Heuristiken zur Ein-Depot-Tourenplanung

Barbara König

Technische Universität München
Institut für Informatik

Diplomarbeit

Heuristiken zur Ein-Depot-Tourenplanung

Barbara König

Aufgabensteller/Betreuer:
Prof. Dr. E. W. Mayr

Abgabedatum: 15. August 1995

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Zusammenfassung

Tourenplanung (Vehicle Routing Problem) ist eine Verallgemeinerung des Traveling Salesman Problem. Den Knoten eines Graphen sind Transfermengen zugeordnet, welche von kapazitäts- und zeitbeschränkten Fahrzeugen ausgehend von einem Depot abgeholt werden müssen, so daß die Gesamtkosten minimal sind.

Das Entscheidungsproblem der Tourenplanung ist NP-vollständig, so daß man nicht mit einem polynomiellen Algorithmus für das Tourenplanungs-Problem rechnen kann und daher auf Heuristiken zurückgreifen muß.

Als Heuristiken werden der Sweep, verschiedene Savings-Verfahren, Cluster-Verfahren, der Giant-Tour-Algorithmus und ein 3-opt-Verbesserungsverfahren vorgestellt.

Die Heuristiken wurden auf verschiedenen Graphen getestet und in ihren Ergebnissen und ihrem Zeitverhalten miteinander verglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Praktische Bedeutung der Tourenplanung

Das Besuchen einer Menge von geographischen Punkten, sei es zum Verteilen bzw. Einsammeln von Gütern, sei es zur Erbringung von Dienstleistungen, ist eine beliebte (und natürlich auch notwendige) Beschäftigung der Menschheit. Daraus ergibt sich der Wunsch, die zu besuchenden Orte in eine optimale Reihenfolge zu bringen, so daß die zurückzulegende Strecke minimal wird.

Anwendungsmöglichkeiten für Tourenplanung gibt es bei Speditionen, überhaupt allen Arten von Auslieferung, bei der Entsorgung und in vielen anderen Bereichen.

Bis vor kurzem wurde die Tourenplanung überwiegend von Disponenten mit hauptsächlich manuellen Methoden durchgeführt. Diese manuelle Arbeit basiert auf Erfahrungen im speziellen Aufgabengebiet und kann durch entsprechende Größe des Tourenplanungs-Problems und durch ganz besondere Einschränkungen und Bedingungen sehr langwierig und ihr Ergebnis sehr unbefriedigend werden.

Die Lösung liegt auf der Hand: Übergang von manueller zu EDV-gestützter Tourenplanung. Indem man die Planungsarbeit automatisiert, ergeben sich Einsparungsmöglichkeiten in der zu fahrenden Strecke und in den entstehenden Kosten. Außerdem werden größere Kundenmenge (Menge von Anfahrpunkten) handhabbar und der Zeitbedarf für die Planung kann verringert werden.

Doch auch die EDV-gestützte Tourenplanung hat ihre Probleme. Wie später gezeigt werden wird, ist es nicht möglich, die gestellten Probleme in annehmbarer Zeit optimal zu lösen, so daß man sich mit heuristischen Algorithmen und suboptimalen Lösungen behelfen muß. Des weiteren stellen die oft sehr speziellen Anforderungen und Einschränkungen, die es in den verschiedenen Branchen gibt, hohe Anforderungen an die Algorithmen, wenn man gleichzeitig die Güte der Lösung im Auge haben muß.

Ich werde in meiner Diplomarbeit verschieden Heuristiken für ein bestimmtes Tourenplanungs-Problem vorstellen und bewerten.

1.2 GIS-gestützte Tourenplanung

Die Algorithmen sind auf der Basis des Geographischen Informationssystems (GIS) ARC/INFO implementiert. Ein GIS ist eine Software, die dazu dient, Daten mit Raumbezug zu erfassen, zu verwalten und zu analysieren.

ARC/INFO besitzt ein Modul, genannt NETWORK, mit dem lineare geographische Objekte behandelt werden können, also auch Straßennetze. Mit Hilfe des GIS ist man in der Lage, reale Straßennetze zu digitalisieren und an Daten bereits digital erfaßter Netze zu gelangen, die zum Teil schon im korrekten Format und mit zahlreichen Straßenattributen vorliegen. Des weiteren hat man mit einem GIS bereits hervorragende Möglichkeiten der Visualisierung der Netze, der Anfahrpunkte und der errechneten Touren und kann sich auf eine vorhandene Grundfunktionalität zur Analyse der Daten stützen.

Auf der schon vorhandenen Funktionalität wurde unter Anpassung der bestehenden Benutzeroberfläche Arctools eine Applikation zur Tourenplanung aufgebaut, an der ich im Rahmen meiner Diplomarbeit mitgearbeitet habe.

Diese Applikation wurde bisher hauptsächlich zur Organisation der Milchabholung von Bauernhöfen eingesetzt.

Kapitel 2

Formalisierung der Problemstellung

2.1 Traveling Salesman Problem (TSP)

Das Traveling Salesman Problem (Problem des Handlungsreisenden) hat seinen Namen von der ursprünglichen Aufgabenstellung: Ein Handlungsreisender muß n Städte, deren Entfernungen untereinander bekannt sind, besuchen und anschließend wieder zum Ausgangsort zurückkehren. Aus verständlichen Gründen will er möglichst wenig Arbeitszeit investieren und beabsichtigt daher, diese Städte in der bestmöglichen Reihenfolge anzufahren.

Wer kann ihm dabei helfen?

2.1.1 Definition

Voraussetzung 2.1 *Sei $G = (V, E)$ ein zusammenhängender gerichteter Graph mit Knotenmenge V, Kantenmenge E und einer Kantenbewertung $c : E \to \mathbb{R}_0^+$ (Längen- bzw. Kostenfunktion).*

Sei $n := |V|$ und $m := |E|$.

Der Abstand zweier Knoten $v_1, v_2 \in V$ ist die Länge des kürzesten Weges zwischen v_1 und v_2 auf dem Graphen bezüglich der Kantenbewertung c. Diese Entfernung wird der Einfachheit halber auch mit $c(v_1, v_2)$ bezeichnet.

Die Stop-Menge $S = \{s_1, \ldots, s_k\}$ sei eine Teilmenge der Knotenmenge V.

Die Begriffe Länge und Kosten eines Weges werden im folgenden synonym verwendet.

Definition 2.1 (Tour) *Gegeben sei Voraussetzung 2.1. Eine Tour π durch alle Stops eines Graphen G ist eine Anordnung der Stops, d.h. eine Bijektion $\pi : \{1, \ldots, k\} \to S$.*

Die Kosten einer Tour π seien definiert als:

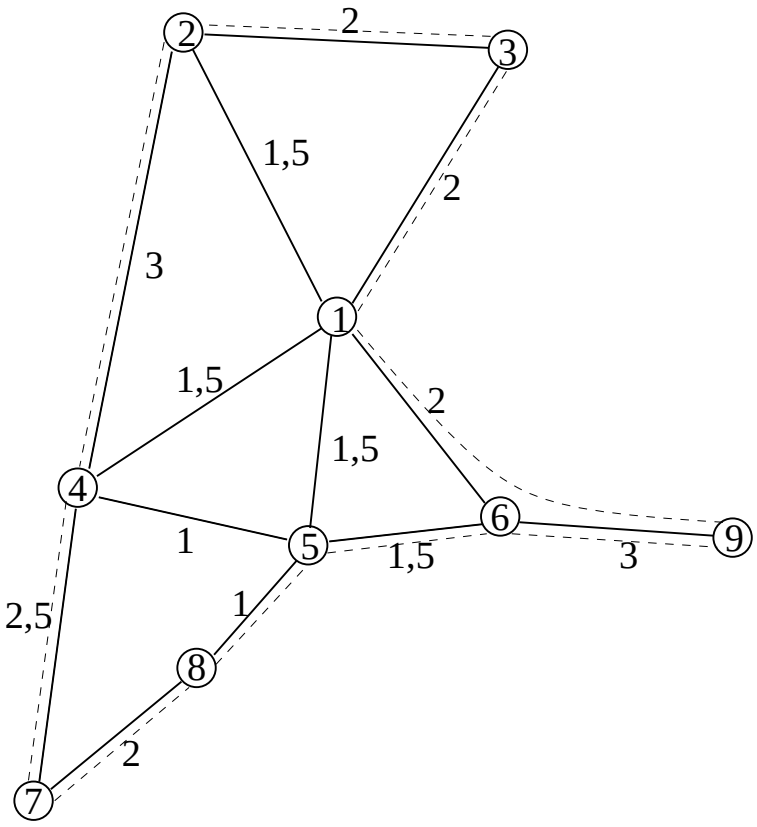


Abbildung 2.1: Beispiel für einen Graphen mit Tour der Länge 22

$$C(\pi) := \sum_{j=1}^{k} c(\pi(j), \pi(j+1))$$

Dabei ist $\pi(k+1) := \pi(1)$.

In Abbildung 2.1 ist ein Beispiel für einen symmetrischen Graphen mit Kostenfunktion (wie in Voraussetzung 2.1) und eine Tour durch diesen Graphen abgebildet. (Alle Kanten des Graphen können in beiden Richtungen benutzt werden.)

Auf nicht-vollständigen Graphen, wie diesem, sind auch Touren denkbar, die einen Knoten (hier: Knoten 6) mehrfach durchlaufen.

Ich werde diesen Beispiel-Graphen im folgenden verwenden, um das Verhalten der Algorithmen zu veranschaulichen.

Problem 2.1 (TSP) *Gesucht wird die kostengünstigste Tour* π *durch alle Stops, d.h. eine Tour* π*, so daß für alle Touren* λ *gilt:*

$$C(\pi) \leq C(\lambda)$$

Indem man den Abstand zwischen allen Paaren von Stops bestimmt (z.B. mit dem Algorithmus von Dijkstra, siehe Abschnitt 4), kann man das TSP in Polynomzeit auf den Fall reduzieren, daß $S = V$ und G vollständig ist.

Denn ist dies nicht der Fall, so löst man das Problem auf dem vollständigen Graphen $G' = (S, S \times S)$ mit der Kostenfunktion c', für die gilt: $c'(s,t) = c(s,t), \forall s,t \in S$. Die Aufgabe besteht in diesem Fall darin, einen minimalen Hamiltonschen Kreis zu finden. (Ein Hamiltonscher Kreis ist eine Tour durch den Graphen, die jeden Knoten genau einmal besucht.)

2.1.2 TSP als ganzzahliges Programm

Das TSP kann auch als ganzzahliges Programm, wie in der Optimierungstheorie formuliert werden. Diese Formulierung dient vor allem als Grundlage exakter Lösungsverfahren (siehe [6]).

Voraussetzung 2.2 *Es liegt die Annahme zugrunde, daß der Graph vollständig und die Stop-Menge gleich der Knotenmenge ist.*
$V = S := \{1, \ldots, k\}$.

Problem 2.2 *Das Problem besteht darin, eine Matrix $X = (x_{ij})_{1 \leq i,j \leq k}$ zu belegen, wobei x_{ij} folgende Bedeutung hat:*

$$x_{ij} = \begin{cases} 1 & \textit{falls die Kante } (i,j) \textit{ zur Tour gehört} \\ 0 & \textit{sonst} \end{cases}$$

Minimiere dabei die Summe der Kosten aller benutzten Kanten, d.h.

$$\sum_{i=1}^{k} \sum_{j=1}^{k} c(i,j) \cdot x_{ij} \tag{2.1}$$

unter Einhaltung der Restriktionen

$$\begin{aligned} x_{ij} &\in \{0,1\},\ i,j \in \{1,\ldots,k\} && (2.2)\\ \sum_{i=1}^{k} x_{ij} &= 1,\ j \in \{1,\ldots,k\} && (2.3)\\ \sum_{j=1}^{k} x_{ij} &= 1,\ i \in \{1,\ldots,k\} && (2.4)\\ X = (x_{ij}) &\in B && (2.5) \end{aligned}$$

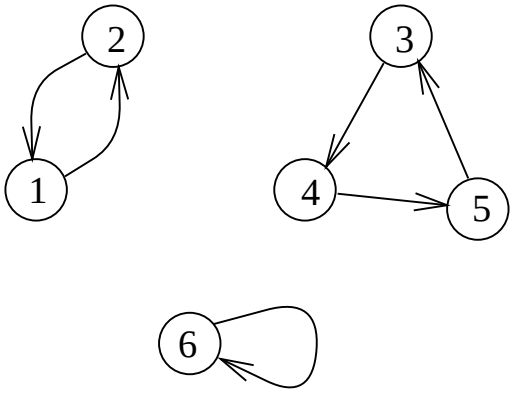


Abbildung 2.2: Beispiel für drei Subtouren

Dabei drücken (2.3) bzw. (2.4) aus, daß jeder Stop genau einmal betreten bzw. verlassen werden muß.

Die Bedingung (2.5) sorgt—bei passender Wahl von B—dafür, daß keine Subtouren gebildet werden, die zum Ausgangspunkt zurückkehren, bevor alle Knoten besucht worden sind. So erfüllen z.B. die Subtouren in Abbildung 2.2 die Bedingungen (2.2) bis (2.4), sind aber keine erwünschte Lösung. Man bezeichnet die Bedingung B daher als Subtour-breaking constraint.

Eine mögliche Definition von B ist beispielsweise:

$$B = \{(x_{ij}) : \sum_{i \in Q} \sum_{j \notin Q} x_{ij} \geq 1, \text{ für alle Mengen } Q \text{ mit : } Q \subset V, Q \neq V, Q \neq \emptyset\}$$

Das bedeutet nichts anderes, als daß bei einer Zerlegung der Knotenmenge in zwei nichtleere Teilmenge mindestens eine Verbindung zwischen beiden Mengen existieren muß.

B verhindert Subtouren, denn existiert eine Subtour, die nicht alle Knoten umfaßt, so belege man Q mit der Menge aller Knoten dieser Subtour, und man sieht sofort, daß diese Lösung nicht in B enthalten sein kann.

Eine Rundtour, die alle Knoten umfaßt, erfüllt umgekehrt auch obige Bedingungen. Denn da sowohl Knoten in Q als auch Knoten in $V \backslash Q$ aufgesucht werden und beide Mengen nichtleer sind, muß es eine Kante auf der Tour geben, deren Anfangsknoten in Q und deren Endknoten in $V \backslash Q$ liegt.

Da Q ein beliebiges Element aus der Potenzmenge von V (außer V selbst und der leeren Menge) sein kann, ergeben sich $2^k - 2$, also exponentiell viele Bedingungen.

Mit weniger Bedingungen, nämlich $(k-1) \cdot (k-2)$, kommt folgende Definition von B aus:

$$B = \{(x_{ij}) : \exists y_i \in \mathbb{R} : y_i - y_j + k \cdot x_{ij} \leq k - 1, \ 2 \leq i \neq j \leq k\}$$

Falls die Lösung des ganzzahligen Programms Subtouren enthält, die nicht alle Knoten des Graphen umfassen, so gibt es eine Subtour, die die 1 nicht beinhaltet. Die Knotenmenge dieser Subtour heiße Q, $Q \neq V$.

Seien die y_i beliebige reelle Zahlen.

Dann gilt:

$$\begin{aligned} \sum_{i,j \in Q, x_{ij}=1} y_i - y_j + k \cdot x_{ij} &= \sum_{i \in Q} y_i - \sum_{j \in Q} y_j + k \cdot |Q| = k \cdot |Q| \\ \not\leq \quad |Q| \cdot (k-1) &= \sum_{i,j \in Q, x_{ij}=1} (k-1) \end{aligned}$$

Das heißt, man kann keine reellen Zahlen y_i finden, so daß eine Lösung des ganzzahligen Programms, die Subtouren enthält, auch in B liegt.

Umgekehrt gibt es aber auch zu jeder Rundtour immer eine Belegung für die y_i. Numeriert man beispielsweise jeden Knoten i in der Tour in der Reihenfolge, in der er besucht wird, beginnend mit einem beliebigen Startknoten und weist diese Nummer y_i zu, so erfüllen die y_i die obigen Bedingungen.

2.1.3 Zusammenhang zwischen Subtour-breaking constraints und dem Maximal Flow Problem in Netzwerken

Betrachtet man die Subtour-breaking constraints genauer, so erkennt man in ihnen das Maximal Flow Problem in Netzwerken und das dazu duale Problem (beschrieben beispielsweise in [4, 38]). Als Ausgangssituation hat man eine Matrix $X = (x_{ij})$, die die Bedingungen (2.2)–(2.4) erfüllt, d.h. die eine Tour durch den Graphen beschreibt, die aber u.U. noch in Subtouren zerfallen kann.

Umformulierung des ersten Subtour-breaking constraint:

$$B = \{(x_{ij}) : \sum_{i \in Q} \sum_{i \notin Q} x_{ij} \leq 1, Q \subset V, Q \neq V, Q \neq \emptyset\}$$

Demgegenüber steht das duale MAXFLOW-Problem:

Der Knoten $q \in V$ sei eine feste Quelle, die Senke $s \in V$ sei variabel.

Minimiere

$$\sum_{i=1}^{k} \sum_{j=1}^{k} x_{ij} \cdot h_{ij} \tag{2.6}$$

unter den Restriktionen:

$$\begin{aligned} w_s - w_q &= 1 && (2.7) \\ w_i - w_j + h_{ij} &\leq 0, \quad i,j \in \{1,\ldots,k\} && (2.8) \\ h_{ij} &\leq 0, \quad i,j \in \{1,\ldots,k\} && (2.9) \end{aligned}$$

Dabei sind die x_{ij} die oberen Schranken für die Kosten, die auf den Kanten des Graphen transportiert werden können. Gehört eine Kante zur Tour, so dürfen dort höchstens die Kosten 1 "entlangfließen", andernfalls ist kein Fluß zulässig.

Die $w_i \in \{0, 1\}$ definieren einen Schnitt durch die Menge der Knoten, d.h. eine Zerlegung in zwei Partitionen. Die h_{ij} legen dann die Kanten fest, die über diesen Schnitt hinweg Kosten transportieren. Der zu minimierende Wert

$$\sum_{i=1}^{k} \sum_{j=1}^{k} x_{ij} \cdot h_{ij}$$

heißt Kapazität des durch die w_i festgelegten Schnitts.

Wenn man für w_i und h_{ij} folgende Festlegungen trifft, so sieht man die Analogie zur Menge B:

$$\begin{aligned} w_i &= \begin{cases} 0 & \text{falls } i \in Q \\ 1 & \text{falls } i \notin Q \end{cases} \\ h_{ij} &= \begin{cases} 1 & \text{falls } (i,j) \in (Q, V \backslash Q) \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

(2.7) sorgt dafür, daß s und q in verschiedenen Partitionen liegen, (2.8) sichert, daß h_{ij} die intendierte Bedeutung hat (Kanten zwischen Q und $V \backslash Q$).

Falls das gefundene Minimum über alle Senken gleich 0 ist, so gibt es einen Schnitt, der Subtouren voneinander trennt, und damit gilt $X \notin B$. Umgekehrt folgt aus $X \in B$, daß das Minimum mindestens 1 sein muß, da es dann keinen Schnitt geben darf, der verschiedene Subtouren voneinander trennt.

Umformulierung des zweiten Subtour-breaking constraint:

Das eigentliche MAXFLOW-Problem lautet:

Maximiere f unter den Restriktionen:

$$\sum_{j=1}^{k} f_{ij} - \sum_{j=1}^{k} f_{ji} = \begin{cases} f & \text{falls } i = q \\ 0 & \text{falls } i \notin \{q, s\} \\ -f & \text{falls } i = s \end{cases} \tag{2.10}$$

$$f_{ij} \leq x_{ij}, \quad i, j \in \{1, \ldots, k\} \tag{2.11}$$

$$x_{ij} \geq 0, \quad i, j \in \{1, \ldots, k\} \tag{2.12}$$

Es wird versucht, möglichst viele Kosten von der Quelle zur Senke zu transportieren, ohne dabei die oberen Schranken x_{ij} zu verletzen. Der Fluß in einem Graphen ist immer kleiner gleich der Kapazität eines beliebigen Schnittes. Daher ist der maximale Fluß f, der von der Quelle q ausgeht und in die Senke s hineinführt gleich der minimalen Kapazität der Schnitte.

Falls das Maximum von f für alle möglichen Senken gleich 1 ist, so kann man eine Verbindung von q zu jedem anderen Knoten finden.

Sei y_i der kumulierte Fluß, der sich auf dieser Verbindung im Knoten mit der Nummer i ansammelt. Es gilt:

$$y_q = 0, y_s = k-1, y_i \in \{0, \ldots, k-1\}$$

Dann ist dies analog zu:

$$B = \{(x_{ij}) : \exists y_i \in \mathbb{R} : y_i - y_j + k \cdot x_{ij} \leq k-1, 1 \leq i \neq j \leq k, j \neq q\}$$

D.h. in Knoten i und j, die auf dem Weg von der Quelle zur Senke aufeinanderfolgen, muß für die Differenz des kumulierten Flusses $y_i - y_j \leq -1$ gelten.

In die Quelle gibt es wegen der Maximalität keinen Fluß, daher wird $j \neq q$ gefordert. Weil der Verlauf einer Tour mit $k-2$ Kanten bereits feststeht, kann in der Bedingung B auch noch $i \neq q$ gefordert werden. (In der ursprünglichen Bedingung B ist $q = 1$.)

2.2 Vehicle Routing Problem (VRP)

Man kann das Problem des Handlungsreisenden durch zusätzliche Einschränkungen beliebig komplizieren.

Die naheliegendste Beschränkung ist zunächst die der Kapazität: Der Reisende kann nur eine beschränkte Anzahl an Gütern mit sich führen. Gleichzeitig ist der Bedarf an Gütern in den Zielorten bekannt. Jede Tour, die der Handlungsreisende unternimmt, muß an einem zentralen Depot, an dem die Güter gelagert werden, beginnen und enden.

Zu Gunsten des Handlungsreisenden soll auch die Zeit, die er insgesamt unterwegs ist, nach oben beschränkt sein.

Ein solches TSP mit zusätzlichen Einschränkungen bezeichnet man als Vehicle Routing Problem (Tourenplanungs-Problem). Die folgende Formalisierung des VRP enthält die für meine spezielle Problemstellung relevanten Einschränkungen.

2.2.1 Definition

Voraussetzung 2.3 *Der Graph G mit Kostenfunktion c und die Stop-Menge S seien gegeben wie in Voraussetzung 2.1.*

Des weiteren sei $D \in V \backslash S$ das Depot.

Jedem Stop ist eine Transfermenge b zugeordnet:

$$b : S \to \mathbb{R}_0^+$$

Neben der Kostenfunktion c wird eine weitere Kantenbewertung eingeführt, die Zeitfunktion t_E, die die Zeit angibt, die man für das Passieren einer Kante benötigt:

$$t_E : E \to \mathbb{R}_0^+$$

Der Zeitabstand zwischen zwei nicht benachbarten Knoten wird analog dem Kostenabstand zwischen nicht benachbarten Knoten definiert.

Zusätzlich wird jedem Stop eine Haltezeit zugeordnet:

$$t_S : S \to \mathbb{R}_0^+$$

Die Haltezeit am Depot ist proportional abhängig von der zu entladenden Menge:

$$t_D : \mathbb{R}_0^+ \to \mathbb{R}_0^+,\ t_D(x) = \tilde{t_D} \cdot x$$

Um die nötige Transfermenge zu befördern, steht ein Fuhrpark $F = \{f_1, \ldots, f_l\}$ zur Verfügung.

Jedem der Fahrzeuge ist dabei eine Kapazität q, eine Einsatzzeit t_F und eine maximale Anzahl von Einsätzen m zugeordnet:

$$\begin{aligned} q &: F \to \mathbb{R}_0^+ \\ t_F &: F \to \mathbb{R}_0^+ \\ m &: F \to \mathbb{N}_0 \end{aligned}$$

Definition 2.2 (Subtour) *Eine Injektion $\pi : \{1, \ldots, p\} \to S,\ p \leq k$ definiert eine Subtour durch p Stops des Graphen G, beginnend und endend beim Depot D.*

Zu einer Tour gehören ihre Kosten C, ihre Transfermenge Q und ihr Zeitbedarf T (Summe aus Fahrzeit und Standzeit):

$$\begin{aligned} C(\pi) &:= c(D, \pi(1)) + \sum_{i=1}^{p-1} c(\pi(i), \pi(i+1)) + c(\pi(p), D) \\ Q(\pi) &:= \sum_{i=1}^{p} b(\pi(i)) \\ T(\pi) &:= t_E(D, \pi(1)) + \sum_{i=1}^{p-1} t_E(\pi(i), \pi(i+1)) + t_E(\pi(p), D) \\ &+ \sum_{i=1}^{p} t_S(\pi(i)) + t_D(Q(\pi)) \end{aligned}$$

Problem 2.3 (VRP ohne Zeitschranken) *Gesucht wird die Gesamtanzahl r der optimalen Touren, dazu die Touren*

$$\pi_i : \{1, \ldots, p_i\} \to S,\ i \in \{1, \ldots, r\}$$

Jeder Stop gehört zu genau einer Tour, d.h. S ist die disjunkte Vereinigung aller Mengen $\pi_i(\{1, \ldots, p_i\}),\ i \in \{1, \ldots, r\}$.

Die Touren müssen mit dem Fuhrpark vereinbar sein, d.h. es muß eine Zuordnung f der Touren zu den Fahrzeugen existieren

$$f : \{1, \ldots, r\} \to F$$

die die Kapazitätsbeschränkungen und die Zahl der maximalen Einsätze einhält, d.h.

$$\begin{aligned} Q(\pi_i) &\leq q(f(i)),\ i \in \{1, \ldots, r\} \\ |\{j : (1 \leq j \leq r) \wedge (f(j) = f_i)\}| &\leq m(f_i),\ i \in \{1, \ldots, l\} \end{aligned}$$

Um die Optimalität der Touren zu gewährleisten, muß für jede Anzahl r' von Touren und für alle dazugehörigen Touren $\lambda_1, \ldots, \lambda_{r'}$, die mit dem Fuhrpark vereinbar sind, gelten:

$$\sum_{i=1}^{r} C(\pi_i) \leq \sum_{i=1}^{r'} C(\lambda_i)$$

Problem 2.4 (VRP mit Zeitschranken) *Die Problemstellung ist exakt wie bei Problem 2.3, mit dem einzigen Unterschied, daß die Vereinbarkeit mit dem Fuhrpark noch von der Einhaltung der Zeitschranken für jedes Fahrzeug abhängt, also*

$$\sum_{f(j)=f_i} T(\pi_j) \leq t_F(f_i),\ i \in \{1, \ldots, l\}$$

Das bedeutet, daß jedes Fahrzeug mehrmals fahren darf, jedoch nur innerhalb der maximal erlaubten Einsätze und der Zeitschranke, die für die Summe der Zeiten aller Einsätze gilt.

Die obigen Voraussetzungen garantieren im Gegensatz zum TSP nicht unbedingt die Existenz einer Lösung. Es mag sein, daß die Anzahl der Fahrzeuge nicht ausreicht, um die Gesamttransfermenge der Stops aufzunehmen. Des weiteren wurde nicht festgelegt, daß die größte Transfermenge eines Stops kleiner sein muß als die maximale Kapazität der Fahrzeuge.

Die Überprüfung, ob überhaupt eine Lösung existiert, ist u.U. so schwer wie die Suche nach einer optimalen Lösung. Das ist z.B. dann der Fall, wenn überhaupt nur die opt. Lösung die Einschränkungen erfüllt.

