Diskrete Mathematik: Graphentheorie

Prof. Dr. Nicola Winter

Version 1.0 (2022/23)

Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

Inhaltsverzeichnis

Literaturempfehlungen

- [KN] **Krumke**,Sven Oliver, **Noltemeier**, Hartmut: Graphentheoretische Konzepte und Algorithmen. Springer Vieweg.
- [TT] **Teschl**, Gerald, **Teschl**, Susanne: Mathematik für Informatiker 1. Springer Verlag, Berlin Heidelberg.

Die mit (*) markierten Definitionen und Algorithmen sind dem ersten Band von [TT] direkt entnommen (ggf. unter Korrektur einiger Tippfehler sowie Textkürzungen und geringfügigen Abweichungen in der Notation).

Grundbegriffe der Graphentheorie

Grundbegriffe der Graphentheorie (1)

Definition (*)

Ein **Graph** $G = (V, E)$ besteht aus einer endlichen Menge V von **Knoten** (engl. vertex) und einer Menge E von **Kanten** (engl. edge) $\{a, b\}$ mit $a, b \in V, a \neq b$.

Eine Kante $\{a, b\}$ verbindet also immer zwei Knoten a, b. Diese beiden Knoten heißen die **Endknoten** der Kante.

Graphen, für die eine graphische Darstellung ohne kreuzende Kanten möglich ist, nennt man **planar**.

Ein Graph, bei dem es zwischen je zwei Knoten eine Kante gibt, heißt **vollständiger Graph**. Der vollständige Graph mit n Knoten wird mit K_n bezeichnet.

Ein Graph $H = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E$ heißt **Teilgraph** von G.

Grundbegriffe der Graphentheorie (2)

Definition (*)

- Wenn 2 Knoten durch eine Kante verbunden sind, so heißen sie **adjazent** oder benachbart.
- Wenn 2 Kanten genau einen gemeinsamen Endknoten haben, so heißen sie **inzident**.
- Auch eine Kante und einen Knoten nennt man **inzident**, wenn der Knoten ein Endknoten der Kante ist.
- Der **Grad** $d(a) = deg(a)$ eines Knotens a ist die Anzahl der Kanten, die inzident mit dem Knoten sind, d.h. die Anzahl der Kanten, die vom Knoten „ausgehen“
  (engl.: Grad = degree).
- Ein Knoten heißt **isoliert**, wenn sein Grad gleich 0 ist (d.h., wenn von ihm keine Kante ausgeht).

Grundbegriffe der Graphentheorie (3)

Definition (*)

Seien $G = (V, E)$ und $H = (V', E')$ zwei Graphen.
Wenn es ein bijektive Abbildung $f : V \to V'$ gibt, so dass $\{a, b\} \in E$ ist, genau dann, wenn $\{f(a), f(b)\} \in E'$ ist, dann nennen wir die Graphen **äquivalent** (oder **isomorph**).

Definition (*)

Ein **gerichteter Graph** oder **Digraph** (engl: directed graph) ist ein Graph $D = (V, E)$, in dem jede Kante eine Richtung besitzt, also durch ein geordnetes Paar dargestellt wird.

Die Kante (auch: **Bogen**), die vom Knoten a zum Knoten b geht ($a, b \in V, a \neq b$), wird nun durch das geordnete Paar (a, b) bezeichnet. Dabei wird a der **Anfangsknoten** und b der **Endknoten** genannt.

Darstellung von Graphen im Computer (1/2)

Definition (*)

Die Knoten eines Graphen $G = (V, E)$ seien durchnummeriert, d.h. $V = \{1, ..., n\}$.
Dann ist die **Adjazenzmatrix** $A = (a_{jk})_{1 \leq j,k \leq n}$ des Graphen gegeben durch

$$a_{jk} = \begin{cases} 1 & \text{falls } \{j, k\} \in E, \\ 0 & \text{sonst} \end{cases}$$

Definition (*)

Die Knoten eines gerichteten Graphen $D = (V, E)$ seien durchnummeriert, d.h. $V = \{1, ..., n\}$.
Dann ist die **Adjazenzmatrix** $A = (a_{jk})_{1 \leq j,k \leq n}$ des gerichteten Graphen gegeben durch

$$a_{jk} = \begin{cases} 1 & \text{falls } (j, k) \in E, \\ 0 & \text{sonst} \end{cases}$$

Sind die Adjazenzmatrizen zweier (Di-)Graphen gleich, dann sind die (Di-)Graphen äquivalent. Äquivalente (Di-)Graphen können aber verschiedene Adjazenzmatrizen haben.

Darstellung von Graphen im Computer (2/2)

Definition

Die Knoten eines Graphen $G = (V, E)$ seien durchnummeriert, d.h. $V = \{1, ..., n\}$, und es sei $E = \{e_1, ..., e_m\}$.
Dann ist die **(Knoten-Kanten-)Inzidenzmatrix** $B = (b_{jk})_{1 \leq j \leq n; 1 \leq k \leq m}$ des Graphen gegeben durch

$$b_{jk} = \begin{cases} 1 & \text{falls Knoten } j \text{ und Kante } e_k \text{ inzident,} \\ & \quad \text{d.h. } j \text{ ist ein Endknoten von } e_k, \\ 0 & \text{sonst} \end{cases}$$

Definition

Die Knoten eines Digraphen $D = (V, A)$ seien durchnummeriert, d.h. $V = \{1, \ldots, n\}$, und es sei $A = \{a_1, \ldots, a_m\}$.
Dann ist die **(Knoten-Kanten-)Inzidenzmatrix** $B = (b_{jk})_{1 \leq j \leq n; 1 \leq k \leq m}$ des Digraphen gegeben durch

$$b_{jk} = \begin{cases} 1 & \text{falls } j \text{ Anfangsknoten der Kante } e_k \text{ ist,} \\ -1 & \text{falls } j \text{ Endknoten der Kante } e_k \text{ ist,} \\ 0 & \text{sonst} \end{cases}$$

Kantenfolgen, Kantenzüge, Wege, Kreise

Definition

Sei $G = (V, E)$ ein Graph.

