsweise des DataGrids in CSharp. Es wurde schliesslich eine **Lösung 3** ‚DijkstraDB2' angefertigt, die der Arbeitsweise der Oberfläche des Visual Basic Programms sehr ähnelt, die Handhabung des DataGrids wirkt jedoch sehr umständlich und unverständlich.

Probleme

Bei der Verwendung mehrerer Tabellen einer Datenbank und deren Verknüpfung hinter einem DataGrid kamen Probleme zu Tage, die in der einschlägigen Literatur nicht beschrieben ist. So können mehrer Tabellen nicht zusammengefasst und angezeigt werden in einem DataGrid wie es bei einer Tabelle der Fall ist.

Eine Lösung ist in dem Buch ‚Visual C# .NET' von Mickey Williams im Verlag ‚Microsoft Press' beschrieben, diese ist jedoch so nicht ausführbar.

Um das Problem zu umgehen, wurde bei ‚DijkstraDB' eine JOIN-Anweisung verwendet, wobei nicht alle Knoten angezeigt werden. Bei ‚DijkstraDB2' hingegen wurde eine neue Datenbankverbindung vom Typ MSDataShape eingerichtet um die SHAPE-Anweisung zu realisieren.

Pseudocode Algorithmus von Dijkstra

Vorbemerkung:
Die **n** (**n>0**) Knoten werden durchgehend von **0** an in Einerschritten bis **n-1** numeriert (ℵℝ), damit können die Knoten eineindeutig den Zeilen- und Spaltenindizes der **Adjazenzmatrix** zugeordnet werden. **Lösung** des Algorithmus ist die ausgefüllte **Adjazenzmatrix w[n,n]** der **kürzesten Wege** vom **Knoten s** nach allen anderen Knoten.

Definition der notwendigen Datenstrukturen:
Anzahl Knoten n (**n>0**) festlegen (einlesen)
Startknoten s mit **0** ≤ **s** < n festlegen (einlesen)
Adjazenzmatrix am[n,n] anlegen (dynamisch)

Wenn **keine Kante** zwischen Knoten **i** und Knoten **j** , dann **am[i,j] = -1** (entspricht ∞)
Wenn **Kante** zwischen Knoten **i** und Knoten **j** , dann **am[i,j] := Entfernung (≥ 0)**

Wegematrix w[n,n] anlegen (dynamisch), entspricht **Adjazenzmatrix** der kürzesten Wege vom Knoten **s** aus (Am Ende repräsentiert die Wegematrix die kürzesten Wege von **s→i**)

Alle Elemente **w[i,j]** mit **0≤i<n** und **0≤j<n** erhalten am Anfang den Wert **-1** (entspricht ∞)

Distanzvektor d[n] anlegen (dynamisch), in **d[i]** mit **0≤i<n** stehen später die kürzesten Entfernungen vom Knoten **s** bis zum Knoten **i**.

Am Anfang wird **d[i] = -1** (entspricht ∞) gesetzt, außer **d[s] = 0**.

Markierungsvektor m[n] anlegen (dynamisch), **unerkundete Knoten i** (0<=i<n) erhalten am Anfang den Wert **m[i]:= true**; wenn **Knoten i erkundet**, dann **m[i]:= false**

Algorithmus Dijkstra:

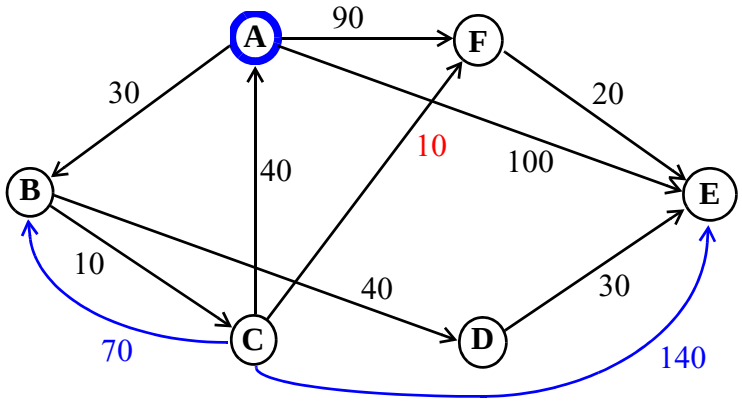
```
wiederhole solange unerkundete Knoten i existieren (ex. Knoten i mit m[i] == false ?)
 wiederhole für alle von-Knoten i=0; i<n; i++
  wiederhole für alle nach-Knoten j=0; j<n; j++
   wenn i ≠ j  und am[i,j] ≥ 0 (Kante i→j ex.) und Knoten i erkundet (m[i] == false)
               und j nicht erkundet (m[j] == true)              // j ist Grenzknoten
     wenn d[j] < 0                      // noch keine Entfernung von Knoten s nach Knoten j
        dann  d[j] = d[i] + am[i, j]
     sonst                                  // Entfernung d[j] ≥ 0 existiert ?
        wenn  d[j] > (d[i] + am[i,j])       // bisheriger Weg nach j länger als Weg über i ?
              dann d[j] = d[i] + am[i, j] (korrigiere Entfernung d[j] auf kleineren Wert)

        wiederhole für alle Knoten k=0; k<n; k++
              wenn wm[k,j] ≥ 0      // Wurde Knoten j bereits über Knoten k erreicht ?
                 dann wm[k,j] = -1  // Knoten j wird nicht mehr über Knoten k erreicht
     end wenn
     wm[i,j] := am[i,j]   (Knoten j wird über Knoten i erreicht, Entfernung i→j := am[i,j])
   end wenn
  end wiederhole
 end wiederhole
end wiederhole
```

Die folgende Funktion wird in der **äußeren Schleife** des Dijkstra-Algorithmus unter "**Wiederhole solange unerkundete Knoten i existieren**" gerufen.

Alle Knoten werden durchlaufen. Wenn Grenzknoten **i** gefunden werden, dann wird derjenige mit **minimalem d[i]** ausgewählt. Dieser Knoten wird in einen "erkundeten Knoten" umgewandelt und **true** zurückgegeben. Wenn kein Grenzknoten mehr gefunden werden kann, dann gibt der Algorithmus **false** zurück und der Dijkstra-Algorithmus wird beendet.

```
bool unerkundete_knoten(){
  min := -1                                   // Anfangswert für Grenzknoten i mit
  wiederhole für alle Knoten i=0; i<n; i++
     wenn Knoten i noch unerkundet (m[i] == true) und Distanz d[i] ≥ 0   // Grenzknoten ?
     dann                                     // Suche Grenzknoten i mit minimalem d[i]
          wenn  min < 0  oder  d[i] < d[min]  // wenn min<0, dann keine Abfrage d[i]<d[min]
                dann  min := i                // min := Grenzknoten i
     end wenn
   end wiederhole
   wenn  min ≥ 0                              // Grenzknoten min mit minimalem d[min]
         dann  { m[min] := false;             // Knoten min ist erkundet
                 return true;                 // unerkundeter Knoten wurde gefunden
         }
   return false                               // kein unerkundeter Knoten mehr vorhanden
}
```

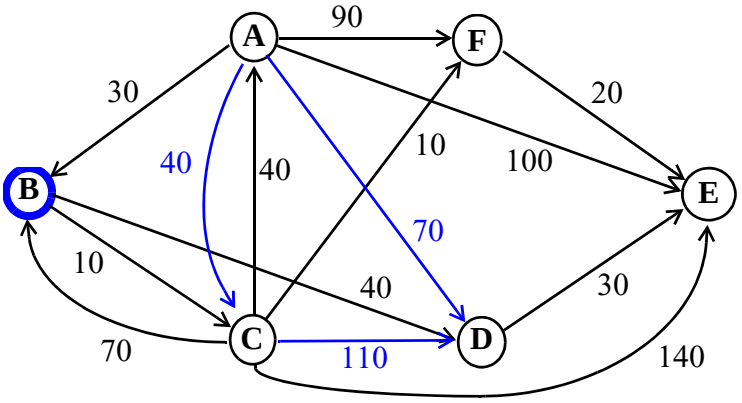
8.14 Algorithmus von Floyd aller kürzesten Wege