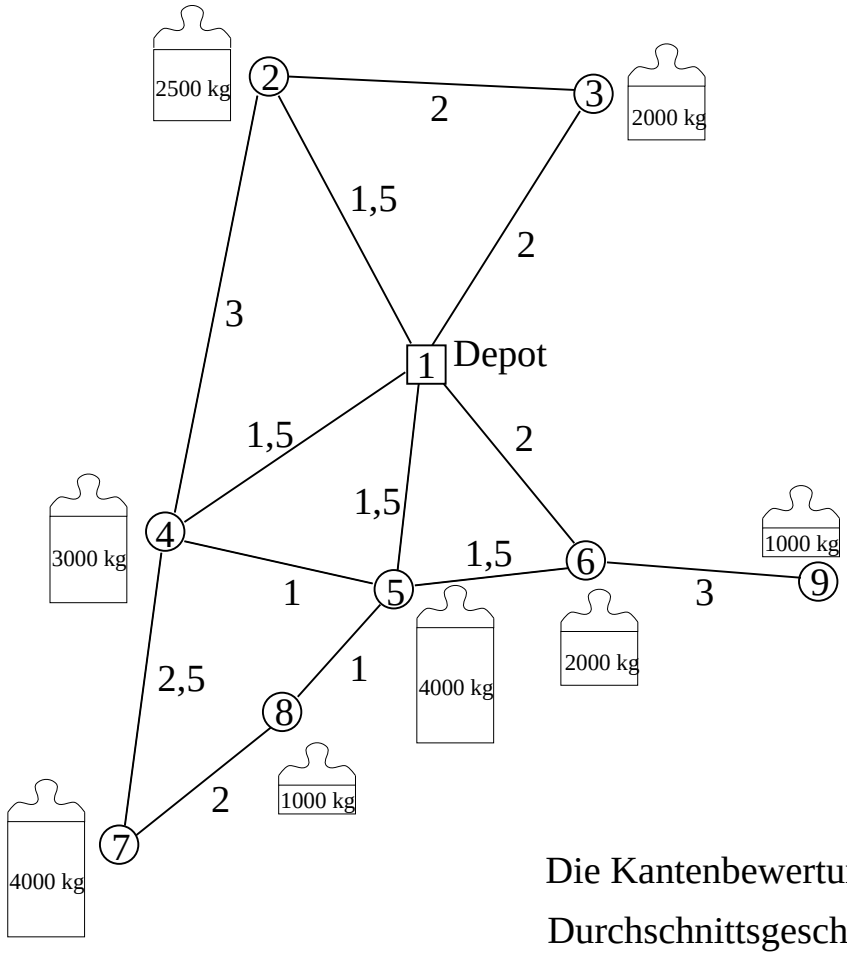
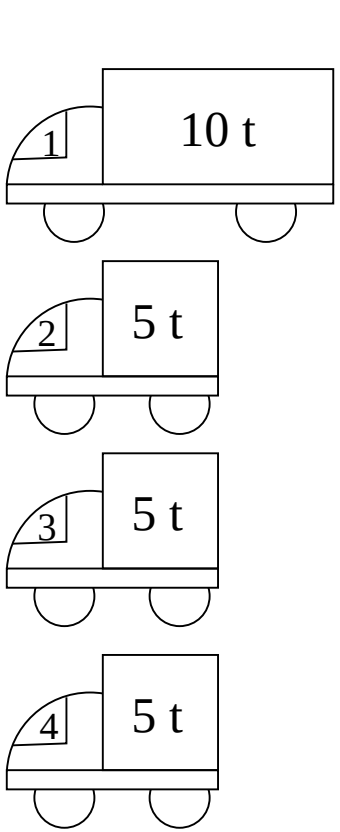


Abbildung 2.3: Beispiel für ein Vehicle Routing Problem



Die Kantenbewertungen sind in km

Durchschnittsgeschwindigkeit auf allen Strassen: 60 km/h

Standzeit pro Stop: 5 min

Weitere Probleme können sich beim VRP mit Zeitschranken ergeben, wenn es keine Lösung geben kann, die in der Lage ist, die Zeitbeschränkung einzuhalten.

Abbildung 2.3 zeigt wieder den Graphen aus Abbildung 2.1 mit graphischer Darstellung der Transfermengen und des Fahrzeugparks als Beispiel für ein Vehicle Routing Problem. Bereits bei diesem relativ kleinen Graphen ist sehr schwer, die optimale Lösung mit "bloßem Auge" zu sehen.

2.2.2 VRP als ganzzahliges Programm

Die folgende Formalisierung ist stark angelehnt an [6].

Voraussetzung 2.4 *Im folgenden gilt die Voraussetzung daß der Graph vollständig und die Stop-Menge plus Depot gleich der Knotenmenge ist.*
$V = \{D\} \cup S := \{1, \ldots, k\}$ *mit* $D = 1$.

Problem 2.5 *Beim VRP hat die zu belegende Matrix noch einen zusätzlichen Index v, der das Fahrzeug, das die entsprechende Kante passiert, bezeichnet und einen Index m, der der Nummer des Einsatzes entspricht.*

$$x_{ij}^{vm} = \begin{cases} 1 & \textit{falls die Kante } (i,j) \textit{ vom Fahrzeug } f_v \textit{ in seinem } m\textit{-ten Einsatz} \\ & \textit{passiert wird} \\ 0 & \textit{sonst} \end{cases}$$

Minimiere dabei die Summe der Kosten aller benutzten Kanten, d.h.

$$\sum_{i=1}^{k} \sum_{j=1}^{k} \sum_{v=1}^{l} \sum_{m=1}^{m(f_v)} c(i,j) \cdot x_{ij}^{vm} \tag{2.13}$$

unter Einhaltung der Restriktionen

$$\begin{aligned}
x_{ij}^{vm} &\in \{0,1\},\ \forall i,j,v,m && (2.14)\\
\sum_{i=1}^{k} \sum_{v=1}^{l} \sum_{m=1}^{m(f_v)} x_{ij}^{vm} &= 1,\ j \in \{2,\ldots,k\} && (2.15)\\
\sum_{j=1}^{k} \sum_{v=1}^{l} \sum_{m=1}^{m(f_v)} x_{ij}^{vm} &= 1,\ i \in \{2,\ldots,k\} && (2.16)\\
\sum_{i=1}^{k} x_{ip}^{vm} - \sum_{j=1}^{k} x_{pj}^{vm} &= 0,\ v \in \{1,\ldots,l\}, p \in \{1,\ldots,k\}, \\
& \quad m \in \{1,\ldots,m(f_v)\} && (2.17)\\
\sum_{i=2}^{k} b(i) \left(\sum_{j=1}^{k} x_{ij}^{vm} \right) &\leq q(f_v),\ v \in \{1,\ldots,l\},
\end{aligned}$$

$$m \in \{1, \ldots, m(f_v)\} \tag{2.18}$$

$$\sum_{i=1}^{k}\sum_{j=1}^{k} t_E(i,j)\left(\sum_{m=1}^{m(f_v)} x_{ij}^{vm}\right) + \sum_{i=1}^{k} t_S(i)\left(\sum_{j=1}^{k}\sum_{m=1}^{m(f_v)} x_{ij}^{vm}\right) +$$

$$\tilde{t_D}\cdot\sum_{i=2}^{k} b(i)\left(\sum_{j=1}^{k}\sum_{m=1}^{m(f_v)} x_{ij}^{vm}\right) \leq t_F(f_v),\ v \in \{1, \ldots, l\} \tag{2.19}$$

$$\sum_{j=2}^{k} x_{1j}^{vm} \leq 1,\ v \in \{1, \ldots, l\},$$

$$m \in \{1, \ldots, m(f_v)\} \tag{2.20}$$

$$\sum_{i=2}^{k} x_{i1}^{vm} \leq 1,\ v \in \{1, \ldots, l\},$$

$$m \in \{1, \ldots, m(f_v)\} \tag{2.21}$$

$$X \in B \tag{2.22}$$

Dabei stehen die Komponenten der Matrix $X = (x_{ij})$ für die Kantenbenutzung, unabhängig von Fahrzeug oder Einsatz:

$$x_{ij} = \sum_{v=1}^{l}\sum_{m=1}^{m(f_v)} x_{ij}^{vm}$$

Die Menge B ist wie in Abschnitt 2.1.2 definiert.

Die Bedingungen (2.15) bzw. (2.16) stellen sicher, daß jeder Knoten außer dem Depot von genau einem Fahrzeug in einem Einsatz bedient wird. Daß jeder Stop, der von einem Fahrzeug betreten wird, auch von diesem verlassen wird, garantiert (2.17). (2.15) und (2.17) bedingen bereits (2.16).

Bedingungen (2.18) bis (2.21) stellen sicher, daß die Einschränkungen an die Fahrzeugbenutzung eingehalten werden. Die Kapazitäts- bzw. Zeitschranken werden durch (2.18) bzw. (2.19) festgelegt, während (2.20) und (2.21) dafür sorgen, daß jedes Fahrzeug das Depot während eines Einsatzes höchstens einmal anfährt. Dabei ist die Bedingung (2.21) überflüssig, da sie bereits von (2.17) und (2.20) garantiert wird.

Des weiteren ist garantiert, daß jedes eingesetzte Fahrzeug auf seiner Route einmal das Depot passiert. Denn wäre dies nicht der Fall, so entstünde eine Subtour, deren Stops nur von einem Fahrzeug in einem Einsatz angefahren werden und zu anderen Stops keinerlei Verbindung haben, da ja jeder Stop nur von genau einem Fahrzeug passiert wird. Diese Subtour wird jedoch von der Bedingung (2.22) verhindert.

Kapitel 3

Komplexitätstheoretische Einordnung

Ich gehe im folgenden nur kurz auf die Grundlagen der Komplexitätstheorie ein. Im wesentlichen stütze ich mich auf Schreibweisen, Definitionen und Sätze in [3], Kap. 1–3.

3.1 TSP und VRP als Entscheidungsproblem

3.1.1 TSP als Entscheidungsproblem

Im folgenden seien die Entfernungen zwischen Knoten immer auf natürlichen Zahlen definiert. Insbesondere gilt für die vorkommende Konstante α, die die Länge einer Tour bezeichnet: $\alpha \in \mathbb{N}$.

In der Komplexitätstheorie sind Probleme Mengen von Zeichenketten, die aus einem Alphabet Σ gebildet werden. Die Aufgabe besteht i.a. darin, zu bestimmen, ob eine Zeichenkette Element dieses Problems ist oder nicht.

Auch aus dieser Sichtweise kann man das TSP betrachten (es gilt Voraussetzung 2.1, G vollständig und $V = S = \{1, \ldots, k\}$):

Definition 3.1 (TSP)

$$\mathsf{TSP} := \{\langle G, c, \alpha\rangle : \exists \text{ Tour } \pi : C(\pi) \leq \alpha\}$$

Die spitzen Klammern ($\langle, \rangle$) stehen für eine geeignete Codierung ihres Inhalts in Σ^*.

Definition 3.2 (P, NP) *P ist die Menge aller Probleme, die von einer deterministischen Turingmaschine in Polynomzeit erkannt werden können. NP ist entsprechend die Menge aller Probleme, die von einer nichtdeterministischen Turingmaschine in Polynomzeit erkannt werden können.*

Satz 3.1 $\mathsf{TSP} \in NP$

Beweisskizze: Die Turingmaschine "rät" eine Tour π, berechnet $C(\pi)$. Ist dieser Wert kleiner als α, so akzeptiert die Turingmaschine die Eingabe. Andernfalls verwirft sie sie.

Definition 3.3 ($\leq_m$, *NP*-hart, *NP*-vollständig) *Ein Problem A ist NP-hart genau dann, wenn es für alle Probleme $B \in NP$ eine Polynomzeitreduktion von B auf A gibt, d.h. eine in Polynomzeit berechenbare Funktion*

$$f : \Sigma^* \to \Sigma^*$$

so daß gilt:

$$x \in B \iff f(x) \in A,\ \forall x \in \Sigma^*$$

(Polynomial time many-one reducibility), in Zeichen: $B \leq_m A$.

Ein Problem A ist NP-vollständig genau dann, wenn A NP-hart ist und $A \in NP$.

$B \leq_m A$ bedeutet anschaulich, daß das Problem B "leichter" ist als A und gelöst werden kann, wenn man A löst und zusätzlich noch polynomiell viel Zeit in Anspruch nimmt. Wenn A *NP*-hart ist, dann bedeutet dies, daß alle Probleme in *NP* mit Hilfe von A gelöst werden können.

Um zu zeigen, daß ein Problem A *NP*-hart ist, reicht es, wegen der Transitivität von $\leq_m$, ein Problem C zu finden, für das bereits bewiesen ist, daß es *NP*-hart ist, und zu zeigen: $C \leq_m A$. Es wird vorausgesetzt, daß das Erfüllbarkeitsproblem für Boole'sche Formeln in konjunktiver Normalform SAT (Satisfiability) *NP*-vollständig ist. (Für den Beweis siehe [12, 3]).

$$\begin{aligned}\mathsf{SAT} \ := \ & \{\langle F\rangle : F \text{ ist } n-\text{stellige Boole}'\text{sche Formel in konjunktiver Normalform} \\ & \wedge\ \exists x_1, \ldots, x_n \in \{0, L\} : F(x_1, \ldots, x_n) = L\}\end{aligned}$$

Im folgenden soll bewiesen werden, daß $\mathsf{SAT} \leq_m \mathsf{TSP}$ und damit die *NP*-Vollständigkeit von TSP. Zunächst werden einige Zwischenergebnisse gezeigt (siehe auch [30]):

Satz 3.2

$$\begin{aligned}\mathsf{CLIQUE} \ &:= \ \{\langle G, k\rangle : G = (V, E) \text{ ist ungerichteter Graph und besitzt mindestens } k \text{ untereinander benachbarte Knoten, d.h. eine } k\text{-Clique}\} \\ \mathsf{SAT} \ &\leq_m \ \mathsf{CLIQUE}\end{aligned}$$

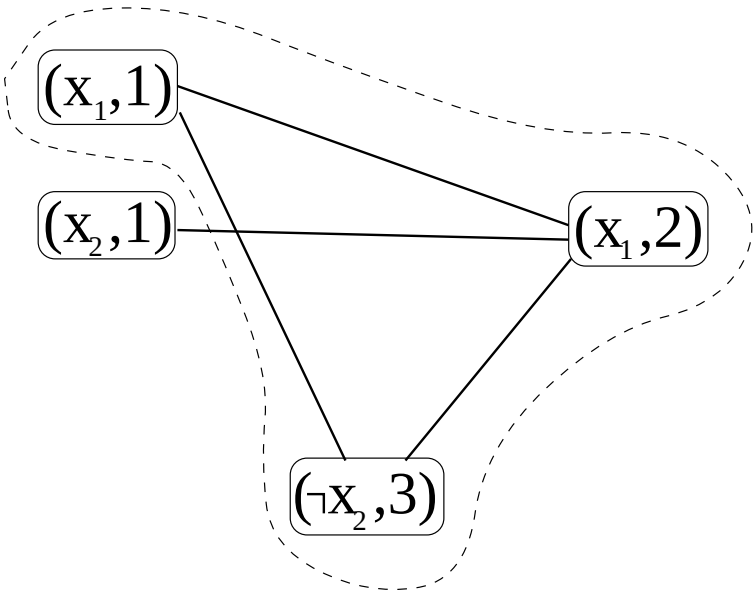


Abbildung 3.1: Graph zur Boole'schen Formel $(x_1 \vee x_2) \wedge x_1 \wedge \neg x_2$ mit markierter 3-Clique

Beweisskizze: Gesucht ist eine passende Reduktionsfunktion $f : \Sigma^* \to \Sigma^*$. Dabei wird die ungültige Codierung einer Boole'schen Formel auf eine ungültige Codierung eines Graphen abgebildet.

Anderenfalls wird die Boole'sche Formel $F = C_1 \wedge \ldots \wedge C_n$, wobei jedes C_i eine Disjunktion von Literalen x_j bzw. $\neg x_j$ ist, abgebildet auf den Graphen $G = (V, E)$ mit:

$$\begin{aligned} V &:= \{(x,i) : x \text{ ist Literal in } C_i\} \\ E &:= \{\{(x,i),(y,j)\} : x \neq \neg y, i \neq j\} \\ k &:= n \end{aligned}$$

D.h. F ist erfüllbar genau dann, wenn es einen Satz von n Variablen—jede in einer eigenen Disjunktion—gibt, die unabhängig voneinander belegt werden können. D.h. die zugehörigen Knoten bilden eine n-Clique (siehe Abb. 3.1).

Satz 3.3

$$\begin{aligned} \mathsf{VERTEX\ COVER} := \{&\langle G', n\rangle : G' = (V', E') \text{ ist ungerichteter Graph und} \\ &\text{es gibt Teilmenge von } n \text{ Knoten } (n\text{-Vertex-Cover}), \\ &\text{so daß jede Kante zu mindestens einem dieser} \end{aligned}$$

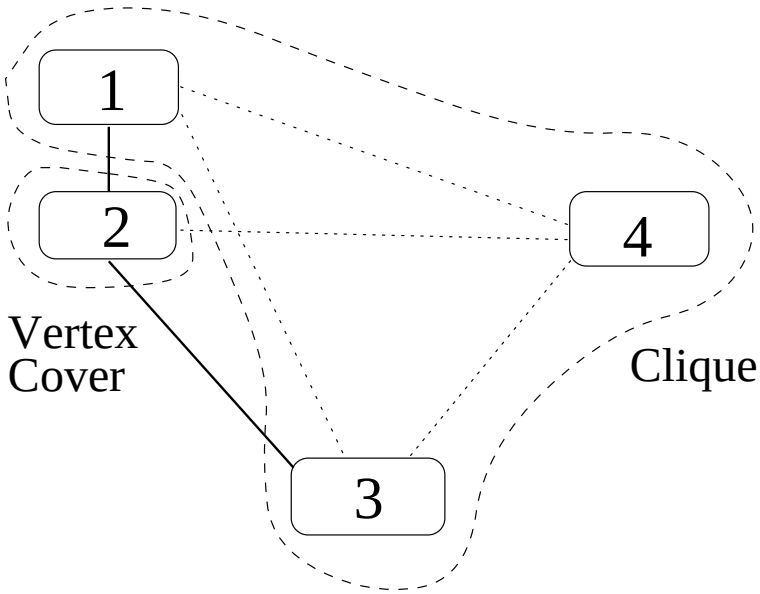


Abbildung 3.2: Vertex Cover und Clique im komplementären Graphen

Knoten benachbart ist}

$$\text{CLIQUE} \leq_m \text{VERTEX COVER}$$

Beweisskizze: Die Reduktion erfolgt durch:

$$\begin{aligned} V' &:= V \\ E' &:= \{\{i,j\} : i \neq j, \{i,j\} \notin E\} \\ n &:= |V| - k \end{aligned}$$

Es gilt:
Es gibt eine k-Clique in G
$\iff$ G' (der komplementäre Graph zu G) eine Menge von k untereinander nicht verbundenen Knoten besitzt, so daß jede Kante von G' mindestens einen Knoten hat, der nicht in dieser Menge liegt
$\iff$ Wenn G' eine Menge von $|V| - k$ Knoten hat, die alle Kanten abdeckt, d.h. ein $|V| - k$-Vertex-Cover

Abbildung 3.2 zeigt den zum Graph in Abbildung 3.1 komplementären Graphen und das dazugehörige Vertex-Cover, bestehend aus einem Knoten.

Satz 3.4

$$\text{VERTEX COVER} \leq_m \text{TSP}$$

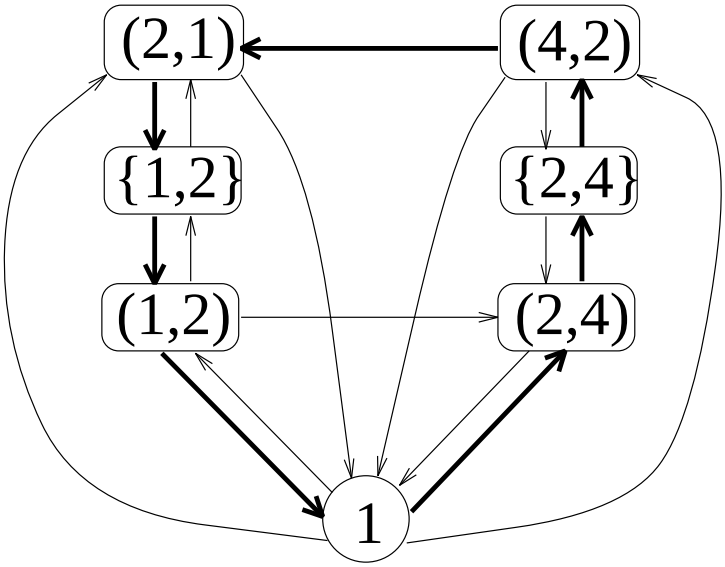


Abbildung 3.3: Reduktion des Beispiels auf TSP mit zulässiger Tour

Beweisskizze: Die Reduktion erfolgt durch:

$$
\begin{aligned}
V &:= \{(i,j), \{i,j\}, (j,i) : \{i,j\} \in E'\} \cup \{1, \ldots, n\} \\
E &:= V \times V \\
K &:= \{((i,j), \{i,j\}), (\{i,j\}, (i,j)) : \{i,j\} \in E'\} \\
&\cup \{((h,i), (i,j)) : \{h,i\}, \{i,j\} \in E', h \neq j\} \\
&\cup \{((i,j), p), (p, (i,j)) : \{i,j\} \in E', 1 \leq p \leq n\} \\
c(e) &:= \begin{cases} 1 & \text{falls } e \in K \\ 2 & \text{sonst} \end{cases} \\
\alpha &:= |V|
\end{aligned}
$$

Da eine Tour durch G über genau α Kanten verläuft, hat sie genau dann Kosten kleiner gleich α, wenn nur Kanten aus K benutzt werden.

In einem Graph, der ein n-Vertex-Cover besitzt, kann jede Kante einem benachbarten Knoten zugeordnet werden. Eine Zuordnung der Kanten $\{i, i_1\}, \ldots, \{i, i_m\}$ zum Knoten i entspricht einem Pfad $j_1, (i, i_1), \{i, i_1\}, (i_1, i), \ldots, (i, i_m), \{i, i_m\}, (i_m, i), j_2,\ 1 \leq j_1, j_2 \leq n$, der nur über Kanten aus K verläuft.

Umgekehrt ergibt sich aus einem solchen Pfad eine Zuordnung der Kanten zu einem Knoten.

Abbildung 3.3 zeigt den zu unserem Beispiel gehörenden Graphen G.

Satz 3.5 TSP *ist NP-vollständig.*

Beweis: Aus den Sätzen 3.2, 3.3, 3.4 und der Transitivität von $\leq_m$ folgt:

$$\mathsf{SAT} \leq_m \mathsf{TSP}$$

d.h. TSP ist *NP*-hart. Mit Satz 3.1 folgt dann die Behauptung.

Das TSP für vollständige Graphen (wie hier behandelt) und das Entscheidungsproblem für die allgemeinere Version des TSP (Graph ist nicht unbedingt vollständig, Stop-Menge ist nicht unbedingt gleich der Knotenmenge: siehe Problem 2.1), im folgenden $\mathsf{TSP}_{\mathsf{allg}}$ genannt, sind gegenseitig aufeinander reduzierbar. ($\mathsf{TSP} \leq_m \mathsf{TSP}_{\mathsf{allg}}$ ist offensichtlich und $\mathsf{TSP}_{\mathsf{allg}} \leq_m \mathsf{TSP}$ ergibt sich aus der Bemerkung nach Problem 2.1.)

D.h. auch $\mathsf{TSP}_{\mathsf{allg}}$ ist *NP*-vollständig.

Schwieriger ist es, die *NP*-Vollständigkeit des euklidischen TSP zu zeigen, d.h. für den Fall, daß die Stops Punkte auf der Ebene und ihre Abstände die euklidischen Entfernungen sind. Ein solcher Beweis findet sich in [37, 21].

3.1.2 VRP als Entscheidungsproblem

Hier gilt im folgenden, ähnlich wie beim TSP: G ist vollständig und $V = S \cup \{D\}$.

Definition 3.4 (VRP ohne Zeitschranken)

$$\begin{aligned}\mathsf{VRP} \;:=\; & \{\langle G, c, b, D, F, q, m, \alpha\rangle : \exists p, \text{Touren } \pi_1, \ldots, \pi_p \text{ vereinbar mit } F : \\ & \sum_{i=1}^{p} C(\pi_i) \leq \alpha\}\end{aligned}$$

Dabei gelten die Voraussetzungen 2.1, 2.2 und 2.3.

Definition 3.5 (VRP mit Zeitschranken)

$$\begin{aligned}\mathsf{VRP}_{\mathsf{time}} \;:=\; & \{\langle G, c, D, F, q, m, t_E, t_S, t_D, t_F, \alpha\rangle : \exists p, \text{Touren } \pi_1, \ldots, \pi_p \\ & \text{vereinbar mit } F \text{ und den Zeitschranken von } F : \sum_{i=1}^{p} C(\pi_i) \leq \alpha\}\end{aligned}$$

Dabei gelten die Voraussetzungen 2.1, 2.2 und 2.3.

Satz 3.6

$$\mathsf{VRP}_{\mathsf{time}} \in NP$$

Beweisskizze: Die nichtdeterministische Turingmaschine rät p, die Touren π_1 bis π_p und eine Zuordnung der Touren zu den Fahrzeugen.

Die Überprüfung auf die Vereinbarkeit mit dem Fuhrpark und den Vergleich der Kosten mit α erfolgt dann deterministisch in polynomieller Zeit.

Satz 3.7

$$\mathsf{TSP} \leq_m \mathsf{VRP}$$

Beweis: Die Reduktion erfolgt durch:

$$\begin{aligned} b(s) &= 1,\ \forall s \in S \\ D &= 1 \\ F &= \{f_1\} \\ q(f_1) &= |S| \\ m(f_1) &= 1 \end{aligned}$$

G, c und α können unbesehen übernommen werden.

Alle Stops können mit einem Fahrzeug bedient werden und die kürzeste Lösung des VRP ist gleich der kürzesten Lösung des TSP.

Satz 3.8

$$\mathsf{VRP} \leq_m \mathsf{VRP}_{\mathsf{time}}$$

Beweis: Die Reduktion erfolgt durch:

$$\begin{aligned} t_E(e) &= 0,\ \forall e \in E \\ t_S(s) &= 0,\ \forall s \in S \\ t_D(x) &= 0,\ \forall x \in \mathbb{R}_0^+ \\ t_F(f) &= 1,\ \forall f \in F \end{aligned}$$

G, c, D, F, q, m und α werden übernommen.

Die Reduktion wurde so angelegt, daß jede Lösung des VRP auch die Zeitschranken einhält.

Satz 3.9 VRP *und* $\mathsf{VRP}_{\mathsf{time}}$ *sind NP-vollständig.*

Beweis: Die Behauptung folgt direkt aus den Sätzen 3.6, 3.7 und 3.9.

3.2 Exaktes TSP und VRP als Entscheidungsproblem

Definition 3.6 (EXACT TSP)

$$\text{EXACT TSP} := \{\langle G, c, \alpha\rangle : \exists \text{Tour } \pi : C(\pi) = \alpha, \forall \text{ Touren } \lambda \text{ gilt} : C(\lambda) \geq \alpha\}$$

D.h. beim EXACT TSP *geht es darum, zu entscheiden, ob die Kosten einer minimalen Tour durch einen Graphen genau α sind.*

Definition 3.7 (Komplexitätsklassen)

$$\begin{aligned} \text{Co-}NP &:= \{A : \Sigma^* \backslash A \in NP\} \\ NP \wedge \text{Co-}NP &:= \{A \cap B : A \in NP, B \in \text{Co-}NP\} \end{aligned}$$

Im folgenden soll gezeigt werden, daß EXACT TSP $NP \wedge$ Co-NP-vollständig ist. Der Satz stammt aus [39], den Beweis habe ich selbst entwickelt.

