Fortgeschrittene Algorithmen

Begrüßung

- Vorstellung
- Themen Semester
- Einstieg + Einführung

Vorstellung Dozent – Uli Siebold

- Studium Informatik
  Karlsruhe & Freiburg
- 10 Jahre bei Fraunhofer (D)
  Dr.-Ing.: Systemmodellierung
  Forschungsgruppe zu:
  Sicherheits- und
  Zuverlässigkeitsanalysen
- 5 Jahre bei CuriX (CH)
  Head of Dev, Head of Res:
  Monitoring-Daten auswerten,
  vorhersagen und Systeme
  resilienter machen

- 44 Jahre
- 5 Kinder
- Hobbies:
  Taekwon-Do, Ausflüge
- Was mir wichtig ist:

  Zuhören,
  Offener Austausch,
  Rückfragen

Vorstellung Dozent – Uli Siebold

- Studium Informatik
  Karlsruhe & Freiburg
- 10 Jahre bei Fraunhofer (D)
  Dr.-Ing.: Systemmodellierung
  Forschungsgruppe zu:
  Sicherheits- und
  Zuverlässigkeitsanalysen
- 5 Jahre bei CuriX (CH)
  Head of Dev, Head of Res:
  Monitoring-Daten auswerten,
  vorhersagen und Systeme
  resilienter machen

- 44 Jahre
- 5 Kinder
- Hobbies:
  Taekwon-Do, Ausflüge
- Was mir wichtig ist:

  Zuhören,
  Offener Austausch,
  Rückfragen

Themenvorstellung

- Datenstrukturen
- (Ausgewählte) Algorithmen
- Entwurfsmethoden
- Analysemethoden

Vorwiegend genutzte Quelle:

Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein: Algorithmen – eine Einführung

+ vereinzelt andere Quellen, auf den Folien angegeben

Strutur / Generelles

- Vorstellung an der Tafel und/oder Folien
- Life-Coding in Java
- Gruppenarbeit
- Einzelarbeit (Programmieren oder Papier)
- Tafelrechnen
- Hausaufgaben → Bonuspunkte 10 % für Klausur möglich

(vermutlich allgemein) bekannte Datentypen

Typname
boolean
char
byte
short
int
long
float
double

https://de.wikibooks.org/wiki/Java_Standard:_Primitive_Datentypen

Datenwort – vereinfacht Wort

Grundsätzlich:

- Ein Datenwort oder einfach nur Wort ist eine bestimmte Datenmenge, die ein Computer in der arithmetisch-logischen Einheit des Prozessors in einem Schritt verarbeiten kann. Ist eine maximale Datenmenge gemeint, so wird deren Größe Wortbreite, Verarbeitungsbreite oder Busbreite genannt. (https://de.wikipedia.org/wiki/Datenwort)

- In Programmiersprachen ist das Datenwort die Bezeichnung für Datentypen

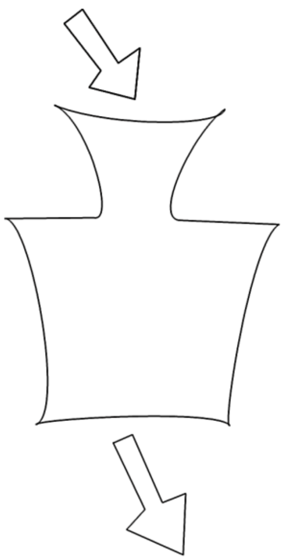
(vermutlich allgemein) bekannte Datentypen

Typname	Größe	Wertebereich
boolean	1 bit (nicht präzise festgelegt)	true / false
char	16 bit	0 ... 65.535 (z. B. 'A')
byte	8 bit	-128 ... 127
short	16 bit	-32.768 ... 32.767
int	32 bit	-2.147.483.648 ... 2.147.483.647
long	64 bit	-2^{63} bis $2^{63}-1$, 0 bis 2^{64} -1[
float	32 bit	+/-1,4E-45 ... +/-3,4E+38
double	64 bit	+/-4,9E-324 ... +/-1,7E+308

https://de.wikibooks.org/wiki/Java_Standard:_Primitive_Datentypen

Algorithmus

- Ein Algorithmus ist eine wohldefinierte Rechenvorschrift, die eine Größe (=Entität, Objekt) oder Menge von Größen verwendet und eine Größe oder Menge von Größen als Ausgabe erzeugt.



Verwendete Größe

Wohldefinierte (Rechen)vorschrift

Ausgegebene Größe

Sortierspiel

Sortierspiel

- ➔ Geht das irgendwie „optimal“?

- ➔ Gibt es ganz allgemein ein Rezept, Aufgaben optimal zu lösen

Was man nicht messen kann, kann man auch nicht verbessern.

Aufwandsmessung

Wir wollen wissen wie lange ein Algorithmus für eine Aufgabe benötigt (Zeit, Rechenschritte). Wir wollen vermutlich auch wissen, wie viel Platz wir im Rechner benötigen (Memory, Storage)

~~Aufwandsmessung~~ → Aufwandschätzungen

Wir wollen wissen wie lange ein Algorithmus für eine Aufgabe benötigt (Zeit, Rechenschritte). Wir wollen vermutlich auch wissen, wie viel Platz wir im Rechner benötigen (Memory, Storage)

Hierzu müssen wir

- Zählen und Rechnen
- mathematische Aufgaben praktisch lösen: approximativ

Wir werden uns also mit Algorithmik und Numerik beschäftigen

Datenstrukturen

Geeignete Datenstrukturen helfen, Aufgabenstellungen (effizient) zu lösen.