Durchlaufe alle Knoten
aktuell: **A**
Vorg.- u. Nachf. von **A**:
{C} -> **A** -> {B, E, F}

Existierende Kante:
W(C,F)=10<40+90
Wert unverändert

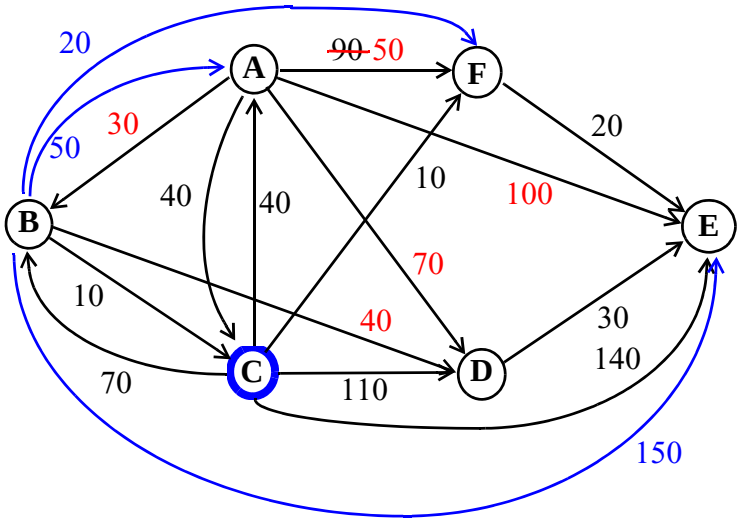
Neue Kanten:
W(C,B)=40+30=70
W(C,E)=40+100=140



aktuell: **B**
Vorg. u. Nachf. von **B**:
{A,C} -> **B** -> {C,D}

Existierende Kanten:
∅

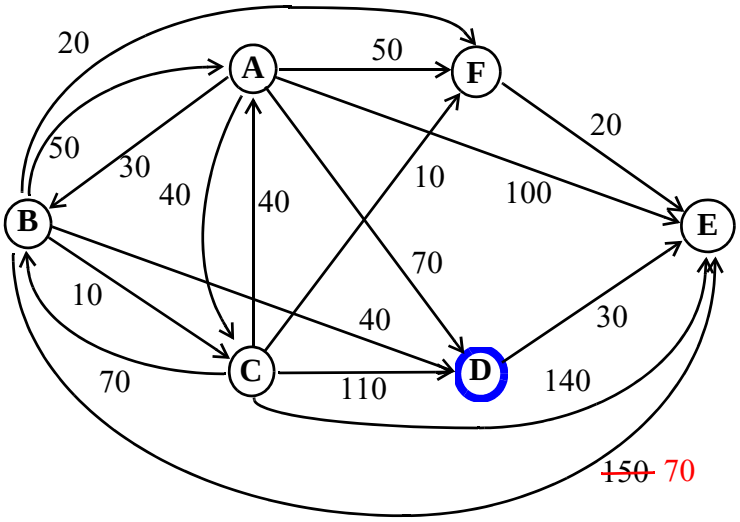
Neue Kanten:
W(A,C)=30+10=40
W(A,D)=30+40=70
W(C,D)=70+40=110



aktuell: **C**
Vorg. u. Nachf. von **C**:
{A,B}->**C**->
{A,B,D,E,F}

Existierende Kanten:
W(A,B)=30 < 40+70=110
W(A,D)=70 < 40+110=150
W(A,E)=100 < 40+140=180
W(A,F)=90 > 40+10=50 änd.
W(B,D)=40 < 10+110=120

