Graphen und Zeiger

Programmieren für Ingenieure
Sommer 2014

Andreas Zeller, Universität des Saarlandes

Algorithmus

- eindeutige *Handlungsvorschrift* zur Lösung eines Problems
- besteht aus endlich vielen, wohldefinierten *Einzelschritten*
- werden typischerweise in *Computerprogrammen* implementiert

Algorithmen

Berechnen	Suchen	Sortieren
• Fibonacci • GgT • Collatz	• Linear • Binär	• Einfügen • Mischen

Sortieren vor Ort

0	1	2	3	4	5	6	7	8	9	10
10	–10	7	2	2	–4	–7	–10	1	4	5

- Wir möchten *innerhalb des Feldes* sortieren
- Annhme: Feld a[0…i–1] ist bereits sortiert
- Wir betrachten das Element a[i]…
- …und fügen es in das sortierte Feld ein

Theremin =

Themen heute

- Graphen
- Tabellen
- Konstanten
- Zeiger

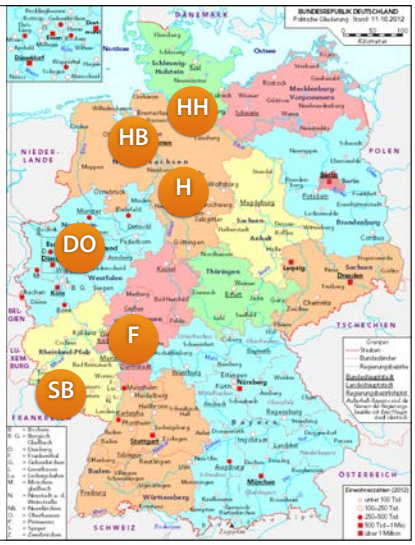
Bild: Tomtom

Der kürzeste Weg

Wer ein echter Saarländer sein will, der will immer nach Hause

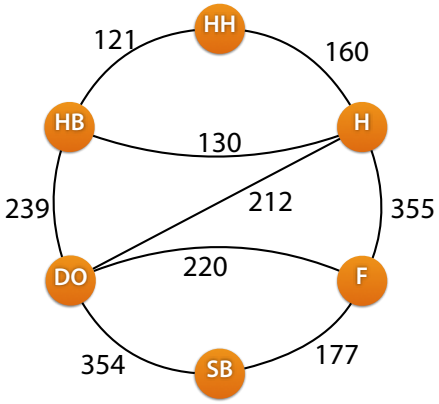
---

Der kürzeste Weg

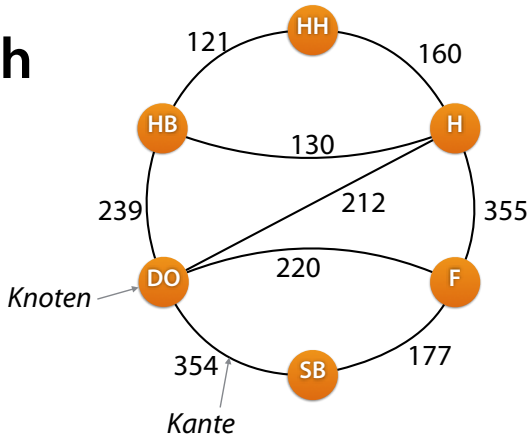


Und wie kommt man nach Hause? Man nimmt eine Karte.

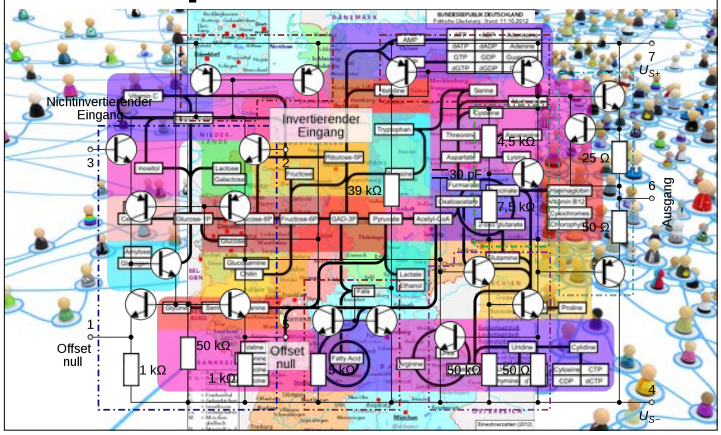
Der kürzeste Weg



Graph

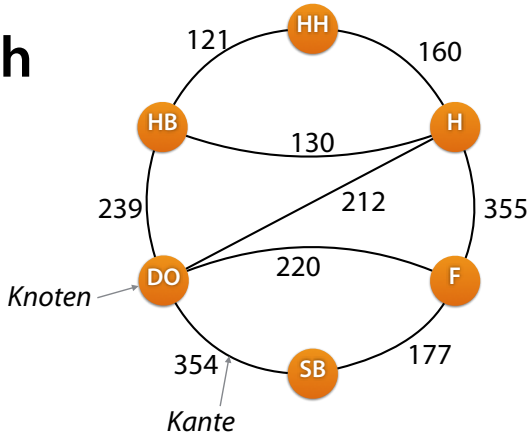


Graphen sind überall



Bilder: Wikipedia, Shutterstock

Graph



Graphen als Tabelle

	SB	DO	F	HB	H	HH
SB						
DO	354					
F	177	220				
HB		239				
H		212	355	130		
HH				121	160	

Diese Tabelle ist eine sogenannte **Adjanzenzmatrix** – sie drückt aus, welche Knoten benachbart (adjazent) sind.

Graphen als Tabelle

	SB	DO	F	HB	H	HH
SB		354	177			
DO	354		220	239	212	
F	177	220			355	
HB		239			130	121
H		212	355	130		160
HH				121	160	

Wir ergänzen die Tabelle um beide Richtungen – es könnte ja Einbahnstraßen geben

---

Graphen als Tabelle

	SB	DO	F	HB	H	HH
SB	0	354	177			
DO	354	0	220	239	212	
F	177	220	0		355	
HB		239		0	130	121
H		212	355	130	0	160
HH				121	160	0

Jeder Ort ist von sich selbst 0 km entfernt

Graphen als Tabelle

	SB	DO	F	HB	H	HH
SB	0	354	177	∞	∞	∞
DO	354	0	220	239	212	∞
F	177	220	0	∞	355	∞
HB	∞	239	∞	0	130	121
H	∞	212	355	130	0	160
HH	∞	∞	∞	121	160	0

Nicht verbundene Orte sind unendlich weit voneinander entfernt

Tabellen in C

- Eine *Tabelle* in C wird so deklariert:

0	1
2	3
4	5

```
int a[2][3] = {
    {0, 1},  // a[0]
    {2, 3},  // a[1]
    {4, 5}   // a[2]
};
```

- Eigentlich ein *Feld aus 3 Feldern* mit je 2 Elementen

Konstanten

- Feldgrößen ändern sich oft und tauchen im restlichen Programm immer wieder auf
- Mit einer *Variablen* habe ich *eine Stelle* im Programm, die die Feldgröße bestimmt:

```
int COLS = 2;
int ROWS = 3;

int a[COLS][ROWS] = {
    {0, 1},  // a[0]
    {2, 3},  // a[1]
    {4, 5}   // a[2]
};
```

Konstanten

- Problem: Feldgrößen müssen *konstant* sein
- Lösung: Variable als *Konstante* markieren

```
const int COLS = 2;
const int ROWS = 3;

int a[COLS][ROWS] = {
    {0, 1},  // a[0]
    {2, 3},  // a[1]
    {4, 5}   // a[2]
};
```

- Ältere Alternative: `#define COLS 2`

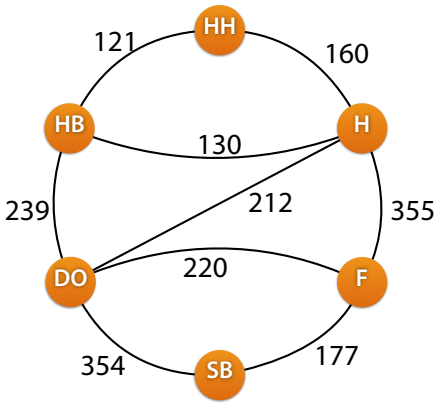
Städte und Distanzen

```
const int CITIES = 6;
const char *names[CITIES] =
{ "SB", "DO", "F", "HB", "H", "HH" };

const int INF = 100000;
const int dist[CITIES][CITIES] =
{
    {0, 354, 177, INF, INF, INF},
    {354, 0, 220, 239, 212, INF},
    {177, 220, 0, INF, 355, INF},
    {INF, 239, INF, 0, 130, 121},
    {INF, 212, 355, 130, 0, 160},
    {INF, INF, INF, 121, 160, 0}
};
```

Tabelle ausgeben

```
void print_dist() {
  for (int i = 0; i < CITIES; i++) {
    for (int j = i; j < CITIES; j++) {
      int distance = dist[i][j];
      char buffer[20];
      sprintf(buffer, "%-3s->%3s:%7dkm",
        names[i], names[j], distance);
      Serial.println(buffer);
    }
  }
}
```

```
SB -> SB:      0km
SB -> DO:    354km
SB ->  F:    177km
SB -> HB: 100000km
SB ->  H: 100000km
SB -> HH: 100000km
DO -> DO:      0km
DO ->  F:    220km
DO -> HB:    239km
DO ->  H:    212km
DO -> HH: 100000km
F  ->  F:      0km
F  -> HB: 100000km
F  ->  H:    355km
F  -> HH: 100000km
HB -> HB:      0km
HB ->  H:    130km
HB -> HH:    121km
H  ->  H:      0km
H  -> HH:    160km
HH -> HH:      0km
```



Die Tabelle enthält jetzt aber noch jede Menge überflüssiges Material – etwa nicht existente Verbindungen (SB → HB) oder banale Verbindungen (SB → SB)

Tabelle ausgeben

```
void print_dist() {
  for (int i = 0; i < CITIES; i++) {
    for (int j = i; j < CITIES; j++) {
      int distance = dist[i][j];
      if (distance > 0 && distance < INF) {
        char buffer[20];
        sprintf(buffer, "%-3s->%3s:%7dkm",
          names[i], names[j], distance);
        Serial.println(buffer);
      }
    }
  }
}
```

Wir müssen also entsprechende Prüfungen einbauen

Break und Continue

- Die Anweisungen “break” und “continue” springen an das Ende der Schleife (“}”)
- “break” setzt die Ausführung *nach* dem Schleifenende fort (= bricht die Schleife ab)
- “continue” setzt die Ausführung *am* Schleifenende fort (= beginnt den nächsten Durchlauf)

