8 Schwere Probleme und Heuristiken

8.1 Das Travelling-Salesman-Problem

Ein für viele logistische Anwendungen relevantes Problem ist das Problem des Handlungsreisenden, auch in der deutschsprachigen Literatur oft als das *Travelling-Salesman-Problem* (kurz: TSP) bezeichnet. Gegeben ist eine Menge von Städten und Abständen zwischen den Städten, modelliert in der Regel als kantengewichteter Graph. Gesucht ist die kürzeste Rundtour, die jede Stadt genau einmal besucht. Abbildung 8.1 zeigt eine kürzeste Tour durch die 20 größten deutschen Städte.

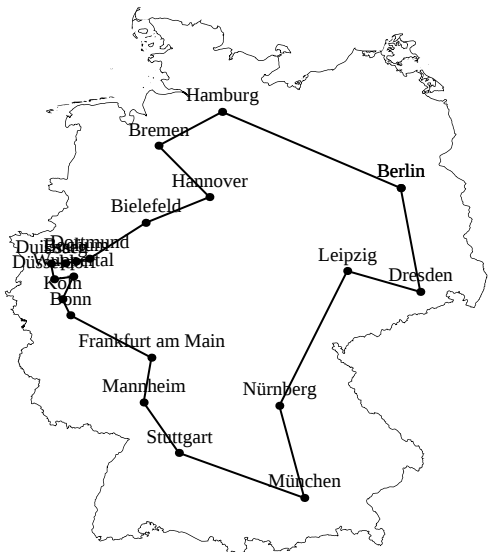


Abb. 8.1: *Eine Lösung des Travelling-Salesman-Problems für die 20 größten Städte Deutschlands. Die Länge dieser Tour beträgt 2430 km. Diese Lösung wurde mit dem in Listing 8.2 gezeigten Code berechnet.*

Das TSP ist ein NP-vollständiges Problem. Man kann also davon ausgehen, dass es keinen effizienten Algorithmus zur Lösung des TSP gibt, d. h. keinen Algorithmus mit polynomieller Laufzeit. Schon für eine Problemgröße von $n = 50$ Städten wäre der für die in Abbildung 8.1 gezeigte Lösung verwendete Algorithmus nicht mehr geeignet eine Lösung innerhalb einer vernünftigen Zeitspanne (etwa zu Lebzeiten der Leser) zu berechnen – siehe hierzu auch Aufgabe 8.4.

8.1.1 Lösung durch Ausprobieren

Die einfachste, aber auch denkbar langsamste Möglichkeit, das TSP zu lösen, besteht darin, alle möglichen Touren, d. h. alle Permutationen der Knotenmenge V, durchzuprobieren und die minimale Tour zurückzuliefern. Eine solche auch oft als *Brute-Force* bezeichnete Lösung zeigt Listing 8.1.

```
1 def TSPBruteForce(graph):
2   nodeList = graph.V()[1:]
3   return min([graph.pathVal(perm + [perm[0]]) for perm in perms(nodeList)])
```

Listing 8.1: *Implementierung des brute-force-Algorithmus, der alle möglichen Touren durchprobiert.*

Die Funktion *perms*(*xs*), wie in Listing B.1 auf Seite 318 gezeigt, liefert eine Liste aller Permutationen der Liste *xs* zurück. Die Methode *pathVal* der Klasse *Graph* (siehe Aufgabe 5.2 auf Seite 151) berechnet den Wert bzw. die Länge eines als Knotenliste übergebenen Pfades. Der Ausdruck *perm*+*perm*[0] erzeugt aus der Knotenpermutation *perm* eine Rundtour.

8.1.2 Lösung durch Dynamische Programmierung

Für das Travelling-Salesman-Problem gilt das sog. *(Bellmannsche) Optimalitätsprinzip*: Eine optimale Lösung setzt sich aus „kleineren“ optimalen Lösungen zusammen. Probleme, für die dieses Optimalitätsprinzip gilt, können durch *Dynamische Programmierung* gelöst werden. In gewissem Sinne muss man ein Problem, das über Dynamische Programmierung gelöst werden soll, genau invers durchdenken als wenn es über Rekursion gelöst werden soll: Während man bei einer rekursiven Implementierung Lösungen gedanklich top-down konstruiert, geht man bei einer Lösung über Dynamische Programmierung bottom-up vor. Man berechnet zunächst die Lösungen der „kleinen“ Teilprobleme und speichert diese Zwischenergebnisse in einer Tabelle. Bei der Berechnung der größeren Teilprobleme (insbesondere des Gesamtproblems) greift man auf die in der Tabelle gespeicherten Werte zurück.

Im Falle des Travelling-Salesman-Problems gilt, dass sich die kürzeste Rundtour über die Knoten aus der Menge S zusammensetzt aus einem Startknoten j und einer um eins kleineren kürzesten Rundtour über alle Knoten aus S, ausgenommen dem Knoten j. Nennen wir $T(i, S)$ den Wert der kürzesten Tour, startend bei Knoten i, die alle Knoten aus S genau einmal besucht und schließlich bei Knoten 1 endet; dann gilt also, dass

$$T(i, S) = \min_{j \in S} \Big(w(i, j) + T(j, S \setminus \{j\}) \Big) \tag{8.1}$$

Modellieren wir die „Tabelle“ T als Python-Dictionary und nehmen an, dass der Graph als Python-Objekt *graph* gegeben sei, so lässt sich dies analog in Python folgendermaßen formulieren:

```
T[(i,S)] = min( graph.w(i,j) + T[(j, diff(S,[j]))] for j in S)
```

(8.2)

Der Wert $T(1, \{2, \ldots n\})$ ist der gesuchte Wert der kürzesten Rundtour.

Formel (8.2) ließe sich zwar direkt in einer rekursiven Implementierung umsetzen, diese ist aber in diesem Fall ineffizient, da eine sehr große Zahl rekursiver Aufrufe entstehen würde[1]. Hier ist also eine Implementierung über Dynamische Programmierung sinnvoll.

[1] Genauer: es wären $|S| - 1$ rekursive Aufrufe notwendig, um die Instanz $T(i, S)$ zu berechnen. Schon der Vergleich mit der rekursiven Implementierung von Quicksort, die bei jeder Instanz höchstens 2 rekursive Aufrufe benötigt, zeigt, dass die $|S| - 1$ Aufrufe sehr „viel“ ist.

Aufgabe 8.1

Geben Sie eine direkt rekursive Implementierung einer Lösung des Travelling-Salesman-Problems in Python an, basierend auf Formel (8.2).

Listing 8.2 zeigt die Verwendung Dynamischer Programmierung bei der Lösung des Travelling-Salesman-Problems.

```
1  def tsp(graph):
2    n = graph.numNodes
3    T = {}
4    for i in range(1,n+1): T[(i,())] = graph.w(i,1)
5    for k in range(1,n-1):
6      for S in choice(range(2,n+1),k):
7        S = tuple(S)    # Listen nicht hashbar ⇒ umwandeln in Tupel
8        for i in diff(range(2,n+1),S):    # for i ∈ S̄
9          T[(i,S)]= min( graph.w(i,j) + T[(j,diff(S,[j]))] ]  for j in S )
10   S = tuple(range(2,n+1))
11   return min( graph.w(1,j) + T[(j,diff(S,[j]))] ] for j in range(2,n+1) )
```

Listing 8.2: *Implementierung eines Algorithmus, basierend auf Dynamischer Programmierung, zur Lösung des Travelling-Salesman-Problems*

Diese Implementierung verwendet ein Dictionary T, um die schon berechneten kürzeren optimalen Touren zu speichern. Die Schlüssel sind hierbei Tupel (i, S) bestehend aus einem Startknoten i und einer Knotenmenge S, die als Tupel repräsentiert ist (in Python ist es nicht möglich, Listen als Schlüsselwerte zu verwenden); $T[(i, S)]$ sollte also immer die kürzeste Rundtour durch Knoten aus S, beginnend bei i, und endend bei Knoten 1 enthalten. In Zeile 4 werden zunächst die „einfachsten“ Einträge in T erzeugt, die nämlich, für die $S = \emptyset$ gilt.

Das in Zeile 6 verwendete *choice*(*range*(2,n+1),k) liefert die Liste aller k-elementigen Teilmengen (jeweils repräsentiert als Python-Listen) der Menge $\{2, \ldots, n\}$ (ebenfalls repräsentiert als Python-Liste) zurück. Eine Implementierung von *choice* – eingebettet in eine kurze Einführung in Binomialkoeffizienten und kombinatorische Grundlagen – findet sich in Listing B.2.

Zunächst berechnet der Algorithmus die Einträge $T(i, S)$ für alle „kleinen“ Teilmengen von $\{2, \ldots, n\}$ – also zunächst für alle 1-elementigen Teilmengen (Schleifendurchlauf für k = 1 der in Zeile 5 beginnenden **for**-Schleife), dann für alle 2-elementigen (Schleifendurchlauf für k = 2), usw. Die eigentliche Berechnung von $T(i, S)$ erfolgt nach Formel (8.2) – Zeile 9 in Listing 8.2 entspricht genau Formel (8.2). Wurden, nach Beendigung der in Zeile 5 beginnenden **for**-Schleife, die Werte $T(i, S)$ aller Touren für alle $S \subseteq \{2, \ldots, n\}$ (und alle $i \in \overline{S}$) berechnet, so kann schließlich der Wert der minimalen Rundtour $T(1, \{2, \ldots, n\})$ in Zeile 11 berechnet werden – dies geschieht wiederum gemäß Formel (8.2).

Aufgabe 8.2

Modifizieren Sie den in Listing 8.2 gezeigten Algorithmus so, dass er – zusätzlich zur Länge der kürzesten Route – die kürzeste Route selbst als Liste von zu besuchenden Knoten zurückliefert.

8.1.3 Laufzeit

Es gibt 2^{n-1} Teilmengen der Menge $\{2, \ldots, n\}$. Für jede dieser Teilmengen S und für jedes $i \in \overline{S}$ muss eine Minimums-Bestimmung durchgeführt werden, die $|S|$ Schritte benötigt. Die Teilmengen S und ebenso deren inverse Mengen $\overline{S}$ haben im Mittel eine Größe von $n/2$ – entsprechend dem Median der Binomialverteilung. Für jede Teilmenge müssen also im Mittel $n/2$ (durchschnittlicher Wert von $|\overline{S}|$) Minimumsbestimmungen durchgeführt werden. Jede Minimumsbestimmung ihrerseits benötigt im Mittel $n/2$ (durchschnittlicher Wert von $|S|$) Schritte um die $|S|$ Schritte miteinander zu vergleichen. Ingesamt benötigt die auf Dynamischer Programmierung beruhende Implementierung *tsp* also

$$(n/2)^2 \cdot 2^{n-1} = O(n^2 2^n)$$

Schritte.

