

Dokumentation Aufgabe 1 – Weniger krumme Touren

LÖSUNGSANSATZ

Gemäß der Aufgabenstellung ist es gefordert, einen Weg durch eine zwei-dimensionale Punkteschar im euklidischen Raum zu definieren, der die Eigenschaft erfüllt, dass der Innenwinkel zwischen zwei Wegabschnitten, die aus drei aufeinanderfolgenden Wegpunkten definiert werden, stets größer als 90° ist. Der Gesamtweg, auf dem jeder Punkt genau einmal „bereist“ werden soll, ist dabei zu minimieren.

Für eine solche Punkteschar lassen sich drei für das Problem relevante Eigenschaften ableiten:

- die Länge der Strecke AB zwischen zwei Wegpunkten ist gleich der Länge der Strecke BA (Annahme der Symmetrie),
- es gilt die Dreiecksungleichung, d.h. der direkte Weg von A nach B ist immer kürzer als der Weg von A über einen dritten Punkt C nach B,
- es kann von jedem Punkt zu einem beliebigen anderen Punkt gereist werden.

Somit handelt es sich bei der Aufgabenstellung um eine Abwandlung des Travelling Salesman Problems (im Folgenden als TSP bezeichnet), bei dem nach der kürzesten Route (also der kürzesten Verbindung aller Punkte bei identischem Start- und Endpunkt) in einem Graphen (im Sinne der Informatik) gesucht wird. Die Abwandlung besteht dabei in der Einschränkung der zugelassenen Winkel zwischen jeweils zwei Wegabschnitten, sowie der Tatsache, dass Start- und Endpunkt nicht identisch sein sollen.

Das eigentliche TSP war schon Gegenstand zahlreicher wissenschaftlicher Abhandlungen, die bewiesen haben, dass dieses Problem als NP-schwer eingestuft werden muss, was bedeutet, dass es nicht optimal in polynomialem Zeit gelöst werden kann.

Die Voraussetzung für die Anwendung einiger bekannter Lösungsansätze des TSPs für diese Aufgabenstellung, ist die Überführung der gegebenen Abwandlungen in eine Instanz des TSPs. Den Aspekt unterschiedlicher Start- und Endpunkte adressiere ich dadurch, dass ich der Punkteschar einen zusätzlichen theoretischen Punkt 0 hinzufüge, dessen Abstand zu allen anderen Punkten als Null definiert ist, der als Start- und Endpunkt einer jeden Tour fungiert. Somit kann, durch nachträgliches Entfernen dieses Punktes, aus einem Rundweg wieder ein nicht-geschlossener Weg erzeugt werden.

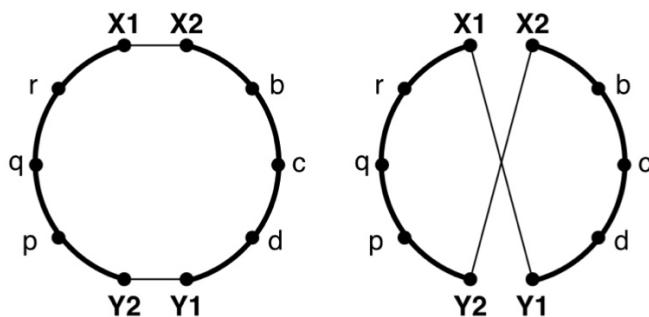
Der Beschränkung der zulässigen Winkel wird dadurch Rechnung getragen, dass ich eine zusätzliche Penalty function einfüge, die für eine Route die Zahl der unzulässigen Winkel angibt.

Meine Lösungsansätze nutzen eine Reihe von für das TSP identifizierten Optimierungsansätzen. Dabei wird stets mit einer zufällig generierten Route, die alle Punkte (inklusive des zusätzlichen Punktes 0) enthält, gestartet und diese anschließend optimiert. Es handelt sich um einen Annäherungsalgorithmus für eine optimale Lösung – es ist jedoch nicht garantiert, dass die optimale Lösung gefunden wird.

Umgesetzt habe ich zwei Versionen der sogenannten k-opt Algorithmus-Konzepte – das 2h-opt-Verfahren (Programm *OpenTSP.java*) und eine Abwandlung der Lin Kernighan Heuristik (Programm *basicLinkKernighan.py*).

Bei den k-opt-Algorithmen werden aus der Ausgangsroute k Wegabschnitte entfernt und durch neue Verbindungen, die bisher nicht Teil der Route waren, jedoch eine neue Route erzeugen, ersetzt. Diese Ersetzung wird für alle möglichen Wegabschnitte durchgeführt, was bei größer werdendem k zu exponentiell steigenden Laufzeiten führt. Es handelt sich so lange um eine Annäherung bis k gleich der Anzahl der gegebenen Punkte n ist. Erst eine n-opt Optimierung würde zur insgesamt optimalen Lösung führen.

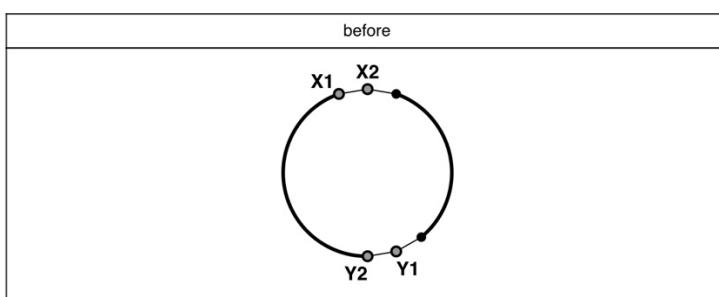
Beim sogenannten 2-opt-Algorithmus, werden genau zwei Wegabschnitte entfernt und die vier Punkte anschließend auf die in Abbildung 1 dargestellt Weise wieder verbunden.



Eine so neu entstandene Route wird als „überlegen“ übernommen, wenn entweder der resultierende Penalty-Wert kleiner ist als für die Ausgangsroute oder bei gleichem Penalty die Länge der Route reduziert wird.

Für diese neue Ausgangsroute werden dann wiederum alle möglichen 2-opt-Moves ausprobiert. Dieser Vorgang wird so lange fortgesetzt, bis keine weitere Optimierung durch 2-opt-Moves mehr möglich ist.

Eine weitere Optimierung ermöglicht der 2h-opt- (bzw. 2,5-opt-) Algorithmus, bei dem zusätzlich die in Abbildung 2 dargestellten Node Shifts geprüft werden.