- Graph

Graph

- Gerichteter Graph

Ein gerichteter Graph (Digraph) ***G*** ist ein Paar ***(V, E)***, wobei ***V*** eine endliche Menge und ***E*** eine binäre Relation auf V ist …

Graph

- Gerichteter Graph

Ein gerichteter Graph (Digraph) ***G*** ist ein Paar ***(V, E)***, wobei ***V*** eine endliche Menge und ***E*** eine **binäre Relation** auf V ist …

➔ Wir sollten uns wirklich erst einmal mit Grundlagen beschäftigen!

Grundlagen

- Mengen

Mengen

- Eine Menge ist ein abstraktes Objekt, das aus der Zusammenfassung einer Anzahl einzelner Objekte hervorgeht. (https://de.wikipedia.org/wiki/Menge_(Mathematik))
- Schreibweisen:

$M = \{blau, gelb\}$ abgekürzt für $M = \{x \mid x = blau\ oder\ x = gelb\}$

$M = \{3, 6, 9, 12, \dots 96, 99\}$
$M = \{x \mid x\ ist\ eine\ durch\ 3\ teilbare\ Zahl\ zwischen\ 1\ und\ 100\}$

$M = \{1, 2, 3, 5, \dots\}$ ist die Darstellung einer unendlichen Menge

Mengen - Beziehungen

- Gleichheit: Zwei Mengen heißen gleich, wenn sie dieselben Elemente enthalten:

$$A = B :\Leftrightarrow \forall x (x \in A \Leftrightarrow x \in B)$$

- Teilmenge: Eine Menge heißt Teilmenge einer Menge B, wenn jedes Element von A auch in ein Element von B ist.

$$A \subseteq B :\Leftrightarrow \forall x (x \in A \rightarrow x \in B)$$

- Differenz:

$$A \backslash \mathrm{B} \coloneqq \{x \mid (x \in A) \land (x \notin B)\}$$

Mengen – Kartesisches Produkt

- Das kartesische Produkt oder auch Produktmenge enthält komplexe Elemente, die nicht Elemente der Ausgangsmengen sind.
- $A \times B \coloneqq \{(a, b) \mid a \in A, b \in B\}$

Binäre Relation

- Formal ist eine binäre Relation R die Untermenge eines Kartesischen Produkts einer Menge:

$$A \times A := \{(a, b) \mid a \in A, b \in A\} = A^2$$
$$R \subseteq A^2$$

- Ist $(a, b) \in R$ so sagt man, a und b stehen in Relation.

- Wichtige Eigenschaften (die jeweils nicht immer gelten **müssen**)

  - Reflexivität
  - Transitivität
  - Symmetrie
  - Antisymmetrie
  - Vollständigkeit

Datenstrukturen

Geeignete Datenstrukturen helfen, Aufgabenstellungen (effizient) zu lösen.

- Graph

Graph: gerichteter Graph

- Ein gerichteter Graph (Digraph) ***G*** ist ein Paar ***(V, E)***, wobei ***V*** eine endliche Menge und ***E*** eine binäre Relation auf V ist.

- ***V*** = Knotenmenge von ***G***, Elemente: Knoten.

- ***E*** = Kantenmenge von ***G***, Elemente: Kanten
  Kantenmenge: geordnet!

  → (u, v) und (v, u) sind **nicht** dieselben Kanten

- Schlingen (Kante von Knoten auf sich selbst) sind möglich

- Darstellung Knoten als Kreis, Kante als Pfeil

Graph: ungerichteter Graph

- Ein ungerichteter Graph ***G*** ist ein Paar ***(V, E)***, wobei ***V*** eine endliche Menge und ***E*** eine binäre Relation auf V ist.

- ***V*** = Knotenmenge von ***G***, Elemente: Knoten.

- ***E*** = Kantenmenge von ***G***, Elemente: Kanten
  Kantenmenge: ungeordnet!

  → (u, v) und (v, u) ist die selbe Kante

- Schlingen (Kante von Knoten auf sich selbst) sind nicht möglich

- Darstellung Knoten als Kreis, Kante als Linie (ohne Pfeilspitze)

Wo finden wir Graphen / Beispiele

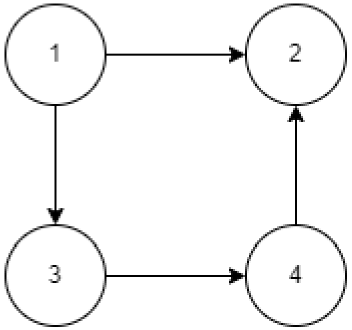
- Tafel ☺

Algorithmen

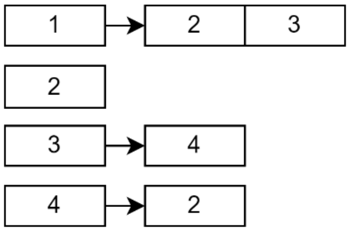
- Graphenalgorithmen

Graphenalgorithmen

- Darstellung als Grafik
  (Kreise und Linien)



- Darstellung im Rechner
  - Adjazenzlisten
    Für jeden Knoten gibt es eine Liste, die damit verbundene Knoten enthält.



