Algorithmen und Datenstrukturen

Praktische Einführung und Programmierung

Stefan Bosse

Universität Koblenz - FB Informatik

Graphen und Graphenalgorithmen

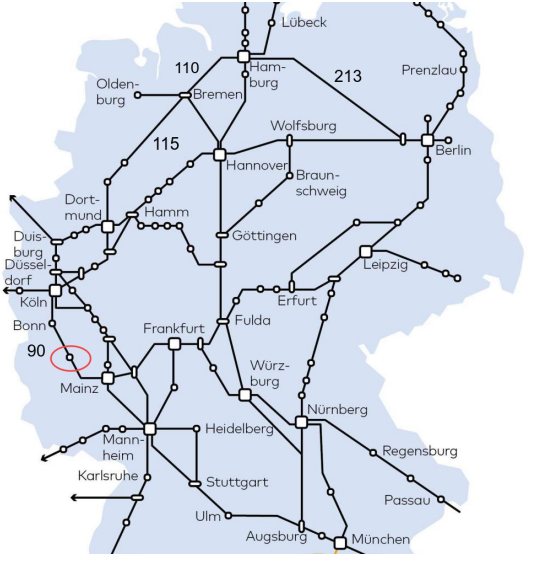


Abb. 1. Warum der ICE im Dörfchen Oberau hält - aber nicht in Koblenz

Grundlagen der Graphen

- Der ungerichtete Graph der voherigen Abbildung zeigt einen vereinfachten Ausschnitt aus dem deutschen Bahnnetz (ICE Trassen).
- An den Kanten sind die Entfernungen in Kilometern angegeben.
  - Es handelt sich also um einen gewichteten Graphen.
  - Der Graph ist zusammenhängend, aber nicht vollständig und nicht eben (also eben Deutsche Bahn)

Vertiefende Literatur

[1] H. Ernst, J. Schmidt, and G. Beneken, Grundkurs Informatik. Springer, 2023.

- Auch die bereits behandelten Bäume sind ein Spezialfall von Graphen. Tatsächlich handelt es sich bei Graphen um eine der wichtigsten Datenstrukturen in der Programmierung, die in vielen Anwendungsgebieten zum Einsatz kommen.
- Ein Graph besteht grob aus mit Kanten verbundenen Knoten.
- Unterschiede gibt es in der Art der Kanten sowie in der Realisierung der Kanten und Knoten in einer Datenstruktur, die sich auf die Effizienz der unterschiedlichen Graphenfunktionen auswirkt.

Grundlagen der Graphen

Graphen können realen Umgebungen abbilden und abstrahieren

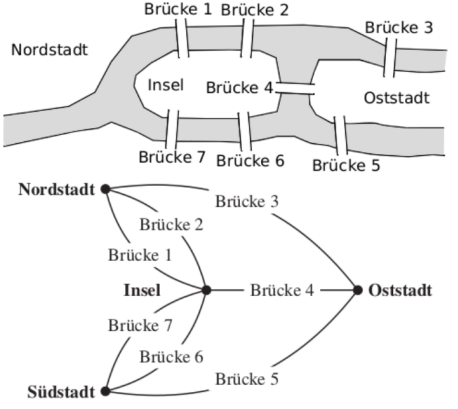


Abb. 2. Das Königsberger Brückenproblem. Oben eine Skizze der Innenstadt von Königsberg im 18. Jahrhundert. Unten eine Repräsentation als Graph, wobei die Stadtteile als Knoten und die Brücken als Kanten dargestellt sind

Taxonomie der Graphen

Arten von Graphen

Graphen werden als eine durch Kanten verbundene Menge von Knoten definiert. Wir werden insbesondere die folgenden drei Klassen von Graphen näher betrachten:

1. Bei **ungerichteten Graphen** wird angegeben, welcher Knoten mit welchem verbunden ist, ohne dass eine Richtung vorgegeben wird. Ist allgemein von Graphen ohne nähere Einschränkung die Rede, sind in der Regel ungerichtete Graphen gemeint. Typische Anwendungen von ungerichteten Graphen sind die Modellierung von Straßenverbindungen (ohne Einbahnstraßen!), der Nachbarschaft von Gegenständen oder eines Telefonnetzes.

2. **Gerichtete Graphen** unterscheiden sich von den ungerichteten dadurch, dass Kanten eine Richtung haben. Auch können zwei Knoten nun durch zwei Kanten (d.h. in jeder Richtung eine) verbunden sein.

Typische Beispiele sind Modelle von Förderanlagen, der Kontrollfluss in Programmen oder auch Petri-Netze (Abluafdiagramme mit Tokens).

Arten von Graphen

3. Ein wichtiger Spezialfall sind **gerichtete azyklische Graphen**, die oft kurz als DAG für englisch directed acyclic graph bezeichnet werden. In einem DAG gibt es keinen geschlossenen Rundweg entlang der gerichteten Kanten.

Beispiele sind die Vorfahrenbeziehung in Stammbäumen und die Vererbung in Programmiersprachen. Ein DAG ist natürlich als Datenstruktur ein normaler gerichteter Graph; relevant ist hier allerdings der möglichst effiziente Test auf Zyklenfreiheit.

- Die Bestandteile von Graphen können mit zusätzlichen Attributen versehen werden, etwa der Länge einer Kante bei der Modellierung eines Straßennetzes. Derartige Graphen werden als gewichtete Graphen bezeichnet.