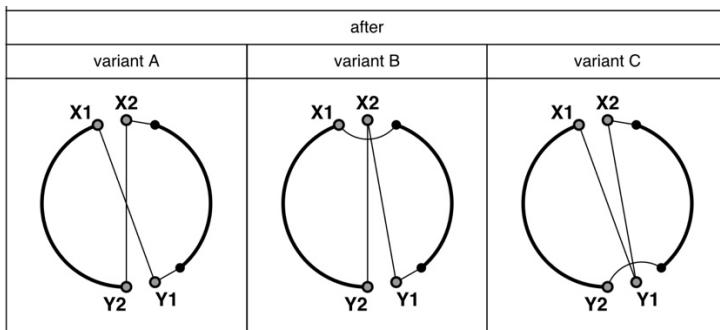


Abbildung 2: Node-Shift Logik im 2h-opt- (bzw. 2,5-opt-) Algorithmus

Analog zum 2-opt-Vorgehen funktioniert auch die 3-opt-Optimierung. Während es beim Entfernen von 2 Wegabschnitten nur zwei Möglichkeiten der neuen Verbindung gibt, sind es bei drei entfernten Abschnitten sieben mögliche neue Verbindungen (siehe Abbildung 3).

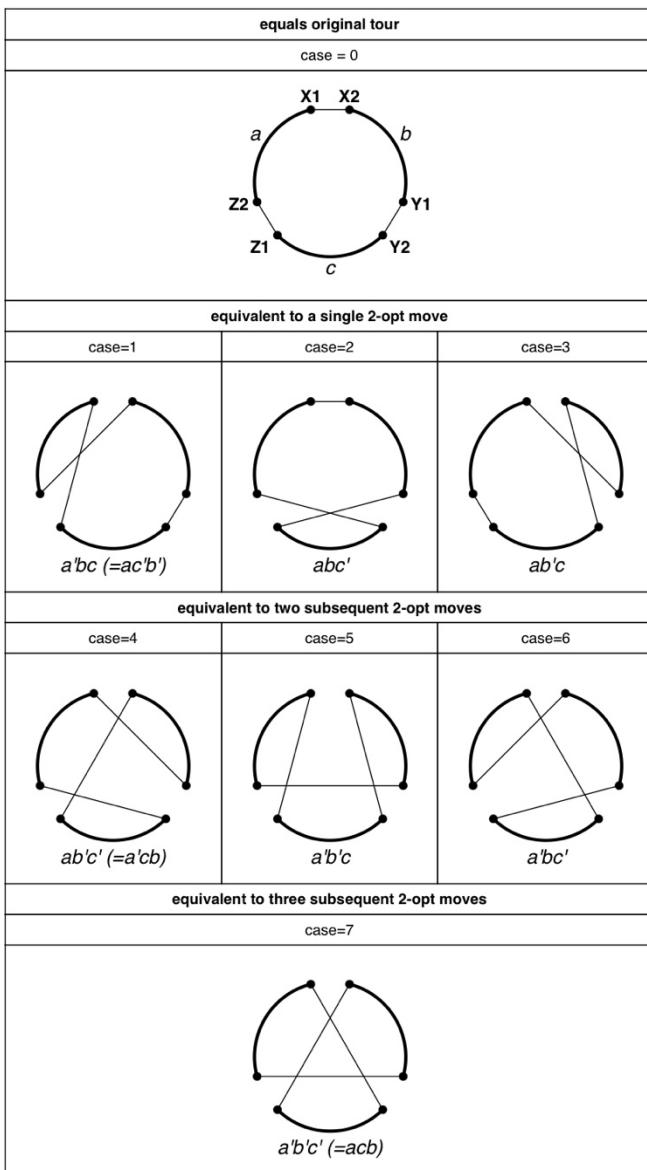


Abbildung 3: Verbindungslogik im 3-opt-Algorithmus

3-opt liefert im Vergleich zu 2-opt und 2h-opt zwar die besseren Ergebnisse, weist dafür jedoch die deutlich längere Laufzeit auf (siehe Darstellung der theoretischen Laufzeiten auf Seite 9).

Eine in meiner Programmierung umgesetzte Möglichkeit der Laufzeitoptimierung bietet dabei die „Don’t Look“-Optimierung, die verhindert, dass bereits geprüfte Wegabschnitte und die jeweiligen Ersetzungen redundant geprüft werden.

Um dem exponentiellen Zusammenhang zwischen der Laufzeit und der Qualität der Ergebnisse bei den k-opt-Verfahren zu entgehen, wurde die Lin Kernighan Heuristik entwickelt, die ich im Programm *basicLinkernighan.py* vereinfacht implementiert habe. Als einen „alternierenden Spaziergang“ bezeichnet die Literatur dazu eine Abfolge von Wegpunkten, sodass kein Wegabschnitt mehrfach vorkommt und das abwechselnd immer ein Wegabschnitt auf der aktuell geprüften Route liegt und der nächste, angrenzende Wegabschnitt nicht darauf liegt. Für jeden Spaziergang wird der sogenannte Gain festgehalten, der sich aus der, aus der Vertauschung der Wegabschnitte ergebenden, Weglängenverbesserung ergibt, vorausgesetzt, dass sich der Penalty-Wert verbessert oder gleichbleibt.

Der Lin-Kernighan Algorithmus baut solche alternierenden Spaziergänge iterativ auf. Ein Knoten wird nur dann einem Spaziergang hinzugefügt, wenn der Gain des Spaziergangs bis zu diesem Punkt insgesamt dadurch nicht negativ wird. Der Beweis in Abbildung 4 zeigt, dass der Abbruch des Spaziergangs an diesem Punkt nicht nachteilig für das Ergebnis insgesamt ist und durch die Wahl eines anderen Startpunkts, der Gain des Spaziergangs bis dahin trotzdem realisiert werden kann.

(b) Sei $P = (x_0, x_1, \dots, x_{2m})$ (Bem: $x_{2m} = x_0$) und sei k der größte Index, sodass $g(x_0, \dots, x_{2k})$ minimal ist.

Sei dann $Q := (x_{2k}, x_{2k+1}, \dots, x_{2m-1}, x_0, x_1, \dots, x_{2k})$.

Für $i = k + 1, \dots, m$ gilt dann:

$$g(x_{2k}, x_{2k+1}, \dots, x_{2i}) = g(x_0, x_1, \dots, x_{2i}) - g(x_0, x_1, \dots, x_{2k}) > 0$$

nach Wahl von k .

