

GREEDY ALGORITHMEN

G1.

(gierige Algorithmen)

a.) "Definition" von greedy Algorithmen

Welche greedy Algorithmen haben wir bisher gesehen?

Beispiele:

- LIST für SCHEDULING $(2 - \frac{1}{m})$ -approximativ)

- bearbeite die jobs in Reihenfolge der Eingabe

repeat $\left\{ \begin{array}{l} - \text{teile den nächsten Job einer Maschine zu} \\ - \text{revidiere diese Entscheidung später nicht.} \end{array} \right.$

- LPT (Longest Processing Time) für SCHEDULING $(\frac{4}{3})$ -approximativ)

- bearbeite die jobs in absteigender Folge der Laufzeiten

repeat $\left\{ \begin{array}{l} - \text{teile den nächsten Job einer geeigneten Maschine zu} \\ - \text{revidiere diese Entscheidung später nicht} \end{array} \right.$

- Greedy Algorithmus für CLIQUE $(\leq \frac{n}{3})$ -approximativ)

repeat $\left\{ \begin{array}{l} - \text{nehme den Knoten mit höchstem Grad} \\ - \text{entferne nicht-adjazente Knoten} \\ - \text{die ausgewählten Knoten werden nicht zurückgenommen} \end{array} \right.$

- Greedy Algorithmus für VERTEX COVER (2) -approximativ)

repeat $\left\{ \begin{array}{l} - \text{nehme beliebige Kante (ihre Endknoten)} \\ - \text{entferne die adjazenten Kanten} \\ - \text{die ausgewählten Endknoten werden später nicht zurückgenommen} \end{array} \right.$

- Greedy Algorithmen für COLORING

Grobe Definition:

Ein Greedy Algorithmus bestimmt eine Lösung iterativ; jedes mal wird eine Entscheidung getroffen, die lokal am vielversprechendsten ist. Getroffene Entscheidungen werden nicht revidiert.

Eine konkreter definierbare Sorte von Greedy Algorithmen:

Besser: ein Versuch Greedy Algorithmen konkreter zu definieren.

Priority Algorithmen

- Jede Eingabe besteht aus Datenelementen; es gibt eine festgelegte vollständige Ordnung auf allen möglichen Datenelementen (man kann z.B. für einen Alg. festlegen, dass längere Jobs höhere Priorität haben als kürzere, unabhängig von konkreten Instanzen)
- In jedem Schritt erhält der Algorithmus das Datenelement das aktuell die höchste Priorität hat, und trifft eine nicht-revidierbare Entscheidung für dieses Datenelement. Die (Prioritäten der) restlichen Datenelemente können sich in jeder Iteration ändern.

Beispiele:

	<u>Datenelement</u>	<u>Vollständige Ordnung</u>	<u>Entscheidung</u>
LPT	Job mit Laufzeit (p_i)	nach fallenden Laufzeiten	Zuweisung zu einer Maschine
Greedy- CLIQUE	Knoten mit Grad ($v, \deg(v)$)	nach fallendem Grad	Hinzunahme zur Clique, falls bisher adjazent
Greedy-VC	Kante (u, v)	beliebig	Hinzunahme der Endpunkte zur Knotenüberdeckung falls bisher nicht adjazent

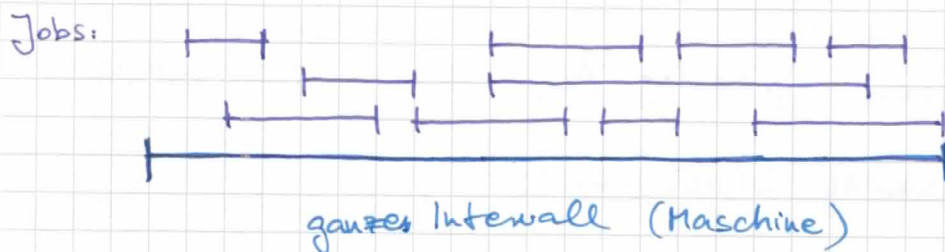
b.) Optimale Greedy Algorithmen (Wiederholung)1. Intervall - Scheduling

Eingabe: n Aufgaben (Jobs) $A_1 A_2 A_3 \dots A_n$
 jeweils mit Startzeiten $s_1 s_2 s_3 \dots s_n$
 und Endpunkten $e_1 e_2 e_3 \dots e_n$

Ausgabe: eine größtmögliche Teilmenge von Jobs
 (d.h. mit den meisten Jobs drin) die ohne
 Überlappen auf einer Maschine ausführbar sind

($A_{i_1} A_{i_2} \dots A_{i_m}$ s.d.

wenn $k \neq l$ dann $[s_{i_k}, e_{i_k}) \cap [s_{i_l}, e_{i_l}) = \emptyset$)



Es gibt einen Greedy Algorithmus, der eine optimale Lösung ausgibt.