  - Adjazenzmatrix (Knoten durchnummeriert)
    $|A| \times |A|$- Matrix $A = (a_{ij})$

$$a_{ij} \begin{cases} 1 \; falls \; (i,j) \epsilon \; E \\ 0 \; sonst. \end{cases}$$

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

Graphenalgorithmen

- Übung auf dem Blatt und Tafel:

  Weg der Nahrung zu Ihnen

- Auswertung / Diskussion:

  Vor- und Nachteile der Darstellungsarten

Graphenalgorithmen

- Übung auf dem Blatt und Tafel:

  Weg der Nahrung zu Ihnen (persönliche Nahrungskette)

- Auswertung / Diskussion:

  Vor- und Nachteile der Darstellungsarten

Finden wir die Ende der Nahrungskette

- Wie könnten wir im Nahrungsketten-Graph das Ende der Nahrungskette finden?

mit einem (oder mehreren) Algorithmen

Transitive Hülle

- Die transitive Hülle ist:
  die Erweiterung der Relation, die zusätzlich alle indirekt erreichbaren Paare erhält (transitiv)

- Der Algorithmus von Warshall kann die transitive Hülle erzeugen:

```
Für k=1 bis n
    Für i=1 bis n
        Falls d[i, k] = 1
            Für j=1 bis n
                Falls d[k, j] = 1
                    d[i, j] = 1
```

- Laufzeit-Komplexität: $O(n^3)$

Universelle Senke

- Stille arbeit

Universelle Senke

- Stille arbeit

	V1	V2	V3	V4	V5	V6
V1	0 →	1 ↓	0	0	0	0
V2	0	0 →	0 →	0 →	0 →	0 →
V3	0	1	0	0	0	0
V4	0	1	0	0	0	0
V5	0	1	0	0	0	0
V6	0	1	0	0	0	0

Exkurs / Voraussetzung: LIFO + FIFO

- Dynamische Mengen

- Stapel
  - **L**ast-**I**n → **F**irst-**O**ut

  ➔ LIFO

- Warteschlange
  - **F**irt-**I**n → **F**irst-**O**ut

  ➔ FIFO

Suchen: Breitensuche

- Gegeben: Graph ***G*** = ***(V, E)***,
- Ziel:
  Alle (erreichbaren) Knoten entdecken, systematisch von einem Startknoten **s** ausgehend

- Beispiele in „realer Welt“ finden → Tafel

Suchen: Breitensuche - Algorithmus

Setup / Initializing

```
Für jeden Knoten u aus G.V-{s}
     u.farbe = weiß
     u.d = infinity
     u.pre = null
s.farbe = grau
s.d = 0
s.pre = null
Queue Q = {}
Q.Enqueue(s)
```

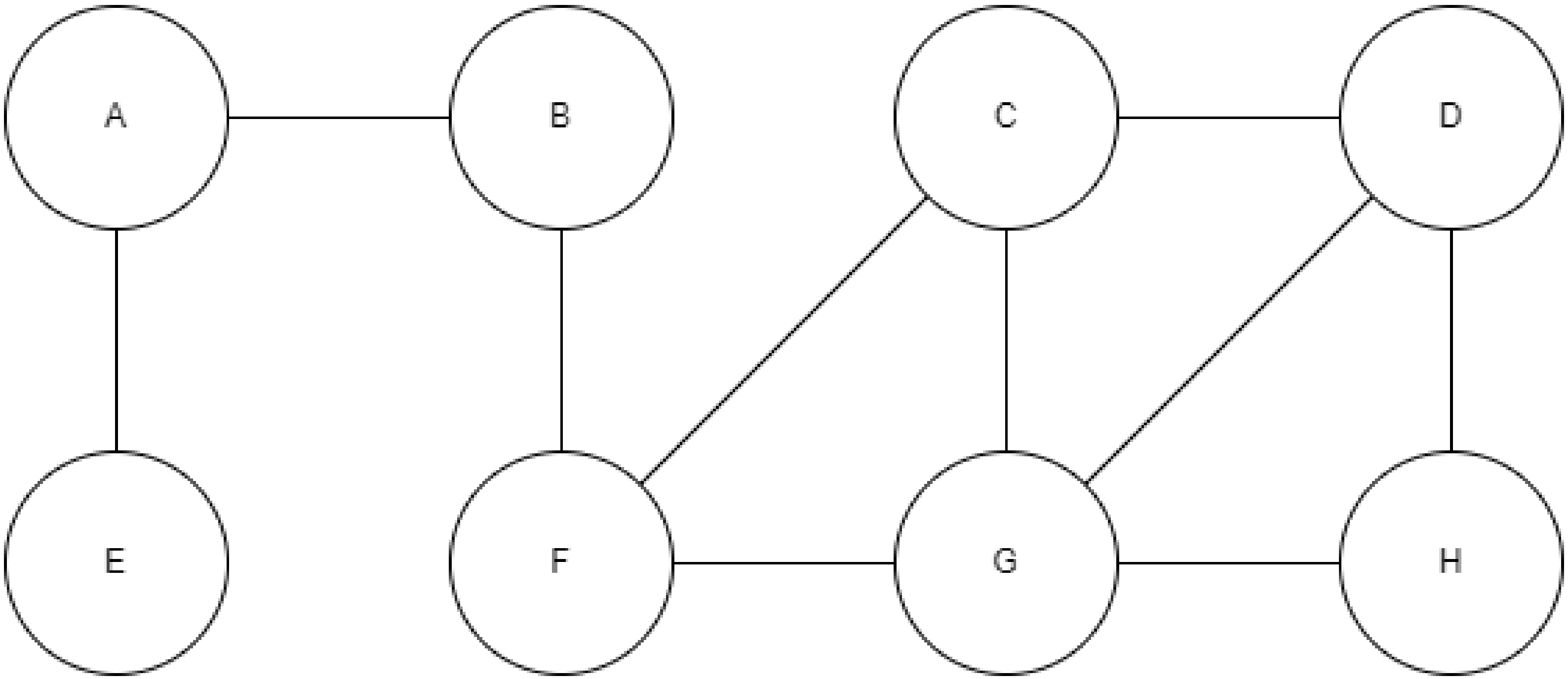
Suchen: Breitensuche - Algorithmus

Breadth First Search (BFS)