Ungerichtete Graphen

Die Definition von (ungerichteten) Graphen erfolgt in Form eines Zweiertupels G = (V, E). Ein Graph besteht aus

- einer endlichen Menge V von Knoten (V für englisch vertices) und
- einer Menge E von Kanten (E für englisch edges).

Jedes e ∈ E ist dabei eine zweielementige Teilmenge der Knotenmenge V.

Def. 1. Ungerichteter Graph

- Diese Definition lässt also keine Schleifen, d.h. Kanten von einem Knoten zu sich selbst, und auch keine mehrfachen Kanten zwischen zwei Knoten (Parallelkanten) zu. Möglich ist auch eine erweiterte Definition, in der jedes e ∈ E eine ein- oder zweielementige Teilmenge darstellt.

Ungerichtete Graphen

Als Beispiel für diese Formalisierung von Graphen betrachten wir den Graph G_u:

$$G_u = (V_u, E_u)$$
$$V_u = \{1, 2, 3, 4, 5, 6\}$$
$$E_u = \{\{1,2\}, \{1,3\}, \{1,4\}, \{2,6\}, \{4,6\}, \{3,6\}, \{5,6\}, \{3,4\}, \{3,5\}\}$$

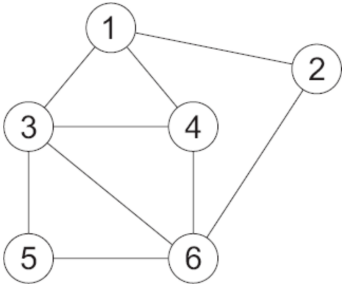


Abb. 3. Biespiel Ungerichteter Graph mit obiger Knoten- und Kantenmenge

Gerichtete Graphen

Die Definition von gerichteten Graphen erfolgt analog zu den allgemeinen Graphen durch ein Zweitupel G = (V, E) mit

- V, einer endlichen Menge von Knoten, und
- E, einer Menge von Kanten.

Def. 2. Gerichteter Graph

Allerdings ist jedes e ∈ E nun ein Tupel (a, b) mit a, b ∈ V. Diese Definition lässt Schleifen (a, a) zu.

Gerichtete Graphen

Als Beispiel für diese Definition gerichteter Graphen betrachten wir den Graph Gg aus Abbildung 16–2:

$$G_g = (V_g, E_g)$$

$$V_g = \{1, 2, 3, 4, 5, 6\}$$

$$E_g = \{(1,2), (1,3), (3,1), (4,1), (3,4), (3,6), (5,3), (5,5), (6,5), (6,2), (6,4)\}$$

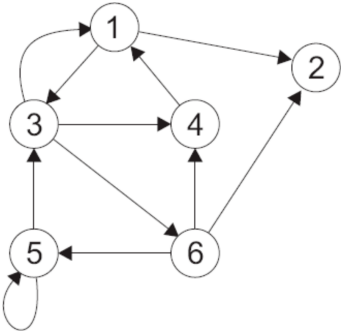


Abb. 4. Biespiel Gerichteter Graph mit obiger Knoten- und Kantenmenge

Gerichtete Graphen

Zustandsautomaten (z.B. Mikroprozessoren oder Kontrollpfade von Logiksystemen) bestehen aus einer endlichen Menge von Zuständen und Übergängen zwischen Zuständen.

- Zustandsautomaten kommen in fast allen Programmen vor. Beispiele sind auch Lexer und Parser.
  - Der Lexer zerlegt einen zusammenhängenden Text in einzelne Tokens (Symbole). Dazu benötigt er eine Sprachstruktur mit Mustern, z.B. ein Muster für die Addition `add := expression + expression`, und `expression` spaltet weiter auf (Baumstrukturen!!), z.B. `expression := Number | Variable | expression`
  - Der Parser setzt nach einem bestimmten Ablauf die Tokens zu Graphenstrukturen zusammen.

Gerichtete Graphen

```
function parse(text) {
  state = {
    instring  : 0,
    incomment : 0,
    last      : '',
    buffer    : ''
  }
  while (next=nexttoken) {
    if (state.incomment) {
      if (next!=NL) continue;
      else state.incomment=0;
    } else if (state.instring==1) {
      if (next!='"') { state.buffer+=next; continue }
    }
    switch (next) {
      case '"':
        state.instring++; if (state.instring==2) { processString(state.buffer); state.instring=0 }
        break;
      case '/': if (state.last=='/') state.incomment++; break;
    }
    state.last=next
  }
}
```

Bsp. 1. Der programmatische Ablauf eines endlichen Zustandsautomaten kann durch eine Zustandsübergangstabelle und als Graph repräsentiert (abstrahiert) werden. Hier sind die Zustände aber im Programmkode "verborgen"

Gewichtete Graphen

Gewichtete Graphen besitzen zusätzlich zu den bisherigen Graphenbestandteilen noch Kantengewichte. Kantengewichte sind den Kanten zugeordnete Gewichtswerte, etwa natürliche Zahlen oder int- oder real-Werte. Ein gewichteter Graph kann nun für natürliche Zahlen als Kantengewichte durch G = (V, E, γ) mit

$$\gamma : \boldsymbol{E} \rightarrow \mathbb{N}$$

definiert werden.

Gewichtete Graphen werfen einige neue Fragestellungen auf, so etwa die Suche nach kürzesten Wegen, dem maximalem Fluss durch einen Graphen oder die Konstruktion minimaler aufspannender Bäume.