Die Datenelemente werden offensichtlich die Jobs sein;
 aber was bestimmt die Priorität?

Man kann leicht einsehen, dass nach ^(aufsteigender) Startzeit, Länge oder Kürze zu sortieren, kein optimales Ergebnis resultiert.

Greedy Intervall-Scheduling

- sortiere die Aufgaben gemäß aufsteigenden Endpunkten

Bezeichne $e^1 \leq e^2 \leq e^3 \leq \dots \leq e^n$ die sortierten Endpunkte,

und $A^1 A^2 A^3 \dots A^n$ die entsprechenden Aufgaben

und $s^1 s^2 s^3 \dots s^n$ die jeweiligen Startpunkte

- Bezeichne J die Menge der ausgewählten Jobs

Setze $J = \emptyset$, und $E = 0$

- FOR $k = 1$ to n DO

IF $s_k \geq E$ THEN $J = J \cup \{A^k\}$ und $E = e^k$
ELSE verwerfe A^k

Theorem: Dieser Greedy Algorithmus ist optimal für INTERVALL SCHEDULING.

Beweis: Wir zeigen durch Induktion über die Anzahl n der Jobs in der Eingabe, dass Greedy eine maximale Anzahl nicht kollidierender Aufgaben in der Lösung erzielt.

Basisfall:

- für $n=1$ Aufgabe ist Greedy optimal -

Induktionsschritt:

- Angenommen, Greedy ist optimal für $\leq n-1$ Jobs in der Eingabe, wir zeigen, dass er auch für n Jobs optimal ist:

Teil 1: A^1 darf in die Lösungsmenge I

Behauptung 1: Es gibt mindestens eine optimale Lösung, die A^1 enthält.

Warum? Nehmen wir eine beliebige optimale Lösung, und sei A^* mit Startpunkt s^* und Endpunkt e^* die erste Aufgabe in dieser Lösung. Falls $A^* = A^1$, die Behauptung gilt. Falls $A^* \neq A^1$, dann $e^* \geq e^1$, weil e^1 so definiert wurde. (e^1 ist minimal).

Deshalb kollidiert A^1 mit anderen Aufgaben der optimalen Lösung nicht (sonst würde A^* auch kollidieren).

Wir tauschen A^* gegen A^1 in der optimalen Lösung, und erhalten somit eine optimale Lösung, die A^1 enthält.

Teil 2: Wir dürfen dann mit Greedy weitermachen

Behauptung 2: Unter allen Lösungen die A^1 enthalten, gibt Greedy eine optimale Lösung aus.

Warum? Lösungen, die A^1 enthalten, enthalten keinen Job der mit A^1 kollidiert, wir dürfen also solche kollidierenden Aufgaben aus der Eingabe I entfernen. Wir entfernen auch noch A^1 , weil er in allen diesen Lösungen drin ist. Sei I' diese neue Instanz mit $\leq n-1$ Aufgaben. Laut Induktionsannahme gibt Greedy eine maximale Anzahl nicht-kollidierender Aufgaben aus I' aus.

□

2. Huffman - Code

Eingabe: Eine Datei bestehend aus Zeichen über dem endlichen Alphabet Σ .

Für $a \in \Sigma$ bezeichne $H(a)$ die Häufigkeit von a in der Datei.

Ausgabe: Bestimme einen Präfixcode

$\text{code} : \Sigma \rightarrow \{0,1\}^*$ so dass die Länge der Kodierung der Datei minimiert wird

D.h. wir kodieren die Buchstaben in Σ durch binäre Werte (z.B. für $a \in \Sigma$ $\text{code}(a) = 0101$) so dass kein Codewort $\text{code}(a)$ Präfix von einem anderen Codewort $\text{code}(b)$ ist. Zu minimieren ist die Länge der kodierten Datei:

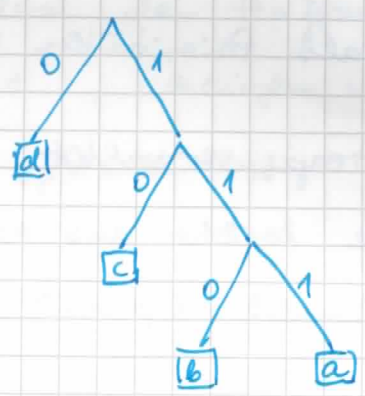
$$\|\text{code}\| = \sum_{a \in \Sigma} H(a) \cdot |\text{code}(a)|$$

\swarrow Summe
 \searrow Zeichenmenge (Alphabet)

Zur Erinnerung:

Jeden Präfixcode kann man als ein Binärbaum, namens Kodierbaum repräsentieren, dessen Kanten durch 0 oder 1 markiert sind, und somit jedes Blatt einem Codewort entspricht. Die Zeichen aus Σ werden 1-1 den Blättern zugeordnet (gemäß ihrem jeweiligen Codewort). Da kein Zeichen Nachfolge eines Anderen ist, ist auch kein verwendetes Codewort Präfix eines Anderen. Welche Zeichen sollten die längeren Codewörter erhalten?