Aufgabe 8.3

Vergleichen Sie die Implementierung in Listing 8.1, die eine Lösung des TSP-Problems durch Ausprobieren aller Möglichkeiten berechnet, mit der Implementierung aus Listing 8.2, die Dynamische Programmierung verwendet.

(a) Zur Berechnung der in Abbildung 8.1 gezeigten Lösung, die kürzeste Rundtour durch die 20 größten Städte Deutschlands zu finden, hat der *tsp*-Algorithmus aus Listing 8.2 auf dem Rechner des Autors etwa 4 Minuten benötigt. Schätzen Sie ab, wie lange der Algorithmus aus Listing 8.1 zur Berechnung dieser Lösung benötigen würde.

(b) Wie viel mal mehr Schritte benötigt der Algorithmus aus Listing 8.1 wie der auf Dynamsicher Programmierung basierende Algorithmus um eine Rundreise durch n Städte zu berechnen?

Aufgabe 8.4

Schätzen Sie ab, wie lange der in Listing 8.2 gezeigte, auf Dynamische Programmierung beruhende Algorithmus benötigen würde, um die kürzeste Rundtour über 30, 40, 50 und 60 Städte zu berechnen.
Gehen Sie wiederum davon aus, dass der in Listing 8.2 gezeigte Algorithmus zur Berechnung einer kürzesten Tour durch 20 Städte etwa 4 Minuten benötigt.

8.2 Heuristiken für das Travelling-Salesman-Problem

Als „Heuristik“ bezeichnet man eine Strategie, um eine „gute“ – jedoch i. A. keine optimale – Lösung eines i. A. schweren Problems in relativ kurzer Zeit zu finden. Hierbei werden spezielle Eigenschaften der Problemstellung ausgenutzt. Aufgrund der NP-Vollständigkeit des Travelling-Salesman-Problems hat man zur Berechnung von Rundtouren über mehr als 30 Städte eigentlich keine andere Wahl als Heuristiken zu verwenden und sich mit einer evtl. nicht-optimalen Lösung zufrieden zu geben – siehe Aufgabe 8.4.

Wir präsentieren im Folgenden mehrere Heuristiken zur Lösung des Travelling-Salesman-Problems, die in allgemeinerer Form auch zur Lösung anderer schwerer Probleme verwendet werden können.

8.3 Greedy-Heuristiken

Mit dem Dijkstra-Algorithmus (siehe Listing 5.5 auf Seite 163) und dem Kruskal-Algorithmus (siehe 5.7 auf Seite 172) haben wir schon zwei sog. Greedy-Algorithmen kennengelernt, die in jedem Schritt einfach die momentan am besten erscheinende Erweiterung zur Lösung wählen. Im Falle des Dijkstra- und Kruskal-Algorithmus gelangt man über diese Greedy-Strategie tatsächlich zur optimalen Lösung.

Dies funktioniert für das Travelling-Salesman-Problem nicht: Eine Greedy-Strategie führt hier i. A. *nicht* zu einer optimalen Lösung – jedoch in vielen Fällen zu einer Lösung die für viele Anwendungen genügend nahe am Optimum liegt. Für das Travelling-Salesman-Problem sind mehrere Greedy-Heuristiken denkbar.

8.3.1 Nearest-Neighbor-Heuristik

Die vielleicht einfachste Möglichkeit besteht darin, von der Stadt aus, in der man sich aktuell befindet, immer die dazu nächstliegende noch nicht besuchte Stadt zu wählen. Diese Heuristik liefert jedoch nur mäßig gute Werte: Verhältnismäßig gute Verbindungen werden relativ früh (aufgrund noch besserer Verbindungen) ausgeblendet; Folge ist, dass gegen Ende einer Nearest-Neighbor-Tour oft sehr lange Wegstrecken in Kauf genommen werden müssen. Im Falle eines nicht vollständigen Graphen (d. h. eines Graphen, bei dem nicht alle Städte miteinander verbunden sind) kann diese Heuristik gar in eine Sackgasse führen.

Die Laufzeit der Nearest-Neighbor-Heuristik beträgt $O(n^2)$ (n Minimumsfindungen aus durchschnittlich $n/2$ Elementen).

Aufgabe 8.5

Implementieren Sie die Nearest-Neighbor-Heuristik für das Travelling-Salesman-Problem und testen Sie diese durch Berechnung der kürzesten Tour durch die ...

(a) ... größten 20 deutschen Städte.

(b) ... größten 40 deutschen Städte.

Hinweis: Die einfachste Möglichkeit, sich einen Graphen zu erzeugen, der die 20 bzw. 40 größten deutschen Städte enthält, besteht in der Verwendung des Python-Moduls *pygeodb*. Mittels *pygeodb.distance* erhält man etwa den Abstandswert zweier Städte.

8.3.2 Nearest-, Farthest-, Random-Insertion

Eine in vielen Fällen etwas bessere Strategie liefert die folgende Greedy-Heuristik: Man beginnt mit einer sehr kurzen (z. B. zwei Städte umfassenden) Tour und man fügt sukzessive weitere Knoten zu der bestehenden Tour möglichst gut ein. Es gibt nun mehrere Möglichkeiten, nach welchen Kriterien der nächste einzufügende Knoten ausgewählt werden kann:

- „Nearest Insertion“: Als nächstes wird derjenige Knoten zur bestehenden Tour hinzugefügt, der zur momentanen Tour den geringsten Abstand hat.
- "Farthest Insertion": Als nächstes wird derjenige Knoten zur bestehenden Tour hinzugefügt, der zur momentanen Tour den größten Abstand hat.
- "Random Insertion": Als nächstes wird zufällig ein noch nicht in der Tour befindlicher Knoten zur Tour hinzugfügt.

Die Abbildungen 8.2 und 8.3 zeigen jeweils Momentaufnahmen bei dem Aufbau einer Tour nach der Nearest- bzw. Farthest-Insertion-Heuristik.

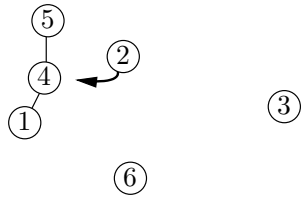


Abb. 8.2: *Momentaufnahme beim Aufbau einer Tour mittels der Nearest-Insertion-Heuristik.*

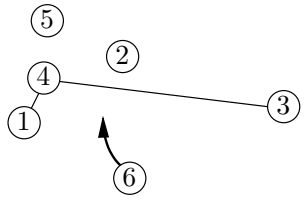


Abb. 8.3: *Momentaufnahme beim Aufbau einer Tour mittels der Farthest-Insertion-Heuristik.*

Tatsächlich liefert schon die Random-Insertion-Heuristik sehr gute Ergebnisse – insbesondere bessere als die Nearest-Insertion-Heuristik. Das folgende Listing zeigt eine Implementierung der Random-Insertion-Heuristik:

```
1  from random import choice
2  def tspRandomInsertion(graph):
3    n = graph.numNodes
4    (w,a,b) = min([(graph.w(i,j), i, j)
5                   for i in range(1,n+1) for j in range(1,n+1) if i≠j])
6    tour = [a,b]
7    while len(tour)<n:
8      v = choice([i for i in range(1,n+1) if i not in tour])
9      pos = min([ (graph.w(tour[i],v) + graph.w(v,tour[i+1]) -graph.w(tour[i],tour[i+1]), i)
10                 for i in range(0,len(tour)-1)  ])[1]
11     tour.insert(pos+1,v)
12   tour = tour + [tour[0]]  # Rundtour daraus machen
13   return pathVal(graph,tour), tour
```

Listing 8.3: *Implementierung der Random-Insertion-Heuristik*

Die Listenkomprehension in den Zeilen 4 und 5 bestimmt die beiden Knoten mit der kürzesten Verbindung im Graphen. Wir beginnen mit einer aus diesen beiden Knoten bestehenden Tour $[a,b]$. In der in Zeile 7 beginnenden **while**-Schleife werden nun sukzessive Knoten zur Tour hinzugefügt, bis schließlich eine komplette Rundtour entsteht. Die Listenkomprehension in Zeile 8 erzeugt alle Knoten, die sich noch nicht in der bisherigen Tour befinden und daraus wird mittels der Funktion *choice* zufällig ein Knoten ausgewählt. In den Zeilen 9 und 10 wird die optimale Einfügeposition in die bestehende Tour bestimmt. Man fügt einfach an derjenigen Position ein, die die bestehende Tour am geringsten vergrößert; man wählt also diejenige Position i der Tour *tour*, die den Ausdruck

$$w(tour_i, v) + w(v, tour_{i+1}) - w(tour_i, tour_{i+1})$$

minimiert. Die Listenkomprehension in den Zeilen 9 und 10 generiert hierzu eine Liste von Tupeln, deren erste Komponente jeweils die zu minimierende Tourvergrößerung ist – die Miniumsbildung läuft auch über diese erste Komponente – und deren zweite Komponente jeweils die Einfügeposition ist. Über die Indizierung $min(\ldots)[1]$ erhalten wir schließlich die zweite Komponente des optimalen Tupels – die optimale Einfügeposition also.

Die Laufzeit des in Listing 8.3 gezeigten Algorithmus ist $O(n^2)$: Es gibt $n-2$ **while**-Schleifendurchläufe und in jedem Schleifendurchlauf muss die (vorläufige) Tour zur Bestimmung der optimalen Einfügeposition durchlaufen werden; deren Länge der vorläufigen Tour ist im i-ten Schleifendurchlauf genau i. Insgesamt sind dies also

$$\sum_{i=0}^{n-2} i = \frac{(n-1)\cdot(n-2)}{2} = O(n^2)$$

Schritte.

Aufgabe 8.6

Implementieren Sie die Nearest-Insertion-Heuristik zum Finden einer möglichst optimalen Lösung des Travelling-Salesman-Problems.

Aufgabe 8.7

Implementieren Sie die Farthest-Insertion-Heuristik zum Finden einer möglichst optimalen Lösung des Travelling-Salesman-Problems.

Aufgabe 8.8

Vergleichen Sie die Güte der gefundenen Lösungen durch die in Listing 8.3 gezeigte Implementierung der Random-Insertion mit den durch ...

- ... Nearest-Insertion
- ... Farthest-Insertion

... gefundenen Lösungen.

Bei der Lösung der vorangegangenen drei Aufgaben konnte man sehen, dass die Nearest-Insertion-Heuristik deutlich schlechtere Ergebnisse liefert als die Farthest-Insertion-Heuristik. Der Grund dafür ist, dass bei der Nearest-Insertion-Heuristik gegen Ende des Algorithmus, wenn nur noch wenige weit entfernte Knoten übrig bleiben, sehr lange Wege entstehen können.