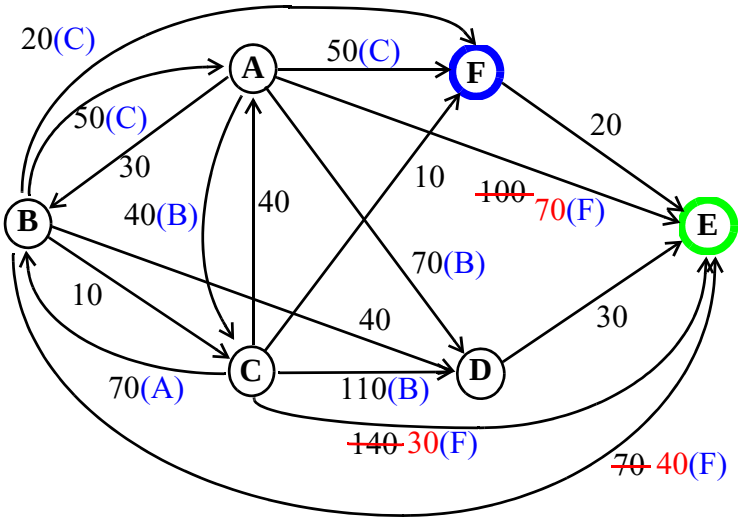
Neue Kanten:
W(B,A)=10+40=50
W(B,E)=10+140=150
W(B,F)=10+10=20



aktuell: **D**
Vorg. u. Nachf. von **D**:
{A,B,C}->**D**-> {E}

Existierende Kanten:
W(A,E)=100 = 70+30 = 100
W(B,E)=150>40+30=70 änd.
W(C,E)=140 = 110+30 = 140

Neue Kanten:
∅



aktuell: **E**
Vorg. u. Nachf. von **E**:
{A,B,C,D,F} ->**E** -> ∅
E hat **keine** Folgeknoten

---

aktuell: **F**
Vorg. u. Nachf. von **F**:
{A,B,C}->**F**-> {E}

Existierende Kanten:
W(A,E)=100 > 50+20=70 änd.
W(B,E)=70 > 20+20=40 änd.
W(C,E)=140 > 10+20=30 änd.

Neue Kanten:
∅

Der **Algorithmus von Floyd** ermittelt **von allen Knoten nach allen Knoten** die **kürzesten Wege**.

Alle Knoten werden **iterativ** durchlaufen. Für jeden Knoten werden **alle Vorgänger** und **alle Nachfolger** betrachtet.

Existiert bereits von einem Vorgängerknoten zu einem Nachfolgerknoten eine Kante, dann wird deren Wert mit der Summe der Werte der Kanten verglichen, die über den aktuellen Knoten führen. Sollte die Summe **kleiner als der Wert der Direktkante** sein, so wird der Wert der Direktkante auf diesen **kleineren Wert** gesetzt.

Existiert zwischen einem Vorgänger und einem Nachfolger **keine Kante**, dann wird eine **neue Kante** mit dem **Wert der Summe der Kanten** vom Vorgänger über den ausgewählten Knoten zum Nachfolger **erzeugt**.

Nachdem alle Knoten iterativ für alle Vorgänger und Nachfolger durchlaufen wurden, sind alle **Knoten durch Kanten verbunden**, **deren Werte die kürzesten Verbindungen** zwischen den betreffenden Knoten angeben.

Im Falle der Wertveränderung oder der Neudefinition einer Kante sollte für die betreffende Kante neben dem Wert **zusätzlich der Knotenname** gespeichert werden, über den der kürzere Weg führt.

Die Kantenmenge teilt sich **disjunkt in Kanten**, die **keinen Zwischenknoten** gespeichert haben, und in Kanten, denen ein **Zwischenknoten** zugeordnet ist.