Definition 3.8 (Komplementäres TSP)

$$\text{Co-TSP} := \{\langle G, c, \alpha\rangle : \exists \text{ keine Tour } \pi : C(\pi) \leq \alpha - 1\}$$

Satz 3.10 Co-TSP *ist* Co-NP*-vollständig.*

Beweisskizze: Es gilt: Co-TSP $\leq_m \Sigma^* \backslash$TSP. Die Reduktion erfolgt durch die Funktion $f : \Sigma^* \to \Sigma^*$ mit:

$$\begin{aligned} f(\langle G, c, \alpha\rangle) &= \langle G, c, \alpha - 1\rangle \\ f(K) &= \langle G', c', \alpha'\rangle \ (\langle G', c', \alpha'\rangle \in \text{TSP beliebig}) \end{aligned}$$

Dabei ist $K \in \Sigma^*$ eine ungültige Codierung eines Graphen mit Kostenfunktion und einer natürlichen Zahl α.

Umgekehrt gilt aber auch: $\Sigma^* \backslash$TSP $\leq_m$ Co-TSP Die Reduktion erfolgt durch die Funktion $f : \Sigma^* \to \Sigma^*$ mit:

$$\begin{aligned} f(\langle G, c, \alpha\rangle) &= \langle G, c, \alpha + 1\rangle \\ f(K) &= \langle G', c', \alpha'\rangle \ (\langle G', c', \alpha'\rangle \in \text{Co-TSP beliebig}) \end{aligned}$$

Dabei ist $K \in \Sigma^*$ eine ungültige Codierung.

$\Sigma^* \backslash$TSP ist nach Definition 3.7 und Satz 3.1 Element von Co-NP. Für jedes Problem $A \in$ Co-NP gilt: $\Sigma^* \backslash A \in NP$ und damit $\Sigma^* \backslash A \leq_m$ TSP (wegen Satz 3.5).

Mit derselben Reduktionsfunktion, mit der $\Sigma^* \backslash A$ auf TSP reduziert wird kann auch A auf $\Sigma^* \backslash$TSP reduziert werden. Damit ist gezeigt, daß $\Sigma^* \backslash$TSP Co-NP-vollständig ist und damit auch Co-TSP, da beide Probleme wechselseitig aufeinander reduzierbar sind.

Satz 3.11

$$\text{EXACT TSP} \in NP \wedge \text{Co-}NP$$

Beweis: Aus den Sätzen 3.1 und 3.10 folgt: $\text{TSP} \in NP$ bzw. Co-$\text{TSP} \in \text{Co-}NP$.

$$\Rightarrow \text{ EXACT TSP} = \text{TSP} \cap \text{Co-TSP} \in NP \wedge \text{Co-}NP$$

Satz 3.12

$$\text{TSP} \leq_m \text{EXACT TSP}$$

Beweisskizze: Es gilt: VERTEX COVER $\leq_m$ EXACT TSP. Um dies zu sehen, muß man nur den Beweis zu Satz 3.4 leicht modifizieren. Denn die Tour durch den Graphen, die gefunden werden muß, um VERTEX COVER zu entscheiden, ist zugleich immer auch die minimale Tour. Es gilt damit also:

$$\text{SAT} \leq_m \text{VERTEX COVER} \leq_m \text{EXACT TSP} \Rightarrow \text{TSP} \leq_m \text{EXACT TSP}$$

Satz 3.13

$$\text{Co-TSP} \leq_m \text{EXACT TSP}$$

Beweisskizze: Die Reduktion erfolgt durch

$$f : \Sigma^* \to \Sigma^*, f(\langle G, c, \alpha \rangle) = \langle G' = (V', E'), c', \alpha' \rangle$$

mit

$$\begin{aligned}
V' &= V \times \{1,2,3,4\} \\
\\
E'_1 &= \{((v_1,1),(v_2,1)) : (v_1,v_2) \in E\} \\
E'_2 &= \{((v_1,2),(v_2,2)),\ldots,((v_{n-1},2),(v_n,2)),((v_n,2),(v_1,2))\} \\
&\quad \text{mit } V = \{v_1,\ldots,v_n\} \text{ (Knoten in beliebiger Reihenfolge)} \\
E'_{34} &= \{((v,1),(v,3)),((v,3),(v,2)), \\
&\quad ((v,2),(v,4)),((v,4),(v,1)) : v \in V\} \\
E' &= E'_1 \cup E'_2 \cup E'_{34} \\
\\
c((v_1,1),(v_2,1)) &= c(v_1,v_2)
\end{aligned}$$

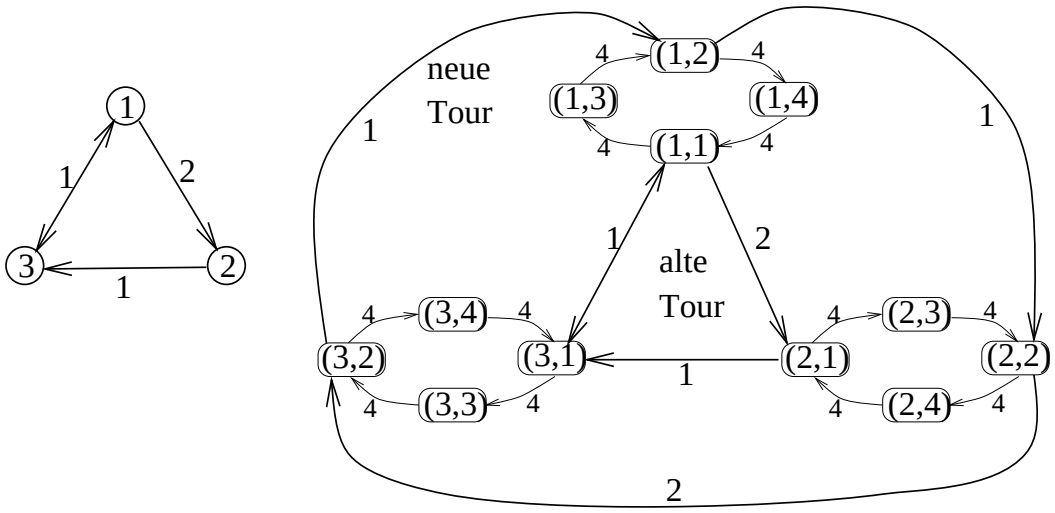


Abbildung 3.4: Beispiel einer Reduktion von Co-TSP mit $\alpha = 4$ auf EXACT TSP mit $\alpha' = (1 + 4 \cdot 3) \cdot 4 = 52$

$$
\begin{aligned}
c((v_1, 2), (v_2, 2)) \quad & \text{beliebig, so daß} \\
& \sum_{i=1}^{n-1} c((v_i, 2), (v_{i+1}, 2)) + c((v_n, 2), (v_1, 2)) = \alpha \\
c(e) \quad &= \quad \alpha, \; \forall e \in E'_{34} \\
\\
\alpha' \quad &= \quad (1 + 4n) \cdot \alpha
\end{aligned}
$$

Der entstehende Graph ist nicht vollständig, kann aber durch Bildung der Distanzmatrix auf einen vollständigen Graphen reduziert werden, ohne daß sich die Länge der kürzesten Tour ändert. Im folgenden argumentiere ich daher auf dem nicht-vollständigen Graphen, der nicht unbedingt einen Hamilton'schen Kreis besitzt, so daß Knoten in einer Tour u.U. mehrfach passiert werden müssen.

Die Idee besteht darin, den Graphen G "künstlich" um eine neue Tour der Länge α über Kanten aus E'_2 zu ergänzen. Der entstehende Graph ist Element von EXACT TSP genau dann, wenn es vorher keine Tour kürzer als α gab, d.h. wenn $\langle G, c, \alpha \rangle \in$ Co-TSP. (Siehe Abbildung 3.4).

Würde man die Kanten aus E'_2 direkt an die ursprünglichen Knoten des Graphen hängen, so bestünde die Gefahr, daß eine Tour kürzer als α entsteht, obwohl es vorher keine solche gab. Aus diesem Grund muß die neue Tour vom bisherigen Graph durch Kanten aus E'_{34} getrennt werden, die alle passiert werden müssen, da sie den einzigen Zugang zu den Knoten der Form $(v, 3)$ und $(v, 4)$ bilden.

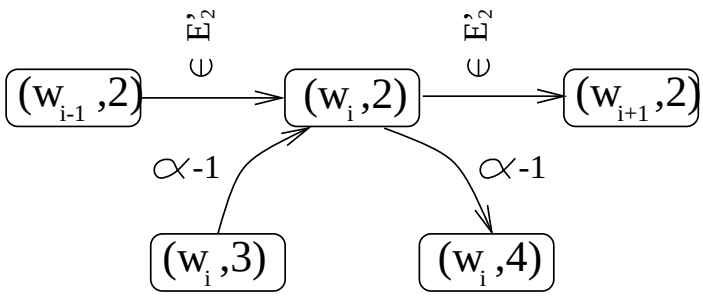


Abbildung 3.5: Wenn eine Kante aus E_2' in der kürzesten Tour passiert wird, dann werden alle Kanten aus E_2' passiert

Der Graph G' enthält immer eine Tour der Länge $\alpha' = (1 + 4n) \cdot \alpha$, nämlich:

$$(v_1, 2), (v_1, 4), (v_1, 1), (v_1, 3), (v_1, 2), (v_2, 2), \ldots, (v_n, 2), (v_n, 4), (v_n, 1), (v_n, 3)$$

Falls der ursprüngliche Graph G eine Tour $w_1, \ldots, w_n$ mit Kosten $\beta < \alpha$ enthält, so enthält der Graph G' die Tour

$$(w_1, 1), (w_1, 2), (w_1, 3), (w_1, 4), (w_1, 1), (w_2, 1), \ldots, (w_n, 1), (w_n, 3), (w_n, 2), (w_n, 4)$$

mit den Kosten $\beta + 4n \cdot \alpha < \alpha'$

Zu zeigen bleibt nun noch, daß G' keine Tour kleiner α' enthält, falls G keine Tour kleiner als α enthält. Im folgenden wird also vorausgesetzt, daß die minimale Tour in G Kosten von mindestens α hat.

Der Beweis erfolgt durch Widerspruch:

Annahme 1: es gibt eine Tour π durch G' mit Kosten $C(\pi) = \beta' < \alpha'$.

Annahme 2: π passiert keine Kanten aus E_2'.

Daraus folgt, daß die Tour im wesentlichen folgendes Aussehen hat:

$$(w_1, 1), (w_1, 2), (w_1, 3), (w_1, 4), (w_1, 1), (w_2, 1), \ldots, (w_n, 1), (w_n, 1), (w_n, 3), (w_n, 4)$$

Läßt man alle Schleifen über Knoten der Form $(w, 2), (w, 3), (w, 4)$ weg, so erhält man eine Subtour $(w_1, 1), \ldots, (w_n, 1)$ der Länge $\beta - 4n \cdot \alpha < \alpha$, die aber einer Tour durch G entspricht. G besitzt aber nach Voraussetzung keine solche Tour. Es ergibt sich ein Widerspruch zu Annahme 2 und es folgt, daß π Kanten aus E_2' passieren muß.

Behauptung: Wenn eine Kante $((w_i, 2), (w_{i+1}, 2))$ aus E_2' passiert wird, dann wird auch die vorhergehende Kante $((w_{i-1}, 2), (w_i, 2))$ passiert. $(w_0 := w_n)$

Es gibt zwei Möglichkeiten, eine Kante aus $((w_i, 2), (w_{i+1}, 2)) \in E_2'$ zu erreichen (siehe auch Abbildung 3.5):

- Über die vorhergehende Kante in E_2'. In diesem Fall ist nichts mehr zu beweisen.
- Über die Kante $((w_i, 3), (w_i, 2))$. In diesem Fall muß der Knoten $(w_i, 2)$ noch einmal betreten werden oder schon einmal betreten worden sein, um die Kante $((w_i, 2), (w_i, 4))$ zu durchlaufen. Dies muß über die vorhergehende Kante aus E_2' geschehen, denn würde die Kante $((w_i, 3), (w_i, 2))$ noch einmal benutzt, so entstünden Kosten von mindestens $(4n + 1) \cdot \alpha = \alpha' > \beta'$ und Widerspruch zu Annahme 1.

Aus der Behauptung folgt, daß alle Kante aus E_2' von π benutzt werden. Damit ergibt sich für die Kosten der Tour: $C(\pi) \geq \alpha + 4n \cdot \alpha = \alpha' > \beta'$. Damit ist Annahme 1 nicht zulässig und der Satz bewiesen.

Satz 3.14 EXACT TSP *ist* $NP \wedge$ Co-NP*-vollständig.*

Beweisskizze: Gezeigt wird, daß es für jedes Problem $C \in NP \wedge$ Co-NP mit $C := A \cap B$, $A \in NP$, $B \in$ Co-NP, eine Reduktion auf EXACT TSP gibt.

Es gilt $A \leq_m$ TSP, da TSP NP-vollständig ist (nach Satz 3.1). Da außerdem TSP $\leq_m$ EXACT TSP gilt (Satz 3.12), folgt: $A \leq_m$ EXACT TSP.

Mit den Sätzen 3.10 und 3.13 folgt analog: $B \leq_m$ EXACT TSP.

Es gibt also Funktionen $f_A : \Sigma^* \to \Sigma^*$ und $f_B : \Sigma^* \to \Sigma^*$, die A bzw. B auf EXACT TSP reduzieren.

Hat man nun ein $x \in \Sigma^*$, für das die Zugehörigkeit zu $A \cap B$ überprüft werden soll, so ist zu entscheiden, ob $f_A(x) = \langle G_A = (V_A, E_A), c_A, \alpha_A \rangle$ und $f_B(x) = \langle G_B = (V_B, E_B), c_B, \alpha_B \rangle$ in EXACT TSP liegen. Dies kann auch zusammen geschehen, indem man zu G_A und G_B einen Graphen G in folgender Weise bildet:

$C := \sum_{v_1, v_2 \in V_A} c(v_1, v_2) + 1$

$G_B' := G_B$, und für die Kostenfunktion c_B' von G_B' gelte: $c_B' := C \cdot c_B$.

M sei die Summe aller Kantenbewertungen der Graphen G_A und G_B'.

Die beiden Graphen G_A und G_B' werden vereinigt. Seien $a \in V_A$ und $b \in V_B$ zwei beliebige, aber feste Knoten aus G_A bzw. G_B'.

Zu jedem Knoten y aus G_B' wird eine Kante von a aus geführt. Für die Bewertung dieser Kante soll gelten: $c(a, y) = M + C \cdot c_B(b, y)$. Analog wird zu jedem Knoten x aus G_A eine Kanten von b mit der Bewertung $c(b, x) = M + c_A(a, x)$ (siehe Abbildung 3.6) gezogen.

Der entstandene Graph wird "vollständig gemacht", indem Knoten, die bisher noch nicht durch eine Kante verbunden waren, gemeinsame Kanten mit dem Abstand der Knoten als Kantenbewertung erhalten. Dieser Graph heißt G und hat die Kostenfunktion c. Er kann in Polynomzeit erzeugt werden.

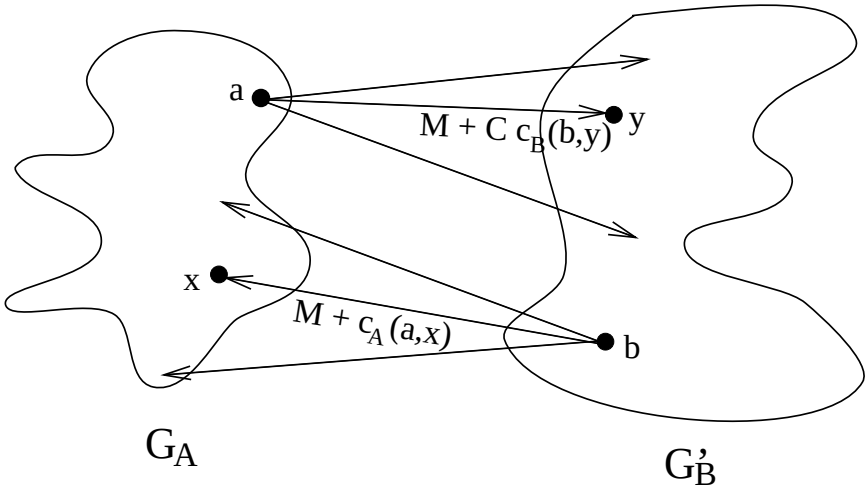


Abbildung 3.6: Konstruktion des Graphen G durch Zusammenfügen von G_A und G'_B

Es soll nun gezeigt werden:

$$\begin{array}{ll} & \langle G_A, c_A, \alpha_A \rangle \in \mathsf{EXACT\ TSP} \ \wedge \ \langle G_B, c_B, \alpha_B \rangle \in \mathsf{EXACT\ TSP} \\ \iff & \langle G, c, \alpha_A + C \cdot \alpha_B + 2M \rangle \in \mathsf{EXACT\ TSP} \end{array}$$

Dazu benötigt man folgende Zwischenergebnisse:

- **Behauptung 1:** Wenn G_A eine Tour der Länge α_A und G_B eine Tour der Länge α_B hat, dann hat G eine Tour der Länge $\alpha_A + C \cdot \alpha_B + 2M$

  Wenn G_B eine Tour der Länge α_B hat, dann hat G'_B eine Tour der Länge $C \cdot \alpha_B$.

  Dann kann man eine Tour in G bilden, die mit a beginnt, über eine der neuen Kanten zum Knoten y (dem Nachfolger von b in der Tour in G'_B) wechselt und dann alle Knoten in G'_B entsprechend ihrer Reihenfolge in der Tour abläuft. Wenn man bei b angelangt ist, dann kann man eine der neuen Kanten benutzen, um zum Knoten x zu gelangen (dem Nachfolger von a in der Tour in G_A). Von diesem ausgehend werden jetzt alle Knoten in G_A besucht, bis die Tour mit dem Erreichen von a beendet ist. (Siehe Abbildung 3.7).

- **Behauptung 2:** Wenn G eine minimale Tour der Länge α hat, dann hat G_A eine Tour der Länge $(\alpha - 2M)$ mod C und G_B hat eine Tour der Länge $(\alpha - 2M)$ div C

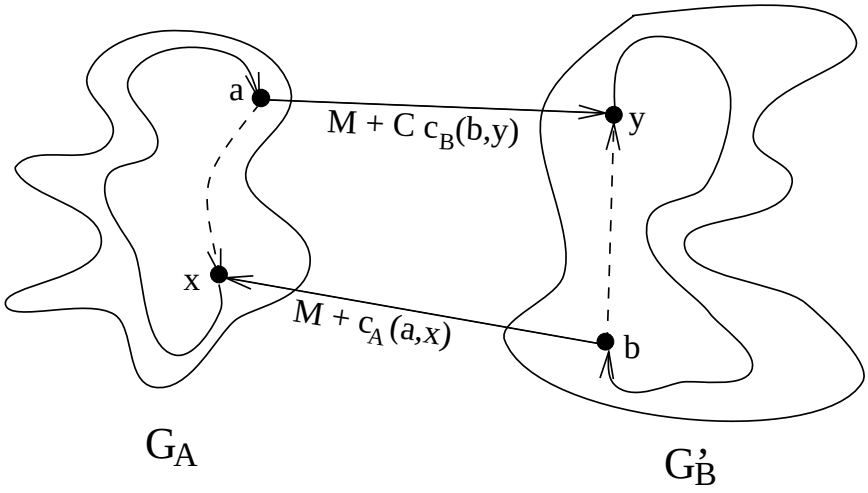


Abbildung 3.7: Verbinden von Touren durch G_A und G'_B zu einer Tour durch G

Wegen der hohen Kosten der verbindenden Kanten wechselt die Tour durch G nur je einmal zwischen Knoten aus G_A zu Knoten aus G'_B und umgekehrt. Sonst könnte man die Knoten innerhalb der Tour umordnen und erhielte eine kürzere Tour.

Im folgenden wird o.B.d.A. angenommen, daß die Tour immer vom Knoten a aus nach G'_B und vom Knoten b aus nach G_A wechselt. Denn ansonsten können die Knoten a und b vor dem Überwechseln in den anderen Graphen besucht und ansonsten übersprungen werden, was wegen der Minimalität der betrachteten Tour und der Dreiecksungleichung dieselben Kosten liefert.

Wenn man nun die beiden Kanten von G_A nach G'_B und umgekehrt aus der Tour streicht, so erhält man in jedem der beiden Teilgraphen einen Pfad, der bereits alle Knoten umfaßt und durch Einfügen der fehlenden verbindenden Kante zu einem Hamiltonschen Kreis vervollständigt werden kann.

Sei α_A die Länge der entstandenen Tour in G_A, $C \cdot \alpha_B$ die Länge der entstandenen Tour in G'_B. Dann gilt:

$$\alpha_A + C \cdot \alpha_B = \alpha - 2M$$

Da $\alpha_A < C$ gilt Behauptung 2.

Aus $\langle G_A, c_A, \alpha_A \rangle \in$ EXACT TSP und $\langle G_B, c_B, \alpha_B \rangle \in$ EXACT TSP folgt dann mit Behauptung 1, daß G eine Tour der Länge $\alpha_A + C \cdot \alpha_B + 2M$ besitzt. Es kann aber keine kürzere Tour geben, denn dann hätte nach Behauptung 2 entweder G_A oder G_B eine kürzere Tour. Es gilt also: $\langle G, c, \alpha_A + C \cdot \alpha_B + 2M \rangle \in$ EXACT TSP.

Aus $\langle G, c, \alpha_A + C \cdot \alpha_B + 2M \rangle \in$ EXACT TSP kann man mit Behauptung 2 schließen, daß G_A eine Tour der Länge α_A und G_B eine Tour der Länge α_B hat. Diese Touren müssen nach Behauptung 1 minimal sein, da G sonst eine kürzere Tour hätte. Es gilt damit: $\langle G_A, c_A, \alpha_A \rangle \in$ EXACT TSP und $\langle G_B, c_B, \alpha_B \rangle \in$ EXACT TSP.

Analog zu EXACT TSP kann man auch EXACT VRP definieren. Wie in Satz 3.11 kann für EXACT VRP die Zugehörigkeit zu $NP \wedge$ Co-NP bewiesen werden. Da EXACT TSP auf EXACT VRP reduziert werden kann, ist EXACT VRP ebenfalls $NP \wedge$ Co-NP-vollständig.

3.3 TSP und VRP als Optimierungsproblem

Wenn man das TSP als Optimierungs- und nicht mehr als Entscheidungsproblem betrachtet, so kann man sich nicht mehr darauf beschränken, Probleme als Mengen von Zeichenketten zu betrachten, sondern muß das TSP als Funktion formulieren.

Definition 3.9 (OPTIMAL TSP) OPTIMAL TSP *sei eine Funktion:*

$$\begin{aligned} \text{OPTIMAL TSP} : \Sigma^* &\rightarrow \mathbb{N} \\ \text{OPTIMAL TSP}(\langle G, c \rangle) &= \alpha \iff \langle G, c, \alpha \rangle \in \text{EXACT TSP} \end{aligned}$$

Für unkorrekte Codierungen eines Graphen G mit Kostenfunktion c ist OPTIMAL TSP *nicht definiert.*

Um OPTIMAL TSP in eine entsprechende Komplexitätsklasse einordnen zu können, benötigt man den Begriff des Orakels.

Definition 3.10 (Orakel-Turingmaschine) *Eine Orakel-Turingmaschine mit Orakel M ist eine deterministische oder nichtdeterministische Turingmaschine, die im Laufe ihrer Berechnung beliebig oft eine Frage an das Orakel stellen darf, d.h. die Antwort auf die Frage $x \in M$ für beliebige $x \in \Sigma^*$ ohne irgendwelchen Aufwand von Rechenzeit oder Speicherplatz erhält.*

Probleme, die von deterministischen Turingmaschinen, die ein Orakel aus NP benutzen, erkannt werden können, bilden die Menge Δ_2^P (oder auch P^{NP}). Die entsprechende Menge für nichtdeterministische Turingmaschinen wird mit Σ_2^P oder NP^{NP} bezeichnet. Sie sind beide Klassen der polynomiellen Hierarchie.

Ebenso kann man Klassen von Funktionen bilden, die von deterministischen bzw. nichtdeterministischen Turingmaschinen mit Orakel aus NP berechnet werden. Diese Klassen werden im folgenden mit $F(P^{NP})$ bzw $F(NP^{NP})$ bezeichnet.

Satz 3.15 OPTIMAL TSP $\in F(P^{NP})$

Beweisskizze: Sei $s := \sum_{v_1,v_2 \in V} c(v_1, v_2)$. s ist auf jeden Fall größer gleich der Länge der minimalen Tour.

Die deterministische Turingmaschine T führt eine binäre Suche für $\alpha \in \{1, \ldots, s\}$ durch. Sobald ein α gefunden ist mit $\langle G, c, \alpha \rangle \in \mathsf{TSP}$ und $\langle G, c, \alpha - 1 \rangle \notin \mathsf{TSP}$, gibt sie α als Ergebnis zurück.

Dazu sind $\mathcal{O}(\log s)$ Schritte erforderlich.

Zu zeigen ist jetzt, daß $\log s$ höchstens polynomiell von $K := |\langle G, c \rangle|$ (Länge der Kodierung des Graphen mit Kostenfunktion) abhängt.

$s \leq |V|^2 \cdot \max_{v_1,v_2 \in V}\{c(v_1, v_2)\}$

Die Anzahl der Knoten im Graphen $|V|$ und die maximale Länge einer Kante müssen in höchstens K Zeichen kodiert sein und können daher höchstens den Wert b^K haben, wobei b von der Art der Kodierung abhängt. D.h.: $s \leq b^{3K}$ und daraus folgt: $\log s = 3K \cdot \log b$. Da b eine Konstante ist, hängt $\log s$ linear von K ab und T arbeitet damit in polynomieller Zeit.

Es gilt daher: $\mathsf{OPTIMAL\ TSP} \in F(P^{NP})$.

Für das Problem der optimalen Tourenplanung, kann die Zugehörigkeit zu $F(P^{NP})$ in analoger Weise gezeigt werden.

$\mathsf{OPTIMAL\ TSP}$ ist sogar $F(P^{NP})$-vollständig. Dieses Ergebnis stammt aus [29].

Um die oben erwähnten Ergebnisse einzuordnen, gebe ich die Inklusionsbeziehungen der behandelten Komplexitätsklassen an:

$$P \subseteq NP \subseteq NP \wedge \text{Co-}NP \subseteq P^{NP} \subseteq NP^{NP}$$

Es ist noch nicht gezeigt worden, ob irgendeine dieser Inklusionen echt ist.

3.4 Approximierbarkeit von TSP

In den vorhergehenden Abschnitten wurde gezeigt, daß bereits die grundlegenden Problemstellungen der Tourenplanung in Komplexitätsklassen liegen, die als "sehr schwer" vermutet werden.

Die besten bekannten Algorithmen für *NP*-harte Probleme haben exponentielle Laufzeit. Es ist zwar noch nicht bewiesen, wird aber allgemein vermutet, daß es für solche Probleme keine polynomiellen Algorithmen gibt ($P \neq NP$).

Nachdem schon viele bei dem Versuch gescheitert sind, polynomielle Algorithmen für *NP*-harte Probleme zu finden, werde ich dies für TSP oder VRP gar nicht erst unternehmen. Da andererseits aber exponentielle Algorithmen für Probleme mit vielen Stops vollkommen unpraktikabel sind, bleibt nur ein Ausweg: Man muß sich mit approximierten, suboptimalen Lösungen begnügen. (Optimale Lösungsverfahren für kleine Stop-Mengen finden sich beispielsweise in [15].)

Suboptimale Lösungen findet man im allgemeinen durch Heuristiken, die (mehr oder weniger) geschickt einen Teil des Lösungsraumes durchsuchen und dabei auf ein lokales Optimum stoßen.