Gewichtete Graphen

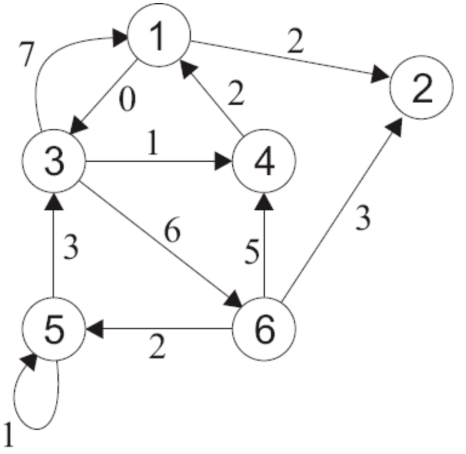


Abb. 5. Beispiel eines gewichteten, gerichteten Graphen. Natürlich gibt es auch gewichtete, ungerichtete Graphen, etwa um Entfernungen in Verkehrsnetzen zu modellieren.

Gewichtete Graphen

Beispiel: Kommunikationsnetzwerk wie das Internet. Die Kanten werden mit Geschwindigkeits- und Durchsatzmetriken annotiert und ermöglichen die Fundung einer optimal schnellen Verbidnung.

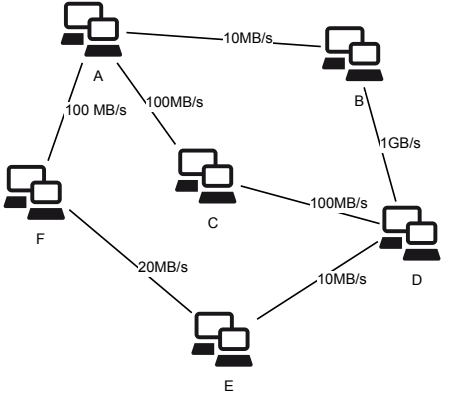


Abb. 6. Beispiel für ein Graph eines Rechnernetzwerkes mit unterschiedlichen Übertragungsgeschwindigkeiten (Bandbreite). Der Pfad A-B-D ist langsamer als der Pfad A-C-D, obwohl die gleiche Anzahl von Knoten entlang der Pfade liegen.

Eigenschaften von Graphen

1. Knotengrad, der für einen gegebenen Knoten i die Anzahl der verbundenen Knoten bezeichnet.
2. Bei gerichteten Graphen kann noch zwischen Ausgangsgrad (die Anzahl der ausgehenden Kanten) und Eingangsgrad (die Anzahl der eingehenden Kanten) unterschieden werden.
3. Pfade: Es existiert ein Pfad zwischen zwei Knoten *u* und *v* wenn ein Knoten *v* von einem Knoten *u* aus erreichbar ist, und eine Kantenfolge von u nach v existiert. Der Pfad kann uni- oder bidrektional sein.

Realisierung von Graphen

Eine Datenstruktur für Graphen kann unterschiedlich aufgebaut werden. Die Realisierungen unterscheiden sich dabei im Platzbedarf gespeicherter Graphen sowie in der Komplexität der erwähnten Basisoperationen.

Knoten- und Kantenlisten

- Eine sehr einfache (und historisch daher als erste verwendete) Speicherung von Graphen erfolgt als eine Liste von Zahlen, etwa in einem Array, einer sequenziellen Datei oder als verkettete Liste.
- Diese Liste von Zahlen dient als Kodierung des Graphen, wobei die Knoten von 1 bis n = |V| durchnummeriert werden.
  - Kanten können dann beispielsweise als Paare von Knotenzahlen dargestellt werden.

Die Realisierung durch Kantenlisten baut eine Zahlenfolge wie folgt auf (Kodierung!):

- Das erste Element definiert die Knotenanzahl, das zweite die Kantenanzahl und der Rest der Zahlenfolge definiert die Liste der Kanten, die jeweils als zwei Zahlen notiert werden.

Knoten- und Kantenlisten

Beispiel:

$$G_g = (V_g, E_g)$$
$$V_g = \{1, 2, 3, 4, 5, 6\}$$
$$E_g = \{(1,2), (1,3), (3,1), (4,1), (3,4), (3,6), (5,3), (5,5), (6,5), (6,2), (6,4)\}$$

wird in Listen (Arrays) kodiert als:

```
Kantenliste :=
[6, 11, 1, 2, 1, 3, 3, 1, 4, 1, 3, 4, 3, 6, 5, 3, 5, 5, 6, 5, 6, 2, 6, 4]
Knotenliste :=
[6, 11, 2, 2, 3, 0, 3, 1, 4, 6, 1, 1, 2, 3, 5, 3, 2, 4, 5]
```

- Die Teilfolge 2, 2, 3 beginnend an der dritten Position in dieser Zahlenfolge bedeutet beispielsweise »Knoten 1 hat Ausgangsgrad 2 und herausgehende Kanten zu den Knoten 2 und 3«.
- Man sieht bereits an diesem Beispiel, dass für Graphen mit vielen Kanten Knotenlisten weniger Speicherbedarf als Kantenlisten benötigen. Genauer benötigen Kantenlisten 2 + 2 · |E| Zahlen und Knotenlisten 2 + |V| + |E| Zahlen.

Adjazenzmatrix

Insbesondere bei sehr vielen Kanten ist eine Speicherung der Verbindungen als n × n-Matrix sinnvoll. Der Wert n entspricht dabei der Knotenanzahl |V|. Eine derartige Matrix wird als Adjazenzmatrix bezeichnet.

Ein typisches Beispiel aus der Digitallogik und Rechnerarchitektur: Der Crossbar Switch!