Und für $i = 1, \dots, k$ gilt:

$$\begin{aligned} & g(x_{2k}, x_{2k+1}, \dots, x_{2m-1}, x_0, x_1, \dots, x_{2i}) \\ &= g(x_{2k}, x_{2k+1}, \dots, x_{2m}) + g(x_0, x_1, \dots, x_{2i}) \\ &\geq g(x_{2k}, x_{2k+1}, \dots, x_{2m}) + g(x_0, x_1, \dots, x_{2k}) \\ &= g(P) > 0 \end{aligned}$$

Also ist Q geeignet. □

Abbildung 4: Beweisführung

Wenn der Endpunkt des Spaziergangs dem Startpunkt entspricht, wird der Spaziergang als geschlossen bezeichnet. Wenn ein geschlossener alternierender Spaziergang mit positivem Gain gefunden wird, werden die Wegabschnitte auf der Route gegen die Abschnitte, die nicht auf der Route liegen ausgetauscht und die Suche wird nur mit den Abschnitten, die jetzt auf der Route liegen fortgesetzt (der Gain wird von der Weglänge abgezogen und ergibt dann die neue Weglänge).

Sobald zu irgendeinem Zeitpunkt kein weiterer Wegpunkt hinzugefügt werden kann (weil zum Beispiel alle angrenzenden Abschnitte schon Teil der Abfolge sind oder sonst kein Rundweg

mehr entstehen), wird der Spaziergang abgebrochen, falls er noch keinen Wegpunkt enthält, oder die Zählvariable i für die Wegpunkte im Spaziergang wird auf das Minimum von $i-1$ und der Backtrackingtiefe gesetzt und die Suche wird von diesem Wegpunkt aus fortgesetzt. Die Backtrackingtiefe ist dabei eine vom Programmierer vorgegebene Steuerungskonstante, mit deren Hilfe die Suchtiefe und damit die Ergebnisqualität und die Laufzeit reduziert oder erhöht werden können.

Mit Hilfe der Unzulässigkeitstiefe als weitere Steuerungsvariable wird vom Programmierer festgelegt, bis zu welchem Zählerwert aus dem alternierenden Spaziergang eine abgeschlossene Route entstanden sein muss, bevor die Suche für den jeweiligen Spaziergang abgebrochen wird.

Der Vorteil des Lin-Kernighan-Algorithmus ist also, dass der k -Wert dynamisch angepasst wird, wodurch die Qualität der Lösung, bei möglichst geringer Laufzeit, erhöht wird.

Die bis hierhin dargelegten Ansätze basieren immer auf einem vollständigen Graphen und anfangs rein zufällig festgelegten Rundwegen. Alternativ ist es auch möglich mittels verschiedener Heuristiken schon eine Vorauswahl an Wegabschnitten zu treffen. Eine Möglichkeit, die sich beim TSP bewährt hat, ist die Delaunay Triangulation – noch etwas komplexer ist die POPMUSIC Metaheuristik. Diese Ansätze habe ich jedoch aufgrund ihrer Komplexität nicht selbst umgesetzt.

Der ursprüngliche von Lin und Kernighan entwickelte Algorithmus enthält noch viele zusätzliche Erweiterungen, die zur Effizienz und der Qualität der Lösungen beitragen. Später wurde er noch von Keld Helsgaun erweitert und als LKH-3-Algorithmus veröffentlicht.

Es gibt noch viele weitere Lösungsansätze auf Linear Programming (siehe Abbildung 5) basierend, die für die Identifikation der optimalen Lösung geeignet sind, bei größeren Datensätzen jedoch zu extrem langen Laufzeiten führen.

minimize :

$$\sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij}$$

subject to :

$$\sum_{\substack{i \in \text{even} \\ i \neq j}} x_{ij} = 1 \quad \text{für alle } j = 1, 2, \dots, n$$

$$\sum_{\substack{k \in \text{even} \\ k \neq j}} x_{jk} = 1 \quad \text{für alle } j = 1, 2, \dots, n$$

$$x_{ik} \in \{0, 1\}$$

$$\sum_{i \in T} \angle_{i_1 i_2 i_3} > 90^\circ$$

Abbildung 5: Mathematisches Modell zum Linear Programming

Weitere exakte Ansätze sind außerdem verschiedene Branch and Bound Ansätze (dabei werden die Bounds entweder mit Hilfe der Reduzierung der Kostenmatrix oder mit Hilfe des

Minimum Spanning Trees berechnet), auf die ich jedoch, aufgrund der ebenfalls extrem langen Laufzeiten, nicht weiter eingehe.

Wichtig ist zu beobachten, dass es Punktescharen gibt, für die keine valide Lösung existiert. Einfachstes Beispiel wäre eine Punkteschar innerhalb eines gleichseitigen Dreiecks (siehe Abbildung 6). Hier müsste nach einem der drei äußeren Punkte wieder ein anderer Punkt innerhalb des Dreiecks angebunden werden. Diese Verbindung hätte immer einen Winkel kleiner oder gleich 60° Grad.

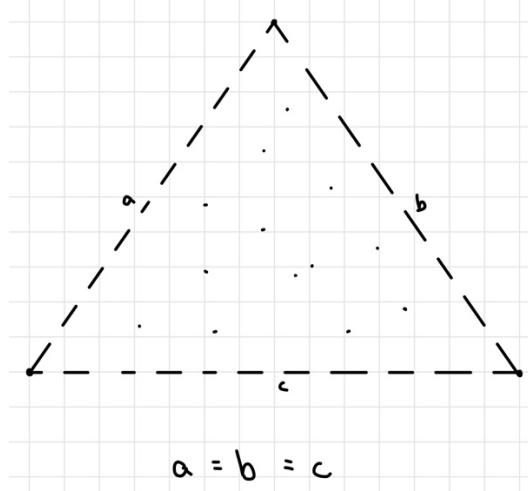


Abbildung 6: Darstellung einer beispielhaften Punkteschar ohne valide Lösung

UMSETZUNG:

Insgesamt habe ich drei verschiedene Programme umgesetzt:

1. *OpenTSP.java* – setzt das 2h-opt-Verfahren um und liefert die beste Laufzeit bei guter Qualität der Näherungslösung
2. *basicLinKernighan.py* – setzt die grundlegende Idee des Lin-Kernighan-Algorithmus um und liefert eine deutlich bessere Ergebnisqualität bei etwas längerer Laufzeit
3. *Penalty_OpenTSP.c* – ist eine Erweiterung des LKH3-Algorithmus, die den Besonderheiten der gegebenen Aufgabenstellung Rechnung trägt. Dies bietet die beste Lösung, beinhaltet jedoch weniger programmiererische Eigenleistung.