Die Aussagen der Komplexitätstheorie darüber, ob es für TSP gute Approximations-Algorithmen gibt, hängen von der speziellen Form des TSP ab. Für das allgemeine TSP (ohne zusätzliche Einschränkungen) gibt es keinen polynomiellen Algorithmus, der eine Tour π liefert und für ein $\epsilon < 1$ garantiert, daß

$$\frac{C(\pi) - C(\text{opt. Tour})}{C(\pi)} \leq \epsilon$$

Wenn jedoch der Graph symmetrisch ist und die Dreiecksungleichung gilt, kann man obere Schranken für die "Schlechtigkeit" des Approximations-Ergebnisses angeben. Die Heuristik von Christofides (siehe Abschnitt 5.1.5) liefert beispielsweise eine TSP-Tour, die höchstens um 50 % schlechter ist, als die optimale Tour. In diesem Fall gilt dann:

$$\frac{C(\pi) - C(\text{opt. Tour})}{C(\pi)} \leq \frac{1}{2}$$

Detailliertere Ergebnisse findet man in [36, 32, 40]

In den Kapiteln 5 und 6 werde ich einige Heuristiken für TSP und VRP vorstellen.

Kapitel 4

Berechnung der Stop-Distanzmatrix

4.1 Algorithmus von Dijkstra

Die Heuristiken zur Tourenplanung, die in den nächsten Kapiteln behandelt werden, gehen i.a. davon aus, daß der zugrundeliegende Graph vollständig ist, bzw. eine Stop-Distanzmatrix für zumindest einen Teil aller Stop-Paare vorliegt. Da dies im allgemeinen jedoch nicht von vornherein vorhanden ist, ist es erforderlich für einen Graphen und eine Stop-Menge wie in Definition 2.1 eine solche Matrix zu berechnen.

Um die Entfernung von einem Stop zu einem oder beliebig vielen anderen Stops zu ermitteln, muß man im wesentlichen immer den gesamten Graphen durchlaufen. Auf diesem Prinzip baut auch der Algorithmus von Dijkstra (siehe [14]) auf, der—von einem Knoten ausgehend—alle Knoten in der Reihenfolge ihrer Entfernung vom Ausgangsknoten betritt.

Im folgenden Algorithmus werden zwei Listen—$\mathcal{G}$ und $\mathcal{R}$—verwendet. $\mathcal{G}$ steht für "gescannt", $\mathcal{R}$ für "reached". Jede besteht aus Tupeln (v, α), wobei v für einen Knoten, α für die Entfernung von v vom Startknoten auf dem bisher kürzesten Weg steht.

Algorithmus 4.1 [Dijkstra]

for every Stop $s \in S$
$\mathcal{G} := \emptyset, \mathcal{R} := \{(s, 0)\}$

repeat
Sei (v, α) das Tupel, das als letztes in $\mathcal{R}$ aufgenommen wurde
Scanne alle Nachbarn von v, die noch nicht Element von $\mathcal{R}$ sind, d.h.
bilde für alle diese Nachbarn v' das Tupel $(v', \alpha' := \alpha + c(v, v'))$
for every Tupel (v', α'), das gerade gebildet wurde

if (es gibt bereits ein Tupel $(v', \beta) \in \mathcal{G}$ für ein beliebiges β) **then**
if $(\alpha' < \beta)$ **then**
Ersetze (v', β) in $\mathcal{G}$ durch (v', α')
else
Verändere $\mathcal{G}$ nicht
endif
else
Nimm (v', α') in $\mathcal{G}$ auf
endif
endfor
Streiche das Tupel (v', α') aus $\mathcal{G}$, welches das kleinste α' besitzt und nimm es in $\mathcal{R}$ auf
until $(\mathcal{G} = \emptyset)$

$\mathcal{R}$ enthält jetzt alle von s aus erreichbaren Knoten des Graphen mit ihrem kürzesten Abstand zu s
Trage die Entfernungen aller Stops in $\mathcal{R}$ in die Distanzmatrix ein
endfor

Abbildung 4.1 zeigt eine Momentaufnahme während des Ablaufs des Algorithmus auf dem Graphen aus Abbildung 2.3. Dabe ist $s = 5$ der Start-Stop der Iteration.

Für die beiden vom Algorithmus verwalteten Listen $\mathcal{R}$ und $\mathcal{G}$ gilt:

$$\begin{aligned} \mathcal{R} &= \{(5,0),(4,1),(8,1)\} \\ \mathcal{G} &= \{(1,1.5),(6,1.5),(7,3),(2,4)\} \end{aligned}$$

Tabelle 4.1 zeigt die entstehende Distanzmatrix.

Satz 4.1 *Mit dem Algorithmus von Dijkstra erhält man eine Stop-Distanzmatrix, die für jedes Paar von Stops die Länge des kürzesten Weges bezüglich c zwischen diesen Stops enthält.*

Beweis: Zu zeigen ist: Für ein Tupel (v', α'), das in $\mathcal{R}$ aufgenommen wird, gilt:

1. α' ist die kürzeste Entfernung vom Ausgangsknoten s zu v'
2. Kein Knoten $v \notin \mathcal{R}$ hat eine kürzeren Entfernung zu s als v'

Der Beweis erfolgt durch vollständige Induktion über $|\mathcal{R}|$
Induktionsanfang: $|\mathcal{R}| = 1$: für $(s, 0)$, gilt die Behauptung.
Induktionsannahme: Die Behauptung gelte für $|\mathcal{R}| \leq p$
Induktionsschritt: Von allen Knoten, die noch nicht in $\mathcal{R}$ aufgenommen wurden, befindet sich einer von denjenigen, die die kürzeste Entfernung zu s

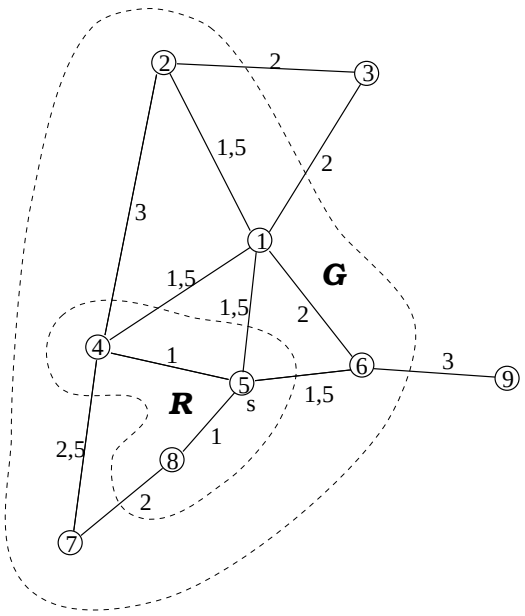


Abbildung 4.1: Ablauf des Algorithmus von Dijkstra

	D	2	3	4	5	6	7	8	9
D	0	1,5	2	1,5	1,5	2	4	2,5	5
2	1,5	0	2	3	3	3,5	5,5	4	6,5
3	2	2	0	3,5	3,5	4	6	4,5	7
4	1,5	3	3,5	0	1	2,5	2,5	2	5,5
5	1,5	3	3,5	1	0	1,5	3	1	4,5
6	2	3,5	4	2,5	1,5	0	4,5	2,5	3
7	4	5,5	6	2,5	3	4,5	0	2	7,5
8	2,5	4	4,5	2	1	2,5	2	0	5,5
9	5	6,5	7	5,5	4,5	3	7,5	5,5	0

Tabelle 4.1: Stop-Distanzmatrix des Beispielgraphen

haben, in der Nachbarschaft eines Knoten aus $\mathcal{R}$. Denn da es keine negativen Kantenbewertungen gibt, steigt die Entfernung zu s an, je weiter der Weg wird.

Da aber alle Nachbarn von Knoten aus $\mathcal{R}$ bereits eingescannt wurden, befindet sich auch einer der gesuchte Knoten mit der entsprechenden Entfernung darunter.

Implementierung: Die Knoten des Graphen werden in einem Array gehalten. In diesem Array werden unter der jeweiligen Knotennummer diejenigen Kanten in einer Liste abgespeichert, die von diesem Knoten ausgehen. Da bei dieser Aufgabenstellung der Graph ein tatsächliches Straßennetz ist, bei dem sich an einer Kreuzung (Knoten) meist nur wenige Straßen treffen, ist es nicht notwendig, den Zugriff auf die Kanten noch weiter zu verbessern.

Die Liste $\mathcal{R}$ kann ebenfalls als Array realisiert werden, da die benötigten Operationen nur der Test auf Enthaltensein eines Elements und das Einfügen eines Elements in die Liste sind. Zu jedem Knoten werden an der Stelle des Arrays, die der Knotennummer entspricht, ein Boole'scher Wert, der angibt, ob der Knoten zu $\mathcal{R}$ gehört, und die Entfernung zum Startknoten abgelegt.

Auf der Liste $\mathcal{G}$ hingegen ist noch eine andere Operation erforderlich: Neben dem Einfügen eines Element muß dasjenige Element gefunden werden, das die kürzeste Entfernung zum Startknoten besitzt. Aus diesem Grunde wurde $\mathcal{G}$ als Heap organisiert, bei dem sowohl das Einfügen eines Elements, als auch das Auslesen des größten Wertes in logarithmischer Zeit (bzgl. Anzahl der Elemente) durchführbar ist. (Zur Beschreibung der Datenstruktur Heap siehe [33]).

Komplexität: Weil $\mathcal{G}$ höchstens n (Anzahl der Knoten) Elemente enhalten kann, hat eine Operation auf $\mathcal{G}$ die Komplexität $\mathcal{O}(\log n)$.

Eine solche Operation wird bei Aufnahme eines Knotens in $\mathcal{R}$ (maximales Element suchen) und bei Einscannen eines Knotens (Element eintragen) ausgeführt. Es werden genau soviel Einscan-Vorgänge durchgeführt, wie es Kanten gibt. Wegen des Zusammenhangs des Graphen gilt: $n \leq m$ Daher ist die Komplexität eines Durchlaufs des Graphen $\mathcal{O}(m \cdot \log n)$.

Dieser Durchlauf wird genau $|S|$-mal durchgeführt. Also ergibt sich als Gesamtkomplexität: $\mathcal{O}(k \cdot m \cdot \log n)$, wobei k die Anzahl der Stops bezeichnet.

Varianten: Außer der Entfernung zweier Knoten bezüglich c wird oft zusätzlich noch ihr Zeitabstand auf dem kürzesten Weg benötigt. Zu diesem Zweck kann in einem Tupel noch zusätzlich die Zeit mitgeführt werden. Ebenso kann in jedem Tupel ein Verweis auf den Vorgänger, von dem aus dieser Knoten eingescannt wurde, vermerkt werden, so daß der kürzeste Weg im Nachhinein nachvollzogen werden kann.

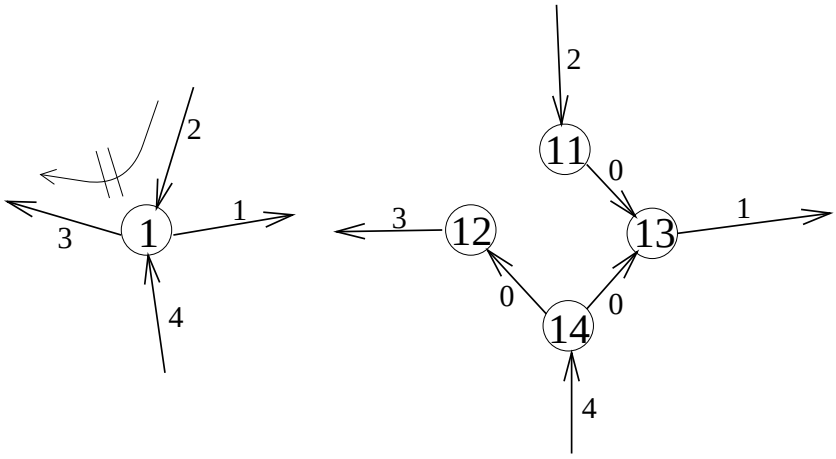


Abbildung 4.2: Modellierung des Abbiegeverbotes $(2, 3, 1)$

4.2 Berücksichtigung von Abbiegeverboten

Definition 4.1 (Abbiegeverbot) *Ein Abbiegeverbot ist ein 3-Tupel* (e_1, e_2, v), $e_1, e_2 \in E, v \in V$. *Durch ein solches Abbiegeverbot wird die Menge der Wege auf dem Graphen* G *derart eingeschränkt, daß ein Weg, der über die Kante* e_1 *den Knoten* v *erreicht, nicht über die Kante* e_2 *weitergeführt werden darf.*

Die Menge aller Abbiegeverbote des Graphen wird mit A *bezeichnet.*

Da sich durch Abbiegeverbote die Entfernungsmatrix zwischen den Knoten ändert, muß der Algorithmus von Dijkstra entsprechend modifiziert werden.

Günstigerweise transformiert man einen Graph G mit Abbiegeverboten A in einen Graph G' ohne Abbiegeverbote, der diese aber implizit in der Knoten-/Kanten-Struktur berücksichtigt:

Der von einem Abbiegeverbot betroffene Knoten wird in (Ausgangsgrad + Eingangsgrad) neue Knoten aufgespalten. Die eingehenden und ausgehenden Kanten enden bzw. beginnen jede an einem eigenen Knoten. Verbindungen mit Kantenbewertung 0 zwischen den neu erzeugten Knoten werden unter Berücksichtigung der Abbiegeverbote gesetzt. Das heißt, Kanten werden von den Knoten eingehender Kanten zu den Knoten ausgehender Kanten geführt, wobei eine Kante nur dann gesetzt wird, wenn ein entsprechendes Abbiegeverbot nicht existiert. (Siehe Abbildung 4.2).

Wenn man den Kanten Bewertungen verschieden von 0 zuweist, so könnte man auch Kosten beim Abbiegen modellieren (z.B. Linksabbiegen ist teurer als Rechtsabbiegen).

Abbiegeverbote haben auch Einfluß auf den Algorithmus von Dijkstra zur Berechnung der Distanzmatrix (Algorithmus 4.1). Die Liste $\mathcal{R}$ der erreichten

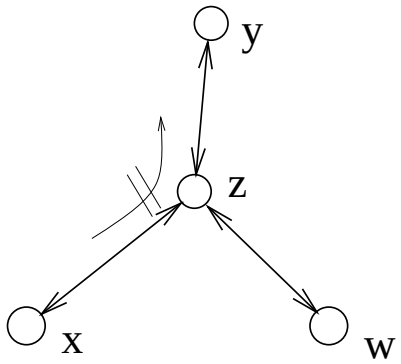


Abbildung 4.3: Beispiel für Graph mit Abbiegeverboten, in dem die Dreiecksungleichung nicht gilt

Knoten muß zu Beginn mit allen Knoten besetzt werden, die durch das Aufspalten des Anfangs-Stops s entstanden sind, falls s aufgespalten wurde. Ansonsten wird sie wie bisher nur mit dem Tupel $(s, 0)$ besetzt.

Der Abstand eines Stops zum Start-Stop s ist der minimale Abstand seiner Knoten zu s.

Zu beachten ist, daß die entstehende Distanzmatrix nicht zwangsläufig die Dreiecksungleichung erfüllt. Bewegt man sich z.B. vom Knoten x zum Knoten y und muß dabei den Knoten z passieren, an dem ein Abbiegeverbot sitzt, so ist man u.U. gezwungen, einen Umweg zu machen, der nicht eintritt, wenn man von x nach z und davon unabhängig von z nach y fährt (siehe Abbildung 4.3). In dem abgebildeten Graphen gilt: $c(x, y) = c(x, z) + c(z, w) + c(w, z) + c(z, y) > c(x, z) + c(z, y)$.

Das bedeutet aber auch, daß Abbiegeverbote an einem Stop keine Bedeutung haben, wenn man an diesem Stop anhält. Dieser Effekt ist aber auch in der Praxis nicht unsinnig, da man ja normalerweise keine Angaben über die Lage des Anfahrpunktes relativ zu den ein- und ausgehenden Straßen hat. Der Aufwand, den man hat, um diesen Punkt zu erreichen, kann in die Standzeit mit einbezogen werden.

Kapitel 5

Heuristiken für TSP

Für die heuristische Lösung des TSP verwendet man i.a. einen zweistufigen Algorithmus, bestehend aus Eröffnungsverfahren und Verbesserungsverfahren.

5.1 Eröffnungsverfahren

Beim Eröffnungsverfahren wird eine Anfangstour berechnet, die in einem 2. Schritt (Verbesserungsverfahren) noch weiter verbessert wird. Es gibt dazu verschiedene Methoden (siehe [28, 6]). Die Worst-Case-Abschätzungen für die Länge der Tour gelten nur für den Fall, daß der Graph die Dreiecksungleichung erfüllt und symmetrisch ist, d.h. $c(v_1, v_2) = c(v_2, v_1),\ \forall v_1, v_2 \in V$. Sie sind ebenfalls aus [28, 6] übernommen.

Die folgenden Algorithmen gehen von Voraussetzung 2.1 aus und nehmen zusätzlich an, daß $V = S = \{1, \ldots, k\}$ und daß G vollständig ist.

5.1.1 Nearest Neighbour

Algorithmus 5.1 [Nearest Neighbour]

Beginne mit einer Tour, die aus einem beliebigem Stop besteht
while (es gibt noch Stops, die nicht zur Tour gehören)
    Wähle aus allen Stops, die noch nicht zur Tour gehören, denjenigen aus, der zum letzten Stop in der Tour am nächsten liegt und hänge ihn an diesen an
endwhile
Verbinde den ersten und letzten Stop der Tour miteinander, um das Ergebnis (Tour π) zu erhalten

Komplexität: Um den nächsten Stop zu einem gegebenen Stop zu finden, muß man im wesentlichen alle (k) Stops betrachten. Da dies insgesamt k-mal passiert, ergibt sich eine Komplexität von $\mathcal{O}(k^2)$.

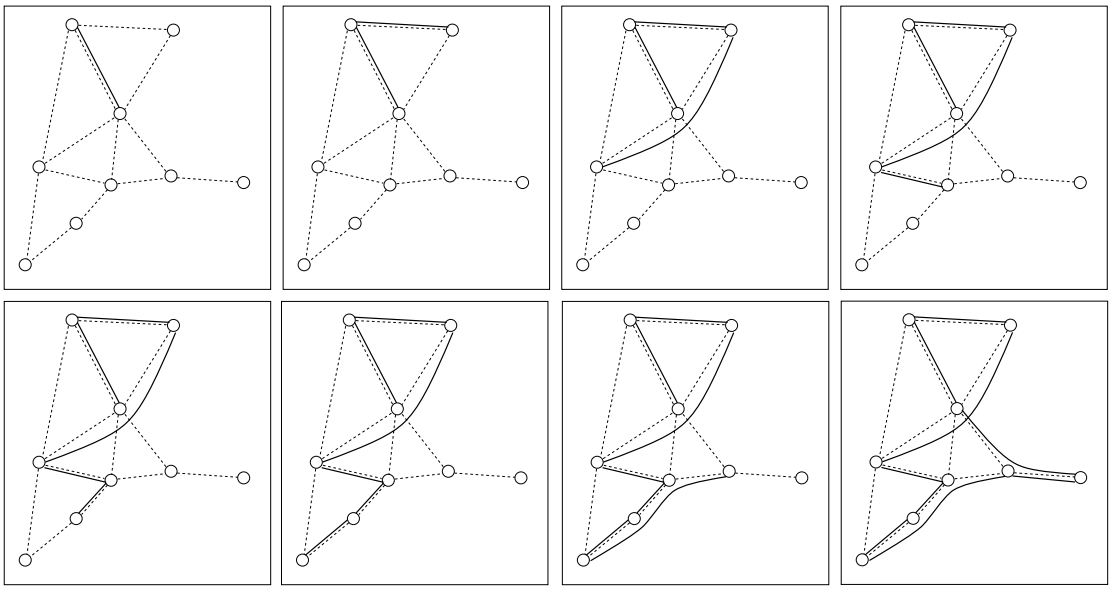


Abbildung 5.1: Momentaufnahmen beim Ablauf des Nearest-Neighbour-Algorithmus

Worst Case:

$$\frac{C(\pi)}{C(\text{opt. Tour})} \leq \frac{1}{2} \cdot k + \frac{1}{2}$$

5.1.2 Nearest Insertion

Algorithmus 5.2 [Nearest Insertion]

Beginne mit einer Tour, die aus einem beliebigen Stop besteht
Wähle einen Stop, so daß der Abstand zum ersten Stop minimal ist
und füge ihn in die Tour ein
while (es gibt noch Stops, die nicht zur Tour gehören)
    Von den Stops, die nicht zur Tour gehören, wähle denjenigen aus, der am nächsten zu irgendeinem Stop in der bereits gebildeten Tour ist
    Suche diejenige Kante $e = (s_1, s_2) \in E$, bei der die kleinsten zusätzlichen Kosten $c(s_1, s) + c(s, s_2) - c(s_1, s_2)$ entstehen, wenn s zwischen s_1 und s_2 in die Tour eingefügt wird
    Füge s zwischen s_1 und s_2 ein
endwhile

Implementierung: In einer Tabelle wird für jeden Stop die kürzeste Entfernung zu den Stops der bereits gebildeten Tour gespeichert. Diese Tabelle wird bei jedem neu zur Tour hinzugefügten Stop erneuert.

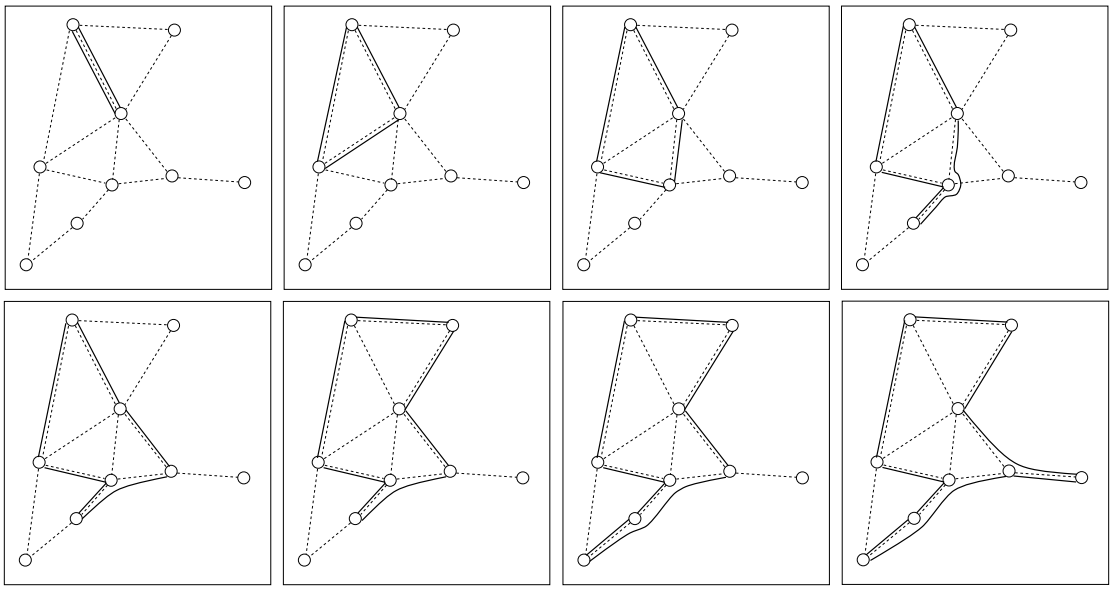


Abbildung 5.2: Momentaufnahmen beim Ablauf des Nearest-Insertion-Algorithmus

Komplexität: Das Erneuern der Tabelle kostet, genauso wie das Suchen des nächsten Stops in dieser Tabelle, $\mathcal{O}(k)$ Schritte. Da diese Operation k-mal erledigt werden muß, ergibt sich eine Komplexität von $\mathcal{O}(k^2)$.

Worst Case:

$$\frac{C(\pi)}{C(\text{opt. Tour})} \leq 2$$

5.1.3 Farthest Insertion

Diese Heuristik ähnelt sehr der Nearest Insertion, mit dem Unterschied, daß anstatt des nächsten Stops immer der am weitesten entfernte Stops gewählt wird. Diesem—intuitiv zunächst seltsam erscheinendem—Vorgehen liegt die Idee zugrunde, daß zunächst der generelle Verlauf der Tour skizziert werden soll und erst danach die Details festgelegt werden.

Algorithmus 5.3 [Farthest Insertion]

Beginne mit einer Tour, die aus einem beliebigen Stop besteht
Wähle einen Stop, so daß der Abstand zum ersten Stop maximal ist
und füge ihn in die Tour ein
while (es gibt noch Stops, die nicht zur Tour gehören)
    Von den Stops, die nicht zur Tour gehören, wähle denjenigen aus, der am weitesten von allen Stop in der bereits gebildeten Tour entfernt ist

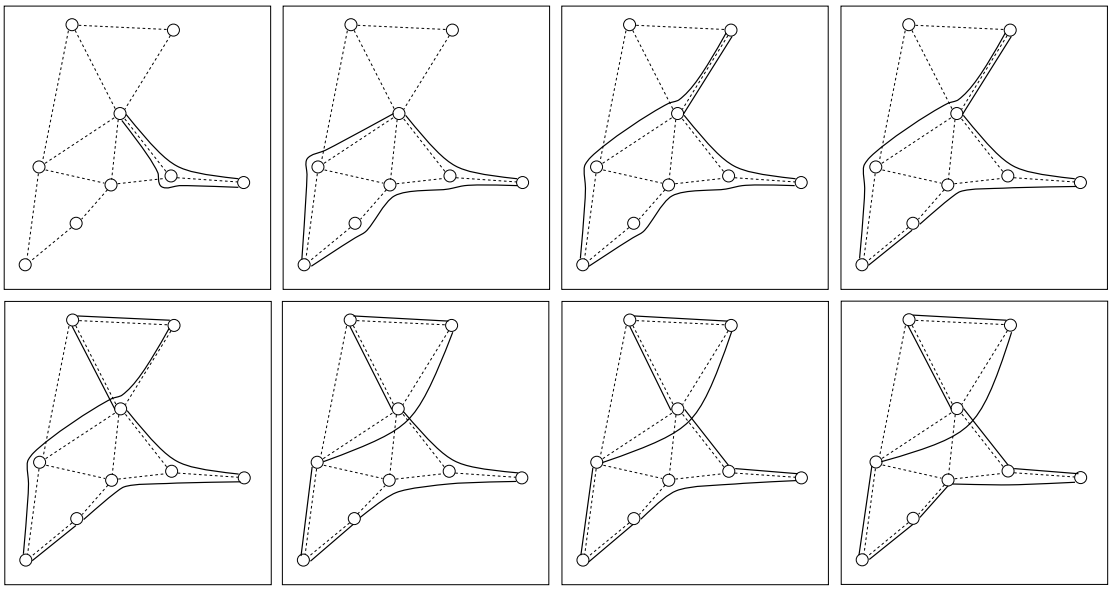


Abbildung 5.3: Momentaufnahmen beim Ablauf des Farthest-Insertion-Algorithmus

Suche diejenige Kante $e = (s_1, s_2) \in E$, bei der die kleinsten zusätzlichen Kosten $c(s_1, s) + c(s, s_2) - c(s_1, s_2)$ entstehen, wenn s zwischen s_1 und s_2 in die Tour eingefügt wird
Füge s zwischen s_1 und s_2 ein
endwhile

Implementierung und Komplexität: analog zu Nearest Insertion (5.1.2).
Worst Case:

$$\frac{C(\pi)}{C(\text{opt. Tour})} \leq \lceil \log k \rceil + 1$$

5.1.4 Savings

Das Savings-Verfahren für VRP ist in Abschnitt 6.2 beschrieben. Wenn man hier ein Fahrzeug einsetzt, dessen Kapazität gleich der Summe der Transfermengen aller Stops ist, kann man mit Hilfe dieses Verfahrens eine einzelne Tour bilden.