- Eine Folge $v_0, e_1, v_1, e_2, \ldots, e_n, v_n$ mit $v_0, v_1, \ldots, v_n \in V$ und $e_1, e_2, \ldots, e_n \in E$ heißt **Kantenfolge**, wenn e_i die Endknoten v_{i-1}, v_i hat $(1 \leq i \leq n)$.
- Eine solche Kantenfolge, bei der alle vorkommenden Kanten verschieden sind, heißt **Kantenzug**.
- Eine solcher Kantenzug, bei dem alle vorkommenden Knoten verschieden sind, heißt **Weg** von v_0 nach v_n.
- Ein geschlossener Weg (d.h. $v_0 = v_n$) heißt **Kreis**.

Bei einer **Kantenfolge** ist also die „Mehrfachbenutzung“ von Kanten erlaubt, bei einem **Kantenzug** nicht.

Zusammenhängende (Di-)Graphen

Definition (*)

Ein Graph *G* heißt **zusammenhängend**, wenn es zwischen je zwei Knoten aus *G* einen Weg gibt.

Ein maximaler zusammenhängender Teilgraph von *G* heißt eine **(Zusammenhangs-)Komponente** von G.

Ein gerichteter Graph heißt **zusammenhängend**, wenn der zugehörige ungerichtete Graph zusammenhängend ist.

Breitensuche (Breadth First Search - BFS)

Gegeben: ein ungerichteter Graph G = (V, E)

1. Man markiere (d.h. durchsuche) einen beliebigen Startknoten.
2. Für jeden im vorigen Schritt (bzw. für jeden im vorigen Durchlauf von Schritt 2.) markierten Knoten, markiere man alle noch nicht markierten benachbarten Knoten.

   Wurde in diesem Schritt ein neuer Knoten markiert, dann wiederhole man Schritt 2 .

   Wurde kein neuer Knoten markiert, dann STOP.

BFS-Algorithmus – Eigenschaften

- Der BFS-Algorithmus durchsucht die Knoten der Breite nach.
- Er markiert, ausgehend von einem beliebigen Startknoten, dessen Zusammenhangskomponente.
- Sind am Ende, d.h. nach maximal $|E|$ Schritten, alle Knoten markiert, dann ist der Graph zusammenhängend.
- Ausgehend vom Startknoten markiert der Algorithmus im k-ten Schritt genau die Knoten, die vom Startknoten durch einen Weg der Länge k, aber keinen kürzeren erreichbar sind. (Beweis mit vollst. Induktion)

Euler-Zug & Algorithmus von Fleury (*)

Definition (*)

Ein **(geschlossener) Euler-Zug** ist ein (geschlossener) Kantenzug, der jede Kante des Graphen genau einmal enthält.

Algorithmus von Fleury (ähnlich zu [TT]) [zur Konstruktion eines Euler-Zugs (sofern vorhanden) ausgehend von einem bel. Startknoten]

1. Starte in einem beliebigen Knoten.
2. Wähle einen benachbarten Knoten und entferne die dabei durchlaufene Kante, wobei der übrigbleibende Restgraph zusammenhängend bleiben muss, sowie den Anfangsknoten der Kante, falls dieser nun isoliert, aber nicht der Startknoten ist. Ist das nicht möglich, dann STOP„Kein Euler-Zug vorhanden“. Besteht der Restgraph nur noch aus dem Anfangsknoten, dann STOP „Euler-Zug gefunden“. Ansonsten: wiederhole 2.

Die entfernten Kanten ergeben in der Reihenfolge ihrer Entfernung einen Euler-Zug.

Bäume

Definition (*)

Ein **Baum** ist ein Graph, der zusammenhängend ist und keine Kreise enthält. In einem Baum heißt ein Knoten v

- **Blatt**, wenn $d(v) = 1$.
- **innerer Knoten**, wenn $d(v) \neq 1$.

Satz (7) (*)

Ein zusammenhängender Graph $G = (V, E)$ mit n Knoten ist genau dann ein Baum, wenn er eine (und damit alle) der folgenden Eigenschaften hat:

1. *G hat genau $n - 1$ Kanten, d.h. $|E| = |V| - 1$, wobei $|E|$ = Kantenanzahl und $|V|$ = Knotenanzahl.*
2. *Entfernt man (irgend)eine Kante, so ist der Restgraph nicht mehr zusammenhängend.*
3. *Zwischen je zwei Knoten gibt es genau einen Weg.*

Bäume zur Speicherung von Daten: Wurzelbäume (*)

Wenn ein Baum einen ausgezeichneten Knoten w besitzt, von dem alle anderen Knoten „abstammen“, dann nennt man ihn **Wurzelbaum** mit **Wurzel** w. (Man „richtet“ die Kanten von der Wurzel weg, so dass es sich um einen **gerichteten Baum** handelt.)