Beispiel:



- code (d) = 0
- code (c) = 10
- code (b) = 110
- code (a) = 111

Huffman's Algorithmus

- anfänglich entspricht jedem Buchstaben ein Einzelknoten

- WHILE $|\Sigma| \geq 2$

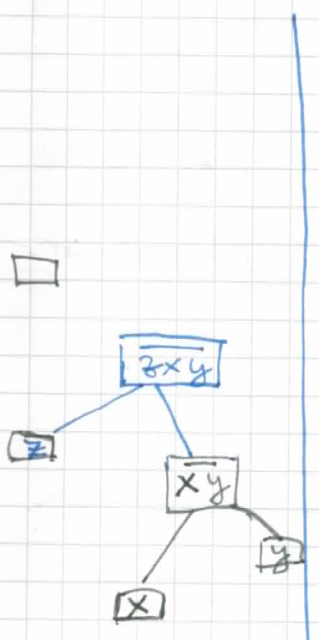
- nimm zwei Buchstaben $x, y \in \Sigma'$ mit den kleinsten Häufigkeiten ($H(x), H(y)$ kleinstmöglich) entferne sie aus Σ'

$\Sigma' = \Sigma' \setminus \{x, y\}$

- erzeuge neues Zeichen $\bar{x}y$ mit $H(\bar{x}y) = H(x) + H(y)$

$\Sigma' = \Sigma' \cup \{\bar{x}y\}$

- erzeuge neuen Knoten im Kodierbaum für $\bar{x}y$, und die Knoten von x bzw. von y seien die zwei Kinder dieses Knotens.



G 8.

Wie immer neue Buchstaben wie $\bar{x}y$ erzeugt werden,
 Die Buchstaben (Zeichen) mit ihren
 $H(\cdot)$ -Werten als Prioritäten werden in
 einem Min-Heap verwaltet. Eine Runde der
 WHILE-Schleife besteht aus ~~es~~ zwei DeleteMin,
 und einer Insert-Operation.

(Die Datenelemente sind die Zeichen mit den
 $H(\cdot)$ -Werten als Prioritäten. Beachte, dass im Laufe
 des Algorithmus neue Datenelemente eingefügt werden.)

Laufzeit: $O(|\Sigma| \cdot \log|\Sigma|)$ ($|\Sigma|$ -mal 2 DeleteMin + 1 Insert Op.)

Theorem: Huffman's Algorithmus ist optimal:
 Er erzeugt einen Präfixcode mit kürzester
 Kodierung der Datei.

Beweis: Wir zeigen durch Induktion über die Anzahl
 der Zeichen $|\Sigma|$, dass $\|\text{code}\| = \sum_{a \in \Sigma} H(a) \cdot |\text{code}(a)|$ minimal ist.

Summe Menge
 der Zeichen

Wir identifizieren jeden Code mit seinem Kodierbaum.
 Seien x und y Zeichen mit kleinsten Häufigkeiten

$$H(x) \leq H(y) \leq H(\cdot) \dots$$

Basisfall:

- für $|\Sigma| = 2$ der Code  minimiert

$\|\text{code}\|$ unter allen Präfix-Codes ✓

Huffman-Code

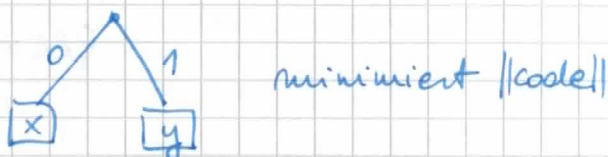
G 8. b.

Thm: Der Algorithmus von Huffman erzeugt einen optimalen Präfix-Code.

Sei (Σ, H) die Eingabe (Alphabet, Häufigkeiten)

Beweis durch Induktion über die Anzahl der Buchstaben im Alphabet $|\Sigma|$

Basissschritt: $|\Sigma| = 2$



Induktionsschritt: Angenommen, der Huffman-Code ist optimal für $n-1$ Buchstaben $|\Sigma| = n-1$, wir zeigen, dass er auch für $|\Sigma| = n$ Buchstaben optimal ist.

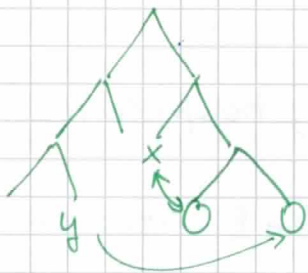
Teil 1: Seien $H(x)$ und $H(y)$ (die) zwei kleinsten Häufigkeiten, dann gibt es einen minimalen Präfix-Code, in dem x und y Geschwister-Blätter sind.