Komplexität: Die aufwendigste Operation ist das Sortieren einer Savings-Matrix mit k^2 Einträgen. Es ergibt sich daher eine Komplexität von $\mathcal{O}(k^2 \cdot \log k)$ (siehe auch Abschnitt 6.2).

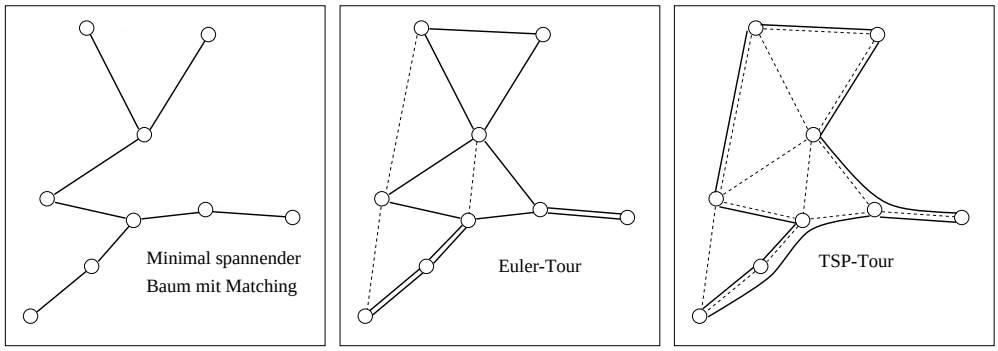


Abbildung 5.4: Momentaufnahmen beim Ablauf der Heuristik von Christofides

5.1.5 Christofides' Heuristik

Von Christofides stammt die folgende Heuristik, die ein erstaunlich gutes Worst-Case-Verhalten hat. Der angegebene Algorithmus ist zunächst nur auf symmetrische Graphen anwendbar. In [2] findet man eine Variante dieses Verfahren, das auch auf gerichteten, asymmetrischen Graphen arbeitet.

Algorithmus 5.4 [Christofides Heuristik]

Finde einen minimal spannenden Baum T von G
Finde alle Stops mit ungeradem Grad in T und berechne ihre Abstände
bezüglich der ursprünglichen Kostenfunktion c von G
Führe ein minimales perfektes Matching auf diesen Stops ungeraden Grades
durch und füge die erhaltenen Kanten zu T hinzu, um eine
Euler-Tour durch den Graphen zu erhalten
Durchlaufe die Euler-Tour und überspringe dabei diejenigen Stops,
die bereits passiert wurden

Komplexität: Der aufwendigste Teil der Heuristik ist i.a. die Durchführung des minimalen Matchings, welches eine Komplexität von $\mathcal{O}(k'^3)$ hat, wobei k' die Anzahl der Knoten ungeraden Grades in T ist. (Für Matching-Algorithmen siehe z.B. [1, 34]).

Die Berechnung des minimal spanndenden Baumes erfordert $\mathcal{O}(k^2 \cdot \log k)$ Schritte (z.B. mit dem Union-Find-Algorithmus), das Berechnen und der Durchlauf der Euler-Tour $\mathcal{O}(k)$ Schritte.

Damit ergibt sich als Gesamtkomplexität $\mathcal{O}(k'^3 + k^2 \cdot \log k)$.

Worst Case:

$$\frac{C(\pi)}{C(\text{opt. Tour})} \leq \frac{3}{2}$$

5.2 Verbesserungsverfahren p-opt

Bei einem p-opt Verbesserungsverfahren wird die Tour solange optimiert, bis eine Vertauschung von p beliebigen Kanten mit p zulässigen anderen keine Verbesserung mehr ergibt.

Diese Optimierung erfolgt einfach durch Ausprobieren (alle möglichen Vertauschungen ausführen).

In Abschnitt 6.5 ist ein 3-opt-Verfahren zur Verbesserung mehrerer Touren beschrieben. Dieses Verfahren läßt sich ohne große Veränderungen auch auf den Fall einer Tour anwenden.

Komplexität: Bei einem p-opt Verfahren versucht man, solange p Kanten gegen p andere zu vertauschen, bis sich eine Verbesserung ergibt. Dabei muß man im schlimmsten Fall $\mathcal{O}(k^p)$ Operationen durchführen (in einer Tour gibt es genausoviel Kanten wie Stops).

Nach einer erfolgten Verbesserung wird das p-opt-Verfahren jedoch wieder von neuem gestartet. Dabei erhöht sich die Komplexität in Abhängigkeit von der Anzahl der Verbesserungen.

Kapitel 6

Heuristiken für VRP

Heuristiken zur Tourenplanung können folgendermaßen unterteilt werden (nach [15]):

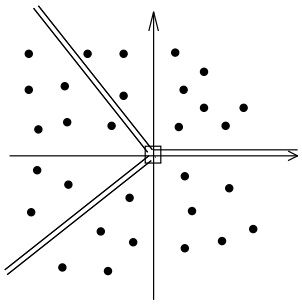
1. Sukzessivverfahren
   (a) Route-first-Cluster-second (z.B. Giant Tour, Abschnitt 6.4)
   (b) Cluster-first-Route-second (z.B. Sweep, Abschnitt 6.1)
2. Parallelverfahren
   (a) Konstruktionsverfahren (z.B. Savings, Abschnitt 6.2)
   (b) Verbesserungsverfahren (z.B. 3-opt, Abschnitt 6.5)

Bei einem *Route-first-Cluster-second-Verfahren* wird zunächst eine Tour durch alle Stops gebildet, anschließend wird diese Tour mit einer Heuristik in Subtouren aufgespalten. Das *Cluster-first-Route-second-Verfahren* geht genau den umgekehrten Weg: Die Stops werden zu Gruppen (Clustern) zusammengefaßt und durch jeden dieser Cluster wird eine Tour gelegt.

Bei *Konstruktionsverfahren* werden die Zuordnung der Stops zu den Touren und die Routenbildung gleichzeitig durchgeführt. *Verbesserungsverfahren* gehen von einem bereits bestehenden Tourenplan aus und versuchen, ihn durch geeignete Modifikationen zu verbessern.

Die im folgenden besprochenen Verfahren sind oben in ihre jeweilige Klasse eingeordnet. Clustering (Abschnitt 6.3) ist eine Kombination aus einem Cluster-first-Route-second und einem Konstruktionsverfahren.

6.1 Sweep



Der Sweep-Algorithmus wurde ursprünglich von Gillett und Miller vorgeschlagen (siehe [23]).

Beim Sweep werden jedem Stop $s \in S$ seine Koordinaten $x(s), y(s)$ auf der euklidischen Ebene zugeordnet. Dabei liegt das Depot im Ursprung des Koordinatensystems.

Es geht nun darum, die Ebene so in Winkelsegmente zu zerlegen, daß alle Stops in einem Winkelsegment, zu einer Tour zusammengefaßt werden können. Dabei ist auf die Einhaltung der Kapazitäts- und Zeitschranken zu achten. Anschließend kann in jedem Segment eine Tour gebildet werden.

Algorithmus 6.1 [Sweep ohne Zeitschranken]

for every Stop $s \in S$
  Berechne den Polarwinkel $\phi(s)$
endfor
Ordne die Stops nach aufsteigenden Polarwinkeln,
so daß $S = \{s'_1, \ldots, s'_k\}$ mit $\phi(s'_i) \leq \phi(s'_j)$ für $i \leq j$
$p := 1, v := 1$
while (es gibt noch Stops, die keinem Fahrzeug zugeteilt sind)
  $m := 1$
  while $(m \leq m(f_v))$
    Dem Fahrzeug f_v werden im m-ten Einsatz die Stops $s'_p, s'_{p+1}, \ldots, s'_u$
    zugeteilt, wobei $u \leq k$ der größte Index ist, so daß $\sum_{i=p}^{u} b(s'_i) \leq q(f_v)$
    $p := u + 1, m := m + 1$
  endwhile
  $v := v + 1$
endwhile
Bilde für jedes Fahrzeug in jedem Einsatz eine Tour durch die ihm
zugeordneten Stops mit einer beliebigen TSP-Heuristik

Komplexität: Der aufwendigste Teil des Algorithmus ist, neben dem Sortieren der Stops nach Polarwinkeln, die TSP-Heuristik. Sortieren benötigt $\mathcal{O}(k \cdot \log k)$

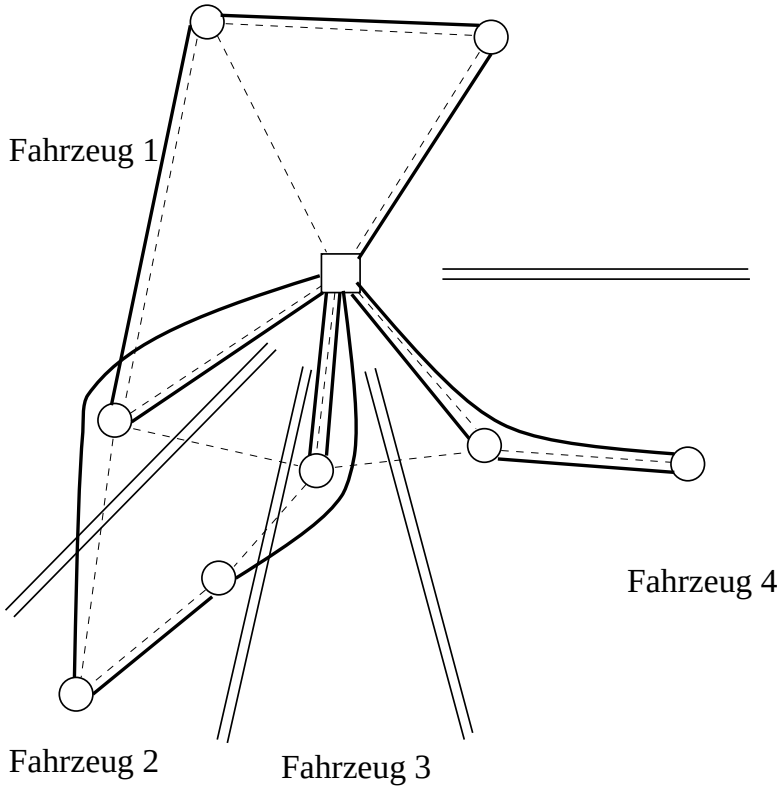


Abbildung 6.1: Ergebnis der Sweep-Heuristik auf dem Beispiel-Graphen

Operationen, die TSP-Heuristik benötigt je nach Algorithmus i.a. quadratische bzw. kubische Zeit in der Anzahl der Stops einer Tour (siehe Kapitel 5).

Maximal können $\max_T := |\{(f_v, m) : f_v \in F, m \leq m(f_v)\}|$ Touren entstehen.

Wenn man die realistische Annahme macht, daß sich die Anzahl Stops pro Tour im wesentlichen nur um einen konstanten Faktor unterscheidet, so muß man im Worst-Case mit $\mathcal{O}(\max_T \cdot \left(\frac{k}{\max_T}\right)^2)$ bis $\mathcal{O}(\max_T \cdot \left(\frac{k}{\max_T}\right)^3)$ Schritten für die gesamte Tourbildung rechnen.

Implementierte Varianten: Indem man die Stops in oder gegen den Uhrzeigersinn sortiert und dabei von jedem Stop aus beginnt, kann man verschiedene Variationen durchspielen und von diesen das beste Ergebnis übernehmen. Die Komplexität erhöht sich dadurch um den Faktor k.

Mögliche Varianten: Die Anordnung der Fahrzeuge und die Reihenfolge ihres Einsatzes bleibt bei diesem Algorithmus dem Anwender überlassen. Man könnte sich jedoch auch vorstellen, die Fahrzeuge nach absteigender Kapazität zu sortieren, um somit im Schnitt weniger Touren zu erhalten.

Zeitschranken: Die Clusterung der Stops entsprechend der Zeitschranken ist wesentlich komplizierter als die Clusterung nach Kapazität. Daher bilde ich zunächst Winkelsegmente entsprechend der Kapazität, bilde eine Tour, überprüfe anschließend die Einhaltung der Zeitschranken und übernehme nur diejenigen Stops, die noch ins Zeitfenster passen:

Algorithmus 6.2 [Sweep mit Zeitschranken]

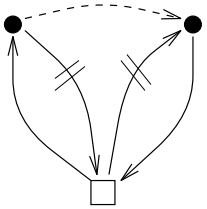
for every Stop $s \in S$
  Berechne den Polarwinkel $\phi(s)$
endfor
Ordne die Stops nach aufsteigenden Polarwinkeln,
so daß $S = \{s'_1, \ldots, s'_k\}$ mit $\phi(s'_i) \leq \phi(s'_j)$ für $i \leq j$
$v := 1$
while (es gibt noch Stops, die keinem Fahrzeug zugeteilt sind)
  $m := 1$
  while $(m \leq m(f_v))$
    Sei t die restliche Zeit des Fahrzeugs f_v
    Dem Fahrzeug f_v werden im m-ten Einsatz alle noch nicht zugeteilten Stops $s'_i, i \leq u$ zugeteilt, wobei $u \leq k$ der größte Index ist, so daß
    $$\sum_{s'_i \text{ nicht zugeteilt}, i \leq u} b(s'_i) \leq q(f_v)$$
    Bilde eine Tour durch die soeben zugeordneten Stops mit einer beliebigen TSP-Heuristik
    if (Zeitbedarf der Tour überschreitet die Zeitschranke t) **then**
      Beende die Tour nach dem Stop, von dem aus die Rückkehr zum Depot gerade noch innerhalb der vorgegebenen Zeitschranke möglich ist

Gib die übriggebliebenen Stops wieder frei, sie sind jetzt keinem Fahrzeug mehr zugeordnet
endif
$m := m + 1$
endwhile
$v := v + 1$
endwhile

Implementierte Varianten: Um das Ergebnis zu verbessern, kann man auf eine "abgeschnittene" Tour noch einmal eine TSP-Heuristik anwenden, um ihre Länge noch zu verkürzen.

In unserem Beispiel (Abbildung 6.1) werden die Zeitschranken bereits eingehalten, daher erzielt der Sweep mit Zeitschranken dasselbe Ergebnis.

6.2 Savings



Der Savings-Algorithmus—ein typisches Konstruktionsverfahren—wurde von Clarke und Wright entwickelt (siehe [11]). Er geht von einer Ausgangssituation aus, bei der jeder Stop einzeln vom Depot aus angefahren wird. Diese Pendeltouren werden, abhängig von den Ersparnissen (Savings), die eine Vereinigung liefert, zusammengefügt, solange die Kapazitäts- und Zeitschranken eingehalten werden.

Definition 6.1 (Saving) *Das Saving $s(s_1, s_2)$ von einem Paar von Stops s_1, s_2 wird durch $s(s_1, s_2) := c(s_1, D) + c(D, s_2) - c(s_1, s_2)$ berechnet.*

Es bezeichnet also die Ersparnis, die man hat, wenn man die Stops nicht mehr einzeln, sondern direkt hintereinander anfährt.

Tabelle 6.1 zeigt die Savings-Matrix des Graphen aus Abbildung 2.3.

Im folgenden wird angenommen, daß man auf eine Liste $\mathcal{S}$ aller Savings zurückgreifen kann, welche absteigend geordnet ist. Die Savings können leicht berechnet werden, wenn man eine Stop-Distanzmatrix besitzt, wie in Kapitel 4 beschrieben.

	2	3	4	5	6	7	8	9
2	—	1,5	0	0	0	0	0	0
3	1,5	—	0	0	0	0	0	0
4	0	0	—	2	1	3	2	1
5	0	0	2	—	2	2,5	3	2
6	0	0	1	2	—	1,5	2	4
7	0	0	3	2,5	1,5	—	4,5	1,5
8	0	0	2	3	2	4,5	—	2
9	0	0	1	2	4	1,5	2	—

Tabelle 6.1: Vollständige Savings-Matrix des Beispiel-Graphen

6.2.1 Simultaner Savings

Alle Touren werden parallel gebildet, es werden immer die Touren mit dem augenblicklich größten Saving zusammengefügt, solange noch Savings zur Verfügung stehen.

Touren dürfen aber nur zusammengefügt werden, wenn sich ein freies Fahrzeug entsprechender Kapazität findet, oder wenn die Transfermenge die minimale Kapazität der Fahrzeuge nicht übersteigt.

Savings, die aufgrund fehlender Fahrzeuge keine Berücksichtigung finden konnten, werden behalten. In weiteren Durchläufen wird versucht, sie erneut zu verwenden, und ein evtl. freigewordenes, passendes Fahrzeug zu besetzen.

Algorithmus 6.3 [Simultaner Savings ohne Zeitschranken]

Bilde die Menge aller Pendeltouren: $M := \{\pi : \{1\} \to S\}$,
wobei S die disjunkte Vereinigung aller $\pi(1), \pi \in M$ ist
Keine der Touren ist fest einem Fahrzeug oder einem Einsatz zugeordnet
$q_{\min} := \min_{f_v \in F}\ q(f_v)$

repeat
  Wähle das größte vorhandene Saving $s(s_1, s_2) \in \mathcal{S}$
  Suche die Touren $\pi_1, \pi_2 \in M$, die s_1 respektive s_2 enthalten
  if $(\pi_1 \neq \pi_2)$ **and**
    (s_1 ist der letzte Stop von π_1, s_2 der erste von π_2) **and**
    ($(Q(\pi_1) + Q(\pi_2) \leq q_{\min})$ **or**
    (es gibt ein Fahrzeug f, das entweder noch frei ist oder sich im Besitz von π_1 oder π_2 befindet mit $Q(\pi_1) + Q(\pi_2) \leq q(f)$))
  then
    Ersetze π_1, π_2 in M durch die Aneinanderreihung beider Touren
    if (π_1 oder π_2 besaßen Fahrzeuge) **then**

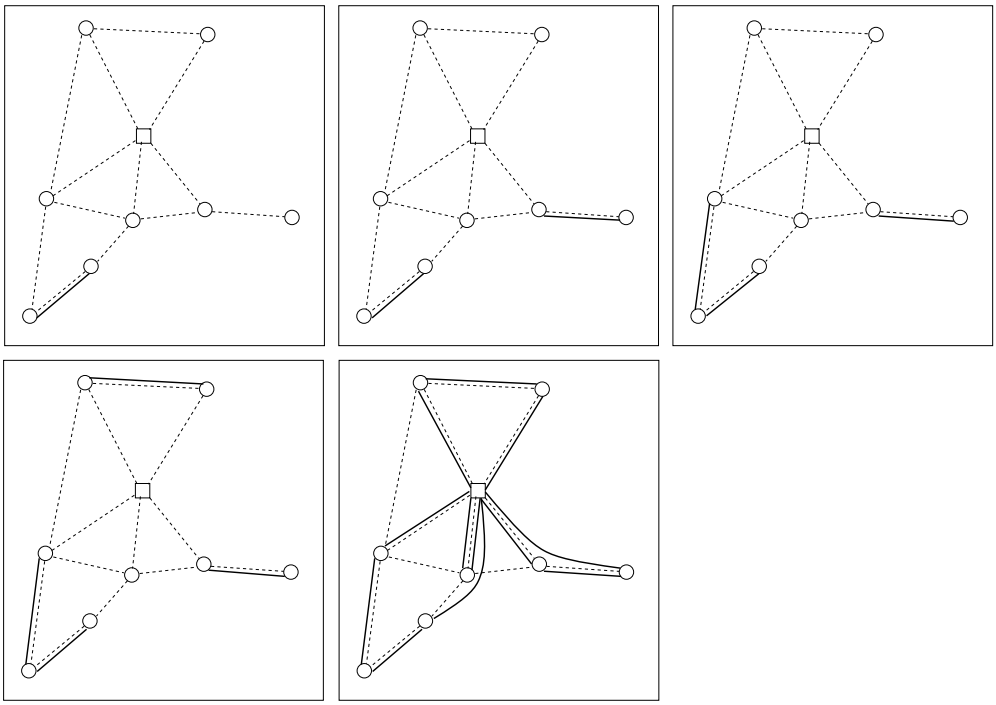


Abbildung 6.2: Ablauf des simultanen Savings auf dem Beispiel-Graphen

gib diese Fahrzeuge frei
endif
if $(Q(\pi_1) + Q(\pi_2) > q_{\min})$ **then**
Besetze von allen passenden freien Fahrzeugen das kleinste
endif
endif
if (das Saving wurde aus Kapazitätsgründen abgelehnt) **then**
Verändere $\mathcal{S}$ nicht
else
Streiche $s(s_1, s_2)$ aus $\mathcal{S}$
endif
until ($\mathcal{S}$ wird nicht mehr kleiner)

Das Endergebnis sind die Touren, die sich in M befinden

Implementierung: Die Savings werden immer in absteigender Reihenfolge durchlaufen. Es genügt daher, sie in einer linearen Liste zu halten. Die Stops werden in einem Array unter ihrer Nummer abgelegt.

Der einzige kritische Schritt im Algorithmus ist die Suche nach π_1 bzw. π_2. Diese Suche kann dadurch verkürzt werden, daß jeder Stop einen Verweis auf die

Tour enthält, in der er sich im Augenblick befindet. Diese Verweise müssen beim Zusammenfügen von Touren natürlich aktualisiert werden. Außerdem wird für jede Tour deren Kapazität abgespeichert.

Das Durchsuchen der Fahrzeuge nach einer ausreichenden Kapazität benötigt wegen der meist geringen Zahl der Fahrzeuge nur wenig Zeit, so daß ich darauf verzichtet habe, die Fahrzeuge in einer speziellen Datenstruktur abzuspeichern. Sie werden einfach in einer linearen Liste gehalten.

Komplexität: Die Savings-Liste $\mathcal{S}$ enthält $\mathcal{O}(k^2)$ Einträge. Im schlimmsten Fall muß man für jeden Saving den Versuch machen, ein passendes Fahrzeug zum Vereinigen der Touren zu finden. Die Liste $\mathcal{S}$ einmal zu durchlaufen kostet daher $\mathcal{O}(\max_T \cdot k^2)$ Operationen. Sie ist nach einem Durchlauf i.a. drastisch reduziert und es erfolgen nach meinen Erfahrungen nur sehr wenige Iterationen, so daß sich die Komplexität im Average Case nicht mehr wesentlich erhöht.

Zeitschranken: Dadurch, daß alle Touren parallel gebildet werden und am Anfang noch nicht bekannt ist, welcher Stop von welchem Fahrzeug bedient wird, ist es sehr schwer, die Einhaltung der Zeitschranken zu überprüfen. Alle Versuche, die ich diesbezüglich unternommen habe (z.B. jede Tour ist von Anfang an einem Fahrzeug fest zugeordnet), erwiesen sich als unpraktikabel. Aus diesem Grund habe ich das Verfahren ohne diese Einschränkung implementiert.

Implementierte Varianten: Um die Touren noch weiter zu verbessern, kann man die durch den Savings gebildeten Einzel-Touren noch einmal einer TSP-Heuristik unterziehen, und überprüfen, ob sich das Ergebnis dadurch verbessert.

Für den Savings-Algorithmus benötigt man im allgemeinen nicht alle möglichen Savings, sondern nur diejenigen zwischen Paaren von Stops, die "nah zusammen" liegen. Es wird angenommen, daß Stops die "weit auseinander" liegen, nicht nacheinander in einer Tour angefahren werden.

Es sei also d' eine feste Cut-off-Distanz. Es werden nur diejenige Savings in die Savings-Liste aufgenommen, für die gilt: $c(s_1, s_2) \leq d'$.

Mögliche Varianten: Indem man Savings (vor allem möglichst große Savings) aus der Savings-Liste $\mathcal{S}$ streicht, kann man verhindern, daß diese Savings benutzt werden. Damit zwingt man den Algorithmus dazu, andere Touren zu bilden, die u.U. stark unterschiedlich sein können. Man kann den Algorithmus mit verschiedenen Savings-Listen durchführen und das beste Ergebnis übernehmen.

6.2.2 Sequentieller Savings

Im Gegensatz zum simultanen Savings können beim sequentiellen Savings die Zeitschranken leicht eingehalten werden. Die Idee besteht darin, die Touren nicht alle gleichzeitig, sondern nacheinander zu bilden.

Die Menge der Pendeltouren M sei definiert wie beim simultanen Savings.

Algorithmus 6.4 [Sequentieller Savings mit Zeitschranken]

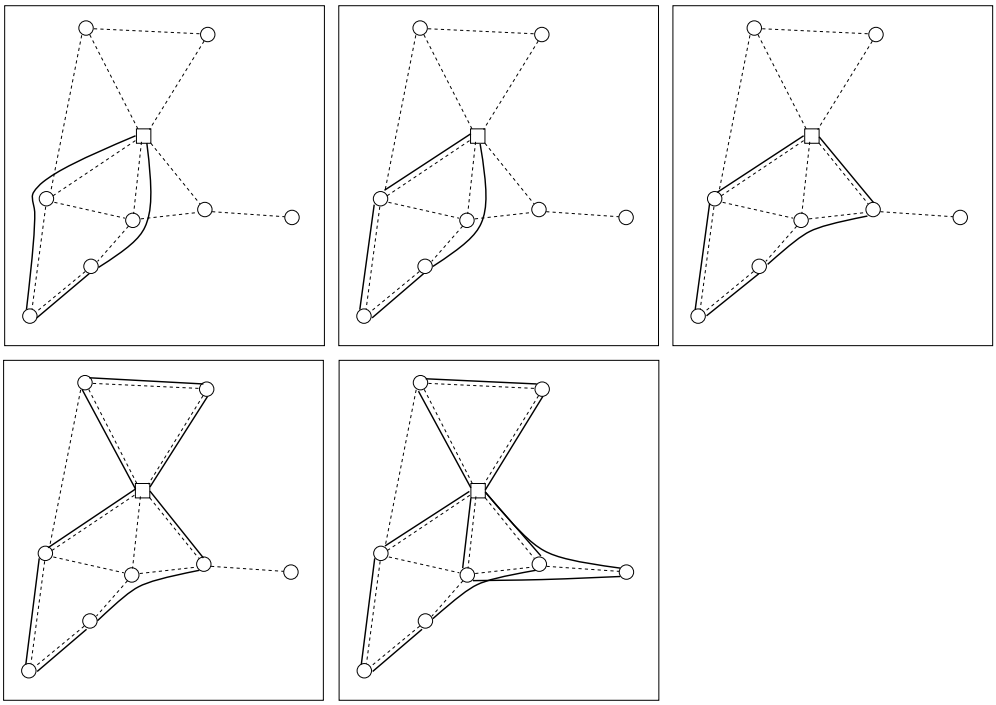


Abbildung 6.3: Ablauf des sequentiellen Savings auf dem Beispiel-Graphen

for every Fahrzeug $f \in F$
  Beginne die Tour zu f, π_f, mit dem größten Saving aus $\mathcal{S}$, dessen Stops noch in Pendeltouren bedient werden
  while (es gibt noch verwendbare Savings in $\mathcal{S}$)
    Suche das größte vorhandene Saving $s(s_1, s_2) \in \mathcal{S}$, das die Tour π_f um einen noch nicht bedienten Stop verlängert (der Stop kann vorne oder hinten an die Tour angehängt werden)
    Suche die Touren $\pi_1, \pi_2 \in M$, die s_1 respektive s_2 enthalten
    if $(Q(\pi_1) + Q(\pi_2) \leq q(f))$ **and**
      $(T(\pi_1) + T(\pi_2) - t_E(s_1, D) - t_E(D, s_2) + t_E(s_1, s_2) \leq t_F(f))$
    then
      Verlängere π_f zu einer Vereinigung von π_1 und π_2
    endif
    Lösche das Saving aus $\mathcal{S}$
  endwhile
endfor

Implementierung: Die Datenstrukturen sind analog zum simultanen Savings aufgebaut. Nur wird hier zu jeder Tour neben der Kapazität noch die Zeit mitgeführt. Zu Anfang muß der Zeitbedarf jeder Pendeltour berechnet werden.