Hierbei helfen uns zwei Konstrukte

Continue

```
void f() {
  for (int i = 0; i < 10; i++) {
    if (i % 2 == 0)
      continue;
    Serial.println("Ungerade");
  }
}
```

ist dasselbe wie

```
void f() {
  for (int i = 0; i < 10; i++) {
    if (i % 2 != 0) {
      Serial.println("Ungerade");
    }
  }
}
```

Break

```
int f() {
  for (int i = 10; i >= 0; i--) {
    if (is_perfect(i)) {
      break;
  }
  return i;
}
```

ist dasselbe wie

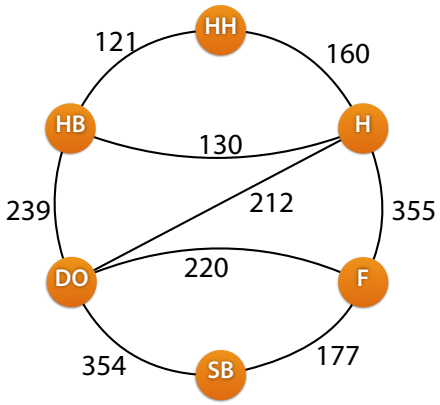
```
int f() {
  for (int i = 10; i >= 0; i--) {
    if (is_perfect(i)) {
      return i;
  }
  return -1;
}
```

Tabelle ausgeben

```
void print_dist() {
  for (int i = 0; i < CITIES; i++) {
    for (int j = i; j < CITIES; j++) {
      int distance = dist[i][j];
      if (distance == 0)
        continue;  // Eigene Stadt
      if (distance >= INF)
        continue;  // Keine Verbindung

      char buffer[20];
      sprintf(buffer, "%-3s->%3s:%7dkm",
          names[i], names[j], distance);
      Serial.println(buffer);
    }
  }
}
```

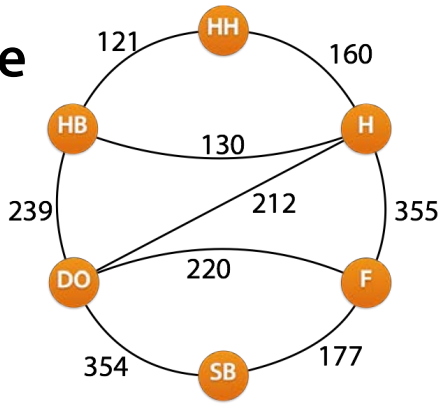
Mit break und continue kann man Sonderfälle (hier: Distanzen 0 und INF) zu Beginn eines Blocks abarbeiten und jeweils mit einem Kommentar versehen. Der Rest des Blocks macht dann die eigentliche Arbeit.

```
SB -> DO:     354km
SB ->  F:     177km
DO ->  F:     220km
DO -> HB:     239km
DO ->  H:     212km
F  ->  H:     355km
HB ->  H:     130km
HB -> HH:     121km
H  -> HH:     160km
```

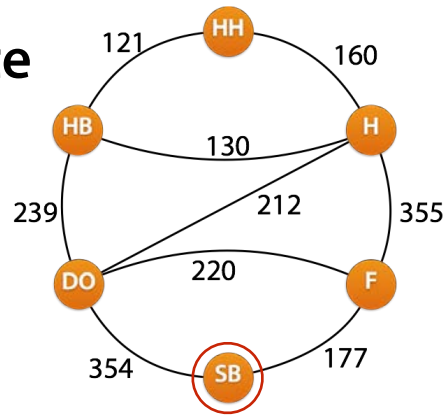


Der kürzeste Weg

- Idee: Wir bestimmen *alle* Pfade…
- …und nehmen den *kürzesten*

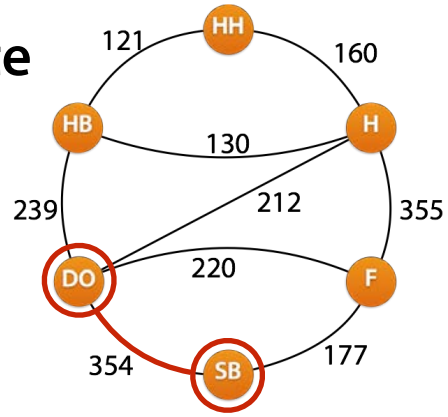


Der kürzeste Weg



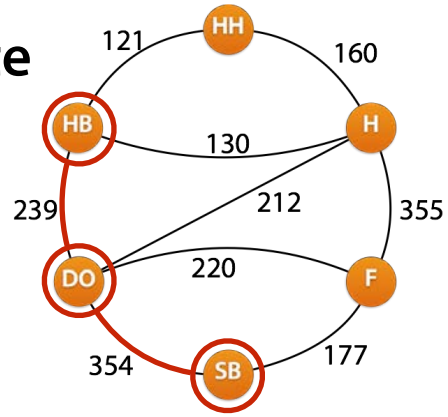
Von jedem Punkt aus prüfen wir der Reihe nach alle Wege, die zu bisher umbesuchten Punkten führen

Der kürzeste Weg



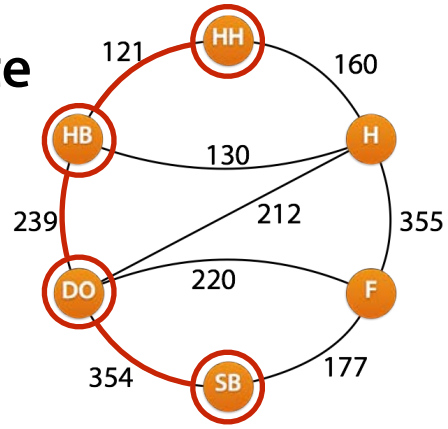
Beginnen wir mit Dortmund

Der kürzeste Weg



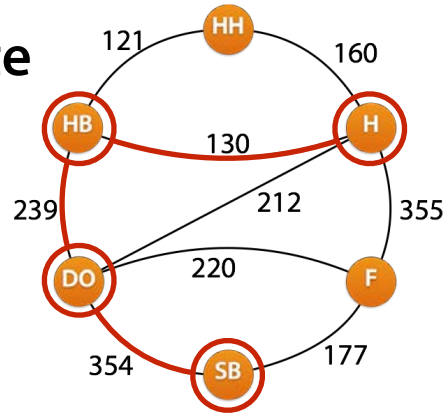
Von Dortmund kommen wir nach Bremen

Der kürzeste Weg



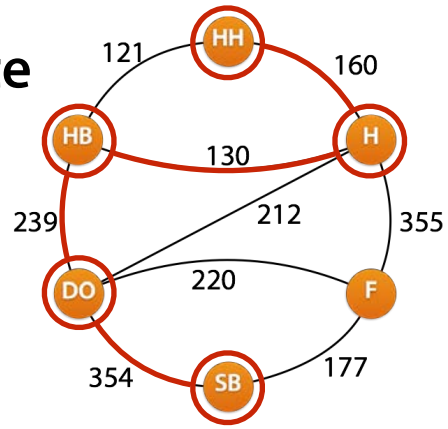
und von Bremen nach Hamburg, wo wir das Ziel erreicht haben.

Der kürzeste Weg



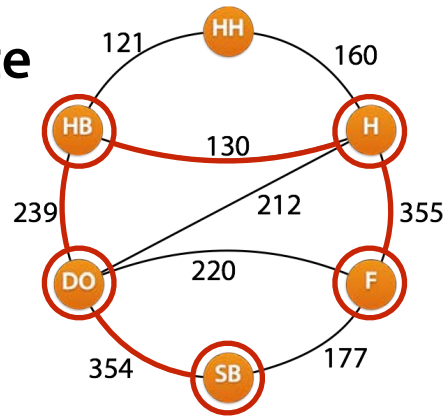
Können wir auch über Hannover gehen?

Der kürzeste Weg



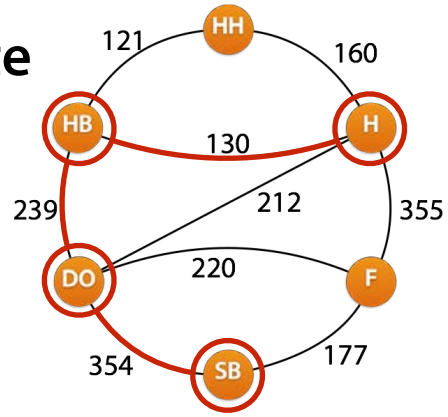
Ja, aber das wird länger.

Der kürzeste Weg



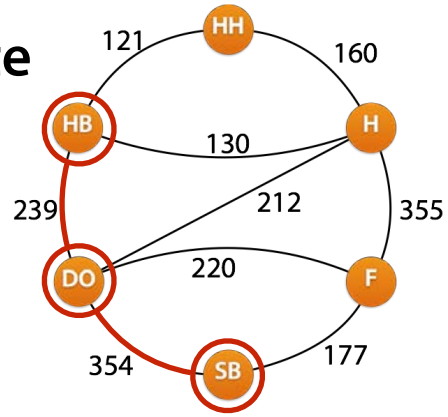
Von Hannover nach Frankfurt? Da kommen wir nicht weiter.

Der kürzeste Weg



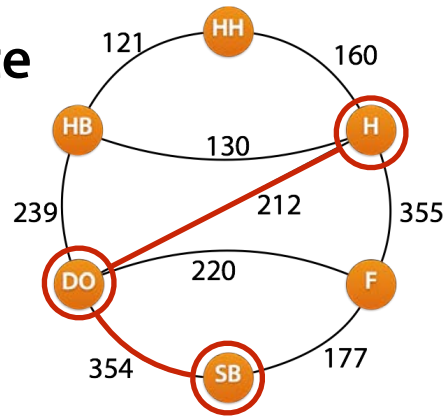
So, das war's mit Hannover

Der kürzeste Weg



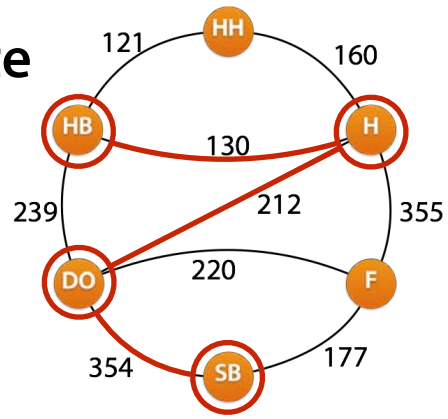
und auch mit Bremen

Der kürzeste Weg

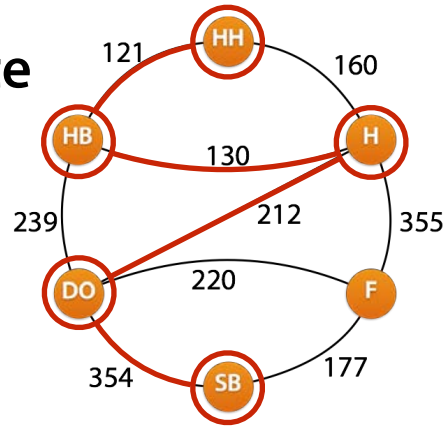


von Dortmund aus gibt es noch Alternativen. Erstmal Hannover.