(d.h. der Alg. darf so anfangen wie der Alg. von Huffman)



In einem minimalen Präfix-Code (Baum) hat jeder innere Knoten 2 Kinder; somit hat jeder Knoten einen Geschwisterknoten, außer der Wurzel

nimm einen optimalen Kodierbaum; er hat
Geschwisterblätter auf der untersten Ebene
(mindestens ein Blatt, das aber ein Geschwister haben soll)



→ falls x nicht eines dieser Blätter
ist, dürfen wir austauschen,
der $\|code\|$ kann dadurch
nicht länger werden (weil $H(x)$ minimal)

Austauschargument → falls y nicht das andere dieser
Blätter ist, dürfen wir austauschen

Teil 2: Unter allen Präfix-Codes in denen x und y
Geschwister-Blätter sind, ist der Huffman-Code
optimal.

(d.h. der Alg. darf mit Huffman weitermachen...)

Wir erstellen eine einfachere Instanz
(um die Induktionsannahme für $n-1$ Buchstaben
zu nutzen)

Original-Eingabe

Σ, H

einfachere Eingabe

Σ', H'

statt x und y nehmen
wir einen
 x -oder- y Buchstaben $\tilde{x}y$
mit Häufigkeit

$$H'(\tilde{x}y) = H(x) + H(y)$$

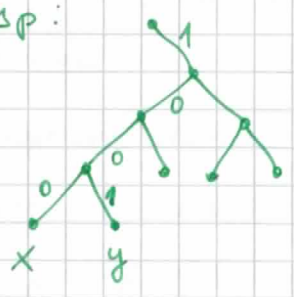
alles andere bleibt
gleich.

ein minimaler Präfixcode für (Σ, H) wo x und y Geschwister sind

"entspricht"

ein minimaler Präfixcode für (Σ', H')

Bsp:



Sei $Code(x) = 1000$
 $Code(y) = 1001$

Wie?

Im Text ist jetzt x -oder- y ein Buchstabe.

In der Kodierung können wir ersetzen ~~mit~~ beide durch 100 (wir brauchen sie nicht zu unterscheiden)

(oder umgekehrt erhalten wir ein Code aus einer Code')

dieser $\|code'\|$ wird genau um $H(x)+H(y)$ kürzer

$$\|code\| = \|code'\| + H(x) + H(y)$$

code ist minimal für (Σ, H)

genau wenn code' für (Σ', H') minimal ist

weil $H(x) + H(y)$ nicht vom Code abhängt.

laut Induktionsannahme ist es optimal

für (Σ, H)



mit

entspricht

für (Σ', H') Huffman-Code zu erzeugen

Huffman weiterzumachen

6.11. $\text{code}()$ ist die Weiter-Berechnung von $\text{code}()$ auf $\Sigma - \{x, y\} + \{\overline{xy}\}$

$\|\text{code}\|$ ist also minimal, wenn über $(\Sigma' + \{ \overline{xy} \})$ $\text{code}()$

mit dem Huffman-Algorithmus weiter berechnet wird

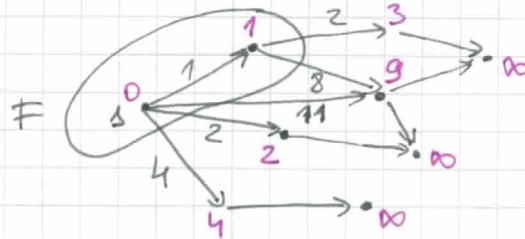
□

3. Das SSSP-Problem (Single-Source-Shortest-Path)

Eingabe: ein gerichteter Graph $G(V, \vec{E})$ mit einer Quelle $s \in V$ und Kantengewichtung

Länge: $E \rightarrow \mathbb{R}_{0,+}$

Ausgabe: bestimme den kürzesten $s-v$ Weg für jeden Knoten $v \in V$ (bzw. seine Länge)



Dijkstra's Algorithmus:

($F \subseteq V$ ist die Knotenmenge mit endgültigen Distanzwerten)
 F wächst in jeder Runde um 1 Knoten

- setze $F = \{s\}$ und $D_F(v) = \begin{cases} 0 & \text{falls } v=s \\ \text{Länge}(s,v) & \text{falls } (s,v) \in E \\ \infty & \text{sonst.} \end{cases}$

- WHILE $F \neq V$

- wähle $v \notin F$ mit minimalem $D_F(v)$
und setze $F = F \cup \{v\}$

- FOR $w \notin F$ so dass $(v,w) \in \vec{E}$

setze $D_F(w) = \min\{D_F(w), D_F(v) + \text{Länge}(v,w)\}$

Beachte: $D_F(v)$ ist die Länge des aktuell kürzesten Weges von s nach v nur über Knoten aus F .