- Wenn $w, ..., x, ..., y$ der (eindeutige) Weg von w nach y ist, so sind x ein **Vorgänger** von y und y ein **Nachfolger** von x.
- Ein benachbarter Vorgänger/Nachfolger heißt **unmittelbarer Vorgänger/Nachfolger**.
- Knoten ohne Nachfolger heißen **Endknoten** oder **Blätter**.
- Die **Länge des Wurzelbaums** ist die Länge des längsten Wegs von der Wurzel zu einem Knoten.
- Wenn jeder Knoten höchstens 2 unmittelbare Nachfolger hat, spricht man von einem **binären Baum**.
  Zu jedem Knoten x kann es einen linken u. einen rechten unmittelbaren Nachfolger x_L bzw. x_R geben. Der Baum mit Wurzel x_L bzw. x_R heißt **linker/rechter Unterbaum** von x.

Suchbaumalgorithmus (*) (1/2)

Gegeben sei eine Menge von Daten mit einer Ordnungsrelation $<$.
Die Daten sind als Knoten eines binären Wurzelbaumes gespeichert:
Für einen festen Knoten y liegen alle Knoten x mit $x < y$ im linken Unterbaum und alle x mit $x > y$ im rechten Unterbaum von y.
Im folgenden jeweils: Startknoten = Wurzel

- **Suche eines Knotens x:**
  Ist x gleich dem aktuellen Knoten y, dann STOP („x gefunden“).
  Ansonsten suche im entsprechenden Unterbaum (links für $x < y$ und rechts für $x > y$) weiter.
  Falls der zugehörige Unterbaum leer ist, dann STOP („x nicht gefunden“).
- **Einfügen eines Knotens x:**
  Suche nach x. Falls x nicht gefunden wird, ordne x als unmittelbaren (linken bzw. rechten) Nachfolger jenes Knotens ein, bei dem die Suche abgebrochen wurde.
  Falls x gefunden wird, STOP („x bereits vorhanden“).

Suchbaumalgorithmus (*) (2/2)

- **Löschen eines Knotens x:**
  Suche nach x. Je nachdem, wie viele unmittelbare Nachfolger x hat, sind drei Fälle zu unterscheiden:

  (1) Ist x ein **Blatt**, so entferne x und die zugehörige Kante mit Endknoten x.

  (2) Gibt es nur **einen Unterbaum**, der in x verwurzelt ist, so ersetze x durch diesen Unterbaum, d.h. entferne die Kante zwischen x und dem unmittelbaren Vorgänger und hänge den Unterbaum an den unmittelbaren Vorgänger.

  (3) Gibt es **zwei Unterbäume**, so suche zunächst das (im Sinn der Ordnung $<$) kleinste Element y im rechten Unterbaum von x (gehe dazu im rechten Unterbaum so lange nach links, bis es keinen linken unmittelbaren Nachfolger mehr gibt). Das so gefundene y ist entweder ein Blatt oder es hat genau einen (rechten) unmittelbaren Nachfolger. Lösche y (gemäß (1) bzw. (2)) und ersetze x durch y.

Suchbaumalgorithmus – Laufzeit

Satz (9)(*)

Die maximale Anzahl an Vergleichen, die der Suchbaumalgorithmus bei der Suche in einem binären Wurzelbaum der Länge l durchführen muss, ist ...

Die Laufzeit des Suchbaumalgorithmus ist also von der Ordnung ...
(auch für das Einfügen und Löschen von Daten).
Es ist somit erstrebenswert, Wurzelbäume mit möglichst geringer Länge zu verwenden.

Definition (Landau-Symbol)

O beschreibt die Größenordnung von Funktionen:
$f \in O(g)$ bedeutet
„f wächst nicht wesentlich schneller als g“, d.h.
g ist eine asymptotische obere Schranke.

Divide and Conquer

1. Daten sortieren
2. Wurzel $w :=$ mittleres Element
   $v := w$
3. linker Nachfolger von $v :=$
   mittleres Element der Hälfte der Elemente kleiner als v
   rechter Nachfolger von $v :=$
   mittleres Element der Hälfte der Elemente größer als v
4. Schritt 3 für den linken und den rechten Nachfolger von v ausführen
5. STOP, wenn die Hälfte der Elemente kleiner v und die Hälfte der Elemente größer v leer sind.

Graphentheoretische Optimierungsprobleme:

Minimal aufspannende Bäume (MST)

Kürzeste Wege Problem (SPP)

Traveling Salesman Problem (TSP)

Cliquen

Unabhängige Mengen

Färbungen

Gewichtete Graphen

Definition (*)

Ein Graph, bei dem jeder Kante $\{a, b\}$ ein Gewicht $w(a, b)$ zugeordnet ist, heißt **gewichteter Graph**.

Das Gewicht der Kante $\{j, k\}$ kann als a_{jk} in der Adjazenzmatrix abgespeichert werden.

Minimal aufspannender Baum
Minimum Spanning Tree (MST)

Definition (*)

Sie G ein zusammenhängender Graph mit n Knoten. Ein Teilgraph, der ein Baum ist und alle n Knoten von G enthält, heißt **(auf-)spannender Baum**.