Der kürzeste Weg

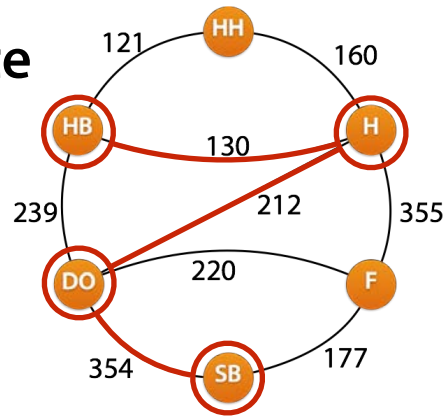


Der kürzeste Weg

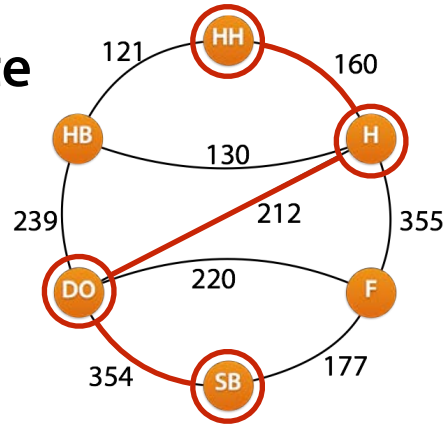


So kommt man auch nach Hamburg

Der kürzeste Weg

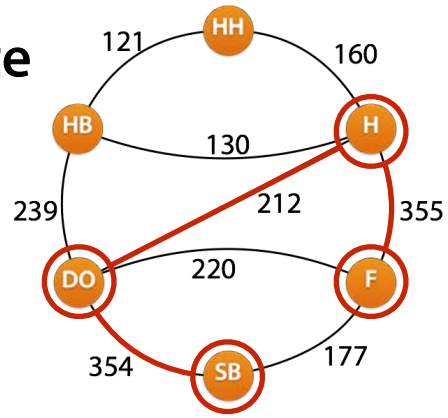


Der kürzeste Weg



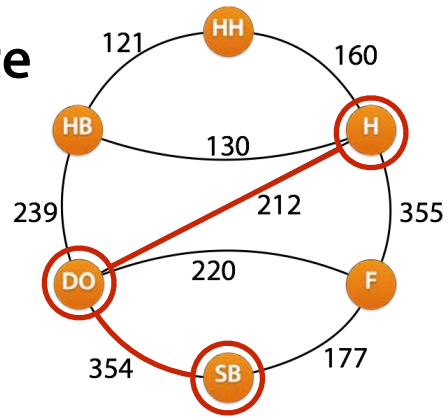
Schneller ist der direkte Weg.

Der kürzeste Weg



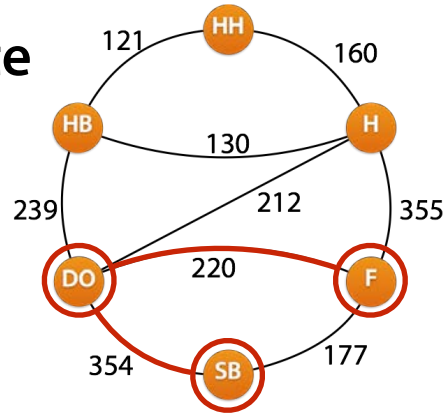
Von Frankfurt aus kommen wir nicht mehr weg.

Der kürzeste Weg



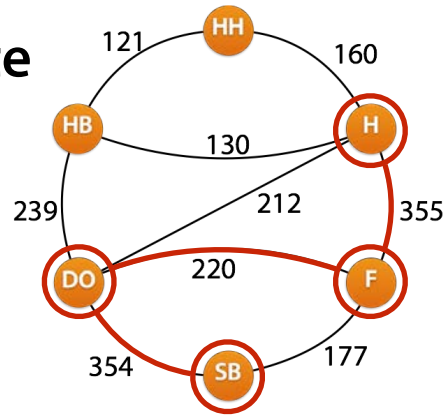
Das waren alle Alternativen aus Hannover.

Der kürzeste Weg



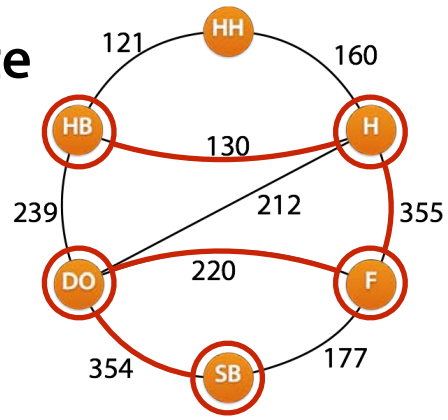
Jetzt haben wir von Dortmund aus Bremen und Hannover probiert. Fehlt noch Frankfurt.

Der kürzeste Weg



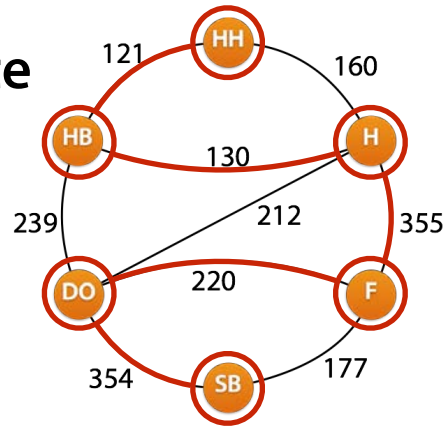
Von Frankfurt aus geht's nach Hannover. (Überall sonst waren wir schon.)

Der kürzeste Weg



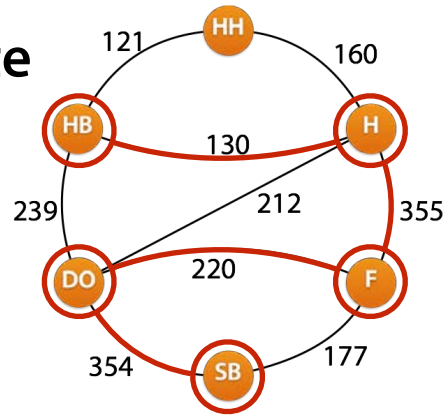
von Hannover können wir über Bremen…

Der kürzeste Weg

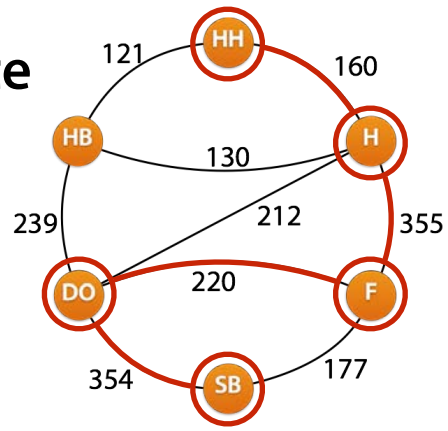


…nach Hamburg

Der kürzeste Weg

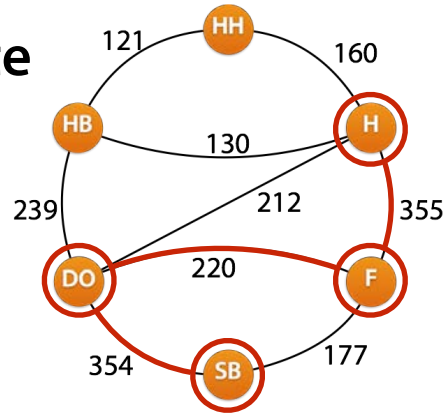


Der kürzeste Weg

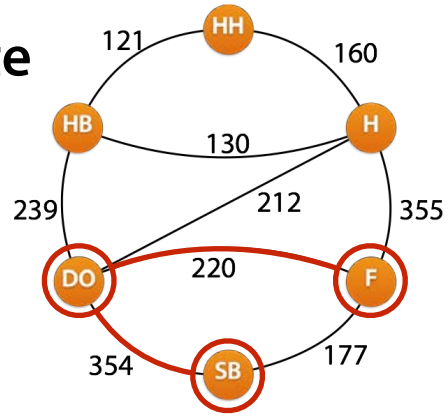


Schneller geht es erneut direkt.

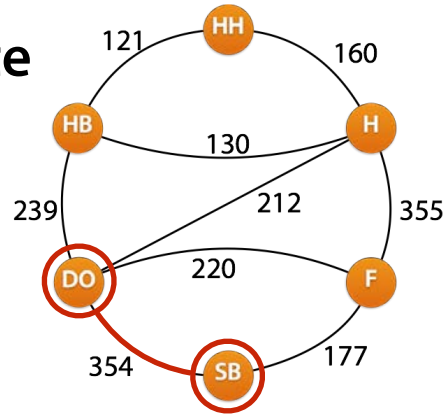
Der kürzeste Weg



Der kürzeste Weg

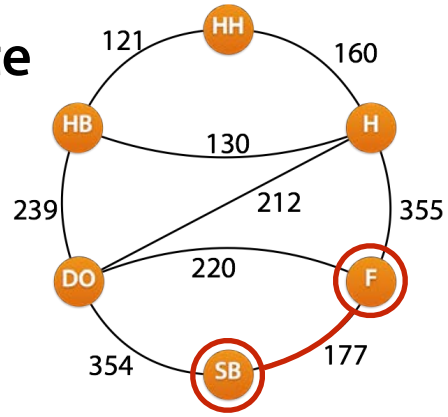


Der kürzeste Weg



Wir haben alle Alternativen über Dortmund durch;

Der kürzeste Weg



jetzt ist Frankfurt dran. (Die Kombinationen sparen wir uns.)