8.3.3 Tourverschmelzung

Eine sich in der Praxis gut bewährende Heuristik ist die der *Tourverschmelzung*: Man wählt zunächst einen beliebigen Startknoten v und generiert $n-1$ Stichtouren zu den verbleibenden $n-1$ Knoten. In jedem Schritt werden zwei der vorhandenen Stichtouren verschmolzen (siehe Abbildung 8.4), und zwar immer so, dass die sich daraus ergebende Kostenersparnis maximal ist. Aus einem Graphen $G = (V, E)$ werden also zwei Touren $tour_i$ (mit Knoten $x \in tour_i$, $\{v, x\} \in E$) und $tour_j$ (mit $u \in tour_j$ und $\{v, u\} \in E$) so gewählt, dass der Ausdruck

$$w(v, u) + w(v, x) - w(u, x) \tag{8.3}$$

maximiert wird.

Folgendes Listing 8.4 implementiert die Tourenverschmelzung: In Zeile 7 wird zunächst mittels *choice* ein Knoten v zufällig aus der Knotenmenge ausgewählt. In Zeile 8 wird der Anfangszustand hergestellt, bestehend aus einer Liste von $n-1$ einelementigen Touren. In jedem Durchlauf der **while**-Schleife ab Zeile 9 werden zwei Touren *t1* und *t2* verschmolzen, indem eine Verbindung zwischen Knoten u und Knoten x eingefügt wird,

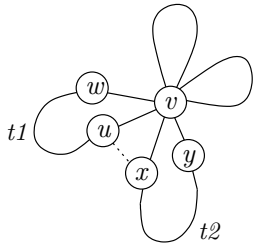


Abb. 8.4: *Sukzessive Verschmelzung von Touren. Die zwei zu verschmelzenden Touren t1 und t2 werden so gewählt, dass die aus der Verschmelzung entstehende Kostenersparnis maximal ist.*

```
1  from random import choice
2  def tspMelt(graph):
3    def melt(t1,t2):
4      return [(graph.w(v,u) + graph.w(v,x) - graph.w(u,x), u==t1[0], t1, x==t2[0], t2)
5              for u in [t1[0],t1[-1]] for x in [t2[0],t2[-1]]]
6    n = graph.numNodes
7    v = choice(graph.V())
8    tours = [[i] for i in range(1,n+1) if i!=v]
9    while len(tours)>1:
10     (fst_u,t1,fst_x,t2) = max([m for t1 in tours for t2 in tours if t1!=t2
11                                                    for m in melt(t1,t2)])[-4:]
12     t1[:] = (t1[::-1] if fst_u else t1) + \
13             (t2 if fst_x else t2[::-1])
14     tours.remove(t2)
15   return [v] + tours[0] + [v]
```

Listing 8.4: *Implementierung der Tourverschmelzung.*

und dafür die beiden Kanten $\{v,u\}$ und $\{v,x\}$ gelöscht werden. Über die Listenkomprehension in den Zeilen 10 und 11 werden die beiden Touren so ausgesucht, dass die Einsparung gemäß Gleichung (8.3) maximiert wird. Die Listenkomprehension erstellt eine Liste aller Verschmelzungen von Touren $t1, t2 \in tours$. Was eine „Verschmelzung" ist, wird durch die ab Zeile 3 definierte lokale Funktion *melt* bestimmt: Nämlich die Liste aller möglichen Verbindungen (davon gibt es 4: Der erste/der letzte Knoten von *t1* kombiniert mit dem ersten/letzten Knoten von *t2*) der beiden Touren. Jede der 4 Kombinationen ist ein 5-Tupel: Die erste Komponente ist die Einsparung, die sich aus der Kombination ergibt. Da die spätere Maximumsbildung sich an der Einsparung orientiert, ist es wichtig, dass dieser Wert an der ersten Stelle steht. Die zweite Komponente gibt an, ob u der erste Knoten aus *t1* ist, die dritte Komponente ist die Tour *t1* selbst, die vierte Komponente gibt an, ob x der erste Knoten aus *t2* ist und die letzte Komponente ist die Tour *t2*. Die Maximumsbildung in Zeile 10 liefert das 5-Tupel mit der maximalen Einsparung und die Indizierung `[-4:]` selektiert die letzten 4 Komponenten dieses 5-Tupels.

In den Zeilen 12 und 13 wird schließlich die Tour *t1* um die Tour *t2* erweitert. Wie dies zu geschehen hat, hängt davon ab, ob sich u, bzw. x, am Anfang oder am Ende der jeweiligen Tour befinden. Schließlich wird in Zeile 14 die Tour *t2* aus *tours* gelöscht. Bleibt schließlich nur noch eine Tour in *tours* übrig, so wird diese eine Tour zusammen mit dem Knoten v als Start- und Endknoten als Rückgabewert von *tspMelt* zurückgeliefert.

Die Laufzeit dieser Implementierung ist $O(n^3)$: Es gibt $n-2$ **while**-Schleifendurchläufe. In jedem Durchlauf werden alle Kombinationen zweier Touren – das sind jeweils $len(tours)^2 - len(tours)$ viele – in Betracht gezogen und die günstigste dieser Kombinationen ausgewählt. Die Laufzeit von *melt* ist eine Konstante, also in $O(1)$. Insgesamt ergibt sich damit als Laufzeit

$$\sum_{i=n-2}^{1} i^2 - i = O(n^3)$$

.

Aufgabe 8.9

Was die Laufzeit betrifft, kann die in Listing 8.4 gezeigte Implementierung der Tourverschmelzung verbessert werden. Anstatt die optimalen Verschmelzungs-Knoten jedesmal neu zu berechnen – wie in den Zeilen 9 und 10 in Listing 8.4 – kann man sich jeweils die optimalen Nachbarn der Anfangs- und Endknoten einer Teiltour merken und – nach einer Verschmelzung – gegebenenfalls anpassen.
Entwerfen Sie eine entsprechend optimierte Version der in Listing 8.4 gezeigten Implementierung und analysieren Sie, welche Laufzeit der Algoritmus nach diese Optimierung hat.

8.4 Lokale Verbesserung

Die Heuristik „lokale Verbesserung“ nimmt eine durch eine andere Heuristik vorgeschlagene Lösung als Ausgangspunkt und nimmt auf dieser (mehr oder weniger gezielte) Veränderungen vor; in diesem Zusammenhang werden diese Veränderungen meist als *Mutationen* bezeichnet. Eine die aktuelle Tour verbessernde Mutation – falls es überhaupt eine solche geben sollte – wird als Ausgangspunkt für die nächste Iteration genommen, usw. Dies wird solange fortgesetzt, bis keine verbessernde Mutation mehr gefunden werden kann. Man beachte, dass im Allgemeinen durch eine lokale Verbesserungsstrategie *nicht* das globale Optimum, sondern lediglich ein *lokales* Optimum erreicht wird.

Für die Lösung des Travelling-Salesman-Problems hat sich in der Praxis das sog. 2-Opt-Verfahren bzw. das allgemeinere k-Opt-Verfahren als praktikabel erwiesen.

8.4.1 Die 2-Opt-Heuristik

Die 2-Opt-Heuristik löscht in einer vorhandenen Tour zwei Kanten und verbindet die dabei frei gewordenen vier Knoten über Kreuz; Abbildung 8.5 zeigt dies graphisch.

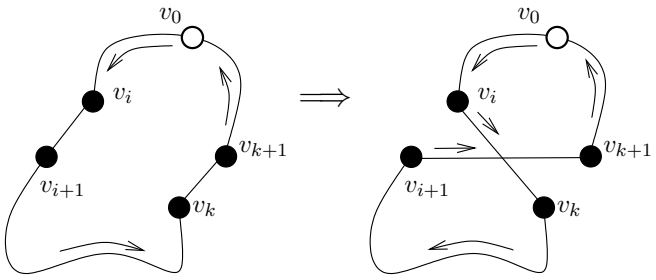


Abb. 8.5: *Eine durch die 2-Opt-Heuristik durchgeführte Mutation einer Tour* $(v_0, v_1, \ldots, v_n, v_0)$*. Zwei Tourkanten* (v_i, v_{i+1}) *und* (v_k, v_{k+1}) *werden gelöscht und stattdessen die Kanten* (v_i, v_k) *und* (v_{i+1}, v_{k+1}) *in die Tour eingefügt; sollte dies eine Verbesserung (bzw. die größte Verbesserung) gegenüber der ursprünglichen Variante ergeben, so wird diese Mutation als Ausgangspunkt für weitere Mutationen verwendet.*

Listing 8.5 zeigt eine Python-Implementierung der 2-Opt-Strategie. Man beachte, dass die Funktion *tsp2Opt* neben dem zugrundeliegenden Graphen einen Algorithmus *heuristik* übergeben bekommt. Die durch diesen Algorithmus berechnete Tour dient (siehe Zeile 3 in Listing 8.5) als Ausgangspunkt für die Durchführung der 2-Opt-Heuristik.

```
1 def tsp2Opt(graph,heuristik):
2   n = graph.numNodes
3   tour = heuristik(graph)
4   while True:
5     (opt, i, k) = max( [(graph.w(tour[i],tour[i+1]) + graph.w(tour[k],tour[k+1]) -
6                          graph.w(tour[i],tour[k]) -graph.w(tour[i+1],tour[k+1]), i,k)
7                            for i in range(n) for k in range(i+2,n) ])
8     if opt<=0: return tour
9     else:      tour = tour[:i+1] + tour[k:i:-1] + tour[k+1:]
```

Listing 8.5: *Implementierung der 2-Opt-Strategie.*

Die Listenkomprehension in den Zeilen 5 bis 7 ermittelt die Mutation der Tour, die sich am ehesten lohnt. Es werden also die beiden Tourkanten (v_i, v_{i+1}) und (v_k, v_{k+1}) mit $i, k \in range(0, n)$ und $i \leq k-2$ ausgewählt, für die die Kostenersparnis

$$w(v_i, v_{i+1}) + w(v_k, v_{k+1}) - w(v_i, v_k) - w(v_{i+1}, v_{k+1})$$

maximal ist. Sollte durch Mutation keine Kostenersparnis mehr möglich sein, d. h. sollte die maximal mögliche Kostenersparnis *opt* kleiner Null sein (dies wird in Zeile 8

geprüft), so wird die 2-Opt-Strategie abgebrochen und die aktuelle Tour zurückgeliefert. Andernfalls wird in Zeile 9 die Mutation durchgeführt. Hierbei muss – das ist in Abbildung 8.5 schön zu sehen – die bisherige Tour bis zu Knoten i übernommen werden (was genau dem Ausdruck $tour[:i+1]$ entspricht), daran Knoten k bis Knoten $i-1$ in umgekehrter Reihenfolge angefügt werden (was genau dem Ausdruck $tour[k:i:-1]$ entspricht) und schließlich alle Knoten ab k ans Ende gehängt werden (was genau dem Ausdruck $tour[k+1:]$ entspricht).