Komplexität: Im wesentlichen muß für jedes Fahrzeug einmal die Savings-Liste $\mathcal{S}$ durchlaufen werden. Daher ergibt sich für die Komplexität wie beim simultanen Savings: $\mathcal{O}(\max_T \cdot k^2)$.

Implementierte Varianten: Auch hier wird versucht, die Einzel-Touren durch eine TSP-Heuristik nochmals zu verbessern.

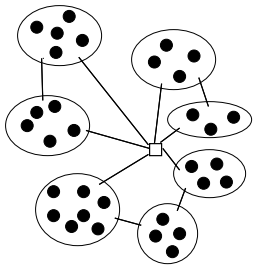
Wie beim simultanen Savings genügt es, wenn nur Savings verwendet werden, die kleiner als ein vorher festgelegte Cut-off-Distanz d' sind.

Als Alternative zum oben vorgestellten Algorithmus kann man die Abbruchbedingung der **while**-Schleife modifizieren. Die Schleife wird bereits dann abgebrochen, wenn die letzte Verlängerung der Tour nur aus Zeit- oder Kapazitätsgründen gescheitert ist.

Mögliche Varianten: Die Anordnung der Fahrzeuge und die Reihenfolge ihres Einsatzes bleibt—wie beim Sweep—dem Anwender überlassen. Man könnte die Fahrzeuge jedoch auch nach absteigender Kapazität sortieren, um somit im Schnitt weniger Touren zu erhalten.

Außerdem können—wie beim simultanen Savings—verschiedene Durchläufe durch Streichen von Einträgen der Savings-Liste durchgeführt werden.

6.3 Clustering mit anschließendem parallelen Savings



6.3.1 Verfahren zur Clusterbildung

Definition 6.2 (Clustering) *Ein Clustering der Stop-Menge S ist eine Partition von S. Ein Cluster ist ein Element dieser Partition, d.h. eine Teilmenge von S.*

Dahinter steckt die noch etwas vage Idee, daß die Stops eines Clusters möglichst "nah zusammen" liegen sollen und daß der Cluster eine vorgegebene Kapazitätsschranke q und Radiusschranke r einhalten soll. Konkreter wird diese Idee, wenn man den Algorithmus zur Clusterbildung betrachtet:

Algorithmus 6.5 [Clustering]

while(es gibt noch Stops in S)
  Wähle einen Kern-Stop und betrachte ihn als einelementigen Cluster C
  repeat
    Wähle den zum Cluster nächsten Stop s', der noch nicht zugeordnet wurde
    if (der Cluster überschreitet C die Kapazitäts- und Radiusbeschränkung nicht, d.h. $\sum_{s \in C \cup \{s'\}} b(i) \leq q$, $\text{radius}(C \cup \{s'\}) \leq r$)
    then
      füge s' zum bisher gebildeten Cluster C hinzu
    endif
  until (der letzte Stop hat die Beschränkungen des Clusters überschritten)
endwhile

Um den Kern-Stop eines Clusters zu wählen, kommen verschiedene Verfahren in Frage, z.B:

- Wähle unter den noch nicht geclusterten Stops denjenigen Stop $s \in S$ mit der maximalen Entfernung vom Depot $c(D, s)$
- Wähle den Stop $s \in S$ mit der größten Transfermenge $b(s)$

Ebenso gibt es verschiedene Möglichkeiten, um den nächsten Stop zum bisher gebildeten Cluster zu finden. Alles hängt davon ab, wie man den Abstand eines Stops zu einem Cluster definiert. Entsprechend ergibt sich auch die Definition des Radius eines Clusters. Als Definitionen wären z.B. möglich:

- Koordinaten des Schwerpunkts des bisher gebildeten Clusters berechnen (alle Stops werden als Punkte gleicher Masse betrachtet).

  Koordinaten eines Stops s seien $x(s), y(s)$. Der Schwerpunkt $\tilde{s}_C$ eines Clusters C sei definiert als:

  $$\tilde{s}_C := (x(\tilde{s}_C), y(\tilde{s}_C)) = (\frac{\sum_{s \in C} x(s)}{|C|}, \frac{\sum_{s \in C} y(s)}{|C|})$$

  Die Entfernung eines Stops s zum Cluster ist die Luftlinienentfernung:

  $$d(s, C) := \sqrt{(x(\tilde{s}_C) - x(s))^2 + (y(\tilde{s}_C) - y(s))^2}$$

- Entfernung zum Cluster wird als Entfernung zum Kern-Stop auf dem Graphen definiert. Sei $\hat{s}_C$ der Kern-Stop des Clusters C.

  $$d(s, C) := c(s, \hat{s}_C)$$

Der Radius eines Clusters wird natürlicherweise definiert als:

$$\text{radius}(C) := \max_{s \in C} \{d(s, C)\}$$

6.3.2 Cluster durch parallen Savings zusammenfügen

Wenn man nun den Abstand zweier Cluster definiert, abstrahiert und die Cluster im weiteren als Stops betrachtet, so kann man das Problem durch einen Savings-Algorithmus lösen.

Definition 6.3 (Cluster-Graph) *Sei $G' := (V', E')$ ein gerichteter, vollständiger Graph mit*

$$E' := S' = \{C : C \text{ Cluster}\}$$

Der Abstand zwischen zwei Clustern C_1, C_2 wird definiert als der minimale Abstand ihrer Stops.

$$c(C_1, C_2) := \min_{s_1 \in C_1, s_2 \in C_2} \{c(s_1, s_2)\}$$

Analog dazu wird der Zeitabstand t_E zweier Cluster definiert.
Die Transfermenge eines Cluster ist:

$$b(C) := \sum_{s \in C} b(s)$$

Die Haltezeit an einem Cluster C ist:

$$t_S(C) := \sum_{s \in C} t_S(s)$$

Wenn man alle anderen Definition wie in Abschnitt 2.2 beläßt, hat man nun das VRP auf ein neues VRP mit weniger Stops reduziert. Hierauf wendet man nun günstigerweise den simultanen Savings an.

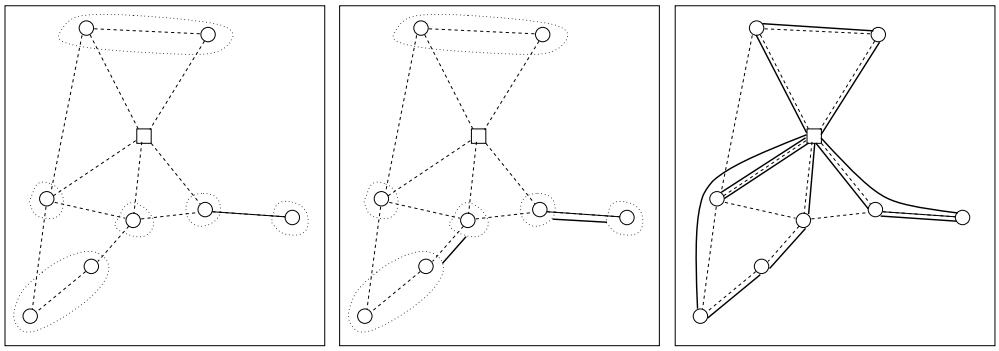
Implementierung: Zu dem Array der Stops wird ein zusätzlicher Eintrag hinzugefügt, der die Zugehörigkeit eines Stops zu einem Cluster angibt.

Die Entfernungen zwischen den Stops werden in einer Distanzmatrix gespeichert (berechnet wie in Kapitel 4 beschrieben). Die Berechnung der Abstände zwischen den Clustern erfordern ein einmaliges Durchlaufen dieser Distanzmatrix. Dabei wird für jedes Paar von Stops überprüft, ob ihre Entfernung kleiner als die bisher kleinste Entfernung ihrer Cluster ist.

Komplexität: Beim Clustering werden $|S| = k$ Stops zugeordnet. Dabei müssen jeweils $\mathcal{O}(|S|)$ Stops durchlaufen werden, um den jeweils nächsten zu finden, damit ergibt sich eine Komplexität $\mathcal{O}(k^2)$. Das Durchlaufen der Stop-Distanzmatrix zur Berechnung der Cluster-Distanzmatrix erfordert ebenfalls $\mathcal{O}(k^2)$ Operationen. Und für das Sortieren der Savings zwischen Clustern benötigt man $\mathcal{O}(|S'|^2 \cdot \log |S'|)$ Zeit. (S' ist die Menge aller Cluster.)

Im weiteren Verlauf des Algorithmus gilt die Komplexität des simultanen Savings (siehe Abschnitt 6.2.1), wobei es jetzt $|S'|$ Stops gibt.

Zeitschranken: Die Berücksichtigung von Zeitschranken ist beim Clustering nicht implementiert. Da der Cluster-Algorithmus auf dem simultanen Savings aufbaut, können auch hier die Zeitschranken nur schwer integriert werden. (Siehe auch die Begründung beim simultanen Savings, Abschnitt 6.2.1).



Kern-Stop ist der Stop mit der aktuell größten Transfermenge, Abstand eines Stops vom Cluster ist der Abstand vom Kern-Stop auf dem Graphen

Abbildung 6.4: Ablauf des Clustering

6.3.3 Berücksichtigung der Tourlängen innerhalb der Cluster

Wird bei der Berechnung der Savings zwischen den Clustern die Strecke, die innerhalb der Cluster zurückzulegen ist, vernachlässigt, so ergeben sich verfälschte Savings.

Um dieses Problem zu umgehen, wird der Abstand zwischen zwei Clustern C_1 und C_2 $c(C_1, C_2)$ folgendermaßen definiert:

Seien $n_1 \in C_1$ und $n_2 \in C_2$ die Stops mit

$$c(n_1, n_2) = \min_{s_1 \in C_1, s_2 \in C_2} \{c(s_1, s_2)\}$$

Sei $f_i \in C_i,\ i \in \{1, 2\}$ der Stop, der vom Depot aus am nächsten liegt, d.h.

$$c(D, f_i) = \min_{s \in C_i} \{c(D, s)\}$$

Analog dazu ist $t_i \in C_i,\ i \in \{1, 2\}$ der Stop der zum Depot hin am nächsten liegt, d.h.

$$c(t_i, D) = \min_{s \in C_i} \{c(s, D)\}$$

(Siehe Abbildung 6.5.)

$$c(C_1, C_2) := c_{f_1,n_1}(C_1) + c(n_1, n_2) + c_{n_2,t_2}(C_2)$$

Dabei ist $c_{x,y}(C_1)$ die Länge (Kosten) der (durch eine Heuristik entstandenenen) Tour, die im Cluster C_1 von x zu y führt und alle Stops des Clusters besucht.

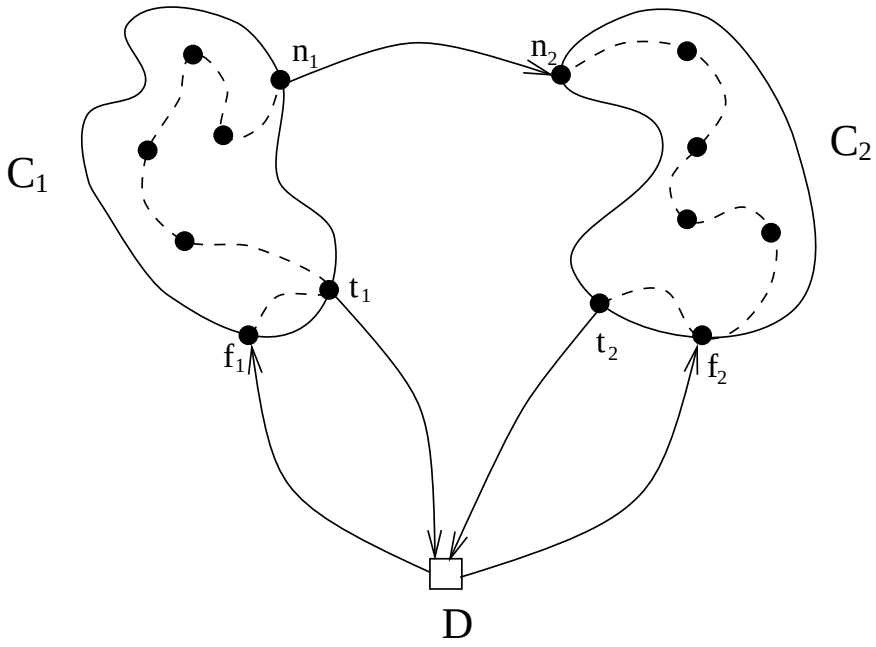


Abbildung 6.5: Berücksichtigung der Tourlängen innerhalb der Cluster

Der Abstand eines Clusters vom Depot ist definiert als:

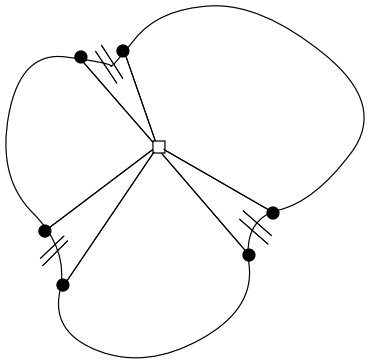
$$c(D, C_1) := c(D, f_1) + c_{f_1,t_1}(C_1)$$

$$c(C_1, D) := c_{f_1,t_1}(C_1) + c(t_1, D)$$

Auf diese Weise können die Savings berechnet werden. Während des Zusammenfügens von Touren können sich die Abstände jedoch ändern, da die Cluster dann nicht mehr in Richtung Depot, sondern in Richtung eines anderen Clusters verlassen werden.

Deshalb müssen alle Savings, die sich beim Zusammenfügen von zwei Touren ändern, aktualisiert und innerhalb der Ordnung der Savings-Liste verschoben werden, d.h. der Savings-Algorithmus muß angepaßt werden.

6.4 Giant Tour



Beim Giant-Tour-Verfahren bildet man eine Rundtour, die einmal durch alle Stops führt (Depot nicht eingeschlossen). Diese "Giant Tour" wird dann nach einer Heuristik in Teiltouren zerstückelt, die die Kapazitäts- und Zeitschranken einhalten.

Algorithmus 6.6 [Giant Tour mit Zeitschranken]

Bilde eine Giant-Tour durch alle Stops mit einer geeigneten TSP-Heuristik
Diese Giant Tour wird nun nach folgendem Verfahren zerlegt:

Variiere einen Faktor h
for every h
  for every Stop s der Giant Tour
    s ist der aktuelle Start-Stop
    Alle Fahrzeuge in allen Einsätzen stehen zur Verfügung
    repeat
      $s' := s$ sei der aktuelle End-Stop
      while (der Giant-Tour-Abschnitt zwischen Start-Stop und End-Stop übertrifft nicht die Kapazität des aktuell größten Fahrzeugs)
        Bilde für den End-Stop s' und jedes noch verfügbare Fahrzeug $f \in F$, das noch ausreichend Zeit besitzt, den Wert

$$w := \frac{s(s, N(s))}{\text{Mittl. Depotabstand aller Stops}} + h \cdot \frac{\text{Ungenutzte Kapazität}(f)}{q(f)}$$

        (Dabei ist $N(s)$ der Nachfolger von s in der Giant-Tour)
        $s' := N(s')$
      endwhile
      Der Giant-Tour-Abschnitt zwischen dem Start-Stop s und dem

End-Stop s' mit dem kleinsten w bildet jetzt eine Subtour
Ihr wird das kleinstmögliche noch vorhandene Fahrzeug zugeteilt
Der nächste Start-Stop s ist der Nachfolger von s': $N(s')$
until (das Ende der Giant Tour ist erreicht)
Die Länge der entstandenen Zerlegung wird berechnet
endfor
endfor

Die kürzeste Zerlegung für alle verwendeten Faktoren h und alle Start-Stops s wird als Ergebnis übernommen

Idee: Die Rückkehr zum Depot ist dann besonders günstig, wenn der Weg zurück zum Depot möglichst kurz, der Abstand zum nächsten Stop möglichst groß (d.h. das Saving möglichst klein) und das Fahrzeug bereits gut ausgelastet ist. Beide Kriterien werden durch h zueinander verschieden gewichtet.

Der Algorithmus ist nicht in der Lage, die Giant Tour optimal zu zerlegen. Im Fall, daß unbeschränkt viele Fahrzeuge von jeder Kapazität zur Verfügung stehen, ist eine solche optimale Zerlegung jedoch möglich (siehe [22, 25]).

Implementierung: Die Giant Tour wird durch ein zum Ring geschlossene Liste ihrer Stops dargestellt. Dadurch ist es möglich, die Giant Tour von jedem beliebigen Stop aus vollständig zu durchlaufen.

Komplexität: Für jeden Stop, an dem man eine Tour beenden möchte, müssen bis zu $\max_T$ verschieden Faktoren w berechnet werden. Dabei gibt es höchstens k Möglichkeiten für den End-Stop einer Tour. Bei einem Durchlauf durch die Giant Tour kann wiederum höchstens $\max_T$-mal eine neue Tour begonnen werden. Das ergibt für einen Durchlauf durch die Gesamttour eine Komplexität von $\mathcal{O}(\max_T^2 \cdot k)$.

Dieser Durchlauf wird von jedem der k Stops begonnen. Außerdem ist noch die Anzahl der verschiedenen h's zu berücksichtigen.

Insgesamt ergibt sich eine Komplexität von:

$$\mathcal{O}(\max_T^2 \cdot k^2 \cdot (\text{Anzahl der verschiedenen Faktoren } h))$$

Implementierte Varianten: Auch beim Giant-Tour-Algorithmus können die durch die Zerschlagung der Giant Tour entstandenen Subtouren noch durch eine TSP-Heuristik verbessert werden.

Im folgenden wird der Ablauf einer Giant-Tour-Iteration auf dem Beispiel-Graphen von Abbildung 2.3 dargestellt.
Die zugrundeliegende Giant Tour ist $3, 2, 4, 7, 8, 5, 6, 9$. Der Stop 3 wurde als Start-Stops gewählt, der verwendete Faktor h ist 2. Der mittlere Depotabstand aller Stops beträgt $2,5$.

Berechung der Werte w zur Bildung der ersten Tour:

Stop s	bestes Fahrzeug	$s(s, N(s))$	ungenutzte Kapazität	w
3	3	1,5	3000	1,8
2	3	0	500	0,2
4	1	3	2500	1,7
7	—		—	—

Berechung der Werte w zur Bildung der zweiten Tour:

Stop s	bestes Fahrzeug	$s(s, N(s))$	ungenutzte Kapazität	w
4	4	3	2000	2,0
7	1	4,5	3000	2,4
8	1	3	2000	1,6
5	—		—	—

Berechung der Werte w zur Bildung der dritten Tour:

Stop s	bestes Fahrzeug	$s(s, N(s))$	ungenutzte Kapazität	w
5	4	2	1000	1,2
6	—		—	—

Berechung der Werte w zur Bildung der vierten Tour:

Stop s	bestes Fahrzeug	$s(s, N(s))$	ungenutzte Kapazität	w
6	2	2	3000	2,0
9	2	0	2000	0,8

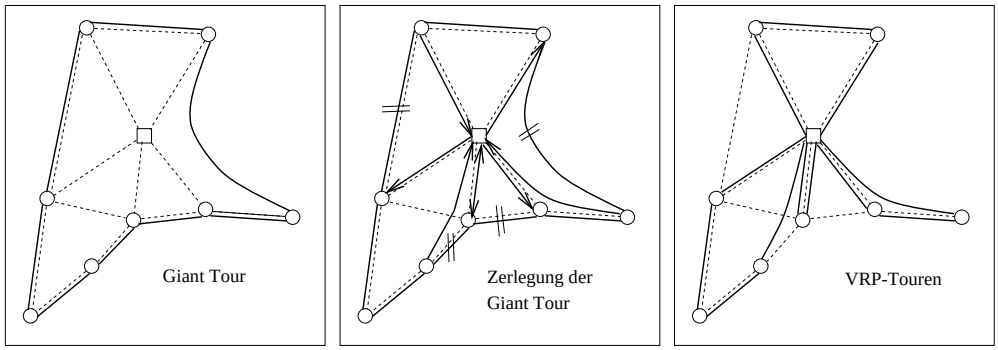
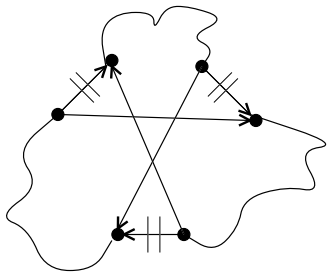


Abbildung 6.6: Ablauf und Ergebnis einer Iteration des Giant-Tour-Verfahrens

6.5 Austauschverfahren zur Verbesserung (3-opt-Verfahren)



Beim 3-opt-Verfahren wird versucht, durch Austausch von drei Kanten einer Tour durch drei andere eine Verkürzung der Tour zu erreichen. Ein 3-opt-Verfahren für eine Tour (TSP-Verbesserungsverfahren) ist beispielsweise in [15] beschrieben, die Idee zu einem 3-opt-Verfahren für mehrere Touren habe ich aus [25] übernommen.

Das 2-opt-Verfahren ist für gerichtete Graphen nicht geeignet, da Abschnitte von Touren bei diesem Verfahren "umgedreht" werden, d.h. in Gegenrichtung durchlaufen werden. Deshalb verwende ich im folgenden immer das 3-opt Verfahren.

Algorithmus 6.7 [3-opt]

Die gebildeten Touren werden zu einer Gesamttour aneinander gesetzt und jeweils durch einen Depot-Stop getrennt

for every (Kombination von drei Kanten innerhalb der großen Gesamttour)
  Die Gesamttour ist jetzt in vier Komponenten zerfallen
  if (durch die Vertauschung der mittleren beiden Komponenten
  ergibt sich eine Verbesserung) **then**
    Durch die Vertauschung können bis zu drei Touren betroffen sein
    if (die neuen Kapazitäten und der neue Zeitbedarf der Touren kann
      durch die vorhandenen Fahrzeuge abgedeckt werden)
    then
      führe die Vertauschung durch und beginne die **for-every**-Schleife von neuem
    endif
  endif
endfor

Eine Vertauschung durch 3-opt ist in Abbildung 6.7 dargestellt.

Wenn die Iteration endet können durch Ersetzung von drei beliebigen Kanten durch drei andere die Touren nicht weiter verbessert werden.

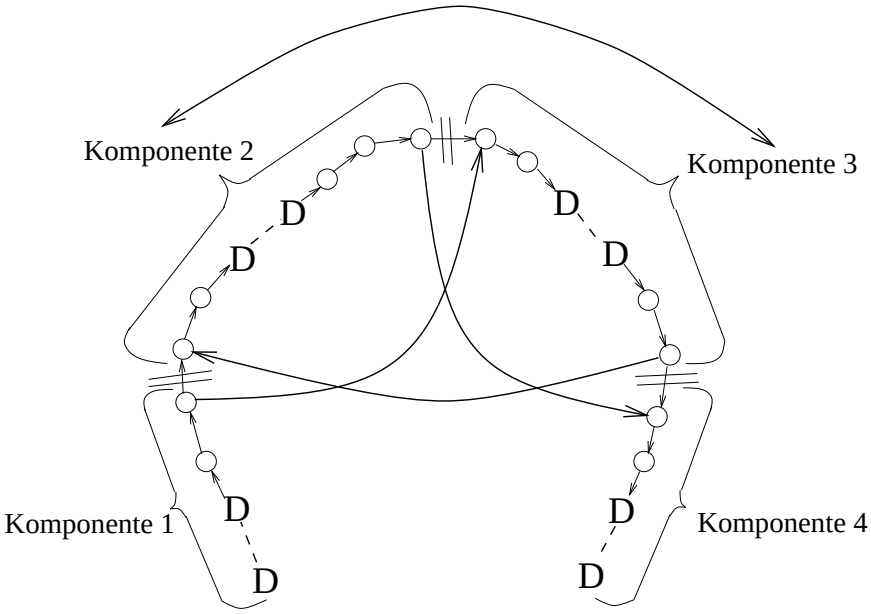


Abbildung 6.7: Der 3-opt-Algorithmus führt eine Vertauschung durch

Implementierung: Die Gesamttour wird, genauso wie die Giant Tour in Abschnitt 6.4, als zum Ring geschlossene Liste dargestellt. Vertauschungen der Abschnitte der Gesamttour erfolgen durch Umhängen von Zeigern.

Komplexität: Eine Tour mit k Stops beinhaltet auch k Kanten. Es gibt $\begin{pmatrix} k \\ 3 \end{pmatrix} = \mathcal{O}(k^3)$ Möglichkeiten, 3 Kanten aus k auszuwählen. Eine Iteration bis zur ersten Vertauschung benötigt daher $\mathcal{O}(k^3)$ Schritte.

Die wirkliche Komplexität hängt aber von der Anzahl der durchgeführten Vertauschungen ab, denn nach jeder Vertauschung beginnt der Algorithmus wieder von neuem. In [15] wird behauptet, daß die Laufzeit des Algorithmus nach empirischen Messungen immer noch $\mathcal{O}(k^3)$ ist, was sich aber bei meinen Versuchen nicht bestätigt hat. Die Anzahl der Vertauschungen hängt nämlich von der Größe des Graphen und der Stop-Menge und der "Schlechtigkeit" der zu verbessernden Touren ab. Dabei können sich bei ungünstigen Ausgangssituationen extrem lange Laufzeiten ergeben. Aus diesen Gründen kann ich keine genauen Angaben für die Worst-Case-Komplexität machen.

Kapitel 7

Verhalten der VRP-Algorithmen

7.1 Allgemeines zur Implementierung und zu den Meßwerten

7.1.1 Implementierung

Die Algorithmen wurden in der Makro-Sprache AML (ARC/INFO Macro Language) des GIS implementiert. AML dient zur Anbindung an die Oberfläche und zur Erzeugung des von ARC/INFO verwendeten Datenbankformats INFO. Außerdem kann man mit AML die NETWORK-Funktionalität (Kürzeste Wege, TSP-Heuristik) des GIS nutzen.

Zeitkritischen Teile der Algorithmen wurden in Kernighan-Richie-C programmiert und in AML eingebettet.

Die Kommunikation zwischen AML und C erfolgt über eine ASCII-Schnittstelle, d.h. die Programme schreiben die zu übergebenden Tabellen als ASCII-Files und lesen sie wieder in die entsprechenden Datenstrukturen ein.

Da AML eine Interpreter-Sprache ist und die Kommunikation über die Schnittstellen relativ viel Zeit in Anspruch nimmt, hat der eigentliche Kern des Algorithmus—das C-Programm—eine verhältnismäßig geringe Laufzeit. Unter diesem Aspekt muß man die Laufzeitmessungen betrachten. Die Laufzeit der einzelnen Programmteile anzugeben macht jedoch nicht viel Sinn, da bei den verschiedenen Verfahren die Aufteilung der Aufgaben zwischen C und AML nicht immer analog gelöst ist.

Als TSP-Heuristik wurde, wo immer eine benötigt wurde, diejenige benutzt, die bereits in ARC/INFO vorhanden ist. Es war nicht herauszufinden, nach welchem Algorithmus sie arbeitet. Man könnte jedoch an die Stelle dieser Heuristik jede andere, wie in Kapitel 5 beschrieben, setzen.

7.1.2 Meßwerte/Ergebnisse

Die Meßwerte/Ergebnisse der Programmläufe zu verschiedenen Problemen und die Erläuterung der Spaltenbeschriftungen in den Meßwert-Tabellen befinden sich in Anhang B. Eine Zusammenfassung der Meßwerte ist weiter unten (Abschnitt 7.3) zusammengestellt.

Die Programmläufe wurden auf einer Sun SPARCstation 10 mit 64 MB Hauptspeicher durchgeführt.