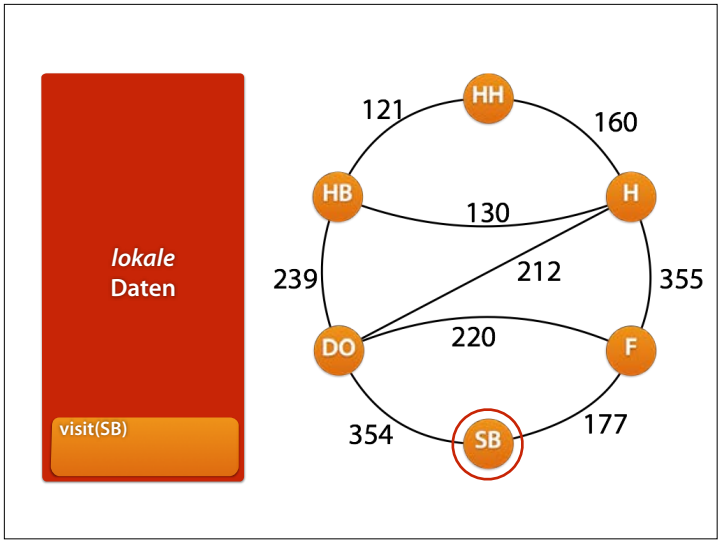
Plan

- Wir *besuchen* einen Knoten *node*
- Von *node* aus besuchen wir alle Nachbarn, die wir noch nicht besucht haben…
- …und besuchen jeweils von dort aus *(rekursiv)* erneut deren Nachbarn.

Städte besuchen

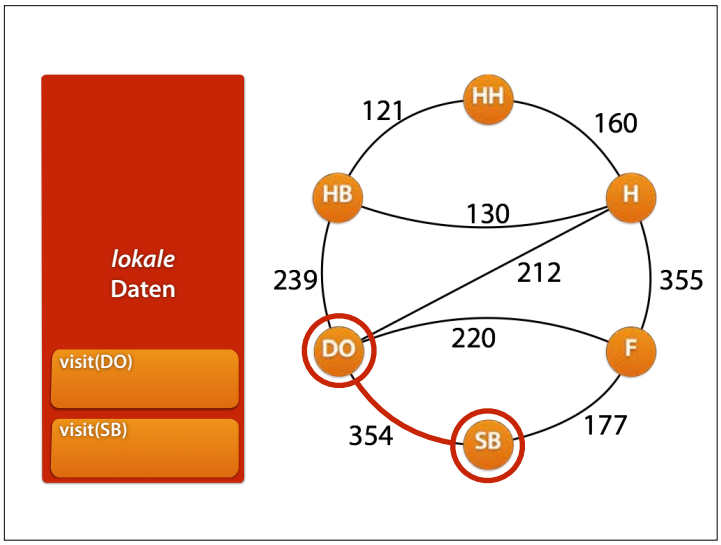
```
int visited[CITIES] = {0, 0, 0, 0, 0, 0};

void visit(int node) {
  visited[node] = 1;
  for (int neighbor = 0; neighbor < CITIES;
       neighbor++)
  {
    if (visited[neighbor])
      continue;

    int distance = dist[node][neighbor];
    if (distance > 0 && distance < INF)
      visit(neighbor);
 }
  visited[node] = 0;
}
```

Mit break und continue kann man Sonderfälle (hier: Distanzen 0 und INF) zu Beginn eines Blocks abarbeiten und jeweils mit einem Kommentar versehen. Der Rest des Blocks macht dann die eigentliche Arbeit.

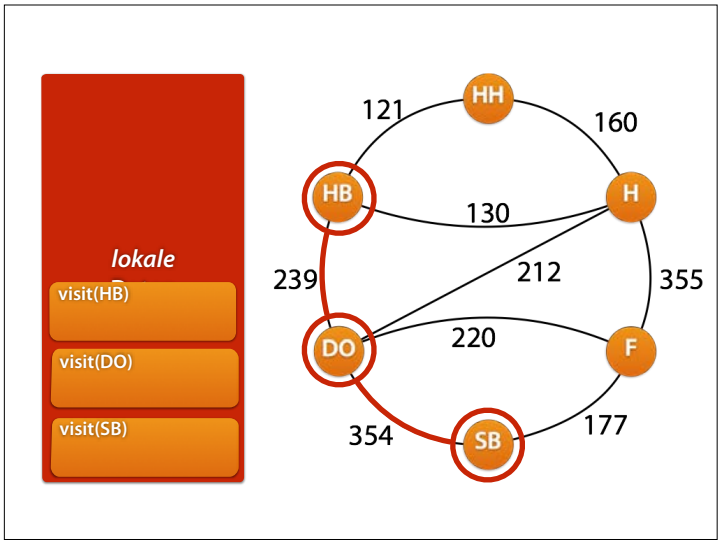


Von jedem Punkt aus prüfen wir der Reihe nach alle Wege, die zu bisher umbesuchten Punkten führen

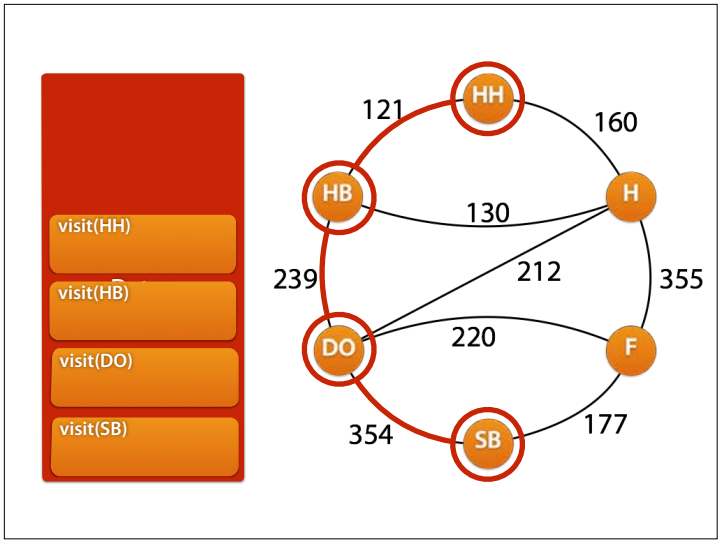
---



Und mit jedem Schritt rufen wir visit() auf dem Nachbarn auf

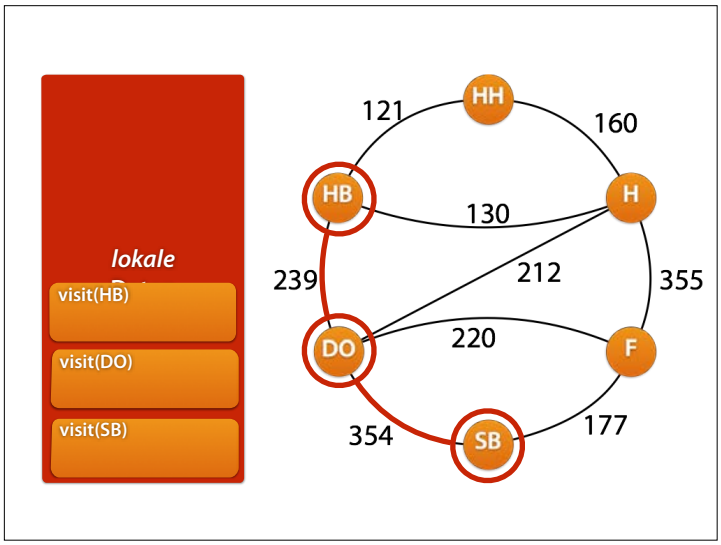


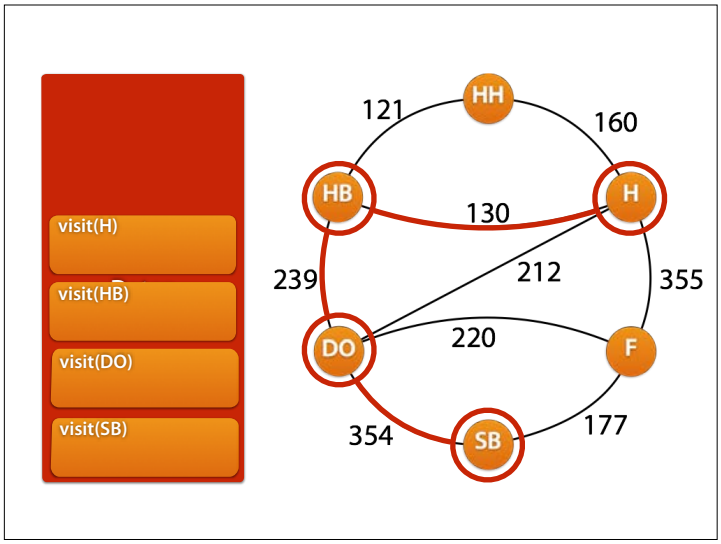
Der Funktionsstapel entspricht dem Pfad, den wir gerade abarbeiten

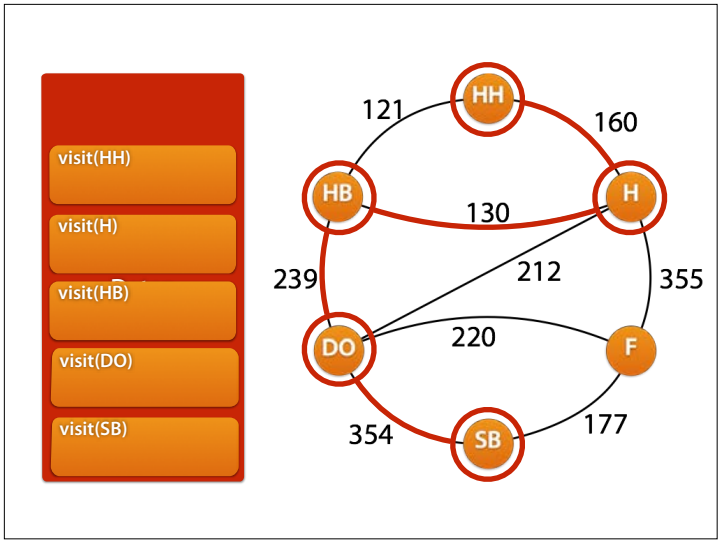


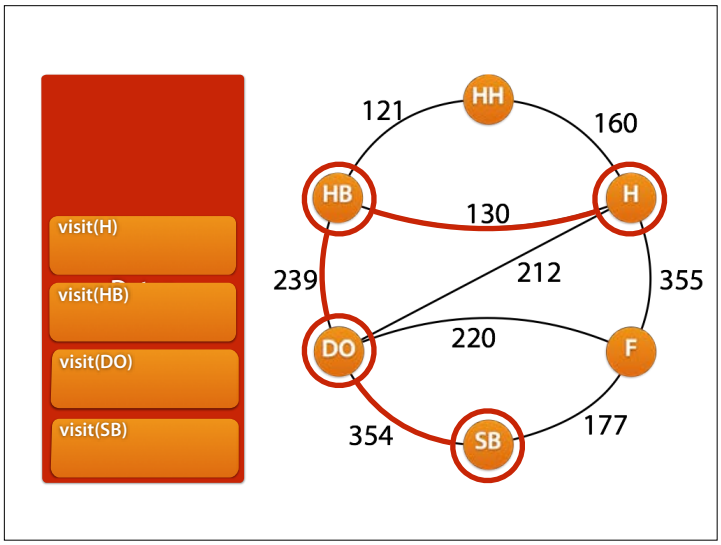
und immer probiert die visit()-Funktion alle bisher unbesuchten Nachbarn