Definition (*)

Ein **minimal (auf-)spannender Baum** T in einem gewichteten Graphen G ist ein aufspannender Baum mit minimalem Gesamtgewicht

$$\sum_{\{a,b\} \in E(T)} w(a,b) \ ,$$

wobei $E(T)$ die Kantenmenge des Baums T ist und $w(a,b)$ das Gewicht der Kante $\{a,b\}$ bezeichnet.

Algorithmus von Kruskal (*)

[zur Bestimmung eines minimal aufspannenden Baums]

Gegeben ist ein zusammenhängender, gewichteter Graph G mit n Knoten.

1. Ordne alle Kanten nach aufsteigendem Gewicht. Beginne mit dem Teilgraphen, der nur aus den n Knoten von G besteht, und füge eine Kante mit minimalem Gewicht hinzu (= die erste Kante aus der Liste der geordneten Kanten).
2. Wähle die nächste Kante $\{a,b\}$ aus der Liste der geordneten Kanten und füge sie zum bisherigen Graphen hinzu, falls die Komponenten von a und b verschieden sind (diese Bedingung stellt sicher, dass durch Hinzunahme von $\{a,b\}$ zu den bereits gewählten Kanten kein Kreis entsteht).
3. Wiederhole Schritt 2. solange, bis $n-1$ Kanten gewählt sind
   (sie ergeben einen minimalen aufspannenden Baum).

Bemerkungen zum Algorithmus von Kruskal (*)

Die Komponenten von a und b müssen natürlich nicht in jedem Schritt neu bestimmt werden:
Zu Beginn besteht die Komponente jedes Knotens nur aus dem Knoten selbst. In jedem Schritt, in dem eine Kante $\{a, b\}$ hinzugefügt wird, müssen dann nur noch die entsprechenden Komponenten von a und b vereinigt werden.
Dann kann man zeigen, dass der Algorithmus von Kruskal (bei geeigneter Datenstruktur für die Mengenoperationen) von der Ordnung $O(|E| \log |E|)$ ist, wobei $|E|$ die Kantenanzahl ist.

Der Algorithmus von Kruskal ist ein **Greedy-Algorithmus**.

Frage: Wie kann man diesen Algorithmus modifizieren, um einen **maximal aufspannenden Baum** zu bestimmen, d.h. einen Baum mit maximalem Gesamtgewicht?

Kürzeste Wege Problem (Shortest Path Problem - SPP)

Definition (*)

Die **Länge eines Weges** W in einem gewichteten Graphen ist die Summe der Gewichte der Kanten, die entlang von W durchlaufen werden:

$$\sum_{\{a,b\} \in E(W)} w(a, b) \ ,$$

wobei $E(W)$ die Kantenmenge des Weges W bezeichnet.
Ein **kürzester Weg** von einem Knoten s zu einem Knoten z ist ein Weg mit minimaler Länge von s nach z.

Spezialfall: $w(e) = 1$ für alle Kanten e.

Algorithmus von Dijkstra (1) (*)

[zur Bestimmung je eines kürzesten Weges von einem Startknoten zu **allen anderen** Knoten]

Gegeben sei ein zusammenhängender Graph mit Knoten $1, 2, \ldots, n$ und Kanten $\{i, j\}$ mit Gewichten $w(i, j) \geq 0$.
Der Algorithmus bestimmt die Längen L_j der kürzesten Wege von 1 nach j für alle Knoten $j = 2, 3, \ldots, n$ und einen aufspannenden Baum mit diesen kürzesten Wegen.
Hierbei wird nach und nach jedem Knoten k entweder

- ein permanentes Label L_k
  (Länge des kürzesten Weges von s nach k)
- oder ein temporäres Label T_k
  (Abschätzung nach oben für den kürzesten Weg von s nach k)

zugeordnet.
In jedem Iterationsschritt gibt das (ggf. temporäre) Label für jeden Knoten k die Länge eines kürzesten Weges von s nach k an, der **nur permanent gelabelte Knoten** verwendet.

Algorithmus von Dijkstra (2) (*)

1. Knoten 1 erhält den permanenten Label $L_1 = 0$.
   Alle anderen Knoten ($j = 2, \ldots, n$) erhalten temp.Label $T_j = \infty$.
   Wähle den Knoten $k = 1$.
2. Update der temporären Label:
   Für jeden zu k benachbarten, temporär markierten Knoten j:
   Bestimme $L_k + w(k, j)$ (= Länge des Weges von 1 nach j über den soeben permanent markierten Knoten k). Ist $L_k + w(k, j)$ kleiner als der bisherige temporäre Label T_j , so wird er der neue temporäre Label: $T_j = L_k + w(k, j)$ und wir notieren k als Vorgänger von j : $V_j = k$.
3. Fixierung eines permanenten Labels:
   Wähle einen Knoten k unter den temporär markierten Knoten, dessen temporärer Label T_k minimal ist. Dieser Knoten erhält den permanenten Label $L_k = T_k$ (damit ist der kürzeste Weg von 1 nach k ermittelt).
   STOP, wenn alle Knoten permanent markiert sind (mithilfe der Vorgänger kann nun der Weg rekonstruiert werden),
   ansonsten wiederhole Schritt 2.