8.4.2 Die 2.5-Opt-Heuristik

Die 2.5-Opt-Heuristik löscht drei Tourkanten, von denen zwei benachbart sind. Dadurch entsteht, wie in Abbildung 8.6 veranschaulicht, (jeweils) eine mögliche Neuverbindung einer so zerfallenen Tour. Die 2.5-Opt-Heuristik prüft, ob es eine Neuverbindung dieser Art gibt, mit der eine bestehende Tour verkürzt werden kann.

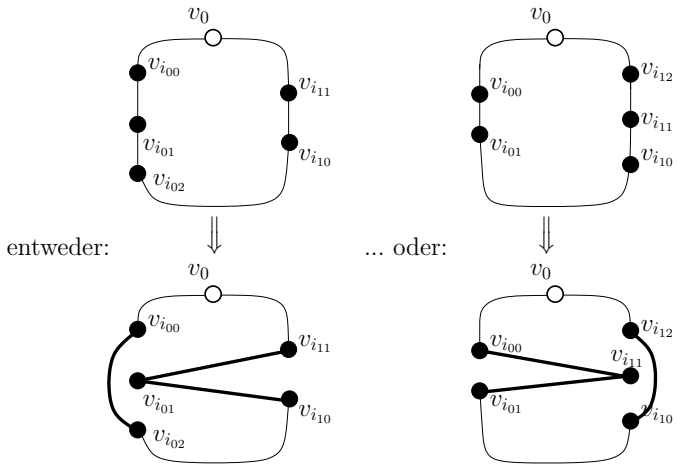


Abb. 8.6: *Die 2.5-Opt-Heuristik erlaubt jeweils genau eine Mutation einer Tour $(v_0, v_1, \ldots, v_n, v_0)$, die durch Entfernung von 3 Kanten (davon zwei benachbarten) Kanten entsteht.*

Die in Listing 8.6 implementierte Funktion *crossTour2_5* erzeugt die in Abbildung 8.6 gezeigte Neuverbindung einer Tour *tour*. Der Parameter i spezifiziert die Kanten, die in der Tour zu entfernen sind. Der in der linken Hälfte von Abbildung 8.6 gezeigten Situation würde der Parameter $i = ((i_{00}, i_{01}, i_{02}), (i_{10}, i_{11}))$ entsprechen, wobei $i_{01} = i_{00} + 1$, $i_{02} = i_{00} + 2$ und $i_{11} = i_{10} + 1$. Der in der rechten Hälfte von Abbildung 8.6 gezeigten Situation würde der Parameter $i = ((i_{00}, i_{01}), (i_{10}, i_{11}, i_{12}))$ entsprechen, wobei $i_{01} = i_{00} + 1$, $i_{11} = i_{10} + 1$ und $i_{12} = i_{10} + 2$.

```
1 def crossTour2_5(tour,i):
2   if len(i[0])==3:
3     return tour[:i[0][0]+1] + tour[i[0][2]:i[1][0]+1] + \
```

```
4             [tour[i[0][1]]]     + tour[i[1][1]:]
5   else:
6     return tour[:i[0][0]+1]           + [tour[i[1][1]]]  + \
7             tour[i[0][1]:i[1][0]+1]  + tour[i[1][2]:]
```

Listing 8.6: *Erzeugung einer Neuverbindung einer durch Löschung von drei (wobei zwei davon benachbart sind) Kanten zerfallenen Tour.*

Mit Hilfe dieser Funktion erfolgt dann die Implementierung der 2.5-Opt-Heursitik so wie in folgendem Listing 8.7 gezeigt:

```
1 def tsp2_5Opt(graph,tour):
2   crTrs = map(lambda i: crossTour2_5(tour,i) ,all2_5Cuts(len(tour)))
3   return min( [(pathVal(graph,c),c) for c in crTrs])
```

Listing 8.7: *Implementierung der 2.5-Opt-Heuristik.*

Zeile 2 wendet die in Listing 8.6 gezeigte Funktion *crossTour2_5* auf *jede* mögliche durch Entfernung von drei Kanten (zwei davon benachbart) zerfallene Tour an. Die Funktion *all2_5Cuts*(n) erzeugt die Spezifikationen aller möglichen Löschungen dreier Kanten aus einer Tour der Länge n. In Zeile 3 wird dann diejenige Neuverbindung mit minimalem Gewicht zurückgeliefert.

Aufgabe 8.10

Implementieren Sie die Funktion *all2_5Cuts*(n), die alle Spezifikationen aller möglichen Löschungen dreier Kanten erzeugt. Beispiel-Anwendungen:

```
>>> all2_5Cuts(10)
>>> [ ((0,1),(3,4,5)),    ((0,1),(4,5,6)),    ... ,   ((5,6,7),(8,9)) ]
```

Aufgabe 8.11

(a) Verwenden Sie statt der *map*-Funktion in Zeile 2 in Listing 8.7 eine Listenkomprehension.

(b) Schreiben Sie die in Listing 8.7 gezeigte Funktion *tsp2_5Opt* so um, dass der Funktionskörper lediglich aus einem **return**-Statement besteht.

Aufgabe 8.12

Implementieren Sie die 2.5-Opt-Heuristik performanter: Überprüfen Sie dazu nicht jedesmal die Länge der gesamten Tour (die durch Neuverbindung entsteht), sondern vergleichen Sie immer nur die Längen der durch Neuverbindung neu hinzugekommenen Kanten mit den Längen der gelöschten Kanten – analog wie in Listing 8.5 realisiert.

8.4.3 Die 3-Opt- und k-Opt-Heuristik

Die k-Opt-Heuristik entfernt k disjunkte Kanten (d. h Kanten ohne gemeinsame Knoten) aus der Tour und versucht die frei gewordenen Knoten so zu verbinden, dass die entstehende Kostenersparnis maximiert wird. Dabei muss man darauf achten, dass durch ungeschicktes Wiederverbinden die ursprüngliche Tour nicht in mehrere Einzeltouren zerfällt. Abbildung 8.7 zeigt alle Möglichkeiten, eine durch Löschung von drei disjunkten

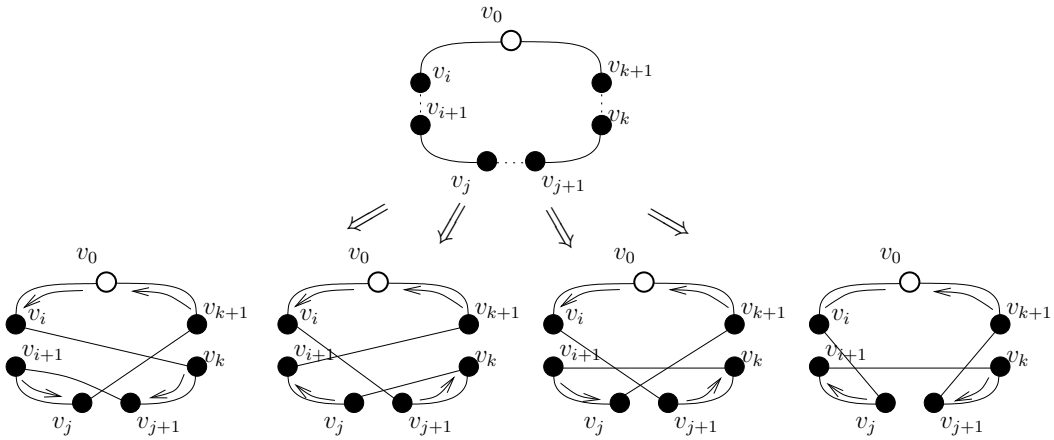


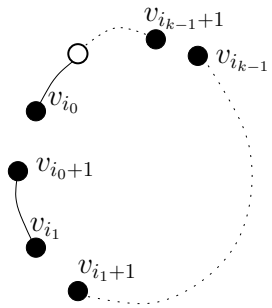
Abb. 8.7: *Es gibt vier Möglichkeiten eine durch Löschung von drei disjunkten Tourkanten zerfallene Tour neu zu verbinden – bzw. sogar acht Möglichkeiten, wenn man ursprüngliche Kanten als Neuverbindungen zulässt, d h. Kanten der Form* (v_m, v_{m+1}), $m \in \{i, j, k\}$ *zulässt.*

Tourkanten zerfallene Tour neu zu verbinden; aus dieser Menge von Neuverbindungen würde man im Laufe einer 3-Opt-Heuristik versuchen, eine verbessernde Neuverbindung auszuwählen.

Aufgabe 8.13

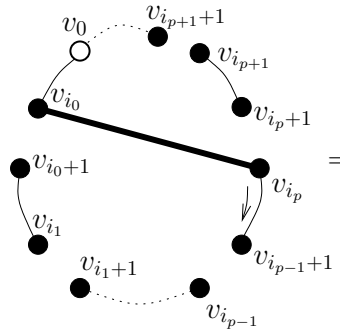
Implementieren Sie die 3-Opt-Heuristik in Python und vergleichen Sie die Güte der berechneten Touren mit denen der 2-Opt-Heuristik.

Wir wollen einen Algorithmus präsentieren, der *alle* möglichen Neuverbindungen einer durch Löschung von k Kanten zerfallenen Tour erzeugt und kümmern uns zunächst darum, wie eine „aufgeschnittene“ Tour repräsentiert werden kann. Abbildung 8.8 zeigt eine Möglichkeit der Repräsentation, die sich in der Implementierung (siehe Listing 8.8) als günstig erweist: die Repräsentation erfolgt als Liste der entfernten Tourkanten – genauer: durch die Liste der Indizes der Tourknoten zwischen denen Kanten entfernt wurden. Abbildung 8.9 zeigt, wie man nach Einziehen einer neuen Tourkante diese Repräsentation anpassen muss: durch Verschmelzung zweier Tupel.



Repräsentiert als $\Longrightarrow$ $[(i_0, i_0+1), (i_1, i_1+1), \ldots, (i_{k-1}, i_{k-1}+1)]$

Abb. 8.8: *Eine an k Kanten aufgeschnittene Tour $(v_0, v_1, \ldots, v_n, v_0)$. Wir werden eine aufgeschnittene Tour durch die Liste der fehlenden Tourkanten repräsentieren. Hierbei ist eine Tourkante jeweils durch ein Tupel der beiden Indizes der Knoten repräsentiert, die diese Kante verbindet. Diese Darstellung ist auch für die spätere Implementierung (siehe Listing 8.8) günstig.*



$\Longrightarrow$ $[\ldots, (i_{p-1}, i_{p-1}+1), (i_p, i_p+1), (i_{p+1}, i_{p+1}+1), \ldots]$
$\Downarrow$
$[\ldots, (i_{p-1}, i_p+1), (i_{p+1}, i_{p+1}+1), \ldots]$

Abb. 8.9: *Die Verwendung der neuen Tourkante (v_0, v_p) zieht in der Repräsentation der fehlenden Tourkanten eine Verschmelzung der Tupel $(i_{p-1}, i_{p-1}+1)$ und (i_p, i_p+1) zum neuen Tupel (i_{p-1}, i_p+1) nach sich.*

Aufgabe 8.14

Wie viele Möglichkeiten gibt es eine durch Löschung von k Kanten zerfallene Tour wieder neu zu verbinden? Geben Sie eine entsprechende von k abhängige Formel an.