---






visit(HH)
visit(H)
visit(HB)
visit(DO)
visit(SB)
HH
HB
H
DO
F
SB
121
160
130
239
212
355
220
354
177


visit(H)
visit(HB)
visit(DO)
visit(SB)
HH
HB
H
DO
F
SB
121
160
130
239
212
355
220
354
177

Demo

Alle Pfade nach HH

```
-> SB (0 km)
    -> DO (354 km)
        -> F (574 km)
            -> H (929 km)
                -> HB (1059 km)
                    -> HH (1271 km) (MINIMUM)
                -> HH (1089 km) (MINIMUM)
                    -> HB (1210 km)
        -> HB (593 km)
            -> H (723 km)
                -> F (1078 km)
                -> HH (883 km) (MINIMUM)
            -> HH (805 km) (MINIMUM)
                -> H (965 km)
                    -> F (1320 km)
        -> H (566 km)
            -> F (921 km)
```

```
        -> DO (397 km)
            -> HB (636 km)
                -> H (766 km)
                    -> HH (926 km)
                -> HH (848 km)
                    -> H (1008 km)
            -> H (609 km)
                -> HB (739 km)
                    -> HH (951 km)
               -> HH (769 km)
                    -> HB (890 km)
        -> H (532 km)
            -> DO (744 km)
                -> HB (983 km)
                    -> HH (1195 km)
            -> HB (662 km)
                -> DO (901 km)
                -> HH (874 km)
            -> HH (692 km) (MINIMUM)
                -> HB (813 km)
                    -> DO (1052 km)
```

Weitere Schritte

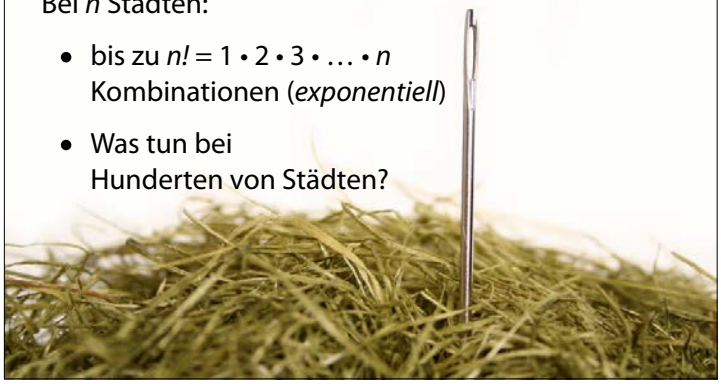
- Wir müssen noch zu jedem Pfad bestimmen
  - was der *bisher kürzeste Pfad* war
  - welche *Städte* (Knoten) enthalten sind

Aber das sparen wir uns, denn wir haben ein Problem

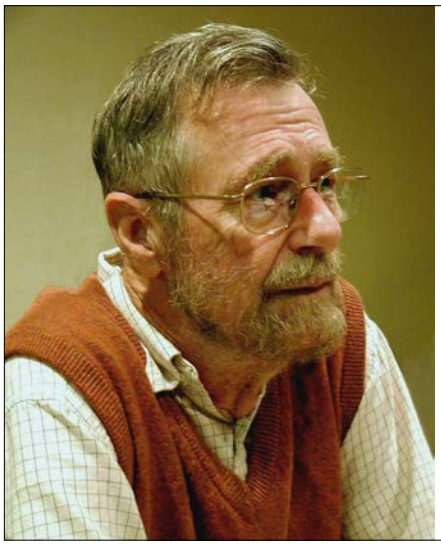
Komplexität

Bei *n* Städten:

- bis zu *n!* = 1 • 2 • 3 • … • *n* Kombinationen (*exponentiell*)
- Was tun bei Hunderten von Städten?



Wenn ich bei jeder Suche erst einmal **alle** Daten durchforsten muss, dauert das sehr lange.



Edsger W. Dijkstra

1930–2002

Edsger Wybe Dijkstra (* 11. Mai 1930 in Rotterdam; † 6. August 2002 in Nuenen, Niederlande) war ein niederländischer Informatiker. Er war der Wegbereiter der strukturierten Programmierung.[1] 1972 erhielt er den Turing Award.

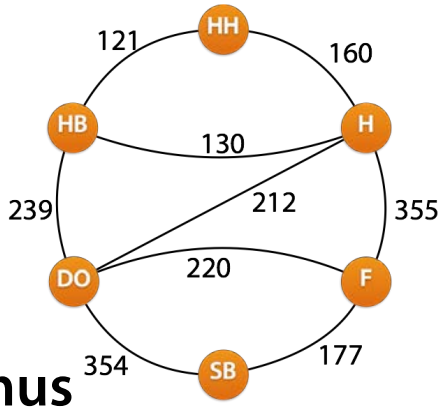
Unter seinen Beiträgen zur Informatik finden sich der Dijkstra-Algorithmus zur Berechnung eines kürzesten Weges in einem Graphen (1959 in einem dreiseitigen Artikel veröffentlicht), die erstmalige Einführung von Semaphoren zur Synchronisation zwischen Threads

---

Dijkstras Algorithmus

- Berechnet kürzeste Wege in einem Graphen
- Idee: Zu jeder Stadt merken wir uns
  - was der *bisher kürzeste Weg* war
  - von *welcher Stadt* er ausging
- Wir beginnen stets mit der kürzesten Distanz

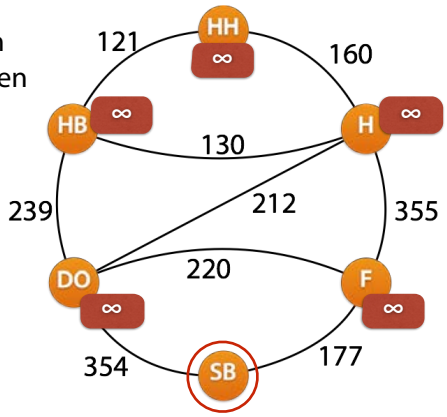
Dijkstras Algorithmus



Von jedem Punkt aus prüfen wir der Reihe nach alle Wege, die zu bisher umbesuchten Punkten führen

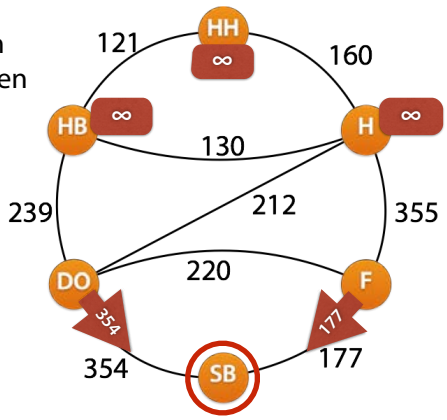
---

- Wir besuchen den bisher unbesuchten Knoten *n* mit der *kürzesten Distanz*
- Bei jedem Nachbarn von *n* aktualisieren wir *Entfernung* und *Richtung*



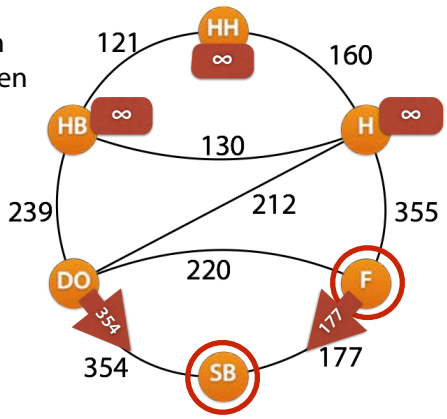
Zu Beginn ist der nächste unbesuchte Knoten SB selbst; dessen Distanz ist 0 km.

- Wir besuchen den bisher unbesuchten Knoten *n* mit der *kürzesten Distanz*
- Bei jedem Nachbarn von *n* aktualisieren wir *Entfernung* und *Richtung*



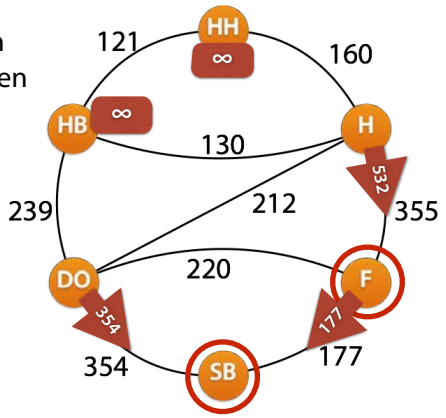
Wir merken uns für jeden Nachbarn die Richtung und Entfernung zum Start

- Wir besuchen den bisher unbesuchten Knoten *n* mit der *kürzesten Distanz*
- Bei jedem Nachbarn von *n* aktualisieren wir *Entfernung* und *Richtung*



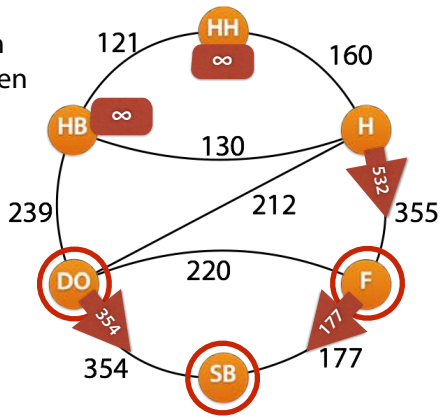
Wir wählen den Knoten mit dem geringsten Abstand zum Start (hier F)

- Wir besuchen den bisher unbesuchten Knoten *n* mit der *kürzesten Distanz*
- Bei jedem Nachbarn von *n* aktualisieren wir *Entfernung* und *Richtung*



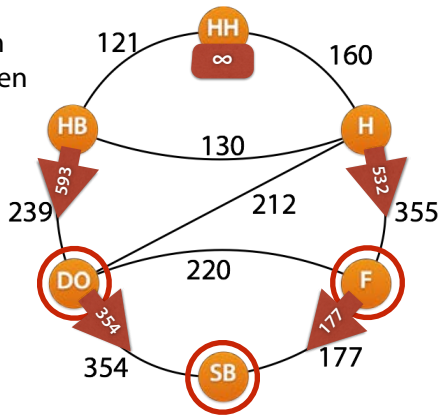
Nun aktualisieren wir Richtung und Entfernung für alle Nachbarn von F. Von H aus geht der kürzeste Pfad zunächst über F; von DO aus lohnt es sich nicht, über F zu fahren.