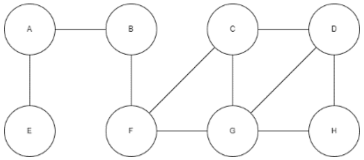
```
While Q != {}
     u = Q.dequeue()
     für jeden Knoten v aus G.Adj[u]
          if v.farbe == weiss
               v.farbe = grau
               v.d = u.d + 1
               v.pre = u
               Q.enqueue(v)
     u.farbe = schwarz
```

Suchen: Breitensuche - Algorithmus

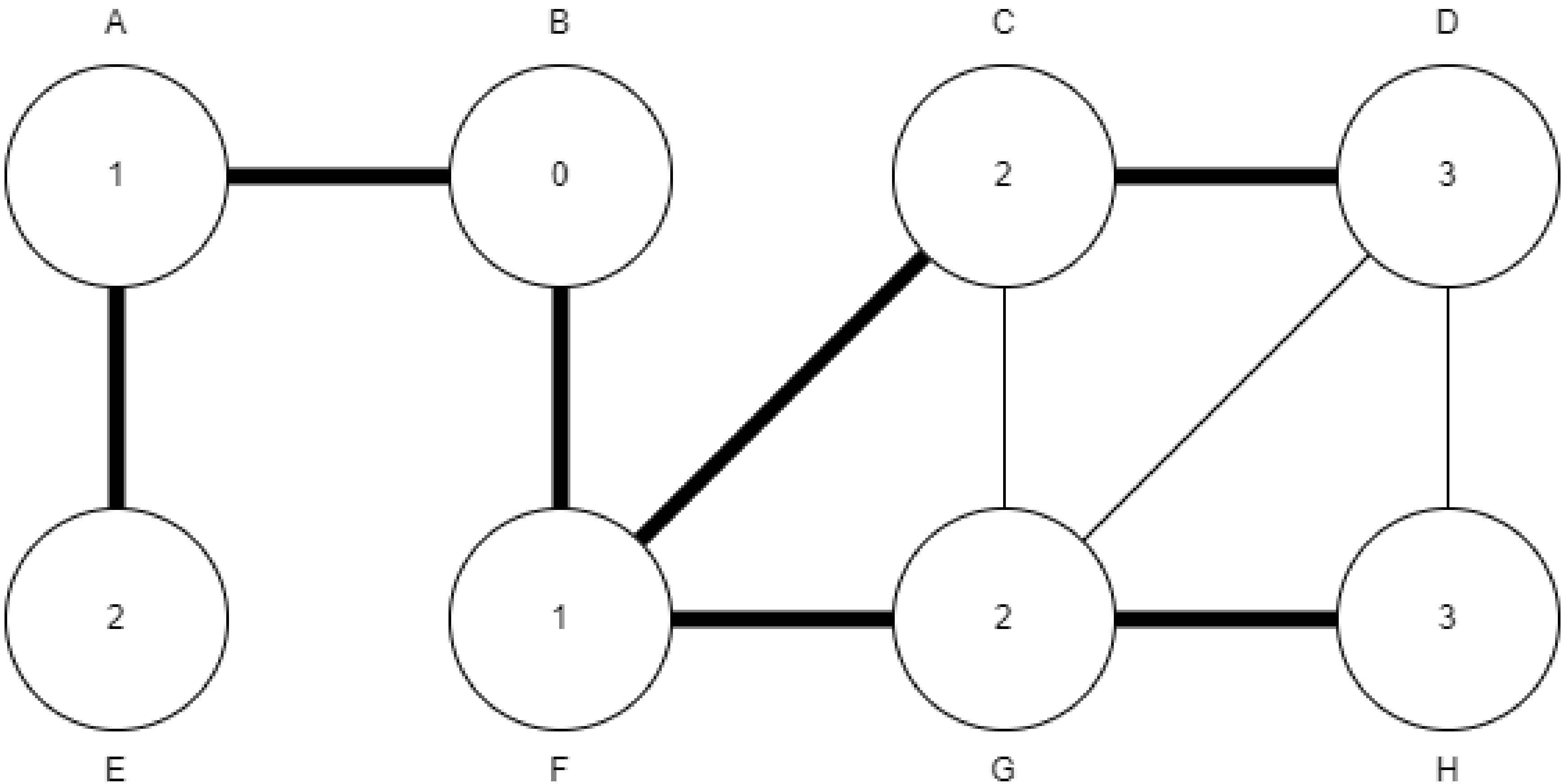
Übung: BFS auf Graph: Startknoten **B**



Suchen: Breitensuche - Algorithmus



Übung: BFS auf Graph: Startknoten **B**



Suchen: Breitensuche

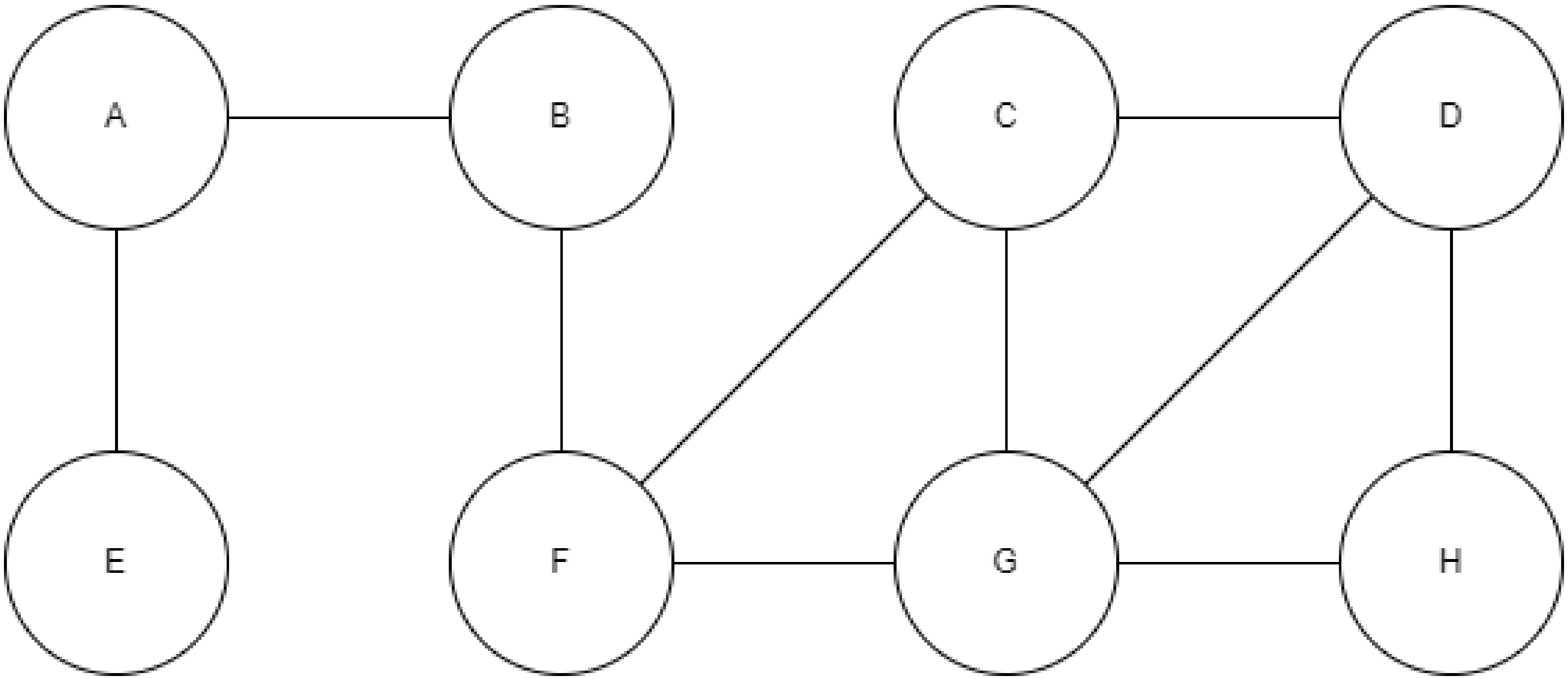
- Erkundet alle erreichbaren Knoten von Startknoten aus
- Findet dabei kürzeste Pfade zwischen S und erkundeten Knoten

- Erzeugt:

  Breitensuchbaum !

Stille Arbeit

- Tiefensuche durchführen auf Graph mit Startknoten B



Suchen: Tiefensuche - Algorithmus

Setup / Initializing

```
Für jeden Knoten u aus G.V-{s}
     u.farbe = weiß
     u.pre = null
zeit = 0

DFS-Visit(G, s)
```

Suchen: Tiefensuche - Algorithmus

Depth First Search (DFS)

```
DFS-Visit(G, u)