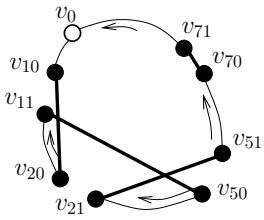
Wir wollen zunächst eine Python-Funktion schreiben, die die Liste aller möglichen Neuverbindungen einer durch Löschung von k (mit $k = len(i)$) Kanten zerfallenen Tour erzeugt. Die in Listing 8.8 implementierte Funktion *allCrosses* liefert die Liste aller möglichen Neuverbindungen einer an den durch den Parameter i spezifizierten Stellen aufgeschnittenen Tour. Die Liste i repräsentiert die Stellen an der die Tour aufgeschnitten ist – und zwar genau so, wie in den Abbildungen 8.8 und 8.9 erläutert; wir gehen also davon aus, dass i eine Liste von Tupeln ist. Eine der (insgesamt 48) Kreuztouren einer Tour, die an den Tourknoten mit Index 10, 20, 50 und 70 aufgeschnitten ist, erhalten wir beispielsweise durch den unten dargestellten Ausdruck (der einfach das zehnte

```
1  def allCrosses(i):
2    if len(i)==1: return [[]]
3    ts = []
4    for p in range(1,len(i)):
5      if p>0:
6        # Rückwärts-Teiltour
7        ts += [[(i[p][0],i[p-1][1])] + x
8                 for x in allCrosses( i[:p-1] + [(i[p-1][0],i[p][1])] + i[p+1:] )]
9      if p<len(i)-1:
10       # Vorwärts-Teiltour
11       ts += [[(i[p][1],i[p+1][0])] + x
12                for x in allCrosses( i[:p] + [(i[p][0],i[p+1][1])] + i[p+2:] )]
13   return ts
```

Listing 8.8: *Funktion, die die Liste aller möglichken Neuverbindungen einer an den durch die Tupel-Liste i spezifizierten Stellen aufgeschnittenen Tour zurückliefert.*

Element, der durch *allCrosses* erzeugten Kreuztourenliste zurückliefert); die Abbildung rechts daneben stellt diese Kreuztour graphisch dar.

```
>>> allCrosses( [(10,11), (20,21), (50,51), (70,71)])[10]
>>> [(20, 11), (50, 21), (51, 70)]
```



Die in Listing 8.8 gezeigte Implementierung erfolgt rekursiv mit Rekursionsabbruch in Zeile 2. Wir gehen davon aus, dass es für eine Tour, der nur eine Kante fehlt, keine neuen Überkreuztouren gibt. Falls i mindestens zwei Tupel enthält, sammeln wir in der Liste *ts* alle Überkreuztouren systematisch auf. Nehmen wir an, die Tour wäre an k Kanten aufgeschnitten und i hätte folglich die Form $[(i_0, i_0+1), \ldots, (i_{p-1}, i_{p-1}+1), (i_p, i_p+1), (i_{p+1}, i_{p+1}+1), \ldots, (i_{k-1}, i_{k-1}+1)]$; siehe Abbildung 8.9 für eine graphische Veranschaulichung. Es gibt – ausgehend von dem Knoten bei dem wir uns aktuell befinden (der aufgrund vorheriger Tupel-Verschmelzungen in i nicht mehr auftaucht) – $2k-2$ mögliche Knoten zu denen wir eine neue „Kreuz"-Kante ziehen können, nämlich i_0+1, i_1, … und i_{k-1}. Die Knoten i_0 und $i_{k-1}+1$ kommen für Neuverbindungen nicht in Frage – eine „Kreuz"-Kante zu diesen Knoten würde bedeuten, die Tour in mehrere Teiltouren zerfallen zu lassen. Die beiden **if**-Anweisungen in den Zeilen 5 und 9 stellen sicher, dass diese beiden Knoten bei dieser Auswahl nicht gewählt werden.

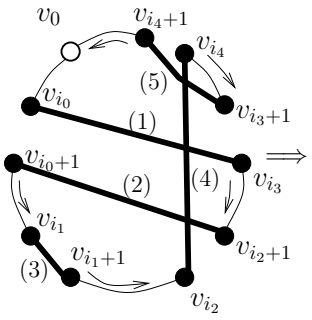
Für jeden dieser $2k-2$ Knoten erfolgt in einer Listenkomprehension ein rekursiver Aufruf an *allCrosses*. Wir erläutern den ersten Fall (die „Rückwärts-Teiltour" in den Zeilen 7 und 8) – die Erläuterungen der „Vorwärts-Teiltour" gehen analog. Dieser Fall entspricht der in Abbildung 8.9 graphisch dargestellten Situation. Die neu eingezogene

Kante geht also zu Knoten mit Tourindex i_p. Von da aus werden die Knoten (relativ zur ursprünglichen Richtung) *rückwärts* bis zum Tourknoten mit Tourindex $i_{p-1}+1$ durchlaufen – daher sprechen wir auch von einer „Rückwarts-Teiltour“. Die Variable x durchläuft in Zeile 8 rekursiv alle Kreuztouren. Der Parameter

$$i[:p-1] + [(i[p-1][0], i[p][1])] + i[p+1:]$$

des rekursiven Aufrufs von *allCrosses* in Zeile 8 repräsentiert die verbleibenden fehlenden Kanten. Diese verbleibenden fehlenden Kanten erhält man durch Verschmelzung zweier Tupel aus i – und zwar genau, wie in Abbildung 8.9 dargestellt.

Abbildung 8.10 veranschaulicht diesen sukzessiven Tupel-Verschmelzungsprozess während des Einziehens neuer Kanten am Beispiel der Wiederverbindung einer durch Löschung von 5 Kanten zerfallenen Tour.



$$[(i_0, i_0+1), (i_1, i_1+1), (i_2, i_2+1), (i_3, i_3+1), (i_4, i_4+1)]$$
$$(1) \Downarrow$$
$$[(i_0, i_0+1), (i_1, i_1+1), (i_2, i_3+1), (i_4, i_4+1)]$$
$$(2) \Downarrow$$
$$[(i_0, i_1+1), (i_2, i_3+1), (i_4, i_4+1)]$$
$$(3) \Downarrow$$
$$[(i_0, i_3+1), (i_4, i_4+1)]$$
$$(4) \Downarrow$$
$$[(i_0, i_4+1)]$$
$$(5) \Downarrow$$
$$[\,]$$

Abb. 8.10: *Jede neu eingezogene Tourkante in einer aufgeschnittenen Tour bewirkt in der Repräsentation der Menge der fehlenden Kanten die Verschmelzung zweier Tupel in ein neues Tupel. In Listing 8.8 geschieht diese Verschmelzung zweier Tupel jeweils in den Zeilen 5, 9, und 13 im Argument des rekursiven Aufrufs von allCrosses.*

Aufgabe 8.15

Vor allem wenn k relativ groß ist (etwa $k > 5$), ist es nicht immer sinnvoll sich systematisch alle Kreuztouren generieren zu lassen; in diesen Fällen tut man besser daran, sich zufällig *eine* der vielen möglichen Kreuztouren auszuwählen. Implementieren Sie eine entsprechende Python-Funktion *randCross*, die – genau wie die Funktion *allCrosses* aus Listing 8.8 – eine Liste der fehlenden Tourkanten als Argument übergeben bekommt und eine zufällig ausgewählte Kreuz-Tour zurückliefert.

Man beachte, dass die Funktion *allCrosses* aus Listing 8.8 unabhängig von einer konkreten Tour ist. Zurückgeliefert werden lediglich Tour*positionen* an denen Kreuzkanten eingefügt werden. Mit Hilfe der in Listing 8.9 gezeigten Funktion *allCrossTours* wird aus den durch *allCrosses* erzeugten Löschpositionen eine konkrete Tour neu verbunden.

```
1 def allCrossTours(tour,i):
2   tours = []
3   for cross in allCrosses(i):
4     t = []
5     for (i0,i1) in cross:
6       t += tour[i0:i1+1] if i0<i1 else tour[i0:i1-1:-1]
7     tours.append(t)
8   return [tour[:i[0][0]+1]+t+tour[i[-1][1]+1:] for t in tours]
```

Listing 8.9: *Die Funktion crossTour wendet die durch allCrosses erzeugten Positionen der Neuverbindungen auf eine bestimmte Tour an.*

Entscheidend ist die Zeile 6: Hier wird auf Basis der in *cross* enthaltenen Tupel die Tour neu verbunden. Ist $i0 < i1$, so entsteht die Vorwärts-Teiltour $tour[i0:i1+1]$, andernfalls entsteht die Rückwärts-Teiltour $tour[i0:i1-1:-1]$. Schließlich werden in Zeile 8 noch an jede so entstandene Tour das Anfangsstück $tour[:i[0][0]+1]$ und Endestück $tour[i[-1][1]:]$ angehängt.

Aufgabe 8.16

Implementieren Sie die k-Opt-Heuristik folgendermaßen:

(a) Schreiben Sie zunächst eine Funktion *randCut*(*n*,*k*), die aus einer Tour mit *n* Knoten zufällig *k* disjunkte Kanten auswählt und die Anfangsknoten dieser Kanten zurückliefert.

```
>>> randCut(100,5)
>>> [16, 30, 73, 84, 99]
```

(b) Schreiben Sie eine Funktion *kOpt*(*graph*,*k*,*m*), die die kOpt-Heuristik implementiert. Für $j = k, k-1, \ldots, 2$ werden jeweils *n*-mal zufällig *j* zu löschende Kanten gewählt; aus dieser entsprechend zerfallenen Tour wird die kürzeste Kreuztour gewählt.

Aufgabe 8.17

Wir wollen eine Variante der kOpt-Heuristik implementieren, die gewährleistet, dass *alle* Kreuztouren, *aller* möglichen Schnitte mit in Betracht gezogen werden.

(a) Implementieren Sie eine Funktion *allCuts*(*n*,*k*), die die Liste aller möglichen Löschungen von *k* Kanten aus einer Tour mit *n* Knoten erzeugt.

(b) Implementieren Sie eine Funktion *kOptAll*(*graph*,*k*), die die k-Opt-Heuristik implementiert und hierbei tatsächlich alle Möglichkeiten durchspielt.