- Wir besuchen den bisher unbesuchten Knoten *n* mit der *kürzesten Distanz*
- Bei jedem Nachbarn von *n* aktualisieren wir *Entfernung* und *Richtung*



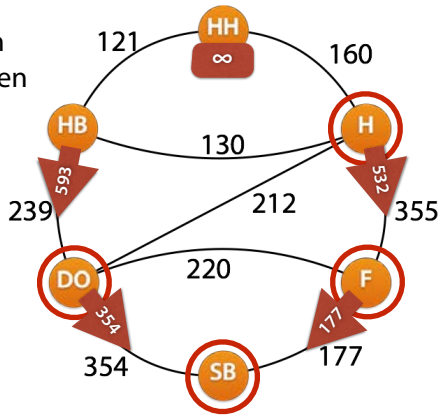
Nun hat DO die kürzeste Entfernung; wir schauen uns seine Nachbarn an. Von H aus lohnt es Sicht nicht, über DO zu fahren; aber wir können HB aktualisieren.

- Wir besuchen den bisher unbesuchten Knoten *n* mit der *kürzesten Distanz*
- Bei jedem Nachbarn von *n* aktualisieren wir *Entfernung* und *Richtung*



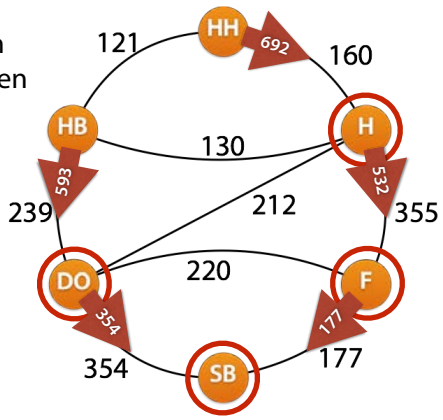
HB: 354 + 239 = 593km.

- Wir besuchen den bisher unbesuchten Knoten *n* mit der *kürzesten Distanz*
- Bei jedem Nachbarn von *n* aktualisieren wir *Entfernung* und *Richtung*



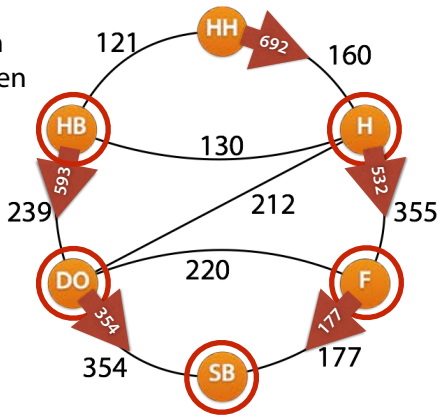
Nun ist H der Knoten mit der geringsten Entfernung.

- Wir besuchen den bisher unbesuchten Knoten *n* mit der *kürzesten Distanz*
- Bei jedem Nachbarn von *n* aktualisieren wir *Entfernung* und *Richtung*



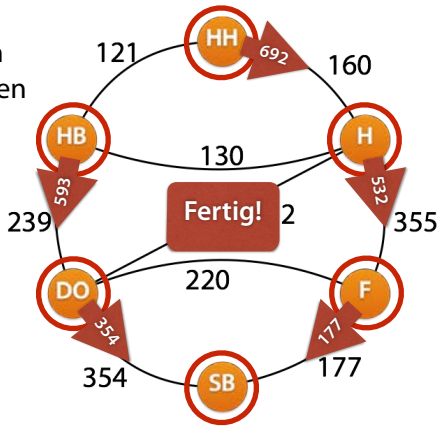
Nun ist H der Knoten mit der geringsten Entfernung, und wir haben HH erreicht. Eigentlich könnten wir jetzt aufhören.

- Wir besuchen den bisher unbesuchten Knoten *n* mit der *kürzesten Distanz*
- Bei jedem Nachbarn von *n* aktualisieren wir *Entfernung* und *Richtung*



Von HB aus ergeben sich auch keine Änderungen.

- Wir besuchen den bisher unbesuchten Knoten *n* mit der *kürzesten Distanz*
- Bei jedem Nachbarn von *n* aktualisieren wir *Entfernung* und *Richtung*



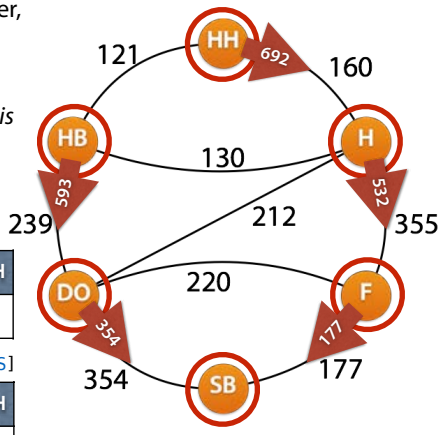
Und wir sind fertig. Für jeden Ort wissen wir jetzt, in welche Richtung wir wie weit fahren müssen.

Komplexität

- Bei n Städten bis zu n^2 Schritte (*quadratisch*)
- In der Praxis *hierarchisches* Vorgehen: erst große Städte, dann lokale Optimierungen

Wenn ich bei jeder Suche erst einmal **alle** Daten durchforsten muss, dauert das sehr lange.

- Wir nutzen zwei Felder, um *Richtung* und *Distanz* abzulegen
- Enthalten das *Ergebnis* des Algorithmus



`int direction[CITIES]`

SB	DO	F	HB	H	HH
SB	SB	SB	DO	F	H

`int shortest_dist[CITIES]`

SB	DO	F	HB	H	HH
0	354	177	593	532	692

Wie speichern wir diese Markierungen?

Dijkstras Algorithmus

```
// Dijkstra's algorithm stores the paths in these fields
// For each city, the direction (as city index) in which to go
int direction[CITIES];

// For each city, the length of the shortest path to TARGET
int shortest_dist[CITIES];

// Compute shortest paths from all cities to TARGET
void shortest_path_dijkstra(int target)
{
    // 1 if already visited
    int visited[CITIES];

    // Initialize fields
    for (int city = 0; city < CITIES; city++) {
        shortest_dist[city] = INF;
        direction[city] = city;
        visited[city] = 0;
    }
    shortest_dist[target] = 0;
```

Nur zur Vollständigkeit – Sie müssen den Code weder vollständig verstehen noch abändern.

```
        shortest_dist[city] = INF;
        direction[city] = city;
        visited[city] = 0;
    }
    shortest_dist[target] = 0;

    // We process all cities until all are visited
    while (target != INF) {
        // Mark target as visited
        visited[target] = 1;

        // Update distances and directions
        // of all unvisited neighbors of target
        for (int neighbor = 0; neighbor < CITIES; neighbor++)
        {
            if (!visited[neighbor] && dist[target][neighbor] < INF)
            {
                // Compute distance from neighbor via TARGET
                int dist_via_target =
                    dist[target][neighbor] + shortest_dist[target];

                // If it's shorter, use TARGET as new direction
                if (dist_via_target < shortest_dist[neighbor])
                {
                    shortest_dist[neighbor] = dist_via_target;
                    direction[neighbor] = target;
                }
            }
        }
```

Wenn Sie aber einen Fehler finden, bitte melden.

```
                // If it's shorter, use TARGET as new direction
                if (dist_via_target < shortest_dist[neighbor])
                {
                    shortest_dist[neighbor] = dist_via_target;
                    direction[neighbor] = target;
                }
            }
        }

        // Compute next target
        // If we find no new target, its value stays INF
        target = INF;

        // As next target, use the unvisited node
        // with the minimum overall distance
        int min_dist = INF;

        for (int city = 0; city < CITIES; city++)
        {
            if (!visited[city] && shortest_dist[city] < min_dist)
            {
                // Shortest distance to target found
                target = city;
                min_dist = shortest_dist[city];
            }
        }
    }
}
```

Tabelle ausgeben

```
int visited[CITIES] = {0, 0, 0, 0, 0, 0};
int direction[CITIES];
int shortest_dist[CITIES];

// fill direction and shortest_dist to target
void shortest_path_dijkstra(int target) {
   // code goes here
}

// Print results for all targets
void print_dijkstra(int target) {
  for (int start = 0; start < CITIES; start++) {
    char buf[100];
    sprintf(buf, "From%3s to%3s:%4d km via%3s",
            names[start], names[target],
            shortest_dist[start],
            names[direction[start]]);
    Serial.println(buf);
  }
}
```

Hier berechnen wir zunächst die kürzesten Wege, und geben sie anschließend aus

Alle Tabellen

```
int visited[CITIES] = {0, 0, 0, 0, 0, 0};
int direction[CITIES];
int shortest_dist[CITIES];

// fill direction and shortest_dist to target
void shortest_path_dijkstra(int target) { ...}

// Print results for all targets
void print_dijkstra(int target) { ... }

void setup() {
  for (int target = 0; target < CITIES; target++)
  {
      shortest_path_dijkstra(target);
      print_dijkstra(target);
  }
}
```

Demo

Alle Tabellen

```
From SB to SB:   0 km via SB
From DO to SB: 354 km via SB
From  F to SB: 177 km via SB
From HB to SB: 593 km via DO
From  H to SB: 532 km via  F
From HH to SB: 692 km via  H

From SB to  F: 177 km via  F
From DO to  F: 220 km via  F
From  F to  F:   0 km via  F
From HB to  F: 459 km via DO
From  H to  F: 355 km via  F
From HH to  F: 515 km via  H

From SB to  H: 532 km via  F
From DO to  H: 212 km via  H
From  F to  H: 355 km via  H
From HB to  H: 130 km via  H
From  H to  H:   0 km via  H
From HH to  H: 160 km via  H

From SB to DO: 354 km via DO
From DO to DO:   0 km via DO
From  F to DO: 220 km via DO
From HB to DO: 239 km via DO
From  H to DO: 212 km via DO
From HH to DO: 372 km via  H

From SB to HB: 593 km via DO
From DO to HB: 239 km via HB
From  F to HB: 459 km via DO
From HB to HB:   0 km via HB
From  H to HB: 130 km via HB
From HH to HB: 212 km via HB

From SB to HH: 692 km via  F
From DO to HH: 360 km via HB
From  F to HH: 515 km via  H
From HB to HH: 121 km via HH
From  H to HH: 160 km via HH
From HH to HH:   0 km via HH
```

Schwenker am Lenker

```
From SB to SB:   0 km via SB
From DO to SB: 354 km via SB
From  F to SB: 177 km via SB
From HB to SB: 593 km via DO
From  H to SB: 532 km via  F
From HH to SB: 692 km via  H
```

Und so wissen Sie stets, wie Sie am schnellsten ins Saarland zurückkommen.