Bei der Bestimmung des **kürzesten Weges** von einem Knoten zu einem anderen müssen **Kanten mit gespeichertem Zwischenknoten** solange rekursiv in Kantenpaare über den Zwischenknoten aufgelöst werden, bis keine Kante des Weges mehr einen Zwischenknoten besitzt. Erst dann wurde der Weg über die ursprünglichen Kanten gefunden.

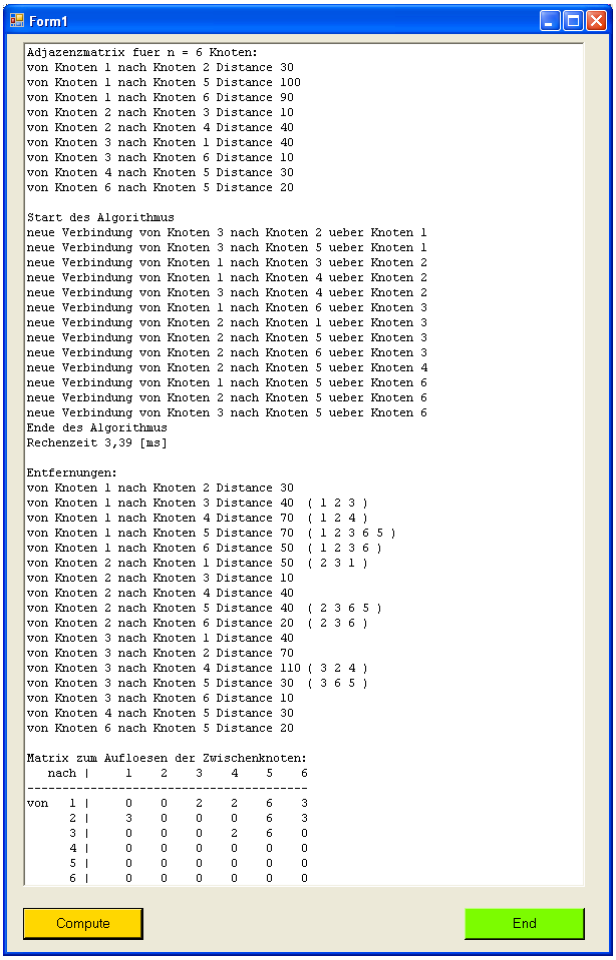
```
// C#.NET - Projekt Floyd1
private double[,] am; // Adjazenzmatrix
private int[,]    zm; // Matrix zum Aufloesen der Zwischenknoten
private int       n;  // Anzahl Knoten
n = 6;                // Anzahl Knoten n = 6

// Instantiierung Matrizen und Vektoren (dynamisch)
am = new double[n,n];
zm = new int[n,n];

// Initialisierung Matrizen und Vektoren
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
        if (i==j) am[i,j] = 0;  // Entfernung zu sich selbst ist 0
        else      am[i,j] = -1; // sonst keine Verbindung
        zm[i,j] = 0;            // Verbindung ohne kein Zwischenknoten
    }
}

// Kanten eintragen (Beispieldaten)
am[0,1] = 30;
am[0,4] = 100;
am[0,5] = 90;
am[1,2] = 10;
am[1,3] = 40;
am[2,0] = 40;
am[2,5] = 10;
am[3,4] = 30;
am[5,4] = 20;

// Floyd Algorithmus iterativ abarbeiten
private void floyd() {
       for (int i=0; i<n; i++) {
              for (int j=0; j<n; j++) {
                   for (int k=0; k<n; k++) {
                     // wenn Verbindugn j->i und i->k existiert
                     if (am[j,i] >= 0 && am[i,k] >= 0)
                       if ( am[j,k] < 0 ||                    // existiert noch gar nicht
                            (am[j,k] > am[j,i] + am[i,k]) ) {  // oder Verbindung ist kürzer
                                am[j,k] = am[j,i] + am[i,k];  // Neue Kante j->k
                                zm[j,k] = i;       // Verbindung j->k ueber Zwischenknoten i
                     }
                   }
              }
       }
}
```

```
// Ausgabe der Ergebnisse
private void output() {
       for (int i=0; i<n; i++) {
              for (int j=0; j<n; j++) {
                 if (i != j && am[i,j] >= 0) {
                     if (zm[i,j] > 0)
                            listBox1.Items.Add(String.Format(
                              "von Knoten {0} nach Knoten {1} Distance {2}\t( {0} {3})",
                              i+1,j+1,am[i,j],zknoten(i,j)));
                     else
                            listBox1.Items.Add(String.Format(
                               "von Knoten {0} nach Knoten {1} Distance {2}",
                               i+1,j+1,am[i,j]));
                 }
              }
       }
}

// Verbindung über Zwischenknoten
private string zknoten(int i, int j) {
       if (zm[i,j] == 0)
              return (j+1).ToString() + " ";
       else
              return zknoten(i,zm[i,j]) + zknoten(zm[i,j],j);
}
```