- Die Adjazenzmatrix ist ähnlich einer Zustandsübergangstabelle einer FSM. Sie kann für gerichtete Graphen direkt erstellt werden.
- Die Idee der Adjazenzmatrix lässt sich direkt auf ungerichtete Graphen übertragen, wobei dann eigentlich nur die eine Hälfte der Matrix gespeichert werden muss, da sich die andere Hälfte durch Spiegelung ergibt.
- Beim Vergleich des Speicherbedarfs muss man nun beachten, dass die Matrixelemente boolesche Werte sind, die im Gegensatz zu Zahlwerten nur ein Bit benötigen.
- Es lassen sich zyklische wie azyklische Graphen darstellen

Adjazenzmatrix

Die folgende Adjazenzmatrix stellt den gerichteten Graphen G_{g} und ungerichteten G_{u} dar:

$$G_G = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

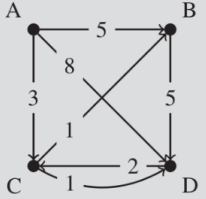
$$G_U = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Adjazenzmatrix

Auch gewichtete Graphen können auf die gleiche Art mit Matrizen beschrieben (und progarmmiert) werden.

[Grundkurs Informatik]

Gezeigt ist ein gerichteter Graph mit Gewichten. Daneben sind die zugehörige Adjazenzmatrix ***A*** und die Bewertungsmatrix ***B*** angegeben:



von: \ nach:	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	1	0	1
D	0	0	1	0

$$\boldsymbol{A} = \begin{pmatrix} 0&1&1&1 \\ 0&0&0&1 \\ 0&1&0&1 \\ 0&0&1&0 \end{pmatrix}, \quad \boldsymbol{B} = \begin{pmatrix} 0&5&3&8 \\ 0&0&0&5 \\ 0&1&0&1 \\ 0&0&2&0 \end{pmatrix}$$

Man beachte, dass die Reihenfolge der Knoten willkürlich ist und auch anders hätte gewählt werden können, was sich in den Matrizen durch Vertauschung von Zeilen und Spalten äußert. Zwei Graphen sind also genau dann isomorph, wenn sich ihre Adjazenzmatrizen durch diese beiden Operationen ineinander überführen lassen.

Abb. 7. Speicherung eines Graphen als Adjazenzmatrix *A* und Bewertungsmatrix *B* die die Gewichte vermerkt.

Für bewertete Graphen kann das Konzept beibehalten werden, wenn anstelle der Nullen und Einsen die Bewertung eingetragen wird. Man spricht dann auch von einer Bewertungsmatrix B.

Mit Matrizen können wir direkt rechnen:

Wege zwischen Knoten

Man kann nun fragen, wie man über direkte Verbindungen hinaus längere Wege zwischen zwei Knoten finden kann. Eine Methode ist die Bildung von Potenzen A^m der Adjazenzmatrix. Die Komponenten der Matrix A^m geben dann die Anzahl der Wege der Länge m vom i-ten zum k-ten Knoten an. Die Komponenten der Matrix A^2 berechnet man nach der Regel „Zeile mal Spalte“ als Skalarprodukte der Zeilen- und Spaltenvektoren der Matrix A mit n × n Elementen:

$$(a_{ik})^2 = \sum_{i=1}^{n} a_{ij} a_{jk}$$

Die obige Formel liefert also die Anzahl der Wege der Länge 2 vom Knoten i zum Knoten k.

Beispiel: $(a_{32})^2=0$, $(a_{14})^2=2$ bedeutet es gibt also keinen Weg der Länge 2 vom dritten zum zweiten Knoten, also von C nach B, aber zweiWege der Länge 2 von A nach D, nämlich A → B → D und A → C → D. Man kann dies direkt am Skalarprodukt ablesen: es ergibt sich genau dann ein Beitrag, wenn die entsprechende Kante existiert. Die Zählung der Indizes beginnt dabei mit 1.

Die Erreichbarkeitsmatrix

Will man wissen, ob unabhängig von der Länge des Weges überhaupt ein Weg von einem Knoten zu einem anderen existiert, so addiert man alle n Potenzen der Adjazenzmatrix und erhält die Erreichbarkeitsmatrix oder Wegematrix:

$$\mathbf{E} = \sum_{i=1}^{n} \mathbf{A}^i$$

> Die Einträge e_{ik} von **E** geben also an, auf wie vielen verschiedenen Wegen ein Knoten von einem anderen Knoten aus erreichbar ist, ohne dass dazwischen liegende Knoten mehrfach besucht werden.

- In der Definition von **E** sind einige Varianten gebräuchlich. Ist man beispielsweise nur an der Existenz eines Weges interessiert, so setzt man $e_{ik} = 1$, wenn der ursprüngliche Eintrag $e_{ik} > 0$ war, sonst bleibt der Eintrag 0 erhalten. Man hat dann also wieder eine effizient speicherbare binäre Matrix.