Zu 1.:

Zuerst sind einige Hilfsfunktionen definiert:

- *edgeLength*: verwendet als Parameter zwei Punkte, in Form von vier doubles und berechnet mit Hilfe des Satz des Pythagoras den Abstand zwischen beiden Punkten
- *length*: verwendet als Parameter zwei Punkte in Form von Integers, die der jeweiligen Nummerierung in der Eingabedatei entsprechen und liefert den abgespeicherten Abstand aus dem *dist-vector* oder 0, falls einer der beiden Punkte der Punkt 0 ist
- *isAcute*: verwendet drei Punkte als Parameter und prüft, ob der Innenwinkel zwischen den Punkten kleiner als 90° ist, indem geprüft wird, ob das *Dotproduct* positiv ist
- *tourLengthCalc*: berechnet die Länge des aktuelle geprüften Rundwegs

- *prev*: verwendet die Nummerierung eines Punktes und liefert den Vorgängerpunkt auf dem aktuellen Rundweg
- *next*: verwendet die Nummerierung eines Punkte und liefert den Nachfolgerpunkt auf dem aktuellen Rundweg
- *acuteAngleCnt*: zählt die Anzahl der invaliden Winkel des Rundwegs mit Hilfe einer einfachen Schleife und der *isAcute*-function
- *swap*: verwendet zwei Indices als Parameter und tauscht diese
- *createRandomTour*: kreiert mithilfe der *swap*-function einen zufälligen Rundweg, indem sie mit dem Rundweg 0 1 2 3 ... n beginnt und anschließend zufällige Indices tauscht
- *flip*: verwendet die Nummerierungen zweier Punkte als Parameter und dreht den Wegabschnitt zwischen diesen beiden Punkten mithilfe der *swap*-function um

Das Hauptprogramm beginnt mit dem Einlesen des Inputs aus der gegebenen txt-Datei und speichert die X- und Y-Werte der einzelnen Punkte in zwei Vektoren. Anhand dieser abgespeicherten Koordinaten wird nun mit Hilfe der *edgeLength*-function eine vollständige Kostenmatrix angelegt.

Die Hauptschleife des Programms gibt an, wie viele Durchläufe des Algorithmus mit unterschiedlichen Startrouten gemacht werden. Bei meinen Testungen waren ungefähr 10.000 Durchläufe notwendig, um, gemessen an den von LKH-3 gelieferten Lösungen, sehr hochqualitative Ergebnisse zu liefern. Je höher der Wert, desto höher ist die Wahrscheinlichkeit einer guten Näherungslösung.

Für jeden Durchlauf wird zuerst eine zufällige Startroute, mit Hilfe der *createRandomTour*-function erstellt. Anschließend wird das Bitarray für die „Don't Look“-Optimierung initiiert. Anfangs sind alle Werte *false*, da noch keine „Moves“ betrachtet wurden. Als nächstes wird die Hauptlösungsfunktion *twohOpt* aufgerufen. Zu Beginn wird die Ausgangslänge der Route mit Hilfe der *tourLengthCalc*-function berechnet. Die Schleife wird nun so oft wiederholt, bis für eine Tour keine 2-opt-Verbesserungen mehr gefunden werden. Die Zählvariable *failures* gibt an, wie oft, ausgehend von einem Punkt, alle möglichen 2-opt-moves keine Verbesserungen ergeben haben, und sobald diese bei der Gesamtanzahl an Punkten n angekommen ist, bedeutet dies, dass es keinen verbessernden 2-opt move mehr für diese Route gibt. Jedes Mal, wenn eine Verbesserung gefunden wird, wird *failures* auf 0 zurückgesetzt und die Suche nach Verbesserungen wird ausgehend vom nächsten Punkt fortgesetzt. Falls das „Don't Look“-Bit dieses Punktes nicht gesetzt ist, wird es nun gesetzt, bevor alle moves ausprobiert werden. Mit Hilfe von fünf Variablen a, b, c, d, e, die die fünf Punkte, die an einem 2h-opt-move (siehe oben) beteiligt sind, darstellen, werden die moves jetzt „simuliert“. Durch die *prev*- und *next*-Funktionen und eine weitere Schleife werden diese Variablen nun so verschoben, dass alle moves „ausgehend“ von b ausprobiert werden, b bleibt natürlich unverändert.

Für jeden der drei Moves wird zunächst geschaut, wie viele invalide Winkel vor dem Move vorlagen und wie viele nach diesem vorliegen würden. Wenn die Zahl danach kleiner oder gleich der Zahl davor ist und somit Weglänge eingespart wird, wird der Move mit Hilfe der *flip*-Funktion durchgeführt. Wie oben gezeigt, entspricht ein 2h-opt-Move eigentlich nur der Umkehrung von Teilstücken der Route. Anschließend werden noch die „Don't Look“-Bits der fünf beteiligten Punkte auf *false* gesetzt und *failures* wird auf 0 gesetzt. Anschließend wird der Schleifendurchlauf, der von b ausgehend alle Moves überprüft, abgebrochen, da nun eine neue Route vorliegt und das ganze Prozedere wird mit neuen b-Werten fortgesetzt, bis zur

Abbruchbedingung. Falls zu einem Zeitpunkt e gleich a wird, wird ebenfalls abgebrochen. Falls die Abbruchbedingung erreicht ist, wird geprüft, ob die 2-opt-Route, die aus diesem Durchlauf resultiert, besser ist als die, all der Durchläufe davor. Sobald alle Durchläufe mit unterschiedlichen Startrouten abgeschlossen sind, wird die beste Route ohne den Punkt 0, mit Länge und Anzahl der invaliden Winkel ausgegeben. Falls diese nicht Null ist, bedeutet dies, dass keine valide Lösung, mit ausschließlich geringen Abbiegewinkeln gefunden wurde. Theoretisch bedeutet dies aber nicht unbedingt, dass eine solche nicht existiert, da nur ein Teil der möglichen Startrouten ausprobiert wurden.

Zu 2.:

Bei meiner Implementation der Lin-Kernighan-Heuristik werden ebenfalls alle unter 1. aufgeführten Hilfsfunktionen genutzt. Zusätzlich gibt es noch drei Klassen:

die *TSP*-Klasse beinhaltet alle Funktionen und speichert alle Variablen, die zur Lösung einer Instanz des abgewandelten TSP notwendig sind,

die Klasse *Tour* enthält alle Informationen zu einer bestimmten Route, also Länge, penalty und die Funktionen, um den Vorgänger und Nachfolger zu ermitteln oder anhand eines alternierenden Spaziergangs temporär die neue Route zu erzeugen

die Klasse *KOpt* verbessert eine Startroute so lange mit Hilfe alternierender Spaziergänge bis keine Verbesserung dieser speziellen Route mehr, innerhalb der vorgegebenen Parameter (siehe oben), gefunden wird.

Insbesondere in der Klasse *KOpt* besteht auch der wichtigste Unterschied zum 2h-opt-Verfahren, denn der grundlegende Ablauf des Programms (Startroute generieren und optimieren) bleibt grundsätzlich gleich. Bei jedem einzelnen Verbesserungsdurchlauf der aktuellen Route wird ein alternierender Spaziergang aufgebaut.