Eigenschaften des Dijkstra-Algorithmus

- Ausgehend vom Startknoten wird ein aufspannender Baum konstruiert.
  In jedem Schritt wird eine kürzeste Verlängerung des aktuellen Teilbaumes gesucht.
- Der Dijkstra Algorithmus ist ein Greedy-Algorithmus.
- Der temporäre Label verbessert sich von Schritt zu Schritt oder bleibt gleich
  und wird – wenn er minimal ist – in einen permanenten Label umgewandelt.
- Man kann zeigen, dass der Dijkstra Algorithmus von der Ordnung $O(|V|^2)$ ist, wobei $|V|$ die Knotenanzahl des Graphen ist.

Floyd-(Warshall-)Algorithmus [zur Bestimmung der jeweiligen Länge eines **kürzesten Weges** von **jedem** Knoten i zu **allen anderen** Knoten]

Gegeben sei ein gerichteter Graph $G = (V, A)$ mit Knoten $1, 2, \ldots, n$ und Kanten (i, j) mit Gewichten $w(i, j)$ (die auch negativ sein können)

begin

for $i = 1, \ldots, n$ do

for $j = 1, \ldots, n$ do

$$d_{ij} = \begin{cases} w(i,j) & \text{falls } i \neq j \text{ und } (i,j) \in A \\ \min\{0, w(i,j)\} & \text{falls } i = j \text{ und } (i,i) \in A \\ 0 & \text{falls } i = j \text{ und } (i,i) \notin A \\ \infty & \text{sonst} \end{cases}$$

$$p_{ij} = \begin{cases} i & \text{falls} (i,j) \in A \\ 0 & \text{sonst (derzeit kein Weg bekannt oder } i = j) \end{cases}$$

for $k = 1, \ldots, n$ do

for $i = 1, \ldots, n$ do

for $j = 1, \ldots, n$ do

$d_{ij} := \min\{d_{ij}, d_{ik} + d_{kj}\}$ u. falls $d_{ij} > d_{ik} + d_{kj}$: $p_{ij} := p_{kj}$

end

Floyd-Algorithmus – (Re-)Konstruktion der in P gespeicherten kürzesten Wege

(Re-)Konstruktion des in P gespeicherten kürzesten (i, j)-Wegs:

$$\begin{aligned} v_q &:= p_{ij} \\ v_{q-1} &:= p_{iv_q} \\ v_{q-2} &:= p_{iv_{q-1}} \\ &\vdots \\ i &= p_{iv_1} \end{aligned}$$

(d.h. man iteriert so lange, bis der Knoten i erreicht ist)

Dann ist $(i, (i, v_1), v_1, (v_1, v_2), v_2, ..., v_q, (v_q, j), j)$ ein kürzester (i, j)-Weg.

Floyd-Algorithmus – Korrektheit

Gegeben sei ein gerichteter Graph $G = (V, A)$ mit Knoten $1, 2, \ldots, n$ und Kanten $\{i, j\}$ mit Gewichten $w(i, j)$ (die auch negativ sein können).
Sei D die $(n \times n)$-Matrix, die der Floyd-Algorithmus generiert.
Dann gilt:

- Der Floyd-Algorithmus liefert genau eine Kürzeste-Weglängen-Matrix D, wenn G keinen gerichteten Kreis mit negativer Länge enthält.
- D enthält einen gerichteten Kreis mit negativer Länge.
  $\Longleftrightarrow$ Ein Diagonalelement von D ist negativ.
- Der Rechenaufwand des Algorithmus ist polynomial, d.h. der Algorithmus ist ein „schneller“ Algorithmus. Der Aufwand des Algorithmus ist $O(n^3)$.

Hamilton-Kreis

Definition (*)

In einem Graphen G heißt ein Kreis, der jeden Knoten (genau einmal) enthält, **Hamilton-Kreis**.

Satz (11)(*)

1. *Wenn ein Graph $G = (V, E)$ mindestens $\frac{1}{2}(|V| - 1)(|V| - 2) + 2$ Kanten enthält, dann besitzt er einen Hamilton-Kreis.*
2. *Wenn in einem Graphen $G = (V, E)$ jeder Knoten mindestens den Grad $\frac{|V|}{2}$ hat, dann besitzt der Graph einen Hamilton-Kreis.*

Das Entscheidungsproblem HAMILTONIAN CIRCUIT:
„Besitzt der Graph G einen Hamilton-Kreis?“
ist *NP*-vollständig.

Das (symmetrische) Traveling Salesman Problem (TSP)

Ein Handlungsreisender soll n Kunden in n verschiedenen Städten besuchen. Zu je zwei Städten ist die Entfernung zwischen diesen beiden Städten angegeben. Gesucht ist eine **Rundreise**, bei der der Handlungsreisende jede Stadt genau einmal besucht und er eine minimale Entfernung zurücklegt.

Graphentheoretische Modellierung:

Traveling Salesman Problem (TSP) - Graphentheoretische Modellierung

Definition (Traveling Salesman Problem (TSP) (*))

Gegeben sei der vollständige, gewichtete Graph K_n mit der Knotenmenge $V = \{1, 2, \ldots, n\}$. Gesucht ist eine Permuation $\pi(1), \pi(2), \ldots, \pi(n)$ der Knoten $1, 2, \ldots, n$ mit minimalem Gesamtgewicht

$$\sum_{i=1}^{n} w(\pi(i), \pi(i+1)) \ ,$$

wobei $\pi(n+1) := \pi(1)$ gesetzt wird.