8.5 Ein Genetischer Algorithmus

Ein genetischer Algorithmus nimmt sich den Evolutionsprozess der Natur als Vorbild. Er besteht aus mehreren *Runden* ($\hat{=}$ *Generationen*); in jeder Runde erzeugt ein genetischer Algorithmus eine ganze Menge von möglichen Lösungen ($\hat{=}$ die *Population* bzw. der *Genpool*), bzw. Teillösungen. Um von Runde i nach Runde $i+1$ zu gelangen, werden die möglichen Lösungen aus Runde i gekreuzt und anschließend nach bestimmten Optimalitätskriterien selektiert; die daraus entstehenden modifizierten Lösungen bilden die Lösungen der Runde $i+1$. Die entscheidende Operation ist die *Kreuzung* (engl.: *Cross-Over*) zweier Lösungen. Im Allgemeinen erfolgt eine Kreuzung zweier Lösungen l und l' so, dass die erste Hälfte der einen Lösung mit der zweiten Hälfte der anderen Lösung kombiniert wird. In vielen Fällen (nicht jedoch beim Travelling-Salesman-Problem) besteht diese Kombination einfach in der Konkatenation[2] der beiden Lösungshälften – in Python darstellbar durch den Konkatenations-Operation „+“. Die beiden Lösungskandidaten für die nächste Runde hätten dann die Form

$$l_{neu} = l\,[0:n/2] + l'\,[n/2:n] \quad ; \quad l'_{neu} = l'\,[0:n/2] + l\,[n/2:n] \tag{8.4}$$

Eine sinnvolle Wahl der Populationsgröße, d. h. der Anzahl der Lösungen in einer Runde, die Selektionskriterien und vor allem die genaue Ausgestaltung des Cross-Overs zweier Lösungen zu einer neuen Lösung, hängt sehr stark von dem konkreten Problem ab. Im Falle des Travelling-Salesman-Problems sind zwei sinnvolle Cross-Over-Techniken der Knoten-Cross-Over und der Kanten-Cross-Over.

8.5.1 Knoten-Cross-Over

Leider kann man die Knoten zweier Touren nicht ganz so einfach kreuzen, wie in Gleichung (8.4) dargestellt – diese einfache Art des Cross-Over würde doppelte oder fehlende Knoten in der entstehenden Tour nach sich ziehen. Man kann dies jedoch einfach verhindern, wenn man beim Anfügen der zweiten Hälfte der zweiten Tour schon vorhandene Knoten überspringt und am Ende alle übriggebliebenen Knoten anfügt. Abbildung 8.11 zeigt diese Art des Cross-Overs an einem Beispiel.

Dies implementiert die Funktion *nodeCrossOver*:

```
1 def nodeCrossOver(tour1,tour2):
2   n = len(tour1)
3   return tour1[:n/2] + \
4          [v for v in tour2[n/2:] if v not in tour1[:n/2]] + \
5          [v for v in tour1[n/2:] if v not in tour2[n/2:]]
```

Listing 8.10: *Implementierung des Knoten-Cross-Over*

8.5.2 Kanten-Cross-Over

Eine meist bessere Möglichkeit besteht darin, die Kanten der beiden zu kreuzenden Touren in einem neuen Graphen G' zusammenzufassen und dann über einen Random-

[2] Konkatenation = Aneinanderhängen, Verketten

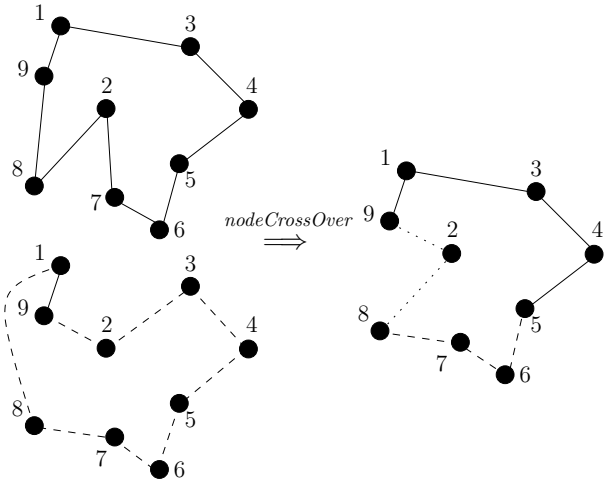


Abb. 8.11: *Knoten-Cross-Over zweier Touren: Die Knoten samt deren Verbindungen (durchgehende Linien) der oben im Bild dargestellten Tour werden übernommen; anschließend werden die fehlenden Knoten samt deren Verbindungen (gestrichelte Linien) der zweiten Hälfte der unten im Bild dargestellten Tour so weit wie möglich übernommen. Ab Knoten „8“ ist dies nicht mehr möglich, denn dessen Tournachfolger, Knoten „2“ wurde schon besucht.*

Walk oder eine andere Heuristik eine Rundtour in diesem Graphen G' zu erzeugen. Abbildung 8.12 veranschaulicht diese Möglichkeit anhand eines Beispiels.

Listing 8.11 zeigt eine Implementierung des Kanten-Cross-Over.

```
1 def edgeCrossOver(graph, tour1,tour2):
2   n = len(tour1) -1
3   G = graphs.Graph(n)
4   for i in range(n-1):
5     for tour in (tour1,tour2):
6       G.addEdge(tour[i],tour[i+1], graph.w(tour[i],tour[i+1]))
7   for tour in (tour1,tour2):
8     G.addEdge(tour[n-1],tour[0], graph.w(tour[n-1], tour[0]))
9   return randomWalk(G,1)
```

Listing 8.11: *Implementierung des Kanten-Cross-Over*

Entscheidend sind die Zeilen 6 und 8: Hier werden (innerhalb der **for**-Schleifen) alle auf den beiden Touren *tour1* und *tour2* befindlichen Kanten in einem neuen Graphen G zusammengefasst. Zurückgegeben wird in Zeile 9 eine zufällige Tour durch den so entstandenen Graphen.

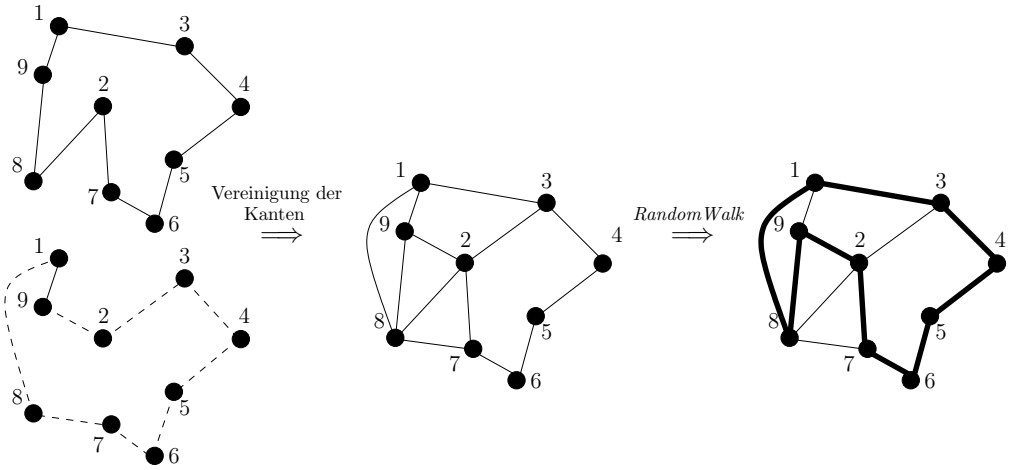


Abb. 8.12: *Kanten-Cross-Over zweier Touren: Die Kanten zweier Touren werden zu einem neuen Graphen vereint; anschließend wird auf dem so entstandenen Graphen ein Random-Walk durchgeführt.*

Aufgabe 8.18

Implementieren Sie die in Zeile 9 in Listing 8.11 verwendete Funktion *randomWalk* folgendermaßen: *randomWalk* soll mit vorhandenen Kanten versuchen eine zufällige Tour zu konstruieren. Sollte es nicht mehr „weitergehen“, weil alle Nachbarn des aktuellen Knotens schon besucht wurden, dann sollte *randomWalk* zurücksetzen und bei einem vorherigen Knoten eine andere Alternative wählen (ein solches Zurücksetzen nennt man auch *Backtracking*).

8.5.3 Die Realisierung des genetischen Algorithmus

Die eigentliche Implementierung des genetischen Algorithmus kann wie in Listing 8.12 gezeigt erfolgen.

Durch den Parameter p kann die Populationsgröße spezifiziert werden; durch den Parameter g kann die Anzahl der Generationen festgelegt werden. In Zeile 2 wird auf Basis der Random-Insertion-Heuristik die erste Generation erzeugt, bestehend aus p unterschiedlichen Touren – es könnten selbstverständlich auch andere Heuristiken verwenden werden, um die initiale Population zu erzeugen, jedoch bietet sich die Random-Insertion-Heuristik dadurch an, dass sie in (nahezu) jedem Durchlauf eine andere Tour liefert. Wir gehen hier davon aus, dass *tspRandIns* immer ein Tupel bestehend aus der Tourlänge und der eigentlichen Tour zurückliefert.

Die **for**-Schleife ab Zeile 3 durchläuft die g Generationen. Wir lassen hier grundsätzlich das beste Drittel der letzten Generation überleben – dies ist jedoch eine mehr oder weniger willkürliche Festlegung mit der man experimentieren kann. Die **while**-Schleife ab Zeile 5 erzeugt dann die restlichen Individuen der neuen Population *newPop*.

```
def tspGen(graph, p, g):
  pop = sorted( [tspRandIns(graph) for _ in range(p)])
  for i in range(g):
    newPop = pop[:p/3] # das beste Drittel überlebt
    while len(newPop)<5*len(pop):
      tours = random.sample(pop,2)
      childTour = edgeCrossOver(graph, tours[0][1], tours[1][1])
      newPop.append((pathVal(graph,childTour)/1000, childTour))
    pop = sorted(newPop)[:p]
  return pop
```

Listing 8.12: *Realisierung des genetischen Algorithmus*

Aufgabe 8.19

Der in Listing 8.12 gezeigte genetische Algorithmus für das Travelling-Salesman-Problem weist folgende Schwäche auf: Die Populationen tendieren dazu, über die Zeit (nach etwa 5 Generationen) genetisch zu verarmen – in diesem Fall heißt das: viele der erzeugten Individuen sind gleich.

(a) Passen Sie den Algorithmus so an, dass sichergestellt wird, dass eine Population keine identischen Individuen enthält.

(b) Man stellt jedoch schnell fest: Der Algorithmus „schafft“ es nach einigen Generationen grundsätzlich nicht mehr, neuartige Individuen hervorzubringen. Passen Sie den Algorithmus so an, dass maximal 50-mal versucht wird ein neues Individuum hervorzubringen – danach wird einfach ein schon vorhandenes Individuum der Population hinzugefügt.