Beispiel rekursive Ausgabe über zknoten(1, 5):

```
Kante:    1 -> 5 = 1 -> 6 -> 5 = 1 -> 3 -> 6 -> 5 = 1 -> 2 -> 3 -> 6 -> 5
über:       6        3             2                 nur noch Elementarkanten
```

Ausgabe von Floyd1:



Algorithmus von Floyd (iterativ)

```
Private a() As Double        'Kostenmatrix kürzester Wege
Private c() As Double        'Adjazenzmatrix des ursprünglichen Graphen
Private p() As Double        'Matrix der Zwischenknoten
Private n As Integer         'Anzahl Knoten

n = 6

ReDim a(1 To n, 1 To n)
ReDim c(1 To n, 1 To n)
ReDim p(1 To n, 1 To n)

Private Function floyd(p() As Double, a() As Double)
Dim i As Integer, j As Integer, k As Integer

  For i = 1 To n                                    'Alle Knoten i
    For j = 1 To n                                  'Knoten j für Kante (j,i)
      For k = 1 To n                                'Knoten k für Kante (i,k)
        If a(j, i) >= 0 And a(i, k) >= 0 Then       'Kanten (j,i) und (i,k) ex. ?
           If a(j, k) < 0 Then                      'Kante (j,k) existiert nicht ?
              a(j, k) = a(j, i) + a(i, k)           'Neue Kante (j,k) über i
              p(j, k) = i                           'Kante (j,k) über i notieren
           Else                                     'Kante (j,k) existiert bereits
              If a(j, i) + a(i, k) < a(j, k) Then   'Weg (j,i)+(i,k) kürzer als Weg (j,k) ?
                 a(j, k) = a(j, i) + a(i, k)        'Kürzere Weglänge (j,k) über i notieren
                 p(j, k) = i                        'Kante (j,k) über i ändern
              End If
           End If
        End If
      Next k
    Next j
  Next i
End Function
```

Algorithmus von Floyd (iterativ)

Aufruf:

```
Private Sub Command1_Click()                    'Compute-Button
  floyd p(), a()                                'Aufruf floyd iterativ
End Sub
```

Rekursives Auflösen einer Kante (i,j) in Elementarkanten:

```
Private Function out(p() As Double, ByVal i As Integer, ByVal j As Integer) As String

  If p(i, j) = 0 Then                                   'Abbruchbedingung
     out = CStr(j) & "  "
  Else
     out = out(p(), i, p(i, j)) & out(p(), p(i, j), j)  'Rekursion
  End If

End Function
```

Beispiel:

Kante: 1 -> 5 = 1 -> 6 -> 5 = 1 -> 3 -> 6 -> 5 = 1 -> 2 -> 3 -> 6 -> 5
über: 6 3 2 nur noch Elementarkanten