- Die Datenelemente sind hier die Knoten v mit den $D_{\neq}(v)$ Werten als Prioritäten. Beachte, dass die $D_{\neq}(v)$ Werte sich ändern im Laufe des Algorithmus und streng genommen, sogar ihre Definition ändert sich, weil F wächst. (adaptiver Priority Algorithmus)
- Deshalb werden die Datenelemente, wie beim Huffman-Algorithmus, in einem Heap (oder in einem sog. Fibonacci-Heap) verwaltet.

Laufzeit: $O(|E| \cdot \log|V|)$ (bzw. $O(|E| + |V| \cdot \log|V|)$)

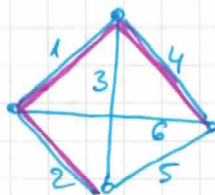
4. Minimaler Spannbaum

Eingabe: ein Graph $G(V, E)$ und Kantengewichtung w_e für jede Kante $e \in E$

Ausgabe: ein Spannbaum von G mit minimalem Gesamtgewicht seiner Kanten

Definition: Ein Spannbaum von $G(V, E)$ ist ein Baum $T(V, E')$ mit der selben Knotenmenge V und $E' \subseteq E$. (Ein Baum ist ein kreisfreier zusammenhängender Graph.)
(Wie viele Kanten enthält E' ?)

minimaler Spannbaum:



Kruskal's Algorithmus (Greedy)

- sortiere die Kanten nach aufsteigendem Gewicht
- setze $E' = \emptyset$
- WHILE $E \neq \emptyset$ DO
 - wähle $e \in E$ mit minimalem w_e
 - $E := E \setminus \{e\}$
 - falls $E' \cup \{e\}$ keinen Kreis enthält
 - $E' := E' \cup \{e\}$
 - sonst verwerfe e

Theorem: Kruskal's Algorithmus findet einen minimalen Spannbaum.

Beweis:

Sei $|V| = n$. Jeder Baum auf n Knoten hat $n-1$ Kanten.

- Sei $T(V, E')$ der Spannbaum ausgegeben von Kruskal's Algorithmus, und $T^*(V, E^*)$ ein minimaler Spannbaum mit maximaler Anzahl gemeinsamer Kanten mit T ($|E' \cap E^*|$ sei größtmöglich). Wenn $E' = E^*$, dann ist $T = T^*$ somit ist T minimal und wir sind fertig.
- Wenn $E' \neq E^*$, dann gibt es mindestens eine Kante in E' die in E^* nicht drin ist (da beide $n-1$ Kanten haben)

c.) Heuristiken für das metrische TSP

(Problem des Handlungsreisenden)

TRAVELING-SALESMAN PROBLEM (TSP)

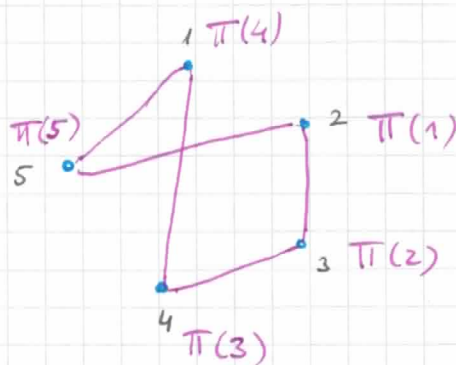
Eingabe: n Orte $\{1, 2, 3, \dots, n\}$ und ein Distanzwert $d(i, j) \geq 0$ zwischen je zwei Orten (so dass $d(i, i) = 0$ und $d(i, j) = d(j, i) \forall i, j$)

Ausgabe: Eine Rundreise minimaler Länge, also eine Permutation $\pi: \{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$ der Orte so dass

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

minimiert wird

Ein Handlungsreisender soll alle Orte aufsuchen, und schließlich nach Hause (zum Startpunkt) zurückkehren, so dass die Länge der gesamten Reise minimiert wird. ($d(i, j)$ kann auch als Reisekosten von i nach j aufgefasst werden)

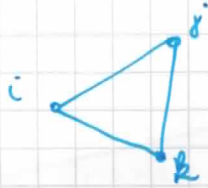


Die Rundreise ist also ~~unabhängig bestimmt~~ durch die ~~Permutation~~ $(\pi(1), \pi(2), \pi(3), \dots, \pi(n))$ und wieder $\pi(1)$ (wobei jeder Ort genau einmal vorkommt (~~...~~)) weil π eine Permutation ist).

G 20.