Das Programm funktioniert rekursiv, sodass, wenn es bei der Wahl eines Wegabschnitts für den Spaziergang nicht mehr weitergeht, zurückgegangen werden kann. Wenn ich also von „Möglichkeiten ausprobiert“ spreche, meine ich, dass, wenn die erste Möglichkeit zu keiner Lösung führt, die nächste ausprobiert wird. Es wird also im Rekursionsstack wieder nach oben gegangen. Somit werden für die Wahl des Startpunkts alle Punkte ausprobiert und anschließend auch die beiden Optionen für den ersten Wegabschnitt, der zu entfernen wäre. Für den ersten Wegabschnitt, der nun wieder hinzugefügt werden soll, werden die 5 nächsten, zu diesem Punkt gehörenden Punkte ausprobiert (Diese Einschränkung stammt von Lin und Kernighan, um den Suchraum zu reduzieren. Es handelt sich hierbei um eine Heuristik, d.h. es besteht keine Garantie, dass die optimale Lösung nicht fälschlicherweise übergangen wird). Während der Wahl dieser Punkte wird immer parallel abgespeichert, wie groß der derzeitige *Gain* ist. Dabei ist anzumerken, dass hier Penalty und Distanz im Sinne des Gains verrechnet miteinander werden. Falls der Innenwinkel kleiner 90° ist, wird die Differenz von 90° und dem Winkel mit 100 multipliziert und zur Weglänge addiert. Dies beruht darauf, dass sich so bessere Ergebnisse in der Praxis ergeben haben, als würde man den Penalty einzeln betrachten.

Die Wahl der weiteren Punkte kann in zwei Funktionen aufgeteilt werden, die sich gegenseitig rekursiv aufrufen. Es gibt die zwei Funktionen *chooseX* und *chooseY*, die jeweils dafür da sind, entweder einen Punkt auszuwählen, der den zu entfernenden Wegabschnitt formt und einen Punkt, der den hinzuzufügenden Wegabschnitt formt. In jedem Durchlauf der *chooseX*-Funktion gibt es zwei Optionen: Vorgänger und Nachfolger aus der Route des letzten Punktes. Anschließend wird geprüft, ob der letzte derzeitige Punkt wieder mit dem Startpunkt

verbunden werden kann, um eine neue bessere Route zu bilden. Dafür wird zuerst geprüft, ob überhaupt eine valide Route entstehen würde und ob der Gain positiv wäre. Wenn beides erfüllt ist, wird die Route aktualisiert. Falls nicht wird wieder ein neues Y gewählt. Sollte es sich gerade nicht um den zweiten Wegabschnitt handeln, der hinzuzufügen ist, wird ausschließlich der nächste Punkt (entfernungsmäßig) ausprobiert. Daraus ergibt sich, dass nur abgebrochen wird den alternierenden Spaziergang zu erweitern, wenn eine Verbesserung gefunden wurde oder wenn „alle Optionen gemäß den getätigten Einschränkungen ausprobiert wurden“.

Letztendlich, nach allen Iterationen mit neuen Startrouten (ich habe aus Laufzeitgründen immer mit ca. 50 Durchläufen getestet), wird auch hier wieder die beste Route ohne den Punkt 0 ausgegeben.

Zu 3.:

LKH-3 basiert prinzipiell auf dem unter 2 beschriebenen Funktionsprinzip. Meine Erweiterung integriert darüber hinaus jedoch die Winkeleingrenzung. Dies wird mit einer eigenen *Penalty*-Funktion realisiert, die die Anzahl der invaliden Winkel zählt. Nun wird ähnlich zum Vorgehen im 2h-opt-Verfahren immer beim Aktualisieren einer Route erst geprüft, ob sich der *Penalty* verbessert hat oder mindestens gleichgeblieben ist, und, falls dies der Fall ist, ob sich die Weglänge verbessert hat.

Nach Ausführen des Algorithmus wird auch hier wieder die beste Route ausgegeben.

LAUFZEIT/SPEICHER:

2-opt (und 2h-opt) time complexity

→ um einen verbessерnden move zu finden, falls existiert

$O(n^2)$ - worst case

→ um mit 2-opt ein lokales Optimum zu finden

$O(2^n)$ - worst case

Lin Kernighan time complexity

→ Lin und Kernighan sprechen von $n^{2,2}$ durchschnittlich, welche LKH-3 auch erreicht

→ im worst case und vor allem bei meiner Implementation ist die Laufzeit wahrscheinlich exponentiell

2-opt (und 2h-opt) Speicher

→ $O(n^2 + n) = O(n^2)$ Kostennmatrix und Tour

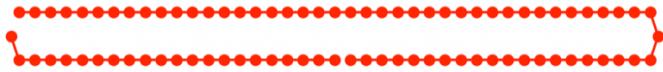
Lin Kernighan Speicher

→ $O(n^2 + n + 2n) = O(n^2)$ Kostennmatrix, Tour und alternierenden Spaziergang

BEISPIELE:

wenigerkrumm1.txt (set1.otsp)

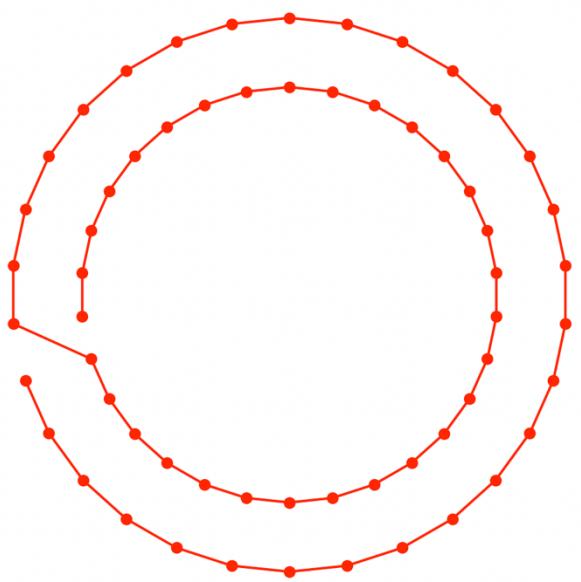
→



Länge: 847.433

wenigerkrumm2.txt (set2.otsp)