Die getesteten Algorithmen Sim. Savings (Abschnitt 6.2.1), Sweep (Abschnitt 6.1) und Giant Tour (Abschnitt 6.4) sind genauso implementiert wie in Kapitel 6 beschrieben. Von dem Seq. Savings und dem Clustering-Algorithmus sind jeweils zwei Varianten implementiert.

Sequentieller Savings:

- Seq. Savings steht für das ursprüngliche Verfahren, wie es in Algorithmus 6.4 beschrieben ist.
- Seq. Savings (Abbr.) hat den Unterschied, daß eine Tour abgebrochen wird, sobald ein Savings gefunden wird, welches die Zeit- oder Kapazitätsschranken übertrifft (das ist die im Anschluß an Algorithmus 6.4 erwähnte Alternative).

Beide Implementierungen sind in Abschnitt 6.2.2 beschrieben.

Clustering:

- Clusters (Schwerp.): Der Kern-Stop ist der Stop mit der augenblicklich größten Entfernung vom Depot. Der Abstand eines Stops vom Cluster ist der Abstand zum Schwerpunkt.
- Clusters (Umg.): Der Kern-Stop ist der Stop mit der augenblicklich größten Transfermenge. Der Abstand eines Stops vom Cluster ist der Abstand zum Kern-Stop auf dem Graphen, d.h. dem Kern-Stop wird seine Umgebung als Cluster zugeordnet.

Die Tourlängen innerhalb der Cluster wurden nur dann berücksichtigt, wenn dies explizit angegeben ist.

Genaueres zu den Varianten des Clustering findet sich in Abschnitt 6.3.

Da man aufgrund des großen Rechenaufwandes für keines der Probleme sagen kann, wie lang die kürzesten Touren sind, können die Ergebnisse auch nicht mit dem Optimum verglichen werden. Ich habe daher bei jeder Meßreihe angegeben, wie weit jedes Ergebnis vom besten Ergebnis aller Heuristiken abweicht. Es

ist daher nicht möglich, die Heuristiken nach ihrer Abweichung vom (unbekannten) Optimum zu bewerten, das Ziel ist vielmehr das Verhalten der Heuristiken untereinander zu betrachten.

Alle verwendeten Verfahren sind deterministisch und verändern ihr Ergebnis nicht bei einer anderen Numerierung oder anderen internen Anordnung der Daten. Daher habe ich für jedes Gebiet und für jeden Algorithmus nur einen Programmlauf durchgeführt.

7.2 Klassifizierung von VRP's

Der Ablauf einer Heuristik wird von den Daten, auf denen sie arbeitet, bestimmt. Es ist anzunehmen, daß Heuristiken in ihrer Suche nach einem lokalen Optimum sich stark von der Struktur des zu lösenden Problems leiten lassen.

Manche Heuristiken verwenden implizit ein Modell der Realität, das manchen Eingaben mehr, anderen weniger entspricht. Entsprechend sollte sich dann die Qualität der Lösungen verhalten.

Im folgenden versuche ich, Vehicle Routing Probleme nach ihrer Struktur zu klassifizieren um Kriterien für den Vergleich von Heuristiken zu erhalten. Das Ziel soll sein, jeder Heuristik bestimmte Klassen von Problemen zuzuordnen, auf denen sie besonders gut/schlecht arbeitet.

Eine Einteilung von Vehicle Routing Problemen ist denkbar nach:

1. Position des Depots
    - zentral
    - am Rand
    - Erreichbarkeit
2. Zusammenhang des Graphen
    - dicht
    - dünn
3. Verteilung der Stops
4. Anzahl Stops pro Tour
    - große Touren (im Verhältnis zur Anzahl der Stops)
    - kleine Touren (im Verhältnis zur Anzahl der Stops)
5. Kapazitätsverteilung der Fahrzeuge
    - homogen

- inhomogen

6. Verteilung der Transfermengen
   - homogen
   - inhomogen

Unter diesem Blickpunkt kann man nun auch die getesteten Probleme einordnen (Abbildungen aller Coverages, d.h. der Straßennetze, Stops und Depots, befinden sich in Anhang refmessung):

Problem	Coverage	Position des Depots	Zusammenhang	Verteilung der Stops
1	Fishnet	zentral	dicht	gleichmäßig
2	Fishnet	zentral	dicht	Cluster
3	Miesbach	zentral	mittel	gleichmäßig
4	Miesbach	zentral	mittel	gleichmäßig
5	Miesbach	zentral	mittel	Cluster
6	Oldenburg	zentral	mittel	gleichmäßig
5	Oldenburg	am Rand	mittel	ungleichmäßig, langgestreckt
8	München	am Rand	dicht	gleichmäsig
5	Salzburg	am Rand	dünn	langgestreckt
5	Miesbach	zentral	mittel	ringförmig

7.3 Zusammenfassung der Meßwerte

Um einen Überblick über die Meßwerte zu erhalten sind im folgenden alle Probleme (bis auf die Probleme 4 und 9, bei denen nur ein Teil der Heuristiken getestet wurde), die Abweichungen der Heuristiken vom besten Ergebnis und ihre Rangfolge bei diesem Problem zusammengestellt.

Bei den Verfahren, bei denen mehrere Meßwerte genommen wurden (Sequentieller Savings, Clustering (Umgebung), Clustering (Schwerpunkt)), wurde jeweils das beste Ergebnis angegeben. Die Ergebnisse, die beim Clustering mit Berücksichtigung der Tourlängen erzielt wurden, gehen nicht in diese Aufstellung mit ein.

Die Ergebnisse, vor allem die Mittelwertbildungen, sind jedoch immer unter dem Aspekt zu betrachten, daß die Beispiele nicht zufällig gewählt sind, sondern dazu dienen sollten, unterschiedliches Verhalten der Heuristiken bei unterschiedlichen Voraussetzungen zu zeigen.

Abweichung vom besten Ergebnis [%]:

Verfahren	1	2	3	5	6	7	8	10	Mittelwert
Sweep	0,00	3,53	7,10	10,54	5,02	22,46	13,43	3,00	8,13
Sim. Savings	1,39	0,00	0,80	0,00	1,46	0,63	0,69	7,55	1,57
Seq. Savings	15,07	4,10	8,53	5,63	7,94	10,81	8,86	5,12	8,26
Clusters (Umg.)	6,62	2,13	0,00	1,63	0,00	3,42	0,00	0,00	1,73
Clusters (Schwerp.)	4,27	2,95	0,82	1,36	0,67	0,16	0,00	5,47	1,96
Giant Tour	9,18	1,50	6,57	4,15	1,19	15,87	6,62	6,09	6,40

Rang des Verfahrens:

Verfahren	1	2	3	5	6	7	8	10	Mittelwert
Sweep	1	5	5	6	5	6	6	2	4,50
Sim. Savings	2	1	2	1	4	2	3	6	2,63
Seq. Savings	6	6	6	5	6	4	5	3	5,13
Clusters (Umg.)	4	3	1	3	1	3	1	1	2,13
Clusters (Schwerp.)	3	4	3	2	2	1	2	4	2,63
Giant Tour	5	2	4	4	3	5	4	5	4,00

7.4 Eignung der Verfahren

Die Einteilung der Verfahren nach ihrer Eignung für bestimmte Problemstellung gestaltet sich als schwierig, was in der Natur des Problems und der Heuristik an sich begründet liegt. Kleine Änderungen in der Aufgabenstellung, die auch statistisch nicht erfaßt werden können, können bereits zu eklatanten Unterschieden im Ergebnis führen.

Ich werde jedoch im folgenden versuchen, die Verfahren in ihrer Eignung so exakt wie möglich gegeneinander abzugrenzen und dies durch Meßwerte zu unterstützen.

Um die Algorithmen gut miteinander vergleichen zu können, gebe ich zu jedem Algorithmus eine alternative Beschreibung basierend auf dem Begriff des Savings (Definition 6.1).

Man würde optimale Touren erhalten, wenn man aus allen Savings der kompletten Savings-Liste $\mathcal{S}$ diejenigen zum Bilden der Touren auswählen würde, deren Summe—unter Einhaltung der Beschränkungen—maximal ist. Für die Länge der optimalen Touren gilt also:

$$\text{Länge(opt.Touren)} = \sum \text{Länge(Pendeltouren)} - \sum \text{optimal ausgewählte Savings}$$

Die optimale Kombination aus Savings zu finden kommt aus Zeitgründen natürlich nicht in Frage. Die Aufgabe der Heuristiken besteht darin, solche Savings zu wählen, deren Gesamtsumme möglichst groß ist. Dabei wird die Savings-Liste vor der Auswahl von Savings zunächst oft heuristisch verkleinert.

Im folgenden nehme ich an, daß als TSP-Heuristik immer das Savings-Verfahren angewandt wird. Denn dann kann man jeden Algorithmus sehr gut auf der Basis von Savings beschreiben.

7.4.1 Sweep

Beschreibung: Beim Sweep wird die Savings-Liste so weit eingeschränkt, daß nur noch Savings, deren beide Stops im selben Winkelbereich liegen, übrigbleiben. Aus den übriggebliebenen Savings wird nun wie beim simultanen Savings eine Teilmenge ausgewählt, um die Touren zusammenzufügen. Dabei entstehen so viele Touren wie Winkelbereiche.

Schwächen/Stärken: Der Sweep ist ein einfach zu überschauendes Verfahren und seine Schwächen sind relativ einfach zu erkennen.

Der Sweep wird aller Voraussicht nach schlechte Ergebnisse liefern, wenn die Winkelbereiche, die jedem Fahrzeug zugeteilt werden, sehr schmal werden oder wenn nur wenige Stops in einem Winkelbereich liegen. In diesem Fall bleiben nur noch wenige benutzbare Savings übrig und die Zuordnung der Stops zu Touren ist sehr eingeschränkt und willkürlich. Der TSP-Heuristik bleiben wenig Möglichkeiten, die Stops in einem schmalen Winkelbereich günstig anzuordnen. (Siehe auch [35]).

Diese schmalen Winkelbereiche entstehen vor allem, wenn geringe Kapazitäten der Fahrzeuge die Bildung vieler Touren erfordern und damit die Größe der Touren im Verhältnis zur Anzahl der Stops klein wird.

Ebenfalls ungünstig für den Sweep ist ein Depot, das am Rand des abzufahrenden Gebietes liegt. In diesem Fall ist bereits der Winkelbereich, in dem überhaupt Stops liegen, eingeschränkt und damit die Winkel kleiner.

Des weiteren kann eine nicht-konvexe Verteilung der Stops dazu führen, daß zwei räumlich voneinander getrennte Gebiete derselben Tour zugeordnet werden

(Für die oben genannten Arten des ungünstigen Verhaltens siehe Abbildung 7.1).

Der Sweep betrachtet die Stops als Punkte in einer Ebene und kann daher nur deren euklidische Entfernung berücksichtigen. In realen Straßennetzen können jedoch Stops in Luftlinie nah beisammen liegen, deren Entfernung auf dem Straßennetz aber sehr groß ist (z.B. in bergigem Gelände). In diesem Fall ist der zugrundeliegende Graph dünn besetzt. Mit solchen dünn besetzten Graphen kann

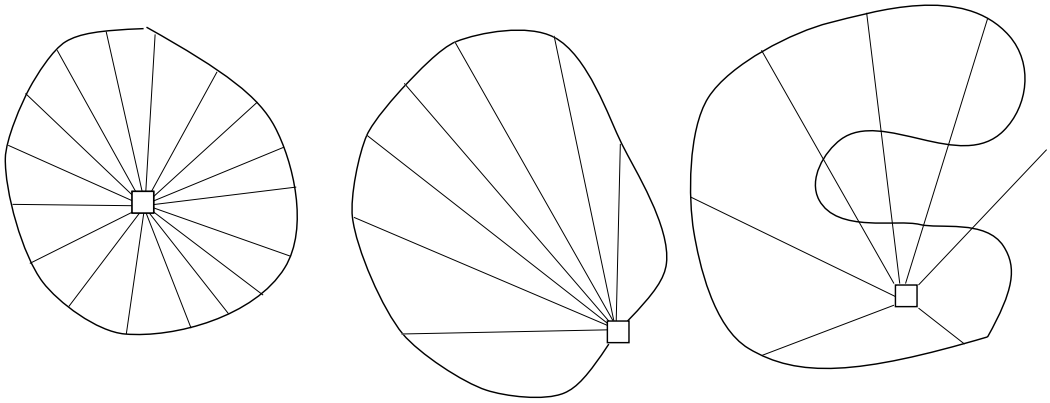


Abbildung 7.1: Beispiele für Probleme, bei denen der Sweep-Algorithmus voraussichtlich schlechte Ergebnisse liefert

der Sweep u.U. große Schwierigkeiten bekommmen, da Stops, die auf dem Netz sehr weit entfernt sind, ähnliche Polarwinkel haben können und damit einer Tour zugeordnet werden können. Die Vorauswahl aus der Savings-Liste wird also ohne Berücksichtigung des tatsächlichen Straßennetzes getroffen.

Die oben aufgeführten Nachteile können auch durch eine Iteration des Sweep (verschiedene Start-Stops) nur bedingt wettgemacht werden.

Meßwerte: Der Sweep liefert also dann gute Ergebnisse, wenn das Depot zentral liegt, der Graph dicht besetzt ist und die Touren im Verhältnis zur Anzahl der Stops groß sind. In diesen Fällen können erstaunlich gute Ergebnisse entstehen, wie in Problem 2, das perfekt auf den Sweep zugeschnitten ist und in dem dieser Algorithmus tatsächlich von allen Heuristiken das beste Ergebnis liefert.

In den Problemen, in denen das Depot zentral liegt und die Stops gleichmäßig verteilt sind, wie in 1, 2, 3 6 und 10 erzielt der Sweep noch recht akzeptable Ergebnisse. Die größte Abweichung vom besten Ergebnis ergibt sich in diesen Problemen mit 7,10 % bei 3, wobei das wohl auf das dünne Straßennetz im Süden zurückzuführen ist.

Bei 5, einem Gebiet, in dem die Stops in Clustern zusammen liegen, schneidet der Sweep relativ schlecht ab, wohingegen in 2, wo die Stops ebenfalls geclustert liegen, der dichte Zusammenhang des Graphen und die relative Größe der Touren noch zu einem akzeptablen Verhalten des Sweep führen.

Sehr schlechte Ergebnisse für den Sweep ergeben sich bei 7 und 8, wo die Depots am Rand liegen.

Zeitschranken: Einer der Vorteile des Sweep ist die Tatsache, daß er Zeitschranken berücksichtigen kann. Allerdings paßt diese zusätzliche Einschränkung nicht wirklich zum Verfahren und muß von außen “aufgezwungen” werden. Das

kann auch zu Problemen führen, da der Sweep bei Fahrzeugen, die viel Kapazität aber nur noch wenig Zeit haben, wieder sehr kleine Winkelsegmente bildet.

Zeitbedarf: Das Sweep-Verfahren ist ausschließlich in AML programmiert und wird bei den Messungen oft iteriert, so daß sich extrem lange Laufzeiten ergeben.

7.4.2 Simultaner Savings

Der simultane Savings ist ein sehr flexibles Verfahren und scheint sich der Struktur des Problems gut anzupassen.

Beschreibung: Er wählt seine benutzten Savings auf sehr naheliegende Weise aus: Alle Savings werden—absteigend geordnet—durchlaufen und alle verwendbaren Savings werden sofort eingesetzt.

Schwächen/Stärken: Probleme ergeben sich jedoch, wenn zwei Touren gebildet werden, die knapp über der halben Kapazität des größten Fahrzeugs liegen. Falls diese Touren nahe beisammen liegen, so können sie dennoch nicht vereinigt werden, da es kein Fahrzeug gibt, welches diese Tour bedienen könnte, und der Savings bereits verbundene Stops nicht mehr trennt. Dadurch können u.U. mehr Touren entstehen, als notwendig sind, oder das Ergebnis verschlechtert sich, da Touren aus Kapazitätsgründen mit ungünstigeren Savings fortgesetzt werden müssen.

Meßwerte: Die Meßwerte zeigen jedoch, daß dieses schlechte Verhalten in der Praxis eher selten eintritt. In den Problemen 2 und 7 bildet der Simultane Savings jedoch eine Tour mehr als die minimale Anzahl an Touren der übrigen Verfahren.

Große Probleme hat der Simultane Savings aber mit Problem 10. Durch die ringförmige Struktur der Stops gibt es sehr viele Stop-Paare mit ähnlich großen Savings. Aus diesem Grund werden sehr viele Touren auf einmal gebildet, die später aus Kapazitätsgründen nicht mehr vereingt werden können. Da es innerhalb des Rings keine Stops gibt, können die “halbfertigen” Touren nicht mehr geeignet fortgesetzt werden. Deshalb zeigt der Simultane Savings hier mit acht Touren (drei mehr als notwendig) und einer großen Abweichung vom besten Ergebnis (7,55 %) hier sein schlechtestes Verhalten.

Zeitschranken: Der große Nachteil dieses Verfahrens ist jedoch, daß Zeitschranken nicht adäquat berücksichtigt werden können. Denn um Zeitschranken einhalten zu können, muß man wissen, welche Tour von welchem Fahrzeug bedient wird und wieviel Zeit dieses Fahrzeug noch zur Verfügung hat. Das kollidiert besonders damit, daß ein Fahrzeug innerhalb seiner Zeitschranke mehrmals eingesetzt werden darf und daher die erlaubte Zeit einer Tour von dem Zeitverbrauch anderer Touren abhängig ist. Ich habe versucht, den Algorithmus so zu modifizieren, daß jede Tour sofort einem Fahrzeug zugeordnet wird. Es werden jedoch i.a. wesentlich mehr Touren parallel gebildet als Fahrzeuge vorhanden sind. Da

Fahrzeuge fehlten, konnten erwünschte Savings nicht berücksichtigt werden und das Ergebnis wurde extrem schlecht.

7.4.3 Sequentieller Savings

Beschreibung: Dadurch, daß der sequentielle Savings immer nur eine Tour auf einmal bildet, ist in jedem Schritt die Menge der Savings auf diejenigen eingeschränkt, die zur Fortsetzung der Tour dienen können.

Aus den Meßwerten, bei denen der sequentielle Savings immer relativ schlecht abschneidet, kann man ablesen, daß diese Einschränkung der Savings-Liste nachteilige Auswirkungen hat.

Geringe Anzahl von Touren: Das Problem des simultanen Savings, daß Touren aus Kapazitätsgrunden nicht mehr vereinigt werden können, kann beim sequentiellen Savings allerdings nicht auftreten, da jede Tour bis zur Kapazitätsgrenze gefüllt wird, bevor die nächste Tour begonnen wird. Daher erhält man beim sequentiellen Savings i.a. weniger Touren als bei der simultanen Version.

Es gibt jedoch auch Fälle, bei denen der sequentielle Savings sogar mehr Touren liefert als der simultane. Dafür ist Abbildung 7.2 ein Beispiel. Alle Stops haben gleichen Abstand vom Depot, so daß die Savings zwischen zwei Stops umso größer werden, je kleiner der Abstand dieser Stops ist. Der simultane Savings berücksichtigt der Reihe nach die Savings $s(1,2), s(3,4), s(6,1), s(4,5)$. Dabei entstehen die beiden Touren $6,1,2$ und $3,4,5$. Der sequentielle Savings jedoch baut zuerst die Touren $1,2,3$, anschließend $4,5$ und der Stop 6 kann aus Kapazitätsgründen keiner Tour mehr zugeordnet werden, so daß drei Touren entstehen. Solche Fälle sehe ich jedoch eher als pathologisch an. In praktischen Fällen hat der sequentielle Savings immer weniger oder gleich viel Touren erzeugt als der simultane Algorithmus.

Schwächen/Stärken: Die sequentielle Tourbildung schränkt die in jedem Schritt zur Verfügung stehenden Savings stark ein, so daß i.a. kleinere Savings verwendet werden und das Ergebnis sich daher verschlechtert. Besonders krasse Abweichungen vom Minimum sind bei den Tabellen 1, 5 und 8 festzustellen (Abweichungen größer als 13 %).

Es gibt jedoch auch hier pathologische Fälle, z.B. das in Abbildung 7.3 abgebildete Vehicle Routing Problem. Der sequentielle Savings bildet die Touren $1,2,3$ und $4,5$ mit einer Gesamtlänge von $16 + 16 = 32$. Mit dem simultanen Savings erhält man jedoch die Touren $1,2,5$ und $3,4$ mit der Länge $22 + 14 = 36$.

Sequentieller Savings mit frühzeitigem Abbruch: Wenn man eine Tour solange fortsetzt, bis keine verwendbaren Savings mehr vorhanden sind (wie in Algorithmus 6.4), dann besteht gegen Ende der Tourbildung die Gefahr, daß Stops mit kleinen Transfermengen an die Tour angehängt werden, die aber sehr weit entfernt sind. Diesem Problem kann man dadurch entgehen, daß man die Tour abbricht, sobald ein anzuhängender Kandidat die Kapazitäts- oder Zeitschranken überschreitet (Verfahren Seq. Savings (Abbr.)). Das kann zwar dazu führen, daß

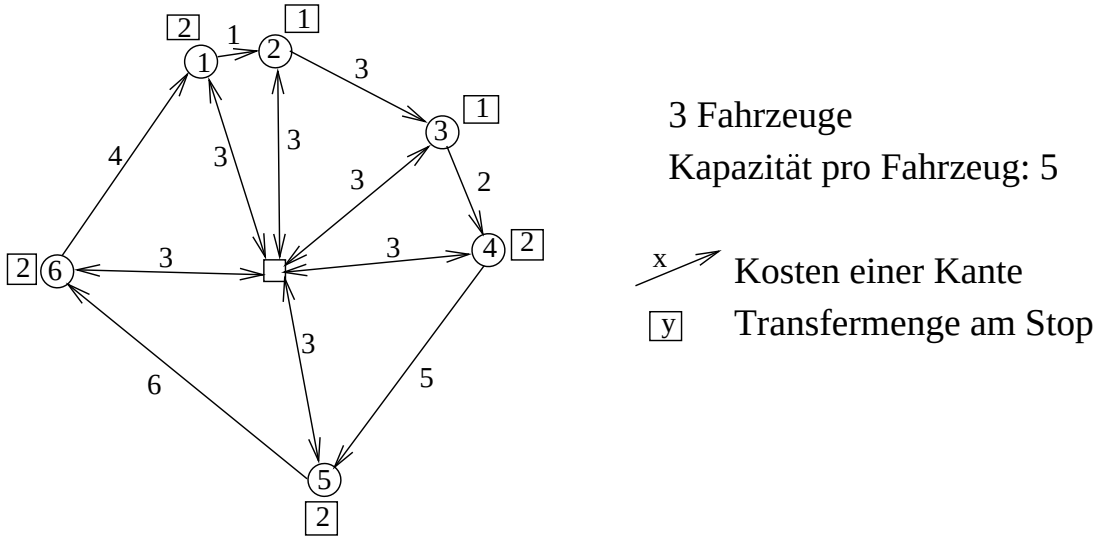


Abbildung 7.2: Beispiel für den Fall, daß der sequentielle Savings mehr Touren erzeugt als der simultane Savings

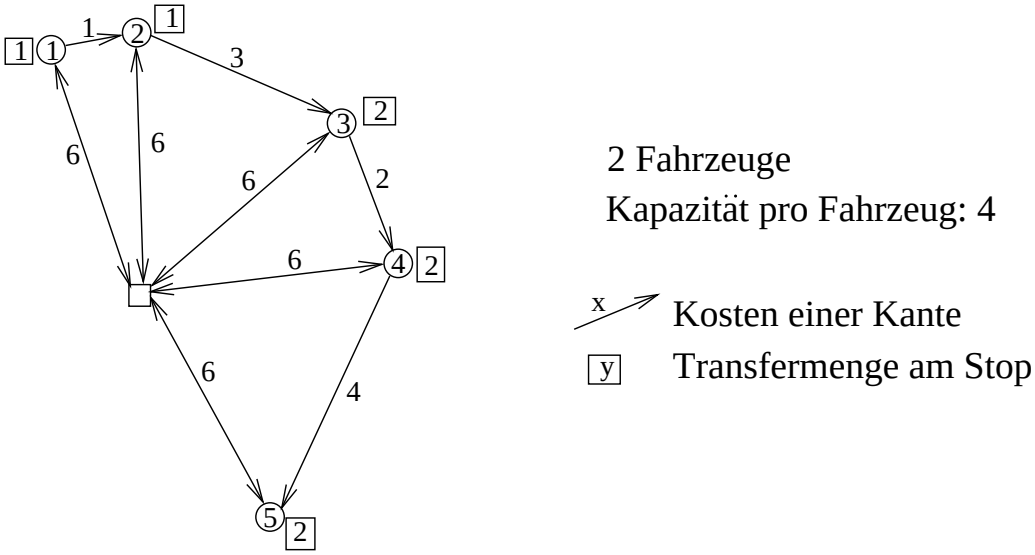


Abbildung 7.3: Beispiel für schlechtes Verhalten des simultanen Savings gegenüber dem sequentiellen Savings

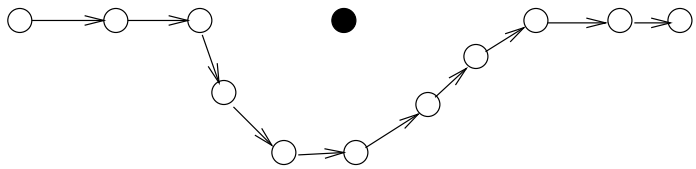


Abbildung 7.4: Sequentieller Savings schafft einen isolierten Stop

mehr Touren entstehen, als unbedingt notwendig (so geschehen bei den Problemen 4, 7, 8 und 9), liefert aber oft bessere Ergebnisse. Besonders deutlich sind die Unterschiede in den Tabellen 5 und 8 zu sehen. Bei 5 sind die Stops zu Clustern zusammengefaßt und die Gefahr, daß ein weit entfernter Stop (aus einem anderen Cluster) noch mitgenommen wird, ist besonders groß. Der sequentielle Savings mit frühzeitigem Abbruch kann natürlich auch schlechtere Ergebnisse liefern, vor allem auch, weil er eventuell mehr Touren bildet. In den Meßwerten liegt sein Ergebnis aber nur selten mehr als einen Prozentpunkt über dem des sequentiellen Savings ohne zusätzliche Abbruchbedingung.

Abhängigkeit von der Größe der Touren: Wenn längere Touren gebildet werden, besteht beim sequentiellen Savings die Gefahr, daß Stops "links liegen" gelassen werden, d.h. isolierte Stops entstehen, in deren Nähe eine Tour vorbeiführt, die aber nicht angefahren werden. Solche Stops müssen später von anderen Touren, die einen großen Umweg machen, bedient werden. Diese Gefahr besteht vor allem dann, wenn relative große Touren gebildet werden, die die meisten anderen Stops im Umkreis eines isolierten Stops "abgrasen" (siehe Abbildung 7.4).

Die Ergebnisse des Algorithmus werden daher auch erkennbar besser, je kleiner die Touren werden. Diese Tendenz kann man in den Tabellen 4 und 9, wo die Kapazitäten der Fahrzeuge sukzessive erhöht wurden, bei beiden Variationen des sequentiellen Savings ablesen. Die Verfahren reagieren allerdings sehr empfindlich auf kleine Änderung, so daß einige "Ausreißer" entstanden sind. (Z.B. hat das Verfahren Seq. Savings (Abbr.) in Tabelle 9 bei Kapazität 3000 zwei Touren mehr gebildet als erforderlich und damit ein sehr schlechtes Ergebnis erzielt.) Dennoch denke ich, daß die Neigung des sequentiellen Savings, bei größeren Kapazitäten (größeren Touren) schlechtere Ergebnisse zu liefern, erkennbar wird.

Zeitschranken: Den sequentiellen Savings zeichnet aus, daß er in der Lage ist, Zeitschranken zu berücksichtigen.