(Die Permutation gibt die Reihenfolge an, in der die Städte besucht werden: nach $\pi(i)$ wird $\pi(i+1)$ besucht.)

Zur Erinnerung: Jede Permutation kann als bijektive Abbildung (in diesem Fall: der Knotenmenge auf sich selbst) aufgefasst werden.

Dynamische Programmierung für das TSP (1/2)

Man versucht, die Lösung des Problems auf kleinere Probleme der gleichen Art zurückzuführen (Rekursion).

Gegeben sei ein vollständiger Graph mit einer Knotenmenge (=Menge von Städten) $S = \{1, 2, \ldots, n\}$ und Kantengewichten (=Entfernungen) $w(i, j) \geq 0$

$L(i, A) :=$ Länge eines kürzesten Weges, der

- in der Stadt i beginnt,
- jede Stadt in der Menge A genau einmal besucht und
- in der Stadt 1 endet.

Dann gilt:

1. Für jede Stadt $i \in S$ ist $L(i, \emptyset) = w(i, 1)$
2. Für jede Stadt $i \in S$ und jede Teilmenge $A \subseteq S \setminus \{1, i\}$ mit $A \neq \emptyset$ ist
   $$L(i, A) = \min\{w(i, j) + L(j, A \setminus \{j\}) \mid j \in A\}$$
3. $L(1, \{2, 3, ..., n\})$ ist die Länge einer optimalen Rundreise.

Dynamische Programmierung für das TSP (2/2)

Gegeben sei ein vollständiger Graph mit einer Knotenmenge (=Menge von Städten) $S = \{1, 2, \ldots, n\}$ und Kantengewichten (=Entfernungen) $w(i, j) \geq 0$

```
begin
   for i = 1, ..., n do
      L(i, ∅) := w(i, 1);
   for k = 1, ..., n − 1 do
      forall A ⊆ S\{1} with |A| = k do
         for i = 1, ..., n do
            if i ∉ A then
               L(i, A) := min{w(i, j)+L(j, A\{j}) | j ∈ A} ;
   return L(1, {2, ..., n}) ;
end
```

Heuristiken für das TSP: Nearest-Neighbour-Approximation

Nearest-Neighbour-Approximation:

1. Man wähle einen Startknoten $i \in V$ und definiere
   $V' := \{i\}$ als die Menge der bereits besuchten Knoten sowie
   $p := i$ als den aktuell besuchten Knoten.
2. Man gehe vom Knoten p zum am nächsten gelegenen, noch nicht besuchten Knoten j, d.h. j sei so gewählt, dass

   $$w(p,j) = \min\{w(p,k) \mid k \in V \setminus V'\}$$

   Man aktualisiere
   $V' := V' \cup \{j\}$ und $p := j$ als den aktuell besuchten Knoten.
3. Gilt $V' = V$, dann sind alle Knoten durchlaufen. Man gehe zurück zum Startknoten und STOP.
   Andernfalls wiederhole man Schritt 2.

Liefert in $O(n^2)$ Schritten eine approximative Lösung.

Heuristiken für das TSP: Farthest-Insert-Approximation

Farthest-Insert-Approximation:
Man nehme in eine Teiltour den am weitesten entfernten, noch nicht besuchten Knoten auf, d.h.:

- Sei V' die Knotenmenge einer bereits bestimmten Teiltour.
- Wir definieren für $i \notin V'$:

  $$w(i, V') := \min\{w(i,j) \mid j \in V'\}$$ (Distanz von i zur Teiltour)

  und wählen $k \notin V'$ so, dass

  $$w(k, V') := \max\{w(i, V') \mid i \notin V'\}$$

  $\Rightarrow$ In jedem Schritt wird ein neuer Knoten k optimal in die schon bestehende Teiltour eingefügt.

Liefert eine approximative Lösung, d.h. eine Näherungslösung, die nicht unbedingt optimal ist.

Heuristiken für das TSP: MST-Approximation

Spanning-Tree-/ Minimum-Tree-Approximation:

1. Man bestimme einen minimal spannenden Baum.
2. Man verdopple die Kanten des Baums.
3. Man bestimme einen **geschlossenen Euler-Zug** C
   (= Kantenzug, der jede Kante genau einmal durchläuft)
   (z.B. mit dem Algorithmus von Fleury)
   und gebe ihm eine Orientierung.
4. Man wähle einen Startknoten i setze $p = i$ und $T = \emptyset$.
5. Man laufe von p entlang der Orientierung von C, bis ein unmarkierter Knoten q erreicht ist.
   Setze $T := T \cup \{(p, q)\}$, markiere q, setze $p := q$.
   (Man sucht also eine „Abkürzung“, wenn man auf einen schon besuchten Knoten trifft.)
6. Sind alle Knoten markiert, dann setze $T := T \cup \{(p, i)\}$ und STOP (T ist eine Tour).
   Andernfalls gehe zu 5.

Heuristiken für das euklidische TSP: Gütegarantien und Laufzeit

Ein symmetrisches TSP heißt **euklidisch**, wenn für die Kantengewichte die Dreiecksungleichung gilt:

$$w(i, k) \leq w(i, j) + w(j, k)$$

Heuristik	Güte (ϵ)	Laufzeit
Nearest Neighbour	$\frac{1}{2}(\lceil \log n \rceil - 1)$	n^2
Nearest Insert	1	n^2
Farthest Insert		n^2
Cheapest Insert	1	n^2
Spanning Tree	1	$n^2 \log n$
Christofides	$\frac{1}{2}$	n^3

Gütegarantie ϵ bedeutet: $w(T_{opt}) \leq w(T_H) \leq (1 + \epsilon) w(T_{opt})$

wobei T_{opt} eine optimale Tour ist und $w(T_{opt})$ ihr Gesamtgewicht sowie T_H eine mit Heuristik ermittelte Tour und $w(T_H)$ deren Gesamtgewicht.