Ihre Aufgabe

Ihre Aufgabe

Ihre Aufgabe

Werte austauschen

- Wir wollen eine Funktion swap(*a*, *b*) schreiben, die die Werte von *a* und *b* vertauscht

```
int x = 1; int y = 2;
swap(x, y);
// x = 2, y = 1
```

Erster Versuch

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```

Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```

Werte austauschen

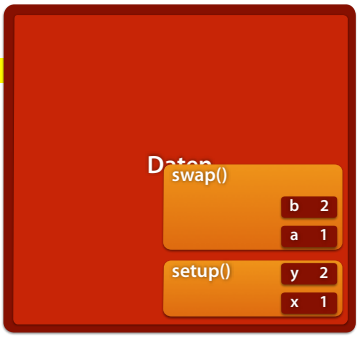
```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```

Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```

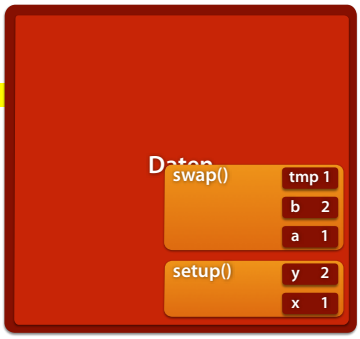
Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```



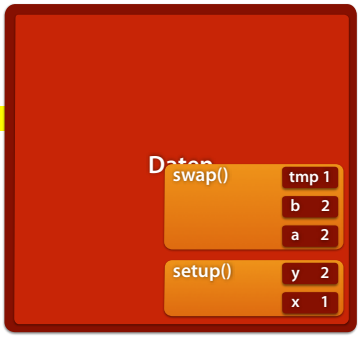
Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```



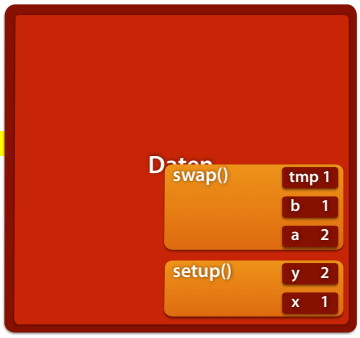
Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```



Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```



So weit, so gut – jetzt haben wir a und b vertauscht. Leider wirkt sich das nicht auf den Aufrufer aus.

Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```

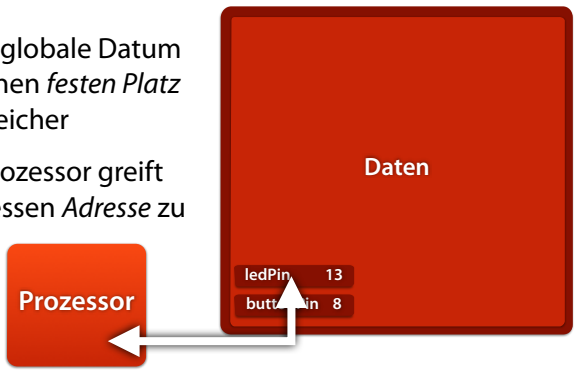
In setup() nämlich bleiben alle Werte so, wie sie waren :-(

Werte austauschen

- Eine Funktion kann die lokalen Variablen einer anderen Funktion nicht verändern…
- …es sei denn, sie benutzt einen *Zeiger*

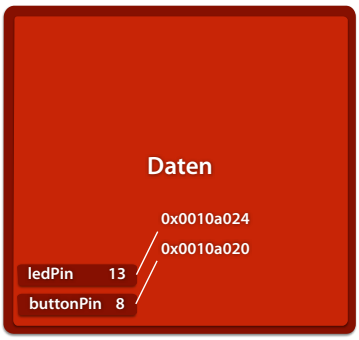
Speicherorte

- Jedes globale Datum hat einen *festen Platz* im Speicher
- Der Prozessor greift auf dessen *Adresse* zu



Speicherorte

- Eine *Adresse* sagt dem Prozessor, wo der Wert zu finden ist
- Eine *Zahl* ähnlich einer Hausnummer
- Die Adresse von *ledPin* könnte etwa **0x0010a024** sein



0x0010a0024

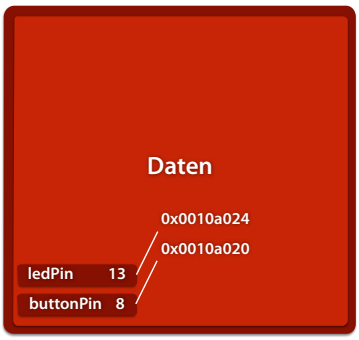
- Hexadezimalzahl = Zahl zur Basis 16
- In C mit Präfix 0x geschrieben
- Ziffern: 0–9 wie bekannt, zudem
  A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
- 0xA3 ist also $10 \cdot 16^1 + 3 = 163$
- 0xAFFE ist
  $10 \cdot 16^3 + 15 \cdot 16^2 + 15 \cdot 16^1 + 14 = 45054$

17432612

- Hexadezimalzahl = Zahl zur Basis 16
- In C mit Präfix 0x geschrieben
- Ziffern: 0–9 wie bekannt, zudem
  A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
- 0xA3 ist also $10 \cdot 16^1 + 3 = 163$
- 0xAFFE ist
  $10 \cdot 16^3 + 15 \cdot 16^2 + 15 \cdot 16^1 + 14 = 45054$

Adressen

- In C liefert ***&x*** die *Adresse* von *x:*
- &ledPin = 0x0010a024
- &buttonPin = 0x0010a020



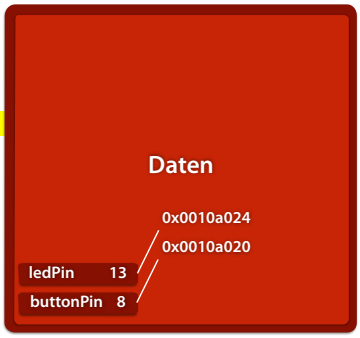
Zeiger

- Ein *Zeiger* ist eine Variable, die die *Adresse* einer Variablen speichert
- Man sagt: Der Zeiger "zeigt" auf die Variable
- Ein Zeiger mit Namen *p*, der auf einen Typ T zeigt, wird als *T *p* deklariert:

```
int *p1 = &ledPin;
```

Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
}
```



Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
}
```

Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
}
```

Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
}
```

Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
}
```

In Wahrheit brauchen wir aber die genauen Adressen gar nicht; es genügt zu wissen, dass p1 auf ledPin zeigt – also die Adresse von ledPin enthält.

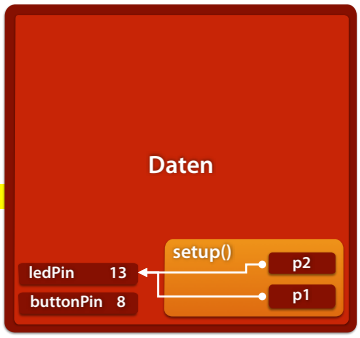
Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
  int *p2 = p1;
}
```

In Wahrheit brauchen wir aber die genauen Adressen gar nicht; es genügt zu wissen, dass p1 auf ledPin zeigt – also die Adresse von ledPin enthält.

Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
  int *p2 = p1;
}
```



Fügen wir noch einen weiteren Zeiger hinzu

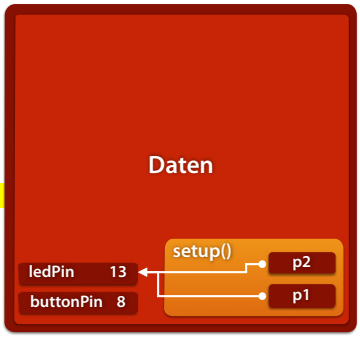
Dereferenzieren

- Der Ausdruck **p* steht für die Variable, auf die *p* zeigt (= die Variable an Adresse *p*)
- Man sagt: Der Zeiger wird *dereferenziert*
- **p* kann wie eine Variable benutzt werden

```
int *p1 = &ledPin;
int x = *p1;   // x = ledPin
*p1 = 25;      // ledPin = 25
```

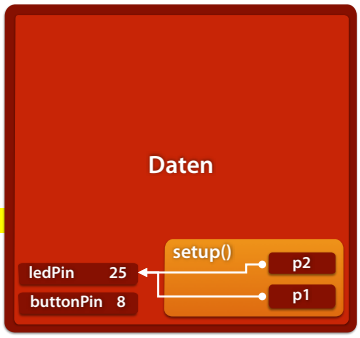
Dereferenzieren

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
 int *p2 = p1;
 *p2 = 25;
 p1 = &buttonPin;
 *p1 = *p2;
}
```



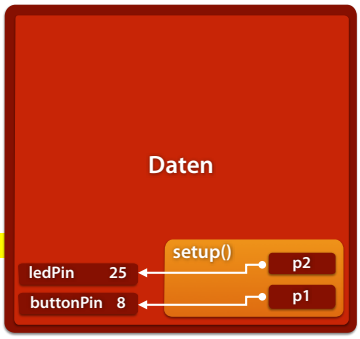
Dereferenzieren

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
 int *p2 = p1;
 *p2 = 25;
 p1 = &buttonPin;
 *p1 = *p2;
}
```



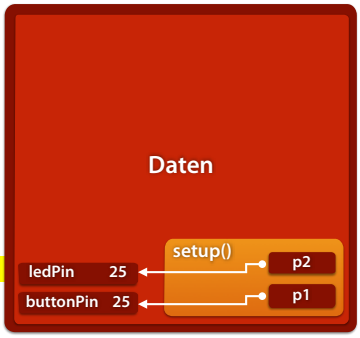
Dereferenzieren

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
 int *p2 = p1;
 *p2 = 25;
 p1 = &buttonPin;
 *p1 = *p2;
}
```



Dereferenzieren

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
 int *p2 = p1;
 *p2 = 25;
 p1 = &buttonPin;
 *p1 = *p2;
}
```



Werte austauschen

- Wir wollen eine Funktion swap(*a*, *b*) schreiben, die die Werte von *a* und *b* vertauscht
- Wir übergeben die *Adressen* von *a* und *b*

```
int x = 1; int y = 2;
swap(&x, &y);
// x = 2, y = 1
```

Tauschen mit Zeigern

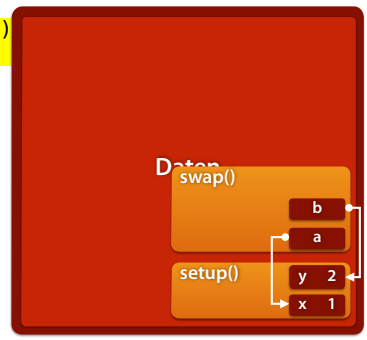
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Werte austauschen

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```

Werte austauschen

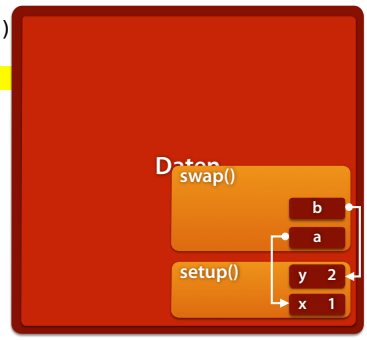
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```



a zeigt nun auf x, b auf y. *a ist der Wert, der an der Adresse von a steht – also der Wert von x.