Erweiterungen: Auch weitere denkbare Einschränkungen, wie z.B. eintägige und zweitägige Abholung (Mehrperiodenproblem) oder Zeitfenster beim Stop scheinen in den sequentiellen Savings relativ gut integrierbar zu sein.

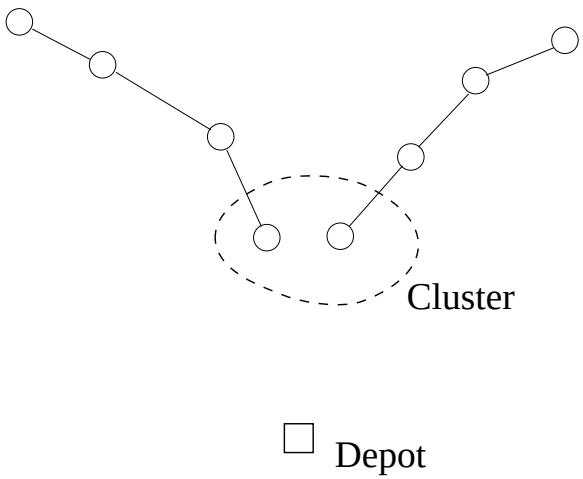


Abbildung 7.5: Fehlverhalten des simultanen Savings, welches durch Clustering verhindert wird

7.4.4 Clustering

Beschreibung: Beim Clustering werden Gruppen von Stops fest zu Clustern zusammengefügt. Diese Cluster kann der Algorithmus nicht mehr trennen. Anders ausgedrückt werden die Savings zwischen Stops eines Clusters allen anderen Savings vorgezogen und in der Reihenfolge der Savings-Liste ganz nach vorne gestellt. Das kann negative und positive Auswirkungen haben.

Schwächen/Stärken: Die Umordnung der Savings kann dazu führen, daß zu viele kleine Savings verwendet werden und dabei ungünstige Touren entstehen. Das passiert vor allem dann, wenn große Cluster entstanden sind.

Wenn die Cluster jedoch relativ klein sind, dann führt dies dazu, daß Stops, die sinnvollerweise sowieso zusammengehören, auch in eine Tour genommen werden. Das macht auch einen Nachteil des simultanen Savings wett, nämlich daß Touren, die nahe beisammen liegen, aus Kapazitätsgründen nicht mehr vereinigt werden dürfen (siehe Abbildung 7.5).

Es ist notwendig, die Größe an die Kapazität der verwendeten Fahrzeuge anzupassen, damit keine Cluster entstehen, die nur mehr schwer zu Touren zusammengefügt werden können.

Meßwerte: An den Meßwerten kann abgelesen werden, daß das Clustering-Verfahren, vor allem bei sehr vielen kleinen Clustern, oft das beste Ergebnis liefert (siehe Tabellen 3, 6, 7 und 8). Dabei ist es nötig, verschiedene Parameterbelegungen durchzuspielen.

Ein besonders gutes Ergebnis erzeugt Clustering bei Problem 10, wo es die Schwächen des Simultanen Savings eliminiert, der Touren aus Kapazitätsgründen nicht mehr vereinigen kann und so viel mehr Touren erzeugt, als notwendig. Die

Zusammenfassung der Stops zu Clustern bestimmt schon in Umrissen den Verlauf der Touren und schränkt damit die Möglichkeiten des anschließenden parallelen Savings ein, die Stops in ungünstiger Weise zusammenzufügen.

Bei Graphen, in denen die Stops bereits in ausgeprägten Clustern zusammenliegen (Probleme 2 und 5), liefert das Cluster-Verfahren keine ausgesprochen guten Ergebnisse und wird auf diesen Straßennetzen jedesmal vom simultanen Savings übertroffen. Meiner Auffassung nach liegt das daran, daß das schlechte Verhalten des simultanen Savings dann nicht mehr so stark ins Gewicht fällt, wenn die Touren, die aus Kapazitätsgründen nicht mehr vereinigt werden können, mit anderen nahegelegenen Stops fortgesetzt werden können. Und da die Stops eines Clusters sowieso sehr nahe zusammen liegen, ist es nicht so entscheidend, welcher Tour ein bestimmter Cluster zugeteilt wird.

In Gebieten, in denen die Stops jedoch großräumiger verteilt sind, fällt es umso mehr ins Gewicht, wenn zwei Stops, die zufällig nahe beisammen sind, nicht in eine Tour zusammengefaßt werden.

Berücksichtigung der Tourlängen innerhalb der Cluster: In Tabelle 7 wurden einige Messungen auch mit Berücksichtigung der Tourlängen innerhalb der Cluster durchgeführt. Damit wurde (mit kleinem Vorsprung) das beste Ergebnis auf diesem Straßennetz erzielt.

Die Ergebnisse sind allerdings zum Teil sogar schlechter als ohne Berücksichtigung der Tourlängen, was zeigt, daß die Suche nach einem guten lokalen Optimum oft auch von Zufällen abhängig ist.

Auch in der Tabelle 6 entstehen nur minimale Verbesserungen und auch teilweise Verschlechterungen durch die Berücksichtigung der Tourlängen, so daß diese Verbesserung des Cluster-Verfahrens, auch wegen des stark erhöhten Zeitbedarfs, eher kleine Vorteile mit sich bringt.

7.4.5 Giant Tour

Beschreibung: Der Giant-Tour-Algorithmus versucht, eine große Rundtour an möglichst günstiger Stelle zu unterbrechen und zum Depot zurückzukehren. Bei der Bildung der Giant-Tour werden zunächst k Savings durch die TSP-Heuristik ausgewählt. Aus diesen Savings versucht man nun wieder eine Teilmenge mit möglichst geringer Summe auszuwählen, die die Giant Tour den Einschränkungen entsprechend zerlegen. Während also beim simultanen Savings die Savings von Anfang an unter Berücksichtigung der Einschränkungen ausgewählt werden, läßt der Giant-Tour-Algorithmus zunächst alle Einschränkungen beiseite, bildet eine große Tour und versucht anschließend, die Einschränkungen durchzusetzen.

Schwächen/Stärken: Der Giant-Tour-Algorithmus geht implizit von einem Modell aus, bei dem das Depot zentral liegt. Wenn das Depot aber am Rand des Gebietes liegt, in dem die Stops liegen, dann ergeben sich wesentlich weniger Möglichkeiten, die Giant Tour günstig zu unterbrechen und zum Depot zurückzukehren. Wenn das Gebiet der Stops zudem noch eine eher langgestreckte Form

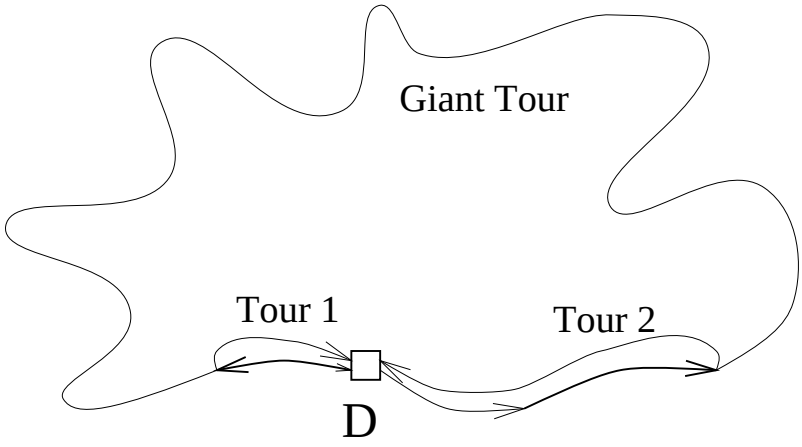


Abbildung 7.6: Schlechtes Verhalten des Giant-Tour-Algorithmus

hat, so kann der Effekt auftreten, daß eine Tour sich vom Depot wegbewegt, dabei ihre Stops aufsammelt, und auf annähernd demselben Weg wieder zum Depot zurückkehren muß (siehe Abbildung 7.6). Das ergibt ein schlechtes Ergebnis, da es anderen Verfahren zumeist gelingt, Touren zu bilden, die sowohl auf dem Hin-, als auch auf dem Rückweg Stops abfahren und damit weniger unnütze Fahrstrecke zu akkumulieren.

Meßwerte: In den Problemen, in denen das Depot zentral liegt, verhält sich das Giant-Tour-Verfahren recht akzeptabel (größte Abweichung vom besten Ergebnis: 7,68 % in Tabelle 1) und ist immer besser als der sequentielle Savings. Bei Problem 4 (Kapazität 30000) erzielt der Giant-Tour-Algorithmus sogar ein besseres Ergebnis als der simultane Savings.

Liegt das Depot am Rand, so ändert sich allerdings das gutmütige Verhalten dieses Verfahrens. Besonders deutlich sieht man dies bei Tabelle 7 (Abweichung von 15,87 % gegenüber dem besten Ergebnis). Auch in Problem 9, in dem das Depot ebenfalls nicht zentral liegt und das Gebiet der Stops relativ langgestreckt ist, ist der Giant-Tour-Algorithmus teilweise schlechter als der sequentielle Savings. Nur bei Problem 8, in dem das Depot ebenfalls nicht am Rand liegt, bedingt wohl die eher quadratische Anordnung der Stops ein noch annehmbares Resultat.

Bei Problem 9 kann man erkennen, daß sich die Größe der Touren auf das Giant-Tour-Verfahren nicht auszuwirken scheint.

Zeitschranken: Der Algorithmus ist, da der Verlauf der Tour schon weitgehend feststeht, auch in der Lage, Zeitschranken zu berücksichtigen. Von allen Verfahren, die Zeitschranken berücksichtigen (Sweep, sequentieller Savings, Giant Tour), liefert das Giant-Tour-Verfahren zumeist das beste Ergebnis.

Erweiterungen: Das Giant-Tour-Verfahren ist, im Gegensatz zum Sweep und zum sequentiellen Savings, ein Verfahren, das die Fahrzeuge nicht in einer

festen Reihenfolge einsetzt, sondern heuristisch eine Auswahl trifft, welche Fahrzeuge eingesetzt werden sollen. Dieses Verfahren wäre auch gut verwendbar für den Fall, daß jedem Fahrzeug Fixkosten und Wegekosten zugeordnet sind. Diese Kosten könnten in die Berechnung des Wertes w im Giant-Tour-Algorithmus integriert werden (siehe Algorithmus 6.6). Am Ende wird dann die Version mit den geringsten Kosten übernommen.

Zeitbedarf: Da die Bildung der Giant Tour viel Zeit in Anspruch nimmt, verbraucht dieses Verfahren—nach dem Sweep—am meisten CPU-Zeit.

7.4.6 Verbesserungsverfahren 3-opt

Beschreibung: 3-opt versucht, die Auswahl der Savings nachträglich zu verbessern, indem versucht wird, drei ausgewählte Savings gegen drei andere, nicht ausgewählte, zu vertauschen. Dabei müssen wiederum alle Beschränkungen eingehalten werden.

Das 3-opt-Verfahren ist in jedem Fall eine gute Ergänzung zu jeder Heuristik.

Meßwerte: An den Ergebnissen kann man ablesen, daß in den meisten Fällen um einige Prozent kürzere Touren entstanden sind. Die Spanne der Verbesserungen reich von 0 % (Probleme 1 und 2) bis 12,07 % (Problem 7, Sweep). Es ist relativ offensichtlich, daß sich schlechte Ergebnis i.a. um einen höheren Prozentsatz verbessern.

Zeitbedarf: Da der Zeitbedarf mindestens kubisch mit der Anzahl der Stops ansteigt, ergeben sich bei Problemen mit vielen Stops, wie z.B. bei 1 und 2, recht lange Rechenzeiten. Außerdem kann man deutlich sehen, daß die Rechenzeit bei größerem Prozentsatz an Verbesserung ansteigt.

7.4.7 Zusammenfassung der Ergebnisse

Die folgende Tabelle soll einen groben Überblick über die Ergebnisse dieses Kapitels und damit über die Eignung der verschiedenen Verfahren geben.

Verfahren	Position des Depots	Zusammenhang	Größe der Touren	Verteilung der Stops	Zeitschranken
Sweep	zentral	dicht	groß	konvex	ja
Sim. Savings					nein
Seq. Savings			klein		ja
Clustering				ringförmig	nein
Giant Tour	zentral				ja

Anhang A

Verzeichnis der Bezeichnungen

Dieses Verzeichnis enthält eine Liste aller Bezeichnungen und Variablen, die häufig vorkommen und nicht unmittelbar an der Stelle, an der sie verwendet werden, definiert sind.

A Menge der Abbiegeverbote

B Menge, die keine Subtouren enthält

C Cluster, Menge von Stops

$C(\pi)$ Kosten einer Tour π

D Depot

E Kantenmenge von G

F Fuhrpark, Menge aller Fahrzeuge

G Zusammenhängender, gerichteter Graph

M Menge aller Pendeltouren

$N(s)$ Nachfolger des Stops $s \in S$ in einer Tour

NP Menge aller Probleme, die mit einer nichtdeterministischen Turingmaschine in polynomieller Zeit gelöst werden können

P Menge aller Probleme, die mit einer deterministischen Turingmaschine in polynomieller Zeit gelöst werden können

$Q(\pi)$ Transfermenge der Tour π

S Stop-Menge

$T(\pi)$ Zeitbedarf der Tour π

V Knotenmenge von G

X Matrix eines ganzzahligen Programms

CLIQUE Problem der Existenz von Cliquen

Co-TSP Komplementäres Traveling Salesman Problem

EXACT TSP Exaktes Traveling Salesman Problem

EXACT VRP Exaktes Vehicle Routing Problem

OPTIMAL TSP Optimales Traveling Salesman Problem

SAT Satisfiability, Erfüllbarkeitsproblem für Boole'sche Formeln

TSP Traveling Salesman Problem

$\mathsf{TSP}_{\mathsf{allg}}$ Allgemeines Traveling Salesman Problem

VERTEX COVER Problem der Existenz eines Vertex Cover

VRP Vehicle Routing Problem

$\mathsf{VRP}_{\mathsf{time}}$ Vehicle Routing Problem mit Zeitschranken

α Länge einer oder mehrerer Touren

$\phi(s)$ Polarwinkel des Stops $s \in S$ zum Depot D

π Tour bzw. Subtour

$b(s)$ Transfermenge des Stops s

$c(e)$ Kantenbewertung von $e \in E$, Abstand zweier Stops

$c(v_1, v_2)$ Abstand der Knoten $v_1, v_2 \in V$

$c_{x,y}(C)$ Kosten der Tour, die im Cluster C von x nach y führt und alle Stops des Clusters besucht

k Anzahl der Stops ($k := |S|$)

l Anzahl der Fahrzeuge ($l := |F|$)

m Anzahl der Kanten ($m := |E|$)

$m(f)$ Maximale Anzahl von Einsätzen des Fahrzeugs $f \in F$

$\max_T$ Summe der Einsätze aller Fahrzeuge

n Anzahl der Knoten ($n := |V|$)

$q(f)$ Kapazität des Fahrzeugs $f \in F$

r Anzahl der Touren beim VRP

radius(C) Radius eines Clusters C

$\tilde{s_C}$ Schwerpunkt eines Clusters C

$\tilde{t_D}$ Haltezeit am Depot pro zu entladender Transfermenge

$t_D(x)$ Haltezeit am Depot bei Transfermenge $x \in \mathbb{R}_0^+$

$t_E(e)$ Zeitbewertung der Kante $e \in E$

$t_E(v_1, v_2)$ Zeitabstand der Knoten $v_1, v_2 \in V$

$t_F(f)$ Einsatzzeit des Fahrzeugs $f \in F$

$t_S(s)$ Standzeit am Stop $s \in S$

$x(s), y(s)$ Koordinaten des Stops $s \in S$

$\mathcal{G}$ Liste der gescannten Knoten (beim Algorithmus von Dijkstra)

$\mathcal{R}$ Liste der erreichten Knoten (beim Algorithmus von Dijkstra)

$\mathcal{S}$ Liste der Savings

Anhang B

Meßwerte/Ergebnisse

Zu jedem Problem, bei dem Meßwerte genommen wurden, gibt es eine Abbildung des Straßennetzes, der Anordnung des Depots und der Stops (wobei ihre Transfermenge proportional zum Radius des Kreises ist).

Nach der Abbildung folgen Tabellen mit den Meßwerten.

Erläuterung der Einträge:

Coverage Name des verwendeten Straßennetzes (Coverage ist der ARC/INFO-Ausdruck für räumliche Daten eines bestimmten Gebietes)

Kanten Anzahl der Kanten im Graphen (m)

Knoten Anzahl der Knoten im Graphen (n)

Stops Anzahl der Stops (k)

Cut [m] Cut-off-Entfernung, wie in Definition 6.1 (in Metern) (d')

Transfersumme Summe der Transfermengen aller Stops (in Mengeneinheiten) ($\sum_{s \in S} b(s)$)

Stops/(Touren $*$ Touren) Verhältnis der Stops pro Tour zur Gesamtanzahl der Touren (ist ein Verhältnis dafür, ob die Touren—verglichen mit der Anzahl der Stops—groß oder klein sind)

$$\text{Stops}/(\text{Touren} * \text{Touren}) = \frac{\text{Stops}}{\min(\text{Spalte Touren})^2}$$

Fuhrpark Auflistung aller einsetzbaren Fahrzeuge mit Kapazität und Anzahl der möglichen Einsätze (F)

Verfahren Name des verwendeten Verfahrens

Cl.gr. Gibt die Kapazitätsbeschränkung eines Clusters an (in Mengeneinheiten) ($b(C) \leq$ Cl.gr.). Ist bei allen Verfahren außer Clustering ohne Bedeutung

Cl.rad [m] Gibt die Radiusbeschränkung eines Clusters an (in Metern) (radius_C). Ist nur für Clustering von Bedeutung

Cluster Anzahl der entstandenen Cluster ($|S'|$). Ist nur für Clustering von Bedeutung

Touren Anzahl der Touren, die vom entsprechenden Algorithmus gebildet wurden

Länge [m] Gesamtlänge (Kosten) aller gebildeten Touren ($\sum C(\pi)$)

A [%] Differenz zum besten Ergebnis in der Spalte "Länge" (in Prozent)

$$\text{A} = \frac{\text{Länge} - \min(\text{Spalte Länge})}{\min(\text{Spalte Länge})} \cdot 100$$

CPU-Zeit [s] CPU-Zeit (in Sekunden), die der Algorithmus verbraucht hat

Nach Verb. mit 3-opt Ergebnis, das entstanden ist, nachdem das Verbesserungsverfahren 3-opt auf das ursprüngliche Ergebnis angewandt wurde.

Die Werte der Spalten Touren, Länge [m] und A [%] sind bereits weiter oben erklärt.

Verb. [%] Verbesserung des Ergebnisses des entsprechenden Algorithmus durch das 3-opt Verfahren (in Prozent)

$$\text{Verb.} = \frac{\text{Länge} - \text{Länge nach 3-opt}}{\text{Länge nach 3-opt}} \cdot 100$$

Bestes Ergebnis Kürzeste Gesamtlänge aller Verfahren (ohne Anwendung von 3-opt)

mit 3-opt Kürzeste Gesamtlänge aller Verfahren (nach Anwendung von 3-opt)

Die Tabellen 3 und 9 haben eine etwas andere Form: In ihnen sind die Ergebnisse der Variation von Fahrzeugkapazitäten festgehalten. Diese Tabellen haben neben einigen der oben erklärten Einträge noch folgende:

Kapazität Kapazität aller verwendeten Fahrzeuge in dieser Testreihe

Abweichung von Sim. Savings [%] Um das Ergebnis einordnen zu können, wird die Abweichung zum Ergebnis des Simultanen Savings mit derselben Kapazität angegeben.

Anhang C

Farb-Abbildungen

Die folgenden Farb-Abbildungen zeigen die Oberfläche der Tourenplanungs-Applikation und die Darstellung geplanter Touren.

Erläuterungen zu den Abbildungen:

- Oberfläche der Applikation mit Problem 3. Die abgebildeten Touren sind das beste Ergebnis, das bei diesem Problem erzielt wurde.
- Oberfläche der Applikation mit Ausschnitt aus Touren zu Problem 9 und gescannter Hintergrundkarte.
- Darstellung des besten Ergebnisses von Problem 8.

Literaturverzeichnis

[1] Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows. Prentice Hall 1993

[2] Akl, S.: The minimal directed spanning graph for combinatorial optimization. Austral. Comput. J. 12(4) (1980), p. 132–136

[3] Balcázar, J; Díaz, J.; Gabarró, J.: Structural Complexity I. EATCS Monographs on Theoretical Computer Science, Springer 1988

[4] Bazaraa, M.S.; Jarvis, J.L.; Sherali H.D.: Linear Programming and Network Flows. John Wiley & Sons 1990

[5] Bodin, L.; Golden, B.: Classification in Vehicle Routing and Scheduling. Networks 11 (1981), p. 95–108

[6] Bodin, L.; Golden, B.; Assad, A.; Ball, M.: Routing and Scheduling of Vehicles and Crews–The State of the Art. Computers & Operational Research 10 (1983), p. 63–211

[7] Bowerman, R.L.; Calamai, P.H.; Hall, G.: The Spacefilling Curve with Optimal Partitioning Heuristic for the Vehicle Routing Problem. European Journal of Operational Research 76 (1994), p. 128–142

[8] Chapleau, L; Ferland, J.A.; Lapalme, G.; Rousseau, J.-M.: A Parallel Insert Method for the Capacitated Arc Routing Problem. OR Letters 3 (1984), p. 95–99

[9] Christofides, N.: Worst-Case Analysis of a new Heuristic for the Travelling Salesman Problem. Management Sciences Research Report 388 (1976), p. 1–5

[10] Christofides, N.; Eilon, S.: An Algorithm for the Vehicle-dispatching Problem. Operational Research Quarterly 20 (1969), p. 309–318

[11] Clarke, G.; Wright, J.: Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. Operations Research 12 (1964), p. 568–581

[12] Cook, S.A.: The Complexity of Theorem-proving Procedures. Proceedings of the Annual ACM Symposium on the Theory of Computing 3 (1971), p. 151–158

[13] Cullen, F.; Jarvis, J.; Ratliff, H.: Set Partitioning Based Heuristics for Interactive Routing. Networks 11 (1981), p. 125–143

[14] Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numeriske Mathematik 1 (1959), p. 269–271

[15] Domschke, W.: Logistik: Rundreisen und Touren. Oldenbourg 1990

[16] Eberhard, U.; Schneeweiß, Ch.; Vaterrodt, H.J.: Tourenplanung für zwei Depots bei offenen Touren. OR Spektrum 6 (1984), p. 39–46

[17] Eberhard, Ulrich: Mehr-Depot-Tourenplanung. Minerva-Publikationen, München 1987–XI, 207

[18] Elcher, Johannes: Die Tourenkonstruktion bei heterogenem Fuhrpark. CH-Verlag Regensburg

[19] Etezadi, T. ; Beasley, J.E.: Vehicle Fleet Composition. Journal of the Operations Research Society 34 (1983), p. 87–91

[20] Fisher, M.; Jaikumar, R.: A Generalized Assignment Heuristic for Vehicle Routing. Networks 11 (1981), p. 109–124

[21] Garey, M.; Graham, R.; Johnson, D.: Some NP-complete geometric problems. Proceedings of the Annual ACM Symposium on the Theory of Computing 8 (1976), p. 10–22

[22] Gheysens, F.; Golden, B.; Assad, A.: A Comparison of Techniques for Solving the Fleet Size and Mix Vehicle Routing Problem. OR Spektrum 6 (1984), p. 49–66

[23] Gillett, B.; Miller, L.: A Heuristic Algorithm for the Vehicle Dispatch Problem. Operations Research 22 (1974), p. 340-349

[24] Golden, B.: Evaluating a Sequential Routing Algorithm. AIIE Transactions 9 (1977), p. 204–208

[25] Golden, B.; Assad, A.; Gheysens, F.: The Fleet Size and Mix Vehicle Routing Problem. Computers & Operational Research 11 (1984), p. 49–66

[26] Golden, B.; Wong, R.T.: Capacitated Arc Routing Problems. Networks 11 (1981), p. 305–315

[27] Golden, B.L.; De Armon, J.S. ; Baker, W.K.: Computational Experiments with Algorithms for a class of Routing Problems. Computers & Operational Research 10 (1983), p. 47–59

[28] Johnson, D.S.; Papadimitriou, C.H.: Performance guarantees for heuristics. The Traveling Salesman Problem (Eds.: Lawler, E.L.; Lenstra J.K.; Rinnoy Kan, A.H.G.; Shmoys, D.B.), Wiley-Interscience 1985, p.145–180

[29] Krentel, M.W.: The Complexity of Optimization Problems. Proceedings of the Annual ACM Symposium on the Theory of Computing 18 (1986), p. 79–86

[30] Lenstra, J.; Rinnoy Kan A.: Computational complexity of discrete optimizations. Annals of Discrete Mathematics 4 (1979), p. 121–140

[31] Lukka, Anita: On method and system design for a problem in Vehicle Routing and Scheduling. Lappeenranta 1987

[32] Lund, C.; Yannakakis, M.: On the hardness of approximating minimization problems. Proceedings of the Annual ACM Symposium on the Theory of Computing 25 (1993), p. 286–295

[33] Mehlhorn, K.: Data Structures and Algorithms 1: Sorting and Searching. EATCS Monographs on Theoretical Computer Science, Springer 1984

[34] Motwani, R.: Average-Case Analysis of Algorithms for Matchings and Related Problems. Journal of the Association for Computing Machinery 41:6 (1994), p. 1329–1356

[35] Paessens, H.: Tourenplanung bei der regionalen Hausmüllentsorgung. Diss., Uni Karlsruhe 1981

[36] Papadimitriou, C.: Computational Complexity. Addison-Wesley 1994

[37] Papadimitriou, C.: The Euclidean Traveling Salesman Problem is NP-complete. Theoretical Computer Science 4 (1977), p. 237–244

[38] Papadimitriou, C.; Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall 1982

[39] Papadimitriou, C.H.; Yannakakis, M.: The complexity of facets (and some facets of complexity). Proceedings of the Annual ACM Symposium on the Theory of Computing 14 (1982), p. 229–234

[40] Papadimitriou, C.H.; Yannakakis, M.: Optimization, approximation, and complexity classes. Proceedings of the Annual ACM Symposium on the Theory of Computing 20 (1988), p. 229–234

[41] Potvin, J.-Y.; Lapalme, G.; Rousseau, J.-M.: A microcomputer assistant for the development of vehicle routing and scheduling heuristics. Decision Support Systems 12 (1994), p. 41–56

[42] Rosenkrantz, D.; Stearns, R.; Lewis, P.: An Analysis of Several Heuristics for the Traveling Salesman Problem. SIAM Journal of Computing 6 (1977), p. 563–581

[43] Ross, G.; Soland, R.: A Branch and Bound Algorithm for the Generalized Assignment Problem. Mathematical Programming 8 (1975), p. 91–103

[44] Roy, S.; Rousseau, J.-M.: The Capacitated Canadian Postman Problem. Canadian Journal of Operations Research and Information Processing 27 (1989), p. 58–73

[45] Taillard, E.: Parallel Iterative Search Methods for Vehicle Routing Problems. Networks 23 (1993), p. 661–673

[46] Tzscharschuch, D.: Das Tourenproblem mit Fahrzeugen unterschiedlicher Ladekapazität. Math. Operationsforschung und Statistik 15 (1984), p. 253–262

[47] Weuthen, H.-K.: Tourenplanung–Lösungsverfahren für Mehrdepot-Probleme. Diss., TU Karlsruhe 1983

[48] Zhu, Peining: Ein flexibles Verfahren zur Lösung kantenorientierter Probleme im Straßenbetriebsdienst. Darmstadt 1989