Cliquen & Cliquenzahl

Definition (Cliquenzahl)

Eine **Clique** des Graphen $G = (V, E)$ ist eine Teilmenge der Knotenmenge $C \subseteq V$, so dass für alle $u, v \in C$ (mit $u \neq v$) die Kante $\{u, v\} \in E$ ist, d.h. die Knoten in C sind paarweise adjazent.

Clique heißt **maximale Clique**, wenn sie keine echte Teilmenge einer größeren Clique ist.

Die **Cliquenzahl** ist die maximale Anzahl von Knoten in einer Clique von G, d.h.

$$\omega(G) = \max\{|C| \mid C \text{ ist Clique von } G\} .$$

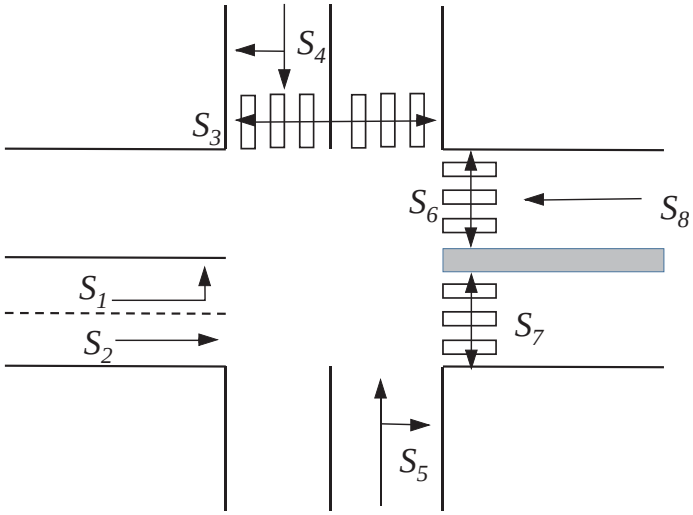
Satz (12)

Die Knotenmenge jedes (endlichen) Graphen lässt sich in disjunkte Cliquen (vollständig) zerlegen.

Beweis: Man nehme z.B. jeden Knoten als Clique (entspricht jeweils einem Teilgraphen K_1). Interessanter sind jedoch Zerlegungen der Knotenmenge in möglichst wenige, paarweise disjunkte Cliquen. □

Cliquen – Anwendungsbeispiel

Gegeben: eine signalgesteuerte Verkehrskreuzung mit Verkehrsströmen $S_1, \ldots, S_n$



Gesucht: eine günstige Planungsfolge ohne Kollisionsgefahr

Modellierung: $G = (V, E)$ mit

Unabhängige Mengen, Unabhängigkeitszahl

Definition (Unabhängige Mengen, Unabhängigkeitszahl)

Eine Teilmenge $U \subseteq V$ heißt **unabhängige Menge** oder **stabile Menge** (in einem Graphen G), falls keine zwei verschiedenen Knoten $u, v \in U$ in G adjazent sind, d.h. falls $\{u, v\} \notin E$ für alle $u, v \in U$ mit $u \neq v$.

Eine unabhängige Menge ist eine **maximale unabhängige Menge**, wenn sie keine echte Teilmenge einer größeren unabhängigen Menge ist.

Für einen Graphen G bezeichnet

$$\alpha(G) = \max\{|U| \mid U \text{ ist unabhängige Menge in } G\}$$

die **Unabhängigkeitszahl** (**Stabilitätszahl**) von G.

Satz (13)

*Ist U unabhängige Menge in G und C eine Clique von G, dann kann U maximal **einen** Knoten aus C enthalten.*

Komplementärgraph

Definition (Komplementärgraph)

Sei $G = (V, E)$ ein Graph. Dann ist $\bar{G} := (V, \bar{E})$ mit

$$\bar{E} := \{\{u, v\} | u \neq v \text{ und } \{u, v\} \notin E\}$$

der zu G **komplementäre Graph** oder auch **Komplementärgraph**.

Frage: Wie hängen Cliquen und unabhängige Mengen zusammen?

Chromatische Zahl (Färbungszahl)

Definition (Chromatische Zahl)

Eine **Färbung** eines Graphen $G = (V, E)$ mit k Farben ordnet jedem Knoten eine Zahl aus $\{1, 2, \ldots, k\}$ zu.
Die Färbung heißt **zulässig**, wenn für jede Kante die beiden Endknoten verschiedene Farben haben.

Die **chromatische Zahl** oder auch **Färbungszahl** $\chi(G)$ des Graphen G ist die kleinste Zahl k, für die eine zulässige Färbung mit k Farben möglich ist.

Färbungen und unabhängige Mengen bzw. Cliquen

Frage: Wie hängen Färbungen und unabhängige Mengen zusammen?

Da in einer Färbung die Knoten jeder Clique unterschiedliche Farben erhalten müssen, ergibt sich folgender Satz:

Satz (14)

Für jeden Graphen G gilt $\chi(G) \geq \omega(G)$.