     zeit = zeit + 1
     u.d = zeit
     u.farbe = grau
     für jeden Knoten v aus G.Adj[u]
          if v.farbe == weiss
               v.pre = u
               DFS-Visit(G, v)
     u.farbe = schwarz
     zeit = zeit + 1
     u.f = zeit
```

Suchen: Vor/Nachteile

Diskussion Tiefensuche / Breitensuche

Suchen: Variante: Beschränkte Tiefensucht

Depth First Search (DFS)

```
DFS-Visit(G, u, maxTiefe)
      zeit = zeit + 1
      u.d = zeit
      u.farbe = grau
      für jeden Knoten v aus G.Adj[u]
            if v.farbe == weiss UND v.tiefe < maxTiefe
                  v.pre = u
                  DFS-Visit(G, v)
      u.farbe = schwarz
      zeit = zeit + 1
      u.f = zeit
```

Iterative Tiefensuche

```
Iterative Tiefensuche (Knoten, Ziel)
{
  IterationsTiefe := 0
  while (IterationsTiefe < unendlich)
  {
    Beschränkte_Tiefensuche (Knoten, Ziel, IterationsTiefe);
    IterationsTiefe := IterationsTiefe + 1;
  }
}
```

Entwurfsmuster: Greedy

Bisher:

uninformiert gesucht (blinde Suche)

Idee:

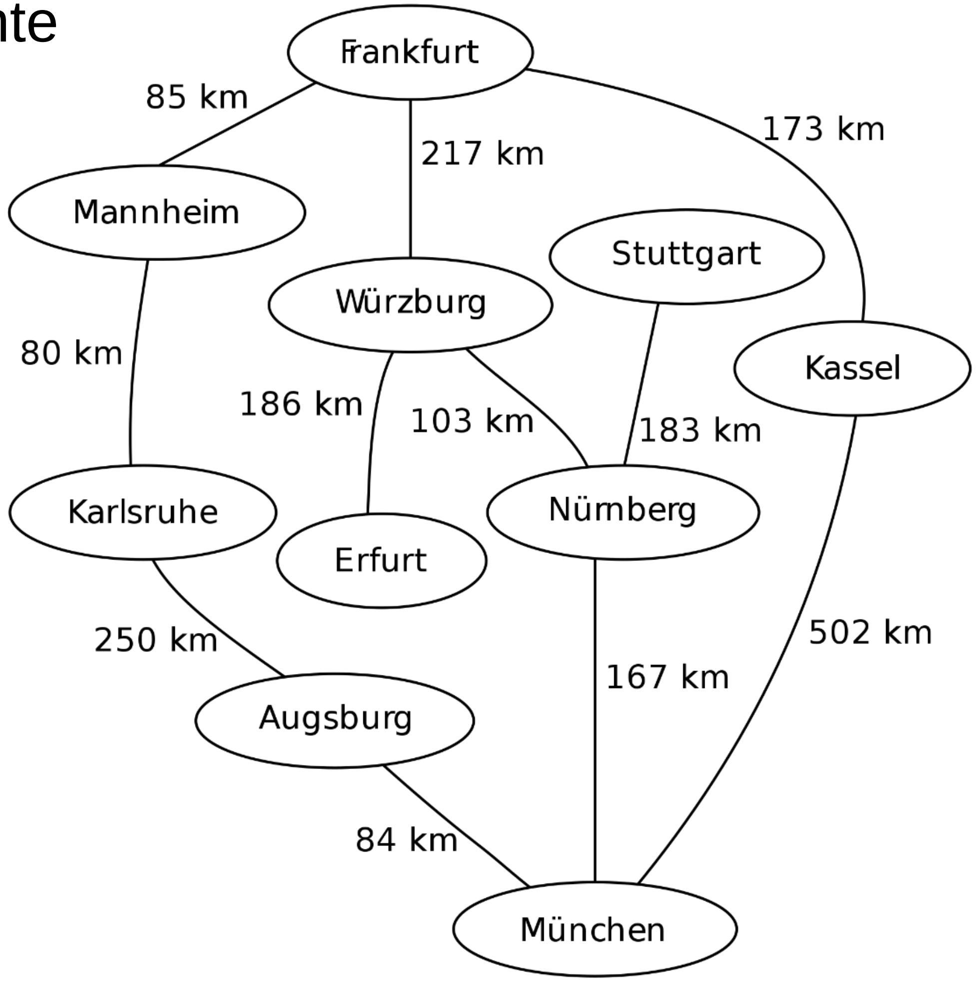
Mehr Informationen nutzen → informierte Suche, Nutzung von Heuristiken

Prinzip:

Versuche in jedem Teilschritt möglichst viel zu erreichen!

Finde kürzeste Wege -- Dijkstra

Zusatzinformation: Kantengewichte



Grobes Prinzip:

Suche in jedem Schritt die jeweils kürzeste neue Teilstrecke.

Aktualisiere Kantengewichte und optimalen Vorgänger für jeden Knoten

→ Dann erzeuge Pfad

Quelle: https://de.wikipedia.org/wiki/Dijkstra-Algorithmus

Finde kürzeste Wege – Dijkstra (Pseudocode)

```
Funktion Dijkstra(Graph, Startknoten):
    initialisiere(Graph,Startknoten,abstand[],vorgänger[],Q)
    solange Q nicht leer:                       // Der eigentliche Algorithmus
        u:= Knoten in Q mit kleinstem Wert in abstand[]
        entferne u aus Q                                // für u ist der kürzeste Weg nun bestimmt
        für jeden Nachbarn v von u:
            falls v in Q:                            // falls noch nicht berechnet
               distanz_update(u,v,abstand[],vorgänger[])   // prüfe Abstand vom Startknoten zu v
    return vorgänger[]
```

```
Methode initialisiere(Graph,Startknoten,abstand[],vorgänger[],Q):
    für jeden Knoten v in Graph:
        abstand[v]:= unendlich
        vorgänger[v]:= null
    abstand[Startknoten]:= 0
    Q:= Die Menge aller Knoten in Graph
```

Quelle: https://de.wikipedia.org/wiki/Dijkstra-Algorithmus

Finde kürzeste Wege – Dijkstra (Pseudocode)

```
Methode distanz_update(u,v,abstand[],vorgänger[]):
    alternativ:= abstand[u] + abstand_zwischen(u, v)   // Weglänge vom Startknoten nach v über u
    falls alternativ < abstand[v]:
        abstand[v]:= alternativ
        vorgänger[v]:= u
```

```
Funktion erstelleKürzestenPfad(Zielknoten,vorgänger[])
 Weg[]:= [Zielknoten]
 u:= Zielknoten
 solange vorgänger[u] nicht null:   // Der Vorgänger des Startknotens ist null
     u:= vorgänger[u]
     füge u am Anfang von Weg[] ein
 return Weg[]
```

Quelle: https://de.wikipedia.org/wiki/Dijkstra-Algorithmus

Finde kürzeste Wege – Dijkstra (Pseudocode)

```
Methode distanz_update(u,v,abstand[],vorgänger[]):
    alternativ:= abstand[u] + abstand_zwischen(u, v)   // Weglänge vom Startknoten nach v über u
    falls alternativ < abstand[v]:
        abstand[v]:= alternativ
        vorgänger[v]:= u
```

```
Funktion erstelleKürzestenPfad(Zielknoten,vorgänger[])
 Weg[]:= [Zielknoten]
 u:= Zielknoten
 solange vorgänger[u] nicht null:   // Der Vorgänger des Startknotens ist null
     u:= vorgänger[u]
     füge u am Anfang von Weg[] ein
 return Weg[]
```