Die Erreichbarkeitsmatrix

[Grundkurs Informatik]

Betrachtet man nochmals Bsp. 13.22, so erhält man für die Folge der Potenzen von $\boldsymbol{A}$ und die Erreichbarkeitsmatrix $\boldsymbol{E}$ bzw. die binäre Erreichbarkeitsmatrix $\boldsymbol{E}_{\text{bin}}$:

$$\boldsymbol{A}^1 = \boldsymbol{A} = \begin{pmatrix} 0&1&1&1 \\ 0&0&0&1 \\ 0&1&0&1 \\ 0&0&1&0 \end{pmatrix}, \quad \boldsymbol{A}^2 = \begin{pmatrix} 0&1&1&2 \\ 0&0&1&0 \\ 0&0&1&1 \\ 0&1&0&1 \end{pmatrix}, \quad \boldsymbol{A}^3 = \begin{pmatrix} 0&1&2&2 \\ 0&1&0&1 \\ 0&1&1&1 \\ 0&0&1&1 \end{pmatrix}, \quad \boldsymbol{A}^4 = \begin{pmatrix} 0&2&2&3 \\ 0&0&1&1 \\ 0&1&1&2 \\ 0&1&1&1 \end{pmatrix}$$

$$\boldsymbol{E} = \sum_{i=1}^{n} \boldsymbol{A}^i = \begin{pmatrix} 0&5&6&8 \\ 0&1&2&3 \\ 0&3&3&5 \\ 0&2&3&3 \end{pmatrix}, \quad \boldsymbol{E}_{\text{bin}} = \begin{pmatrix} 0&1&1&1 \\ 0&1&1&1 \\ 0&1&1&1 \\ 0&1&1&1 \end{pmatrix}$$

Knoten A kann offensichtlich von keinem anderen Knoten aus erreicht werden, da die gesamte erste Spalte von $\boldsymbol{E}$ Null ist.

Abb. 8. Bestimmung der Erreichbarkeitsmatrix an einem Beispiel. Die Potenzen der Matrix A ergebn sich aus dem Matrixprodukt!

Enthält eine ganze Spalte der Erreichbarkeitsmatrix nur die Einträge 0, so kann dieser Knoten von keinem anderen Knoten aus erreicht werden. Sind alle Einträge einer Zeile 0, so kann von dem entsprechenden Knoten aus kein andere Knoten erreicht werden.

Kürzeste und längste Wege

Weiter mit den Adjazenzmatrizen ...

- Bei einem gewichteten Graphen kann man anstelle der Anzahl der Wege auch eine Summe von Gewichten längs der Kanten des Weges eintragen und diese als Weglängen interpretieren.
- Dies kann etwa das Minimum der Summe der Gewichte sein, wenn man den kürzesten Weg zwischen je zwei Kanten bestimmen möchte, oder auch das Maximum der Summe der Gewichte, wenn der längste Weg gesucht ist.
- Der Eintrag 0 bedeutet dabei nach wie vor, dass kein Weg vorhanden ist.

Ausgehend von Bsp. 13.22 und 13.23 ergeben sich die Matrizen $\boldsymbol{E}_{\min}$ und $\boldsymbol{E}_{\max}$ der kürzesten bzw. längsten Wege zu:

$$\boldsymbol{E}_{\min} = \begin{pmatrix} 0 & 4 & 3 & 4 \\ 0 & 8 & 7 & 5 \\ 0 & 1 & 3 & 1 \\ 0 & 3 & 2 & 3 \end{pmatrix}, \quad \boldsymbol{E}_{\max} = \begin{pmatrix} 0 & 5 & 12 & 10 \\ 0 & 8 & 7 & 5 \\ 0 & 1 & 8 & 6 \\ 0 & 3 & 2 & 8 \end{pmatrix}$$

Abb. 9. Berechnung der kürzesten und längsten Wege

Der Floyd-Warshall-Algorithmus

```
Nummeriere die Knoten des Graphen von 1 bis n durch und
trage die Längen der Wege von Knoten i zu Knoten j in die
Erreichbarkeitsmatrix e(i,j) ein. Existiert kein solcher
direkter Weg, trage ein Symol für „unendlich“ ein (Infinity).

Wiederhole für k := 1 bis n
  Wiederhole für i := 1 bis n
   Wiederhole für j := 1 bis n
    Wenn e(i,k) + e(k,j) < e(i,j) dann
      Setze e(i,j) := e(i,k) + e(k,j)
      Ersetze die zu e(i,j) gehörende Knotenliste durch die Knotenliste, die durch
      Verbinden der zu e(i,k) und zu e(k,j) gehörenden Knotenlisten entsteht.
ENDE
```

Alg. 1. Floyd-Warshall-Algorithmus zum Auffinden kürzester Wege in einem Graphen

Längste Pfade

Das Problem des längsten Pfades ist NP-schwer oder NP-vollständig, abhängig von zusätzlichen Randbedingungen. Eine einfache Modifikation des Floyd-Warshall- oder Dijkstra-Algorithmus wird nicht funktionieren.

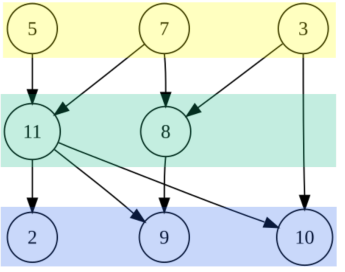
- Wenn jeder Knoten immer wieder besucht werden kann gibt es unendlich viele unendlich lange Pfade

https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/

Topologisches Sortieren

- Linearisierung eines Graphens (nur DAG), d.h. Einordnung der Knoten in eine Liste mit einer Ordnung unter der Bedingung:

> dass für jede gerichtete Kante (u,v) vom Knoten u zum Knoten v dann u in der Reihenfolge vor v steht.