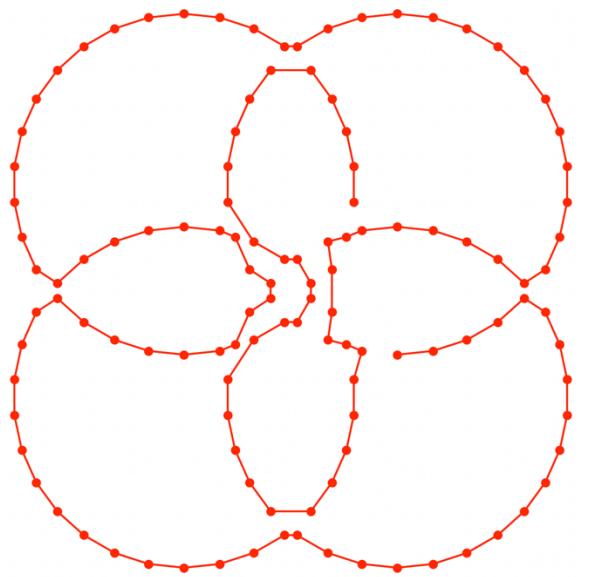
→



Länge: 2183.664

wenigerkrumm3.txt (set3.otsp)

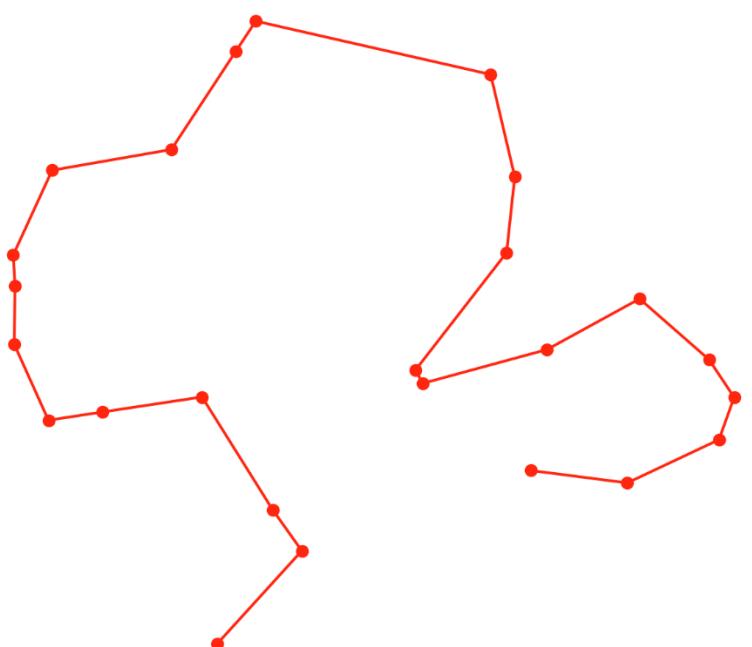
→



Länge: 1848.083

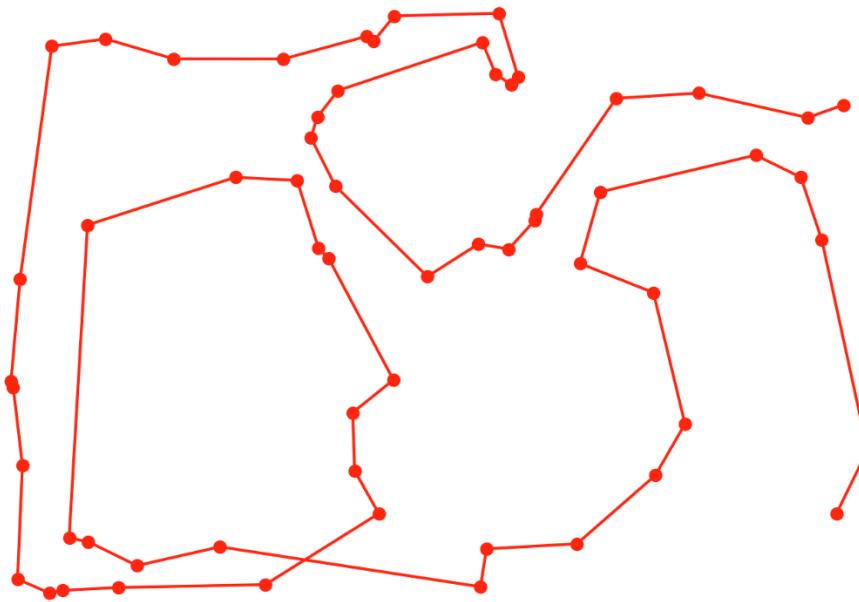
wenigerkrumm4.txt (set4.otsp)

→



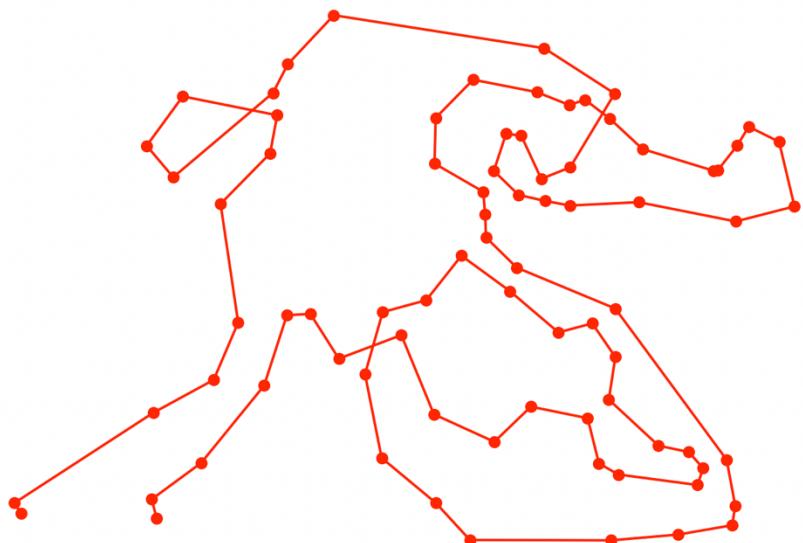
Länge: 1205.068

wenigerkrumm5.txt (set5.otsp)
→



Länge: 3257.920

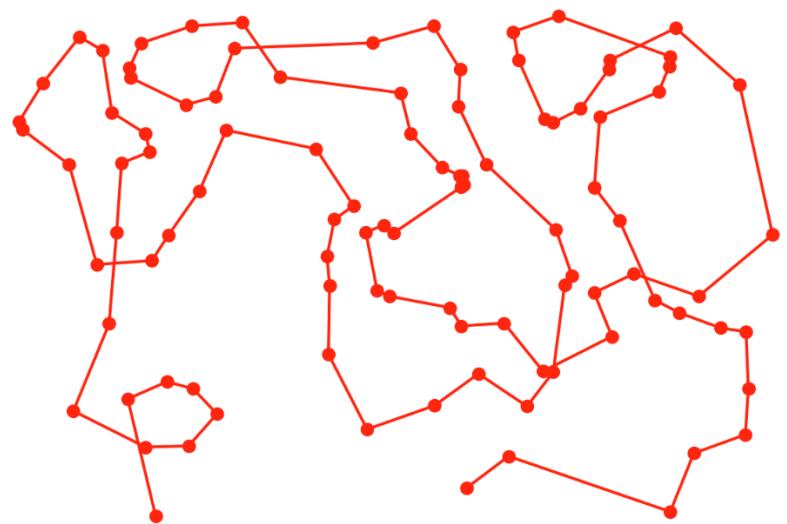
wenigerkrumm6.txt (set6.otsp)
→



Länge: 3477.188

wenigerkrumm7.txt (set7.otsp)