Quelle: https://de.wikipedia.org/wiki/Dijkstra-Algorithmus

Hausaufgaben / Übung / Selbststudium

- Multigraph → zu ungerichtetem Graph:
  (Finden Sie heraus, was ein Multigraph ist und lösen Sie dann folgende Aufgabe)
  Geben Sie je einen Algorithmus an, der einen Multigraph in einen „äquivalenten“ ungerichteten Graph transformiert.
  - Variante A: Ausgehend von einer Adjazenzlistendarstellung
  - Variante B: Ausgehend von einer Adjazenzmatrixdarstellung

- Geben Sie je einen Algorithmus an, der einen gerichteten Graphen transponiert.
  - Variante A: Ausgehend von einer Adjazenzlistendarstellung
  - Variante B: Ausgehend von einer Adjazenzmatrixdarstellung

Selbststudium – Vorbereitung / Appetizer

- Schauen Sie sich bitte den A* Algorithmus an

- Vor- / Nachteile werden wir „besser“ beleuchten

Multigraph

- Sind in einem Graphen zwei Knoten mit mehreren Kanten verbunden, spricht man von einem Multigraphen.
- Äquivalenz zwischen zwei Graphen definieren wir hier intuitiv:
  - Gleiche Knoten
  - Gleiche Kanten
  - Wenn in Graph A (a, b) existiert, so existiert in Graph B auch (a, b) und umgekehrt

G‘ (E‘, V‘) = G“ (E“, V“) genau dann wenn E‘ = E“ und
(a, b) in V‘ genau dann wenn (a, b) in V“

Multigraph Umwandlung zu gerichtetem Graph

- Pseudocode Tafel !

➔ Jemand macht Foto und lädt es in Moodle hoch

Gerichteten Graphen Transponieren

- Transponiert man einen Graphen G = (V, E) so erhält man G‘ = (V, E‘)

  mit (v, u) in E‘ genau dann wenn (u, v) in E

Gerichteten Graphen Transponieren

- Pseudocode an Tafel

➔ Abfotografieren und in Moodle hochladen

A* Algorithmus

Ähnlich zu Dijkstra Algorithmus

Zusätzlich wird Entfernung zum Ziel geschätzt

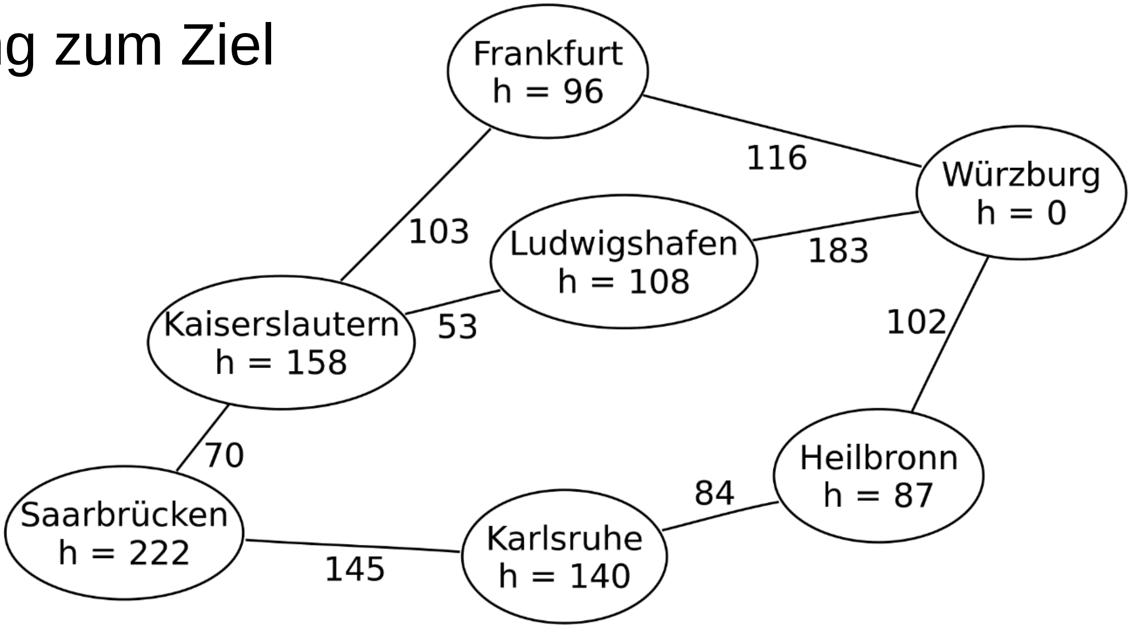
Heuristik (h)

Auswahl des nächsten Knoten über

f(x) = g(x) + h(x)

g(x) = bisherige Kosten

h(x) = heuristik



Heuristik

- Eine Heuristik ist zulässig, wenn sie die Kosten nie überschätzt
- Eine Heuristik heißt monoton, wenn sie jede Kante unterschätzt
- Eine Heuristik h2 heißt mehr informiert als h1
  wenn $h2(n) \leq h1(n) \leq h^*(n)$ für alle n