Aufgabe 8.20

Der Algorithmus in Listing 8.12 verwendet für zur Implementierung eines genetischen Algorithmus das Kanten-Cross-Over als Reproduktionsart. Implementieren Sie eine Variante, die stattdessen das Knoten-Cross-Over verwendet und vergleichen Sie die Qualitäten der Ergebnisse für die beiden Reproduktionstechniken.

8.6 Ein Ameisen-Algorithmus

Ähnlich, wie sich genetische Algorithmen ein Vorbild an der Funktionsweise natürlicher Prozesse nehmen, tun dies auch Ameisen-Algorithmen, die das Verhalten eines Schwarmes bei der Suche nach Lösungen simulieren – vorzugsweise für Lösungen von Problemen der kombinatorischen Optimierung. Die Heuristiken, die wir in diesem Abschnitt beschreiben, sind auch unter dem Namen „Ant Colony Optimization“ (kurz: „ACO“) bekannt.

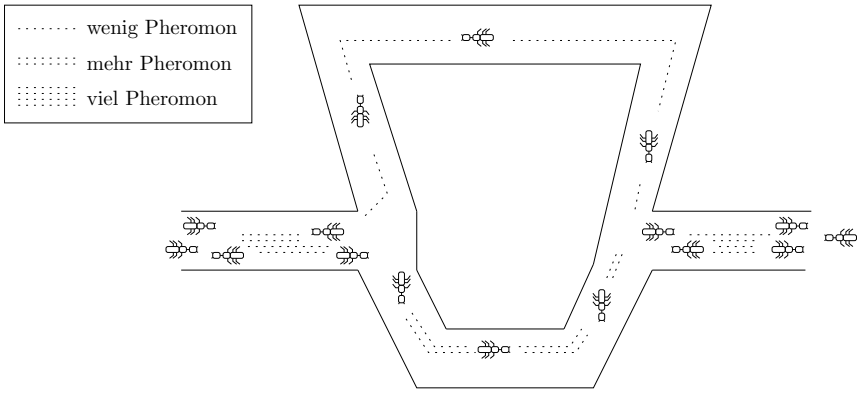


Abb. 8.13: *Je mehr Pheromon sich auf einem bestimmten Pfad befindet, desto größer ist die Wahrscheinlichkeit, dass die Ameisen den entsprechenden Pfad wählen. Da das Pheromon nach einer gewissen Zeit verdunstet, ist die Pheromonkonzentration auf dem längeren Pfad geringer als auf dem kürzeren; die Ameisen wählen also nach einer gewissen Zeit mit größerer Wahrscheinlichkeit den kürzeren Pfad.*

Auf der Wege-Suche nach Nahrung verhalten sich Ameisen in der folgenden Art und Weise: Die einzelnen Tiere (bei der Implementierung in eine Software-System auch gelegentlich als „Agenten“ bezeichnet) suchen die Umgebung zunächst zufällig ab. Findet ein Tier eine Nahrungsquelle, so kehrt es zum Nest zurück und hinterlässt eine *Pheromonspur*[3]. Je größer die Pheromonkonzentration auf einem Pfad, desto größer ist die Wahrscheinlichkeit, dass eine bestimmte Ameise diesen Pfad wählt. Pheromone sind allerdings flüchtig und verdunsten nach einer gewissen Zeit. Je mehr Zeit eine bestimmte Ameise benötigt, um einen Pfad abzulaufen, desto mehr Zeit haben auch die hinterlassenen Pheromone um zu verdunsten. Dies ist genau der Grund, warum Ameisen in der Lage sind, kürzeste Wege zu finden. Abbildung 8.13 veranschaulicht diesen Sachverhalt.

Typisch für Schwärme (wie eben Ameisen, oder große Vogel- oder Insektenschwärme) ist die Beobachtung, dass das Verhalten des Schwarmes nicht durch einen Anführer oder durch hierarchische Beziehungen unter den einzelnen „Agenten“ zustande kommt. Es gibt keine zentrale Abstimmung und jeder „Agent“ in einem Schwarm folgt denselben einfachen Regeln. Ein solches *emergentes* (d. h. aus sich selbst heraus entstehendes) Verhalten bezeichnet man oft als *Schwarm-Intelligenz*. Es hat sich gezeigt, dass die Methoden der Schwarm-Intelligenz und insbesondere die simulierte Verhaltensweise von Ameisen, eine der effizientesten Methoden liefert, eine gute Lösung für das TSP zu finden.

Um ein lokales Optimum des Travelling-Salesman-Problems durch einen simulierten Ameisen-„Schwarm“ zu suchen, muss der dem TSP-Problem zugrundeliegende Abstandsgraph wie folgt konservativ erweitert werden: Ein Kante (i, j) muss neben dem

[3] Als Pheromon bezeichnet man eine spezielle Art flüchtiger Dufthormone, die Insekten – speziell: Ameisen – zur Orientierung dienen.

Gewicht $w(i,j)$, das den Abstand der beiden Knoten i und j repräsentiert, noch ein weiteres Gewicht $p(i,j)$ haben. Der Wert $p(i,j)$ repräsentiert hierbei Menge an Pheromon, die sich auf der Kante (i,j) befindet.

8.6.1 Erster Ansatz

Jede Ameise durchläuft den Graphen komplett. Der jeweils nächste Knoten j von Knoten i aus wird gemäß einer bestimmten Wahrscheinlichkeit gewählt, die sich aus der Entfernung des nächsten Knotens und dem Pheromongehalt der entsprechenden Kante ergibt – je höher hierbei der Pheromongehalt $p(i,j)$ der Kante und je geringer der Abstand $w(i,j)$, desto wahrscheinlicher wird der Knoten j als nächster Knoten auf der Rundtour gewählt.

Übergangsregel. Nennen wir $\Pr_k(i,j)$ die Wahrscheinlichkeit, dass die auf Knoten i befindliche Ameise k als Nächstes den Knoten j wählt. Es erweist sich als günstig diese Wahrscheinlichkeit folgendermaßen festzulegen:

$$\Pr_k(i,j) = \begin{cases} \dfrac{p(i,j) \cdot \dfrac{1}{w(i,j)^\beta}}{\displaystyle\sum_{v \in \Gamma_k(i)} p(i,v) \cdot \frac{1}{w(i,v)^\beta}}, & \text{falls } j \in \Gamma_k(i) \\ 0, & \text{sonst} \end{cases} \tag{8.5}$$

wobei $\Gamma_k(i)$ die Menge der Knoten bezeichnet, die von Ameise k von Knoten i aus erreichbar sind. Über den Parameter β kann man bestimmen, wie sich der Abstandswert w gegenüber der Pheromonmenge p bei der Bestimmung der Wahrscheinlichkeit $\Pr_k$ verhält: Je größer β gewählt wird, desto größer fällt die Pheromonmenge der Kante (i,j) in Gewicht und desto mehr wird der Abstandswert bei der Entscheidung darüber, welcher Knoten als Nächstes gewählt wird, ausgeblendet.

Um also zu berechnen, mit welcher Wahrscheinlichkeit die Kante (i,j) gewählt wird, wird das Verhältnis zwischen Länge und Pheromongehalt der Kante (i,j) durch die Summe der Verhältnisse aller von Knoten i aus erreichbaren Kanten geteilt.

Implementierung der Übergangsregel. Listing 8.13 zeigt den Python-Code zur Simulation einer Ameise. Im Gegensatz zu allen vorigen Anwendungen, müssen wir hier *zwei* Werte je Kante speichern: eine Entfernung und eine Pheromon-Konzentration. Der Einfachheit halber vermeiden wir Anpassungen an der in Abschnitt 5.1.2 beschriebenen *Graph*-Klasse, sondern gehen einfach davon aus, dass ein Gewicht *graph.w*(i,j) einer Kante (i,j) des betrachteten Graphen aus zwei Komponenten besteht: Die erste Komponente *graph.w*(i,j) [0] speichert die Entfernung zwischen Knoten i und Knoten j, die zweite Komponenten *graph.w*(i,j) [1] speichert den Pheromongehalt der Kante (i,j).

In Variable i ist immer der als Nächstes zu besuchende Knoten gespeichert. Diese wird zunächst in Zeile 4 zufällig gewählt. Die **while**-Schleife in Zeile 5 wird solange durchlaufen, bis alle Knoten des Graphen von der Ameise besucht wurden. Die Liste *tour* enthält

die bisherige von der Ameise gelaufene Tour in Form eine Knotenliste. Die Knotenliste *js* enthält immer die noch zu besuchenden Knoten, entspricht also dem Ausdruck Γ_k in Formel (8.5). Die in Zeile 8 definierte Liste *ps* enthält die (noch nicht normierten) Übergangswahrscheinlicheiten: $ps[k]$ enthält die relative Wahrscheinlichkeit, dass als Nächstes der Knoten $js[k]$ gewählt wird; dies entspricht genau dem Teilausdruck $p(i,j) \cdot \frac{1}{w(i,j)^\beta}$ aus Formel (8.5). Die Funktion *chooseIndex* (siehe Aufgabe. 8.21) wählt auf Basis von *ps* per Zufallsentscheidung den nächsten Knoten aus, den die Ameise besucht.

```
1  def ant(graph):
2    def w(i,j): return graph.w(i,j)[0]
3    def p(i,j): return graph.w(i,j)[1]
4    tour = [] ; n = graph.numNodes ; i = randint(1,n)
5    while len(tour)<graph.numNodes-1:
6      tour.append(i)
7      js = [ j for j in range(1,n+1) if j not in tour ] # Liste der verbleibenden Knoten
8      ps = [ p(i,j)*1./(w(i,j)**beta) for j in js ]     # Liste der Wahrscheinlichkeiten
9      i = js[chooseIndex(ps)]                           # Nächster Knoten
10   tour = tour + [tour[0]]                             # Rundtour!
11   return tour, pathVal(graph,tour)
```

Listing 8.13: *Simulation einer Ameise*

Aufgabe 8.21

Implementieren Sie die Funktion *chooseIndex*, die eine Liste von Zahlen $[x_1, \ldots x_n]$ übergeben bekommt und mit Wahrscheinlichkeit p_i die Zahl i zurückliefert, wobei

$$p_i = \frac{x_i}{\sum_{k=1}^{n} x_k}$$

Pheromon-Anpassung. Wurde der Graph von allen Ameisen vollständig durchlaufen, wird der Pheromongehalt folgendermaßen angepasst: Zum Einen verflüchtigt sich ein Teil des Pheromons; zum Anderen erhöht jede Ameise das Pheromon auf den von ihr verwendeten Kanten umgekehrt proportional zur Länge der von ihr gelaufenen Tour. Bei einer langen Tour wird das Pheromon also um einen geringen Betrag erhöht, während bei einer kurzen Tour das Pheromon um einen verhältnismäßig großen Betrag erhöht wird.

$$p(i,j) := (1-\alpha) \cdot p(i,j) + \sum_{k=1}^{m} \Delta p_k(i,j) \tag{8.6}$$

wobei

$$\Delta p_k(i,j) = \begin{cases} \frac{1}{pathVal(tour_k)}, & \text{falls } (i,j) \in tour_k \\ 0, & \text{sonst} \end{cases}$$

Hierbei ist:

- $pathVal(t)$: die Länge der Tour t
- $tour_k$: die von Ameise k gegangene Tour
- m: die Anzahl der verwendeten Ameisen
- α: der Zerfallsparameter – je größer α, desto flüchtiger ist das modellierte Pheromon.