2. das metrische TSP (Δ -TSP)

In dieser Einschränkung des TSP wird gefordert, dass die Distanzwerte $d()$ die Dreiecksungleichung erfüllen:



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

Def: Eine Distanzfunktion $d()$ über alle ~~Paare~~^{Element}-Paare einer ~~Menge~~ Menge heisst eine Metrik, falls

1. $d(i,j) = 0 \iff i=j$

2. $d(i,j) = d(j,i)$

Symmetrie

3. $d(i,k) \leq d(i,j) + d(j,k)$ Dreiecksungleichung
o. Transitivität

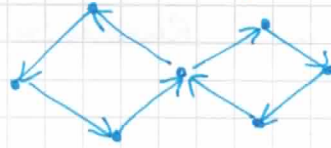
(Metrische Distanzen nicht mit den Euklidischen Distanzen verwechseln! Euklidische Distanzen entsprechen einer Metrik, aber es gibt viele andere Metriken, z.B. Hamming-Distanz, oder $d(i,j) = 1 \iff i \neq j$ sind auch Metriken.)

Das metrische TSP kann man um konstanten Faktor effizient approximieren.

Heuristiken

Die ersten beiden Heuristiken suchen eine sog. Euler-Tour über alle Orte. Hierfür brauchen sie als Erstes einen Eulerschen Graphen über alle Orte (zusammenhängender Graph mit geradem Grad für alle Knoten)
Die Euler-Tour kann man anschließend kürzen, um jeden

Ort nur einmal zu besuchen. Dank der Dreiecksungleichung, wird jede solche Abkürzung die Länge der Tour tatsächlich kürzen (zumindest nicht verlängern)
(Achtung! Nur kurze Euler-Touren mit möglichst wenigen Kanten sind brauchbar)



Zur Erinnerung:

Eine Euler-Tour in einem Graphen ist eine Tour über adjazente Kanten, (Folge von jeweils benachbarten Knoten, die im Startknoten endet), die jede Kante genau einmal durchläuft, (die Knoten dürfen mehrmals besucht werden).

Theorem (Euler): Ein zusammenhängender, ungerichteter Graph $G(V, E)$ enthält eine Euler-Tour



alle Knoten haben geraden Grad

Beweis:

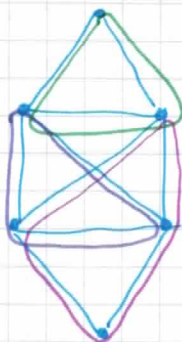
⇓ diese Richtung ist trivial, wenn die Tour in einen Knoten reinläuft, muss über eine neue Kante wieder rauslaufen



Wir laufen von Kante zur Kante, solange es geht. Jedes mal, wenn es nicht weitergeht, haben wir einen Kreis (von Kanten) geschlossen. Da der Graph zusammenhängend ist, können all diese 'Kreise' (Touren) letztendlich in eine einzige Tour geschmelzt werden.

(wenn unser Kreis noch nicht jede

Kante durchläuft, kann er verlängert werden (iterativ) um einen weiteren adjazenten Kreis)



G22. Betrachte die Orte im TSP Problem als eine Menge von Knoten.

Die nächsten beiden Heuristiken erstellen einen zusammenhängenden Graphen mit geradem Knotengrad für jeden Knoten und berechnen eine Euler-Tour mit Abkürzungen auf diesem Graphen, als Rundreise über die Orte.

i.) Die Spannb Baum-Heuristik

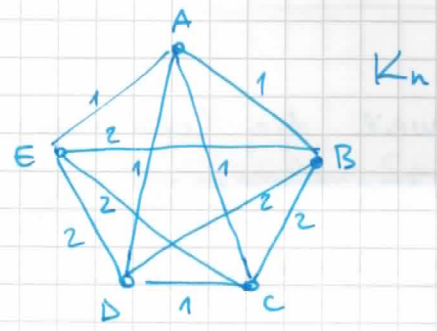
- Betrachte den vollständigen Graphen über alle Orte als Knoten, und mit den Distanzen $d(i, j)$ als Kantengewichten / Längen K_n
- Bestimme einen minimalen Spannb Baum in diesem vollständigen Graphen mit Kruskal's Algorithmus T

(Der Spannb Baum spannt zwar alle Orte, es gibt noch aber Knoten mit ungeradem Grad ...)

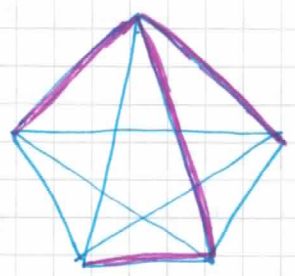
- verdoppele jede Kante im Spannb Baum \hat{T}
(jetzt haben wir einen Spanngrafen mit geradem Knotengraden)
- nimm eine Euler-Tour in \hat{T} ; z.B.
ein Präorder Durchlauf (Tiefensuche) und dann der Rückkehr zur Wurzel entspricht einer Euler-Tour.
- Mache Abkürzungen wo nötig, um jeden Ort nur einmal zu besuchen (beim ersten Mal).