$h^*(n)$ tatsächliche Kosten von n zum Ziel

A* Algorithmus

```
declare openlist as PriorityQueue with Nodes
declare closedlist as Set with Nodes

program a-star
   openlist.enqueue(startknoten, 0)
   repeat
      currentNode := openlist.removeMin()

      if currentNode == zielknoten then
         return PathFound

      closedlist.add(currentNode)

      expandNode(currentNode)

   until openlist.isEmpty()

   return NoPathFound
end
```

A* Algorithmus

```
function expandNode(currentNode)
    foreach successor of currentNode

        if closedlist.contains(successor) then
            continue

        tentative_g = g(currentNode) + c(currentNode, successor)

        if openlist.contains(successor) and tentative_g >= g(successor) then
            continue

        successor.predecessor := currentNode
        g(successor) = tentative_g

        f := tentative_g + h(successor)
        if openlist.contains(successor) then
            openlist.updateKey(successor, f)
        else
            openlist.enqueue(successor, f)
    end
end
```

(wichtige) Eigenschaften von Algorithmen

- vollständig:
  falls eine Lösung existiert, wird sie gefunden
- optimal:
  es wird immer eine optimale Lösung gefunden
- korrekt:
  wenn eine Lösung gefunden wird, existiert diese auch (z.B. Pfad)

- Laufzeit (bzw. Anzahl Operationen)
- Speicherbedarf
- optimal effizient:
  es gibt keinen anderen Algorithmus, der (unter gleichen Bedingungen) schneller ist

Kosten- / Laufzeitschätzung

Definition Kostenfunktion:

Gegeben sei ein Algorithmus A. Die Kostenfunktion $k : \mathbb{N} \rightarrow \mathbb{R}^+$ ordnet jeder Problemgröße n den Ressourcenbedarf (z.B. Anzahl Operationen) k(n) zu, die A zur Verarbeitung der Eingabe der Größe n benötigt.

Kosten- / Laufzeitschätzung

O-Notation: Drückt eine Größe aus, die eine bestimmte „Ordnung“ nicht übersteigt.

Definition O-Notation:

Es seien f und g zwei Kostenfunktionen. Wenn es eine Konstante c ∈ $\mathbb{R}$ und ein $n_0 \in \mathbb{N}$ gibt, so dass

f(n) ≤ c ·g(n) für alle n ≥ n_0,

dann schreiben wir $f \in O(g)$
(oder $f(n) \in O(g(n))$)

Laufzeitanalyse (Velocity)

O-Notation, wichtigste Regeln

- $c = O(1)$
- $c * f(n) = O(f(n))$
- $O(f(n)) + O(f(n)) = O(f(n))$
- $O(O(f(n))) = O(f(n))$
- $O(f(n)) * O(g(n)) = O(f(n) * g(n))$
- $O(f(n)) + O(g(n)) = O(\max\{ f(n), g(n) \})$

Suche des Maximums

```
Array values = [5, 12, 4, 2, 8, 7, …]
```

← n

```
pos = 0;
max = values[pos];
while pos < values.length
     if values[pos] > max
          max = values[pos]
return max
```

$O(2) + O(4) * O(n) \rightarrow O(1) + O(1) + O(n) \rightarrow O(n)$

Eigenschaften bisheriger Algorithmen

- Transformationen:
  - Transitive Hülle
  - Multigraph -> Gerichteter Graph
  - Transponieren

- Suchen
  - Breitensuche
  - Tiefensuche
  - Tiefensuche iterativ
  - Dijkstra
  - A*

➔ Tabelle an Tafel → Eigenschaften sammeln

Aufgabe

Schiebepuzzle-Rätsel mit Suche lösen

6	5	3
1		2
4	7	8

1	2	3
4	5	6
7	8	

➔ Tafel

Branch and Bound

```
begin
    activeset :={Ø};
    bestval:=NULL (max/min);
    currentbest:=NULL;

    while activeset is not empty do
        choose a branching node, node k ∈ activeset;
        remove node k from activeset;
        generate the children of node k, child i, i=1,. . . ,nk,
                and corresponding optimistic bounds obi;
        for i=1 to nk do
                if obi worse than bestval then kill child i;
                else if child is a complete solution then
                        bestval:=obi, currentbest:=child i;
                else add child i to activeset
        end for
    end while
end
```

Branch and Bound

```
begin
    activeset :={Ø};
    bestval:=NULL (max/min);
    currentbest:=NULL;

    while activeset is not empty do
        choose a branching node, node k ∈ activeset;
        remove node k from activeset;
        generate the children of node k, child i, i=1,. . . ,nk,
                and corresponding optimistic bounds obi;
        for i=1 to nk do
                if obi worse than bestval then kill child i;
                else if child is a complete solution then
                        bestval:=obi, currentbest:=child i;
                else add child i to activeset
        end for
    end while
end
```

Entscheidungen
Optimale Lösungen
suchen

Zusammenfassung

- Datenstruktur Graph
- Graphenalgorithmen
  - Transformationen
  - Suchen
    - Uninformiert
    - Informiert (Heuristik)
- Entwurfsmuster: Greedy
- Eigenschaften von Algorithmen
- Anwendungsbeispiel (Schiebepuzzel)

Randomisierte Algorithmen

- Teile des Algorithmus nutzen Zufall(szahlen).

- Zwei Klassen:
  - Las Vegas
  - Monte Carlo

Randomisierte Algorithmen

```
findingA_LV(array A, n)
begin
    repeat
        Randomly select one element out of n elements.
    until 'a' is found
end
```

```
findingA_MC(array A, n, k)
begin
    i := 0
    repeat
        Randomly select one element out of n elements.
        i := i + 1
    until i = k or 'a' is found
end
```