Implementierung der Pheromon-Anpassung. Wir teilen die Umsetzung von Formel (8.6) auf zwei Funktionen auf. Am Ende eines Zyklus, nachdem alle Ameisen über den Graphen gelaufen sind, lässt die in Listing 8.14 gezeigte Funktion *vapourize* Pheromon auf jeder Kante „verdunsten“. Die Zuweisung in Zeile 5 entspricht hierbei dem ersten Summanden in Formel (8.6).

```
1 def vapourize(graph):
2   for i in range(1,graph.numNodes+1):
3     for j in range(1,graph.numNodes+1):
4       (w,p) = graph.w(i,j)
5       p_neu = (1.-alpha)*p
6       graph.addEdge(i,j,(w,p_neu))
```

Listing 8.14: *Diese Funktion lässt einen durch α bestimmten Teil von Pheromon auf jeder Kante von „graph“ verdunsten.*

Die in Listing 8.15 gezeigte Funktion *adapt* setzt den zweiten Summanden aus Formel (8.6) um. Jede Ameise erhöht auf den Kanten „ihrer“ Tour den Pheromonwert um den Kehrwert der Länge der Tour. In Zeile 5 in Listing 8.15 wird diese Anpassung berechnet.

```
1 def adapt(graph, tour, L_k):
2   L_kInv = 1./L_k
3   for i in range(len(tour)-1):
4     (w,p) = graph.w(tour[i],tour[i+1])
5     p_neu = p + L_kInv
6     graph.addEdge(tour[i],tour[i+1],(w,p_neu))
```

Listing 8.15: *Diese Funktion erhöht Pheromon auf den Kanten einer Tour „tour“ antiproportional zur Länge L_k dieser Tour.*

Implementierung eines ACO-Zyklus. Als einen ACO-Zyklus bezeichnen wir einen kompletten Durchlauf aller Ameisen durch den Graphen zusammen mit der anschließenden Pheromon-Anpassung. Listing 8.16 zeigt die Implementierung eines ACO-Zyklus.

```
1 def acoCycle(graph):
2   tours = [ant(graph) for _ in range(M)]
3   vapourize(graph)
4   for (t, tl) in tours: adapt(graph,t, tl)
5   tours.sort(key=lambda x:x[1])
6   return tours[0][1] # Länge der kürzesten Tour
```

Listing 8.16: *Implementierung eines ACO-Zyklus: Alle M Ameisen durchlaufen den Graphen; anschließend werden die Pheromone auf den Kanten angepasst.*

Zunächst werden in Zeile 2 die M Ameisen „losgeschickt“ und die von ihnen gelaufenen Touren in der Liste *tours* aufgesammelt. Der Aufruf von *vapourize* in Zeile 3 lässt anschließend Pheromon verdampfen. In Zeile 4 werden die Pheromon-Werte auf allen Touren entsprechend dem zweiten Summanden aus Formel (8.6) erhöht. In Zeile 5 werden die Touren ihrer Länge nach sortiert, um schließlich die Länge der kürzesten Tour zurückzuliefern.

Aufgabe 8.22

In Zeile 5 in Listing 8.16 werden die Touren ihrer Länge nach sortiert, um schließlich die kürzeste Tour zurückzuliefern, die in diesem Zyklus von einer Ameise gelaufen wurde.

(a) Es gibt jedoch eine schnellere Methode – zumindest was die asymptotische Laufzeit betrifft – die kürzeste Tour zu erhalten. Welche?

(b) Implementieren Sie mit Hilfe dieser Methode eine schnellere Variante von *acoCylceH*.

(c) Führen mit Hilfe von Pythons *timeit*-Modul Laufzeitmessungen, um zu prüfen, ob *acoCycleH* tatsächlich performanter ist als *acoCylce*.

8.6.2 Verbesserte Umsetzung

Beim bisherigen Vorgehen durchläuft jede Ameise die Knoten des Graphen komplett; dann wird die Pheromonmenge auf allen Kanten aktualisiert und anschließend eine weitere Iteration durchgeführt, usw. Mit diesem Vorgehen kann man – bei Wahl geeigneter Parameter – zwar gute Touren finden, jedoch ist die Methode zu aufwändig, als dass sie auf große Probleme (mit mehr als 100 Knoten) angewendet werden könnte.

Wir stellen im Folgenden pragmatische Verbesserungen und Erweiterungen vor, mit denen auch größere TSP-Probleme in angemessener Zeit bearbeitet werden können.

Modifikation der Übergangsregel. Über eine Zufallszahl q_0 wird bestimmt, ob Formel (8.5) verwendet wird, oder ob einfach nicht-probabilistisch die „beste“ (in Bezug auf Länge und Pheromongehalt) Kante gewählt wird. Für die Bestimmung des nächsten

Knotens j, ausgehend von einem Knoten i ergibt sich also für Ameise k die folgende neue Formel:

$$j = \begin{cases} \max_{v \in \Gamma_k(i)} \left\{ \frac{p(i,v)}{w(i,v)^\beta} \right\}, \text{ falls } random() \leq q_0 \\ \text{Bestimme } j \text{ aus (8.5)}, \quad \text{sonst} \end{cases} \tag{8.7}$$

wobei *random*() eine Zufallszahl auf dem Interval $[0, 1)$ ist.

Einführung einer lokalen Pheromon-Anpassung. Zusätzlich zur im nächsten Abschnitt beschriebenen (globalen) Pheromon-Anpassung, kommt nun noch eine lokale Pheromon-Anpassung: Von jeder Ameise wird auf den von ihr besuchten Kanten eine Pheromon-Anpassung folgendermaßen durchgeführt:

$$p(i,j) = (1 - \rho) \cdot p(i,j) + \rho \cdot p_0 \tag{8.8}$$

Hierbei ist:

- p_0 Eine Pheromon-Konstante. Ein möglicher einmalig berechneter Wert hierfür, der sich in Experimenten bewährt hat, ist:

  $$p_0 = \frac{1}{pathVal(tour_{nn})}$$

  wobei $tour_{nn}$ die durch die Nearest-Neighbor-Heuristik gefundene „optimale“ Rundtour durch den Graphen ist.
- ρ Weiterer Zerfallsparameter

Implementierung der modifizierten Übergangsregel und lokalen Pheromon-Anpassung. Listing 8.17 zeigt den modifizierten Python-Code zur Simulation einer Ameise. Die Ameise gehorcht der in Formel (8.7) beschriebenen modifizierten Übergangsregel. Diese wird in den Zeilen 11 bis 15 umgesetzt. Zusätzlich wird auf jeder gegangenen Kante mittels der lokalen Funktion *adaptLocal* eine lokale Pheromon-Anpassung durchgeführt; dies geschieht zum Einen in Zeile 16 innerhalb der **while**-Schleife, und in Zeile 18 für die zuletzt einfügte Kante zurück zum Ausgangsknoten. Die ab Zeile 5 definierte Funktion *adaptLocal* realisiert genau die in Formel (8.8) beschriebene lokale Anpassung.

```
1  def ant(graph):
2    tour = [] ;  n = graph.numNodes ; i = randint(1,n)
3    def w(i,j): return graph.w(i,j)[0]
4    def p(i,j): return graph.w(i,j)[1]
5    def adaptLocal(i,j):
6      p_neu = (1-rho)*p(i,j) + rho*p_0
7      graph.addEdge(i,j,(w(i,j),p_neu))
8    while len(tour)<graph.numNodes-1:
9      tour.append(i) ; i_old = i
10     js = [j for j in range(1,n+1) if j not in tour]
```

```
11      if random()<q_0:
12        i = max(js, key=lambda j: p(i,j)*1./(w(i,j)**beta))
13      else:
14        ps = [ p(i,j)*1./(w(i,j)**beta) for j in js]
15        i = js[chooseIndex(ps)]
16      adaptLocal( i_old , i)
17    tour = tour + [tour[0]]
18    adaptLocal(tour[-2],tour[-1])
19    return tour, pathVal(graph,tour)
```

Listing 8.17: *Simulation einer Ameise, die der modifzierten Übergangsregel gehorcht.*

Modifikation der (globalen) Pheromon-Anpassung. Formel (8.6) wird so angepasst, dass nicht mehr alle, sondern nur noch die kürzeste Tour der aktuellen Iteration betrachtet wird.

$$p(i,j) := (1-\alpha)\cdot p(i,j) + \Delta p(i,j) \tag{8.9}$$

wobei

$$\Delta p(i,j) = \begin{cases} \frac{1}{pathVal(tour_{gb})}, & \text{falls } (i,j) \in \ tour_{gb} \\ 0, & \text{sonst} \end{cases}$$

Hierbei ist $tour_{gb}$ die global-beste Tour der aktuellen Iteration.

Implementierung der modifizierten Pheronom-Anpassung. Listing 8.18 zeigt die Implementierung der globalen Pheromon-Anpassung, basierend auf einer bestimmten durch eine Ameise gegangenen Tour *tour* der Länge *L_k*.

```
1  def adaptGlobal(graph, tour, L_k):
2    L_kInv = 1./L_k
3    for i in range(len(tour)-1):
4      (w,p) = graph.w(tour[i],tour[i+1])
5      pNeu = p + L_kInv
6      graph.addEdge(tour[i],tour[i+1],(w,pNeu))
```

Listing 8.18: *Die Funktion adaptGlobal implementiert die globale Pheromon-Anpassung*

In der **for**-Schleife ab Zeile 3 werden die Pheromone auf allen Kanten der Tour *tour* um den Kehrwert *L_kInv* der Länge *L_k* der Tour erhöht. Hierbei ist *p* die alte Pheromonmenge und *pNeu* die neu berechnete Pheromonmenge; in Zeile 6 wird schließlich der alte Pheromonwert mit dem neuen überschrieben.

Aufgabe 8.23

Wenden Sie den „verbesserten“ Ameisenalgorithmus, auf das Suchen einer kurzen Rundtour durch die 100 größten Städte Deutschlands an und vergleichen Sie Ergebnisse mit denen anderer Heuristiken (etwa der Nearest-Neighbor-Heuristik, der Farthest-Insertion-Heuristik oder der Tourverschmelzung). Halten Sie hierbei – um eine gute Vergleichbarkeit zu gewährleisten – die Berechnungszeiten möglichst gleich lang.