Topologisches Sortieren = Scheduling Problem

- 5, 7, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 3, 5, 7, 8, 11, 2, 10, 9 (lexicographic by incoming neighbors)
- 5, 7, 3, 8, 11, 2, 10, 9 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

Graphen als dynamische Datenstrukturen

- Die bisherigen Graphendarstellungen waren statische Realisierungen, da eigentlich keine dynamischen Erweiterungen im Sinne verketteter Listen vorgesehen waren (obwohl etwa Knotenlisten natürlich auch als verkettete Liste realisiert werden können).
- Es gibt nun eine Reihe von Möglichkeiten, Graphen als dynamische Datenstrukturen zu realisieren.
  - Eine nahe liegende Variante ist es, wie bei einem Baum die Knoten als Elemente einer dynamischen Struktur aufzufassen und die Kanten als Verzeigerung zu realisieren. Dies bildet direkt die Graphenstruktur ab und resultiert in einer komplex verzeigerten Struktur mit vielen Zyklen (und damit Fehlermöglichkeiten bei der Programmierung).

Wir werden uns eine andere Realisierung anschauen, die auch als Adjazenzliste bezeichnet wird. Ein Graph wird dabei durch |V| +1 verkettete Listen realisiert. Die Basisstruktur bildet die Liste aller Knoten. Für jeden Knoten wird eine Liste der Nachfolger entlang gerichteter Kanten abgespeichert, so dass man für n = |V| und m = |E| insgesamt image Listenelemente erhält.

Graphen als dynamische Datenstrukturen

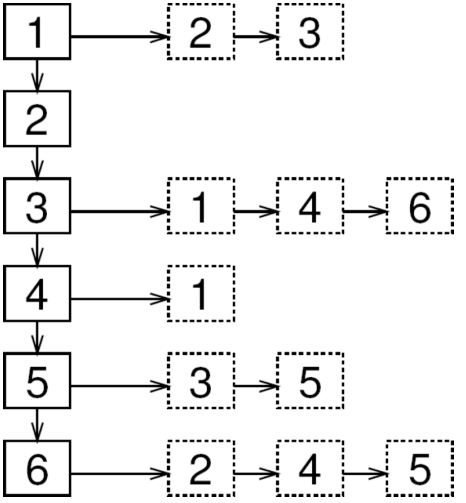


Abb. 10. Graph als dynamische Datenstruktur

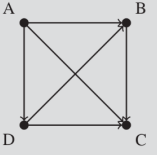
Verkettete Speicherung eines Graphen

Forward-Star Datenstruktur

- Eine sehr platzeffiziente Möglichkeit zum Speichern von Graphen ist der Forward Star. Dabei wird zunächst jedem der n Knoten ein Index zugewiesen.
- Nun werden in einem als Knotenliste bezeichneten Array K Verweise auf Positionen in einem zweiten Array, die Nachfolgerliste N, eingetragen.
  - Diese enthält, beginnend mit dem ersten Nachfolger von Knoten 1 der Reihe nach alle Nachfolger von Knoten 1, sofern dieser überhaupt einen hat.
  - Danach kommen alle Nachfolger von Koten 2 usw. Die Nachfolgerliste enthält somit genau einen Eintrag für jede der m Kanten.
- Die in der Knotenliste eingetragenen Verweise K_i geben die Position an, an der in der Nachfolgerliste die zugehörigen Nachfolger beginnen.

Forward-Star Datenstruktur

Die $n = 4$ Knoten {A, B, C, D} des Graphen sind mit den Indizes 1 bis 4 durchnummeriert. Die Knotenliste K enthält für jeden Knoten die zugehörige Position in der Nachfolgerliste N. Knoten 3 hat Ausgangsgrad 0. Der letzte Eintrag der Knotenliste enthält den Eintrag $m + 1 = 7$ mit der Anzahl $m = 6$ der Kanten.

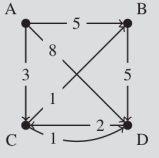


Knotenindex i:	1	2	3	4		
Knotenliste K:	1	4	5	5	7	
Nachfolgerliste N:	2	3	4	3	2	3

Abb. 11. Repräsentation eines Graphen als Forward Star

Forward-Star Datenstruktur

Gezeigt ist ein Beispiel für die verkettete Speicherung eines gerichteten Graphen mit Gewichten in Form des Bildes des Graphen, einer tabellarischen Darstellung sowie einer ausführlichen graphischen Darstellung der Verkettung:



Knotenliste	Kantenliste mit Gewichten
A	5, B; 3, C; 8, D
B	5, D
C	1, B; 1, D
D	2, C

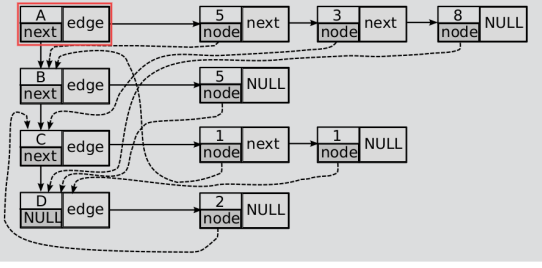


Abb. 12. Der Graph wird direkt als verzeigerte Datenstruktur abgebildet. Jeder Knoten hat eine Nachfolgerliste. Algortihmen auf diese Datenstruktur anzuwenden ist wegen Zyklen gefährlich und aufwendig Zyklen zu erkennen.

Vergleich und Komplexität

Alle vorgestellten Realisierungsvarianten sind äquivalent; jede Darstellung kann also in jede andere ohne Informationsverlust transformiert werden. Im Wesentlichen erfordert dies das Auslesen der einen Darstellung und anschließend das Erzeugen der jeweils anderen Darstellung.