Werte austauschen

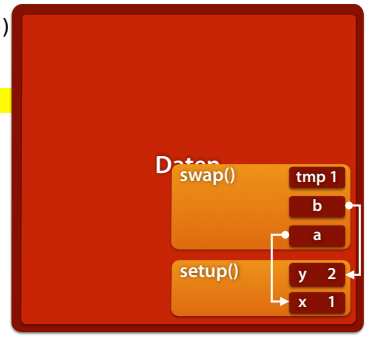
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```



a zeigt nun auf x, b auf y. *a ist der Wert, der an der Adresse von a steht – also der Wert von x.

Werte austauschen

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```

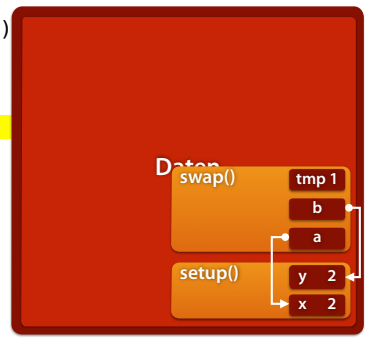


Wir können nun *a einen neuen Wert zuweisen – und verändern damit x

---

Werte austauschen

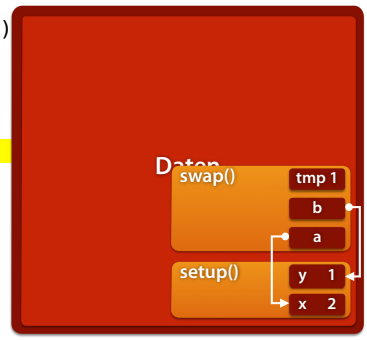
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```



*b verändert analog die Variable y.

Werte austauschen

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```



Werte austauschen

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```

Fertig!

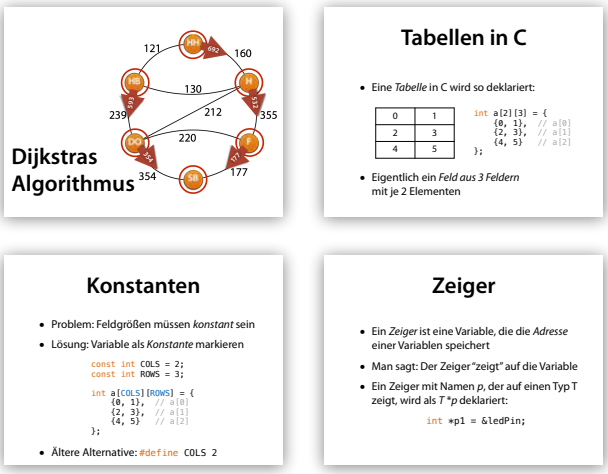
… und am Ende sind x und y (wie geplant) vertauscht!

Spaß mit Zeigern

Mit Zeigern kann man

- eine Funktion Variablen *verändern* lassen
- *Freispeicher* verwalten
- auf *Felder* zugreifen
- *komplexe Datenstrukturen* aufbauen

Mehr in der nächsten Woche



Tabellen in C

- Eine *Tabelle* in C wird so deklariert:

0	1
2	3
4	5

```
int a[2][3] = {
    {0, 1},  // a[0]
    {2, 3},  // a[1]
    {4, 5}   // a[2]
};
```

- Eigentlich ein *Feld aus 3 Feldern* mit je 2 Elementen

Konstanten

- Problem: Feldgrößen müssen *konstant* sein
- Lösung: Variable als *Konstante* markieren

```
const int COLS = 2;
const int ROWS = 3;

int a[COLS][ROWS] = {
    {0, 1},  // a[0]
    {2, 3},  // a[1]
    {4, 5}   // a[2]
};
```

- Ältere Alternative: `#define COLS 2`

Zeiger

- Ein *Zeiger* ist eine Variable, die die *Adresse* einer Variablen speichert
- Man sagt: Der Zeiger "zeigt" auf die Variable
- Ein Zeiger mit Namen *p*, der auf einen Typ T zeigt, wird als *T *p* deklariert:

```
int *p1 = &ledPin;
```

Handouts

Tabellen in C

- Eine *Tabelle* in C wird so deklariert:

0	1
2	3
4	5

```
int a[2][3] = {
    {0, 1},  // a[0]
    {2, 3},  // a[1]
    {4, 5}   // a[2]
};
```

- Eigentlich ein *Feld aus 3 Feldern* mit je 2 Elementen

Städte und Distanzen

```
const int CITIES = 6;
const char *names[CITIES] =
{ "SB", "DO", "F", "HB", "H", "HH" };

const int INF = 100000;
const int dist[CITIES][CITIES] =
{
    {0, 354, 177, INF, INF, INF},
    {354, 0, 220, 239, 212, INF},
    {177, 220, 0, INF, 355, INF},
    {INF, 239, INF, 0, 130, 121},
    {INF, 212, 355, 130, 0, 160},
    {INF, INF, INF, 121, 160, 0}
};
```

Break und Continue

- Die Anweisungen "break" und "continue" springen an das Ende der Schleife ("}")
- "break" setzt die Ausführung *nach* dem Schleifenende fort (= bricht die Schleife ab)
- "continue" setzt die Ausführung *am* Schleifenende fort (= beginnt den nächsten Durchlauf)

Hierbei helfen uns zwei Konstrukte

Tabelle ausgeben

```
void print_dist() {
  for (int i = 0; i < CITIES; i++) {
    for (int j = i; j < CITIES; j++) {
      int distance = dist[i][j];
      if (distance == 0)
        continue;  // Eigene Stadt
      if (distance >= INF)
        continue;  // Keine Verbindung

      char buffer[20];
      sprintf(buffer, "%-3s->%3s:%7dkm",
          names[i], names[j], distance);
      Serial.println(buffer);
    }
  }
}
```

Mit break und continue kann man Sonderfälle (hier: Distanzen 0 und INF) zu Beginn eines Blocks abarbeiten und jeweils mit einem Kommentar versehen. Der Rest des Blocks macht dann die eigentliche Arbeit.

Dijkstras Algorithmus

```
// Dijkstra's algorithm stores the paths in these fields
// For each city, the direction (as city index) in which to go
int direction[CITIES];

// For each city, the length of the shortest path to TARGET
int shortest_dist[CITIES];

// Compute shortest paths from all cities to TARGET
void shortest_path_dijkstra(int target)
{
    // 1 if already visited
    int visited[CITIES];

    // Initialize fields
    for (int city = 0; city < CITIES; city++) {
        shortest_dist[city] = INF;
        direction[city] = city;
        visited[city] = 0;
    }
    shortest_dist[target] = 0;
```

Nur zur Vollständigkeit – Sie müssen den Code weder vollständig verstehen noch abändern.

```
        shortest_dist[city] = INF;
        direction[city] = city;
        visited[city] = 0;
    }
    shortest_dist[target] = 0;

    // We process all cities until all are visited
    while (target != INF) {
        // Mark target as visited
        visited[target] = 1;

        // Update distances and directions
        // of all unvisited neighbors of target
        for (int neighbor = 0; neighbor < CITIES; neighbor++)
        {
            if (!visited[neighbor] && dist[target][neighbor] < INF)
            {
                // Compute distance from neighbor via TARGET
                int dist_via_target =
                    dist[target][neighbor] + shortest_dist[target];

                // If it's shorter, use TARGET as new direction
                if (dist_via_target < shortest_dist[neighbor])
                {
                    shortest_dist[neighbor] = dist_via_target;
                    direction[neighbor] = target;
                }
            }
        }
```

Wenn Sie aber einen Fehler finden, bitte melden.

```
                // If it's shorter, use TARGET as new direction
                if (dist_via_target < shortest_dist[neighbor])
                {
                    shortest_dist[neighbor] = dist_via_target;
                    direction[neighbor] = target;
                }
            }
        }

        // Compute next target
        // If we find no new target, its value stays INF
        target = INF;

        // As next target, use the unvisited node
        // with the minimum overall distance
        int min_dist = INF;

        for (int city = 0; city < CITIES; city++)
        {
            if (!visited[city] && shortest_dist[city] < min_dist)
            {
                // Shortest distance to target found
                target = city;
                min_dist = shortest_dist[city];
            }
        }
    }
}
```

Wege ausgeben

```
int visited[CITIES] = {0, 0, 0, 0, 0, 0};
int direction[CITIES];
int shortest_dist[CITIES];

// fill direction and shortest_dist to target
void shortest_path_dijkstra(int target) {
    // code goes here
}

// Print results for all targets
void print_dijkstra(int target) {
  for (int start = 0; start < CITIES; start++) {
    char buf[100];
    sprintf(buf, "From%3s to%3s:%4d km via%3s",
            names[start], names[target],
            shortest_dist[start],
            names[direction[start]]);
    Serial.println(buf);
  }
}
```

Hier berechnen wir zunächst die kürzesten Wege, und geben sie anschließend aus

Zeiger

- Ein *Zeiger* ist eine Variable, die die *Adresse* einer Variablen speichert
- Man sagt: Der Zeiger "zeigt" auf die Variable
- Ein Zeiger mit Namen *p*, der auf einen Typ T zeigt, wird als *T *p* deklariert:

```
int *p1 = &ledPin;
```

Dereferenzieren

- Der Ausdruck **p* steht für die Variable, auf die *p* zeigt (= die Variable an Adresse *p*)
- Man sagt: Der Zeiger wird *dereferenziert*
- **p* kann wie eine Variable benutzt werden

```
int *p1 = &ledPin;
int x = *p1;  // x = ledPin
*p1 = 25;     // ledPin = 25
```

Werte austauschen

- Wir wollen eine Funktion swap(*a*, *b*) schreiben, die die Werte von *a* und *b* vertauscht
- Wir übergeben die *Adressen* von *a* und *b*

```
int x = 1; int y = 2;
swap(&x, &y);
// x = 2, y = 1
```

Tauschen mit Zeigern

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```