(Achtung: wenn der kürzeste Weg von j nach k entlang Ort i führt, und i schon besucht wurde früher, darf die Rundreise natürlich an i vorbeiführen. z.B. in der Eukl. Ebene, oder in einem Straßennetzwerk / Graphen.)

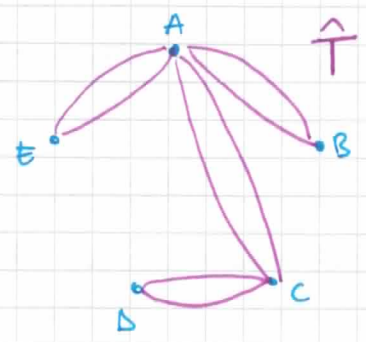
Beispiel:



die Orte mit ihren Distanzwerten

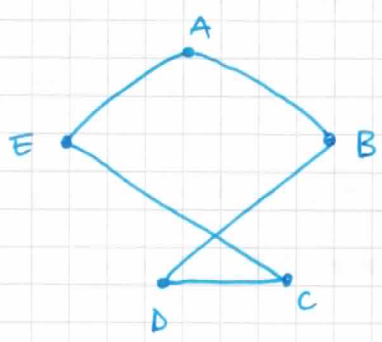


ein minimaler Spannbaum
T



verdoppelte Kanten und eine Euler-Tour:

A E A C D C A B A



die Rundreise =
Euler-Tour mit
Abkürzungen

A E C D B(A)

Theorem: Die Spannbaum-Heuristik ist 2-approximativ.

Wann? Sei l_{SH} die Länge der Rundreise ausgegeben von der Spannbaum-Heuristik, und l_{opt} die optimale Länge. Sei noch $l(T)$ die Gesamtlänge der Kanten im Spannbaum.



Es gelten:

$l_{\text{opt}} > l(T)$ weil die (optimale) Rundreise mit
sogar minus eine Kante auch ein Spannbaum ist,
und T ein minimaler Spannbaum ist.

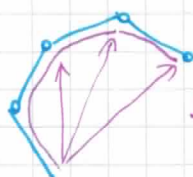
$l_{\text{ST}} \leq 2 \cdot l(T)$ weil die Länge der Euler-Tour
genau $2 \cdot l(T)$ ist, und l_{ST} hat noch Abkürzungen im
Vgl. zur Euler-Tour.

(Beachte, dass ohne der Dreiecksungleichung eine „Abkürzung“
zu einer Verlängerung führen könnte



(ohne Dreiecksungleichung wird „Vorbeifahren“ auch
nicht definiert).

Mit Dreiecksungleichung kann aber eine Abkürzung
nicht länger werden als vorher ohne Abkürzung:



man kann die
Dreiecksungleichung
iterativ, oder durch Induktion
anwenden

Wir erhalten: $l_{\text{ST}} \leq 2 \cdot l(T) < 2 \cdot l_{\text{opt}}$ also die
Spannbaum-Hennstrikte ist 2-approximativ.

Der folgende Algorithmus definiert den Eulerschen
Graphen (d.h. mit den geraden Knotengraden)

raffiniert, und erreicht einen noch besseren

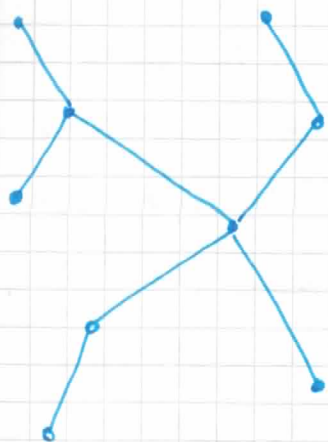
Approximationsfaktor:

ii.) der Algorithmus von Christofides für Δ -TSP

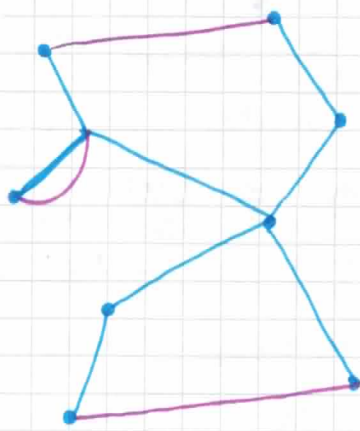
- bestimme einen minimalen Spannbaum T im vollständigen Graphen K_n mit Kantenlängen $d(i, j)$

(es gibt noch Knoten in T mit ungeradem Grad)