Operation	Kanten-liste	Knoten-liste	Adjazenz-matrix	Adjazenz-liste
Einfügen Kante	$O(1)$	$O(n+m)$	$O(1)$	$O(1)$ / $O(n)$
Löschen Kante	$O(m)$	$O(n+m)$	$O(1)$	$O(n)$
Einfügen Knoten	$O(1)$	$O(1)$	$O(n^2)$	$O(1)$
Löschen Knoten	$O(m)$	$O(n+m)$	$O(n^2)$	$O(n+m)$

[addp]

Abb. 13. Vergleich der Komplexität. Hierbei gilt n = |V| und m = |E|.

Komplexität der Berechnung der Erreichbarkeitsmatrix

- Die Komplexität für die Berechnung der Erreichbarkeitsmatrix eines Graphen mit n Knoten durch Summierung der ersten n Potenzen der Adjazenzmatrix ist von der Ordnung $O(n^4)$, also polynomial mit einem recht hohen Exponenten. Dies ergibt sich, da $n - 1$ Matrixmultiplikationen auszuführen sind, wobei die Komplexität der Multiplikation zweier $n \times n$ Matrizen von der Ordnung $O(n^3)$ ist.

- Der Floyd-Warshall-Algorithmus hat die Komplexität $O(n^3)$, was sich aus den drei ineinander geschachtelten Schleigen mit n Durchläufen sofort ertgit.

Das Löschen eines Knotens löscht auch zugehörige Kanten, so dass hier der Aufwand in allen Varianten unvermutet hoch ist. Zu den einzelnen Varianten sind einige Bemerkungen angebracht:

- Bei Kantenlisten ist sowohl das Einfügen von Kanten (Anhängen zweier Zahlen) und von Knoten (Erhöhung der ersten Zahl um 1) besonders günstig, falls das Anhängen von Zahlen nicht zum Umkopieren eines Feldes führt. Das Löschen von Kanten kann das Zusammenschieben in einer Liste bedeuten; das Löschen von Knoten eine erneute Durchnummerierung der Knoten.
- Bei Knotenlisten ist insbesondere das Einfügen von Knoten (Erhöhung der ersten Zahl und Anhängen einer 0) günstig.
- In der Matrixdarstellung ist insbesondere das Manipulieren von Kanten eine sehr effizient ausführbare Operation. Der Aufwand bei Knoteneinfügung hängt von der Realisierung der Matrix ab, hier wurde von einem Kopieren der Matrix in eine größere Matrix ausgegangen.
- **Bei der Realisierung als Adjazenzliste ergibt sich unterschiedlicher Aufwand, je nachdem ob die Knotenliste als Feld (mit Direktzugriff) oder als verkettete Liste (mit sequenziellem Durchlauf) realisiert wird.**

Durchsuchen von Graphen

- Das systematische Durchsuchen eines Graphen ist durchaus keine triviale Aufgabe.
- Da es sich um einen wichtigen Teilaspekt vieler Anwendungen handelt, werden gibt es einige Algorithmen zum Durchsuchen von Graphen.
- Verwendet werden solche Verfahren beispielsweise um einen Weg von einem gegebenen Startknoten zu einem Zielknoten zu finden, der ggf. bestimmten Kriterien genügen soll (wie: gefunden werden soll der kürzest mögliche Weg).

Tiefensuche

Bei der Tiefensuche (Depth First Search) in einem Graphen besucht man von einem Startknoten k ausgehend dessen nächsten noch nicht besuchten Nachfolger, d.h. den als nächsten in der zu k gehörenden Nachfolgerliste befindlichen Knoten und setzt dort die Suche rekursiv fort. Gerät man dabei in eine Sackgasse, so muss der Weg bis zum ersten Knoten zurückverfolgt werden, von dem aus eine alternative Wahl möglich ist (Backtracking).

Breitensuche

Bei der Breitensuche (Breadth First Search) besucht man von einem Knoten ausgehend zuerst alle direkten Nachfolger, d. h. die in der zugehörigen Kantenliste enthaltenen Knoten, bevor deren noch nicht besuchte Nachfolger besucht werden. Auch bei der Breitensuche müssen Knoten als schon bearbeitet markiert werden können, wozu wie schon bei der Tiefensuche der Eintrag „Status“ verwendet wird. Anstelle eines Stapels wird bei der Breitensuche ein Puffer (Warteschlange, FIFO) für die temporäre Speicherung benötigt.

A∗-Algorithmus

- Problematisch bei der uniformen Kosten Suche ist, dass zur Bestimmung kürzester Wege der zu expandierende Knoten nur auf Basis der schon entstandenen Kosten ab dem Startknoten ausgewählt wird, in der Annahme, dass es besser ist zunächst die Wege mit aktuell geringen Kosten weiter zu betrachten.
- Solche Wege können aber kurz sein und in die falsche Richtung gehen, d. h. weg vom Ziel.
- Das Verfahren bemerkt dies erst, wenn andere Wege, die näher am Ziel sind, wieder besser bewertet werden. Man bezeichnet solche Suchverfahren als uninformierte Suche, weil sie keine Informationen darüber haben, wie nahe sie während der Suche dem Ziel schon gekommen sind (ohne den Weg zum Ziel tatsächlich zu berechnen).

A∗-Algorithmus

Der A∗ -Algorithmus kombiniert die besten Aspekte zweier anderer Algorithmen:

- Dijkstra-Algorithmus: Dieser Algorithmus findet den kürzesten Weg zu allen Knoten von einem einzigen Quellknoten.
- Greedy Best-First-Suche: Dieser Algorithmus untersucht den Knoten, der dem Ziel am nächsten zu sein scheint, basierend auf einer heuristischen Funktion.