Aufgabe: Geben Sie einen Graphen an, für den hierbei Ungleichheit gilt.

Satz (15) (Vierfarbensatz)

Für jeden planaren Graph G gilt $\chi(G) \leq 4$

Sequential Coloring Algorithmus

Färbungsalgorithmus: „Sequentielles Färben“ bzw. „Sequential Coloring“

Basisidee: Man ordnet die Knoten des Graphen in eine (zunächst beliebige) Reihenfolge $v_1, v_2, \ldots, v_n$.
Dann weist man für $i = 1, 2, \ldots, n$ dem Knoten v_i die jeweils kleinste zulässige Farbe zu.

Gegeben: ein ungerichteter Graph G = (V, E) in Adjazenzlistendarstellung

1. Wähle eine Folge $v_1, v_2, \ldots, v_n$ aller Knoten.
2. Setze $f(v_1) := 1$.
3. Für $i = 2, 3, \ldots, n$ führe folgenden Schritt durch:
   Setze $f(v_i) := q$, d.h. färbe v_i mit q, wobei $q \in \mathbb{N}$ die kleinste natürliche Zahl sei, so dass für alle Kanten $\{v_i, v_j\} \in E(G_i)$ gilt: q ist nicht Farbe von v_j.
   $E(G_i) := \{e \in E \mid e \text{ verbindet zwei Knoten aus } \{v_1, v_2, ..., v_i\} \subseteq V\}$

$f : V \to 1, 2, \ldots, n$ ist dann eine zulässige Färbung.

Sequential Coloring Algorithmus – Eigenschaften (1/2)

Sei $\Delta := \Delta(G)$ der **Maximalgrad** von G, d.h. der maximale im Graphen G vorkommende Grad.

Satz (16)

Der Sequential Coloring Algorithmus angewendet auf einen Graphen $G = (V, E)$ benötigt höchstens $\Delta(G) + 1$ Farben. Die Laufzeit ist linear, d.h. $O(|V| + |E|)$.

Beweis: Ein Knoten $v_i (1 \leq i \leq n)$ hat höchstens $d(v_i) \leq \Delta$ inzidente Kanten in G. $\Rightarrow$ $f(v_i) \leq d(v_i) + 1$ und $\max_i f(v_i) \leq \Delta + 1$.
Da der Algorithmus jede Kante höchstens zweimal betrachtet, ergibt sich eine Laufzeit von $O(|V| + |E|)$. □

Sequential Coloring Algorithmus – Eigenschaften (2/2)

Der Sequential Coloring Algorithmus liefert allerdings

- nicht immer eine optimale Färbung,
- i.a. unterschiedliche Farbanzahlen in Abhängigkeit von der verwendeten Reihenfolge der Knoten.

Es lässt sich jedoch zeigen, dass man bei geeigneter Reihenfolge der Knoten mit der optimalen Farbanzahl auskommt.

Beweis: Sei f eine optimale Färbung von G, d.h. $f : V \to \{1, 2, \ldots, \chi(G)\}$.
Wir definieren die **Farbklassen** $f^{-1}(i) := \{v \in V | f(v) = i\}$.
Die Reihenfolge für die Knoten wählen wir entsprechend der Farbklassen $f^{-1}(1), f^{-1}(2) \ldots, f^{-1}(\chi(G))$, wobei wir Knoten gleicher Farbe beliebig anordnen dürfen.
Durch Induktion folgt, dass der Algorithmus eine Färbung $\widetilde{f}$ mit $\widetilde{f}(v) \leq i$ für alle $v \in f^{-1}(i)$ erzeugt.
$\Rightarrow$ Der Sequential Coloring Algorithmus benötigt höchstens $\chi(G)$ Farben.

Tiefensuche (Depth First Search - DFS)

Tiefensuche (Depth First Search (DFS)) (1/2)

Gegeben: ein ungerichteter Graph G=(V, E) mit $V=\{1,2,...,n\}$ in Adjazenzlistendarstellung

$N[k]$: Liste der zu k adjazenten Knoten. (Nach Benutzung einer Kante wird der zugehörige Endknoten aus $N[k]$ entfernt.)

In $num[n]$ wird festgehalten, in welcher Reihenfolge die Knoten von G erreicht werden.

1. Setze:
   $k := 1$ (aktuell besuchter Knoten)
   $p(1) := 0$ und $p(j) := -1$ für $2 \leq j \leq n$
   ($p(j)$ soll der Knoten sein, der vor j besucht wurde.)
   Iterationsindex $i := 1$
   $num(1) := 1$

2. Falls $N[k] = \emptyset$ gehe zu Schritt 4 (denn alle zu k inzidenten Kanten wurden benutzt).

Tiefensuche (Depth First Search (DFS)) (2/2)

3. Wähle $j \in N[k]$ und entferne j aus $N[k]$.
   Falls $p(j) = -1$, mache einen Vorwärtsschritt zu einem neuen Knoten, d.h. setze
   $p(j) := k$
   $k := j$
   $i := i + 1$
   $num(k) := i$ (denn k wurde im i-ten Schritt erreicht).
   Gehe zu Schritt 2.

4. Falls $num(k) = 1$, dann STOP (denn alle Kanten der Wurzel wurden benutzt).
   Andernfalls: Setze $k := p(k)$ und gehe zu Schritt 2 (Rückschritt zu altem Knoten).