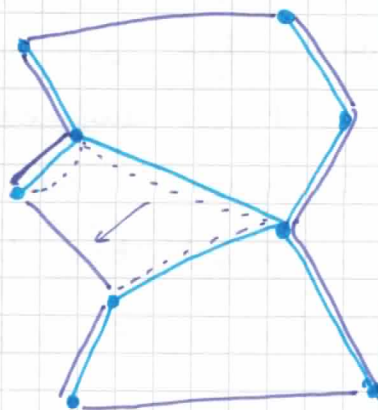
- Bezeichne U die Menge der Knoten mit ungeradem Grad in T ($|U|$ ist gerade, weil die Summe aller Knotengrade immer gerade ist $2 \cdot |E| = 2 \cdot (n-1) \Rightarrow$ gerade)
- Berechne ein perfektes Matching mit minimalem Gesamtlänge seiner Kanten über die Knoten von U . Sei M dieses Matching.
- Nimm die Kanten in M zu den Baumkanten in T hinzu: jetzt hat jeder Ort geraden Grad. (Weil die Knoten in U jetzt $+1$ Grad haben von M .)
- bestimme eine Euler-Tour in $T \cup M$, und dann mache Abkürzungen um jeden Ort genau einmal zu besuchen.



T



T ∪ M



Rundreise

G26.

Theorem: Der Algorithmus von Christofides ist $\frac{3}{2}$ -approximativ.

Beweis: Sei l_{CH} die Länge der Rundreise ausgegeben von Christofides

Wir wissen:

$$l(T) < l_{OPT}$$

$$\text{und } l_{CH} \leq l(T) + l(M)$$

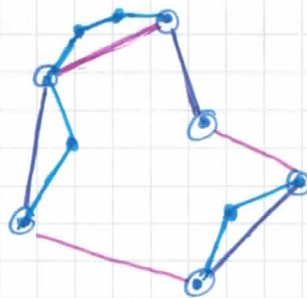
(wobei $l(M)$ ist die Gesamtlänge des optimalen Matchings über die Knotenmenge U)

Wir zeigen noch, dass $l(M) \leq \frac{l_{OPT}}{2}$

Sei l_{OPT}^U die Länge einer optimalen Rundreise über die Menge $U \subseteq \{1, 2, \dots, n\}$ (der Knoten mit ungeradem Grad in T)

Dann ist $l(M) \leq \frac{l_{OPT}^U}{2}$

- nicht U
- ⊙ U



weil die optimale Rundreise über U in zwei Matchings aufgeteilt werden kann, und mindestens eins davon hat Länge $\leq \frac{l_{OPT}^U}{2}$, also ein optimales Matching über U ist auch nicht länger als $\frac{l_{OPT}^U}{2}$.

Weiterhin gilt $l_{OPT}^U \leq l_{OPT}$ weil eine optimale (ganze) Rundreise, auf die Knoten von U gekürzt werden kann. Somit gilt $l(M) \leq \frac{l_{OPT}^U}{2} \leq \frac{l_{OPT}}{2}$.

Schließlich: $l_{CH} \leq l(T) + l(M) \leq l_{opt} + \frac{l_{opt}}{2} = \frac{3}{2} l_{opt}$

□

Laufzeit: $O(n^3)$ (Die Berechnung eines perfekten Matchings mit minimalem Gewicht braucht $\Theta(|U|^3) = O(n^3)$ Zeit und dies dominiert die Laufzeit; die anderen Schritte sind linear.) $\rightarrow O(n)$

[Bemerkung: es wurde gezeigt (Zoll) dass mit einer geschickter Wahl des Spannbaums ein kleinerer Approximationsfaktor erreichbar ist, wenn die Metrik ~~dunkel~~ Distanzen in einem Graphen definiert wird.]

(greedy)
Die folgenden beiden Heuristiken haben schlechteren Approximationsfaktor im Worst-Case als Christofides.

iii.) Nearest-Insertion Heuristik 2-approximativ

- sei $V = \{1, 2, \dots, n\}$

- sei $\{i, j\}$ ein Paar von Orten mit ~~minimaler~~ ^{maximaler} Distanz $d(i, j)$

- setze $V := V \setminus \{i, j\}$

- sei die (partielle) Rundreise $R = (i, j, i)$

- REPEAT

- nimm einen ~~Ort~~ ^{Ort $k \in V$ weitestem} mit ~~kleinstem~~ ^{kleinstem} Abstand zu einem

Ort in der aktuellen partiellen Rundreise R

sei $V := V \setminus \{k\}$

- füge k zwischen zwei benachbarten Orten in der partiellen Rundreise ein, so dass dies den kleinsten Anstieg der Länge der Rundreise resultiert

UNTIL $V = \emptyset$

iv.) Farthest-Insertion Heuristik

$6.5 \leq \text{Approx-Faktor} \leq \log n + 1$