A∗-Algorithmus

- A∗ findet immer eine Lösung, sofern diese existiert. Das Verfahren ist optimal-effizient.
- Es lässt sich zeigen, dass kein anderer Algorithmus existiert, der bei der Suche weniger Knoten expandiert, wenn die Bedingungen für die heuristische Funktion eingehalten werden.
- Deswegen ist A∗ das Standardverfahren für alle Anwendungen, bei denen kürzeste Wege gesucht werden müssen, so z.B. bei der Routenplanung in Navigationsgeräten oder der Wegplanung in Computerspielen.
- Die Zeitkomplexität von A∗ ist exponentiell in der **Länge des kürzesten Pfads zum Ziel**.
- Das Hauptproblem in der Praxis ist jedoch der (ebenfalls exponentielle) Speicherverbrauch, da alle Knoten im Speicher gehalten werden müssen.

Schlüsselkonzepte in der A∗-Suche

[https://www.datacamp.com/tutorial/a-star-algorithm]

Die Effizienz des A∗ -Algorithmus beruht auf seiner intelligenten Bewertung von Pfaden unter Verwendung von drei Schlüsselkomponenten: g(n), h(n) und f(n). Diese Komponenten arbeiten zusammen, um den Suchprozess auf die vielversprechendsten Pfade zu lenken.

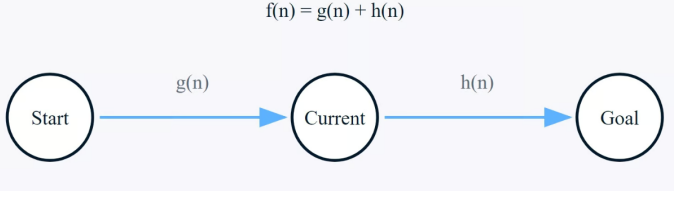


Abb. 14. A∗ Kostenfunktionen: Pfadkosten g, heuristische Funktion h

```
function A_Star(start, goal) {
    // Initialize open and closed lists
    openList = [start]          // Nodes to be evaluated
    closedList = []             // Nodes already evaluated
    // Initialize node properties
    start.g = 0                 // Cost from start to start is 0
    start.h = heuristic(start, goal)  // Estimate to goal
    start.f = start.g + start.h       // Total estimated cost
    start.parent = null               // For path reconstruction
    while openList is not empty:
        // Get node with lowest f value - implement using a priority queue
        // for faster retrieval of the best node
        current = node in openList with lowest f value
        // Check if we've reached the goal
        if current = goal:
            return reconstruct_path(current)
        // Move current node from open to closed list
        remove current from openList
        add current to closedList
        // Check all neighboring nodes
        for each neighbor of current:
            if neighbor in closedList:
                continue  // Skip already evaluated nodes
            // Calculate tentative g score
            tentative_g = current.g + distance(current, neighbor)
            if neighbor not in openList:
                add neighbor to openList
            else if tentative_g >= neighbor.g:
                continue  // This path is not better
            // This path is the best so far
            neighbor.parent = current
            neighbor.g = tentative_g
            neighbor.h = heuristic(neighbor, goal)
            neighbor.f = neighbor.g + neighbor.h
    return failure  // No path exists
}
```

Alg. 2. A∗-Algorithmus

A∗-Algorithmus

Was ist wichtig?

- Es werden Listen geführt um noch nicht und breits besuchte Knoten zu merken

Initialisierungsphase

Der Algorithmus beginnt mit der Erstellung von zwei wesentlichen Listen:

1. Die offene Liste beginnt nur mit dem Startknoten
2. Die geschlossene Liste beginnt leer

Jeder Knoten speichert vier kritische Informationen:

1. g: Die tatsächlichen Kosten vom Startknoten
2. h: Die geschätzten Kosten für das Ziel
3. f: Die Summe von g und h
4. parent: Verweis auf den vorherigen Knoten (für Pfadrekonstruktion)

Hauptschleife

Der Kern von A∗ ist seine Hauptschleife, die fortgesetzt wird, bis entweder:

1. Das Ziel ist erreicht (Erfolg)
2. Die offene Liste wird leer (Fehler - es existiert kein Pfad)

Während jeder Iteration wird der Algorithmus:

1. Wählt den vielversprechendsten Knoten (niedrigster f-Wert) aus der offenen Liste aus
2. Verschiebt es in die geschlossene Liste
3. Untersucht alle benachbarten Knoten

Nachbarbewertung

Für jeden Nachbarn der Algorithmus:

1. Überspringt Knoten, die sich bereits in der geschlossenen Liste befinden
2. Berechnet einen vorläufigen G-Wert
3. Aktualisiert Knotenwerte, wenn ein besserer Pfad gefunden wird
4. Fügt den offenen Liste neue Knoten hinzu

Wegrekonstruktion

Sobald das Ziel erreicht ist, arbeitet der Algorithmus rückwärts durch die übergeordneten Referenzen, um den optimalen Pfad vom Start zum Ziel zu konstruieren.

Diese systematische Vorgehensweise stellt sicher, dass A* immer den optimalen Weg findet, wenn:

1. Die heuristische Funktion ist zulässig (niemals überschätzt)
2. Zwischen dem Start- und dem Zielknoten existiert tatsächlich ein Pfad