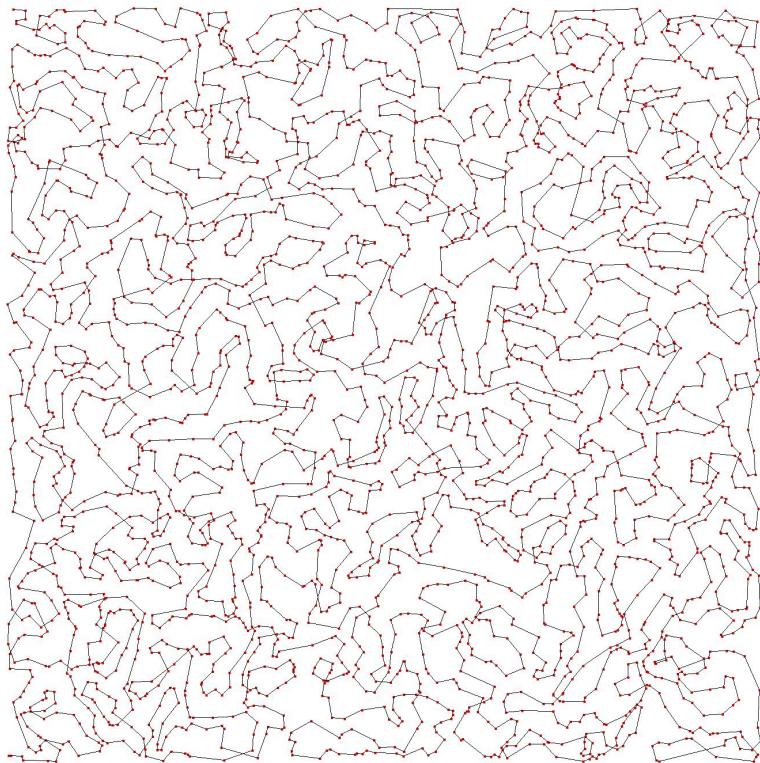
→



Länge: 4150.641

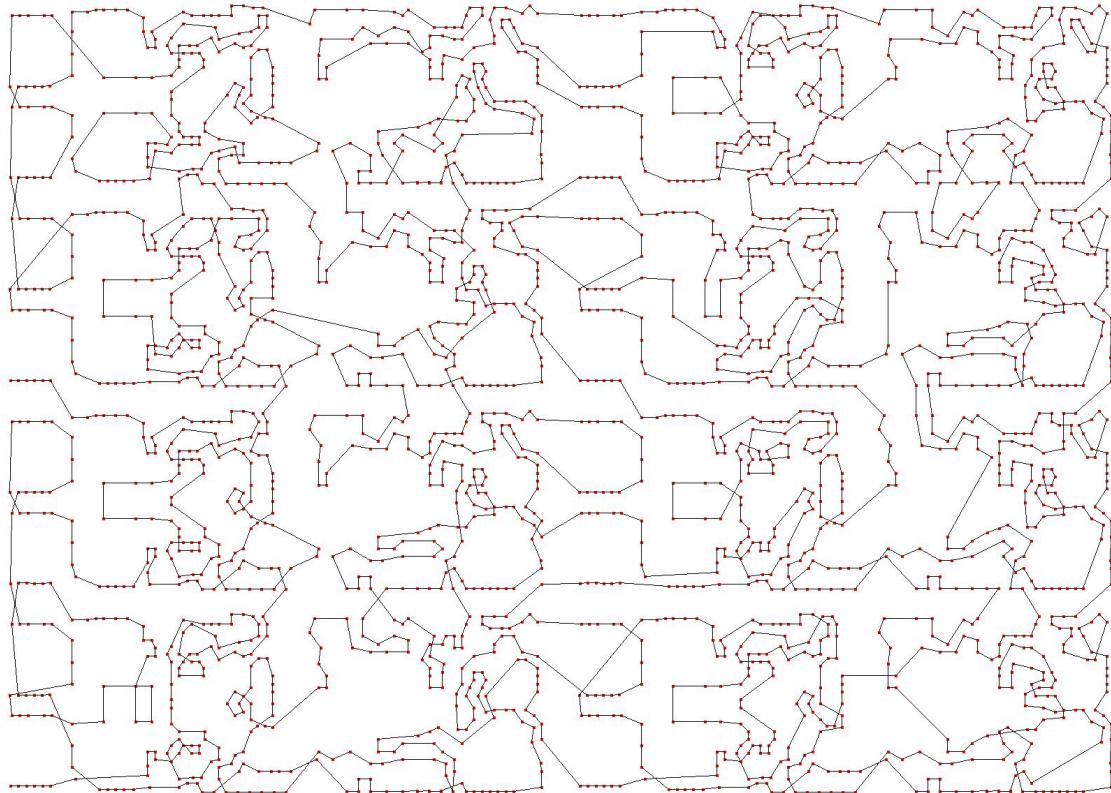
E3k.0.tsp

→



Länge: 49193833.40

pr2392.tsp
→



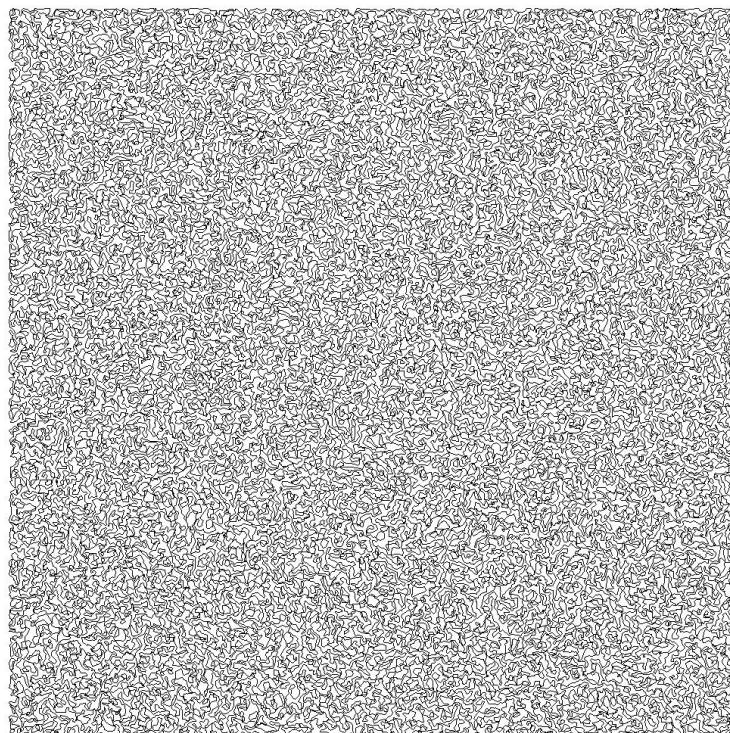
Länge: 464430.60

mona-lisa100K.tsp
→



Länge: 6289800.51

E100K.tsp
→



Länge: 272042626.36

QUELLCODE:

OpenTSP.java

```
1 // Umgesetzt in Java
2
3 // imports für benötigte externe Pakete
4 import java.io.File;
5 import java.util.Scanner;
6 import java.util.Random;
7
8 // Definierung der Hauptprogrammklasse
9 public class OpenTSP {
10
11     // Definierung (+Initialisierung) wichtiger Konstanten und Variablen
12     static final int RUNS = 100;
13
14     static double tourLength, bestTourLength = Double.MAX_VALUE;
15     static int bestAcuteAngles = Integer.MAX_VALUE;
16     static int n;
17     static double[] x, y;
18     static double[] dist[];
19     static int[] tour, bestTour, index;
20     static boolean[] dontLook;
21     static Random rand = new Random();
22
23     // Funktion zur Berechnung der Entfernung zweier Punkte
24     static double edgeLength(double x1, double y1, double x2, double y2) {
25         return Math.hypot(x1 - x2, y1 - y2);
26     }
27
28     // Funktion, die die Entfernung zweier Punkte wiedergibt, entweder aus der abgespeicherten Matrix oder neu berechnet
29     static double length(int i, int j) {
30         if (dist != null)
31             return dist[i][j];
32         if (i == n - 1 || j == n - 1)
33             return 0.0;
34         return edgeLength(x[i], y[i], x[j], y[j]);
35     }
36
37     // Funktion, die die Gesamtlänge einer Tour berechnet und wiedergibt
38     static double tourLength() {
39         double sum = length(tour[n - 1], tour[0]);
40         for (int i = 1; i < n; i++)
41             sum += length(tour[i - 1], tour[i]);
42         return sum;
43     }
44
45     // Funktion, die die Anzahl an invaliden Innenwinkel zählt
46     static int acuteAngles() {
47         int acuteAngles = 0;
48         for (int i = 0; i < n; i++)
49             if (isAcute(prev(i), i, next(i)))
50                 acuteAngles++;
51         return acuteAngles;
52     }
53
54     // Funktion, die zufällig einen Rundweg erzeugt
55     static void createRandomTour() {
56         for (int i = 0; i < n; i++)
57             index[tour[i]] = i;
58         for (int i = 1; i < n; i++)
59             swap(i, rand.nextInt(i));
60     }
61
62     // Funktion, die den Vorgänger eines Punktes in dem Rundweg wiedergibt
63     static int prev(int v) {
64         return tour[index[v] > 0 ? index[v] - 1 : n - 1];
65     }
66
67     // Funktion, die den Nachfolger eines Punktes in dem Rundweg wiedergibt
68     static int next(int v) {
69         return tour[index[v] < n - 1 ? index[v] + 1 : 0];
70     }
71
72     // Funktion, die zwei Punkte des Rundwegs vertauscht
73     static void swap(int i, int j) {
74         int temp = tour[i];
75         index[tour[i]] = tour[j] = i;
76         index[tour[j]] = temp = j;
77     }
78
79     // Funktion, die einen Abschnitt des Rundwegs dreht
80     static void flip(int b, int d) {
81         int i = index[b], j = index[d];
82         while (i != j) {
83             swap(i, j);
84             if (++i == n) i = 0;
85             if (i != j && --j < 0)
86                 j = n - 1;
87         }
88     }
89
90     // Funktion, die prüft, ob ein Innenwinkel valide ist
91     static boolean isAcute(int a, int b, int c) {
92         // falls der Dummy Node dabei ist, ist der Winkel immer valide
93         if (a == n - 1 || b == n - 1 || c == n - 1)
94             return false;
95         // prüft, ob das Dotproduct positiv ist
96         return (x[a] - x[b]) * (x[c] - x[b]) +
97                (y[a] - y[b]) * (y[c] - y[b]) > 0;
98     }
```

```
100 // Lösungsfunktion, die den 2h-opt Algorithmus implementiert
101 static void twohOpt() {
102     tourLength = tourLength();
103     int failures = 0, b = -1;
104     int bestAcuteAngles = Integer.MAX_VALUE;
105
106     // Schleife solange es Verbesserungsmöglichkeiten gibt
107     while (failures++ < n) {
108         if (++b == n)
109             b = 0;
110         // Initialisierung der Don't Look Bits
111         if (dontLook[b])
112             continue;
113         dontLook[b] = true;
114         // Initialisierung der beteiligten Punkte für eine Bewegung
115         int a = prev(b), c = next(b), d;
116         while ((d = c) != prev(a)) {
117             c = next(d);
118
119             // erste Möglichkeit die Knoten wiederzuverbinden
120             // Berechnung der invaliden Winkel vor der Bewegung
121             int acuteBeforeA =
122                 (isAcute(prev(a), a, b) ? 1 : 0) +
123                 (isAcute(a, b, next(b)) ? 1 : 0) +
124                 (isAcute(d, c, next(c)) ? 1 : 0) +
125                 (isAcute(prev(d), d, c) ? 1 : 0);
126             // Berechnung der invaliden Winkel nach der Bewegung
127             int acuteAfter =
128                 (isAcute(d, a, prev(a)) ? 1 : 0) +
129                 (isAcute(c, b, next(b)) ? 1 : 0) +
130                 (isAcute(b, c, next(c)) ? 1 : 0) +
131                 (isAcute(a, d, prev(d)) ? 1 : 0);
132             // Bedingung, ob der Rundweg aktualisiert werden soll
133             if (acuteAfter <= acuteBeforeA) {
134                 // hierfür Berechnung des Gain in Bezug auf die Weglänge
135                 double gain = length(a, b) + length(c, d) -
136                     length(a, d) - length(b, c);
137                 if (acuteAfter < acuteBeforeA ||

138                     (acuteAfter == acuteBeforeA &&
139                      tourLength - gain < tourLength)) {
140                     flip(b, d); // Ausführen der Bewegung
141                     tourLength -= gain;
142                     dontLook[a] = dontLook[b] = dontLook[c] =
143                         dontLook[d] = false;
144                     failures = 0; // Zurücksetzen von failures
145                     if (!false)
146                         System.out.printf("A: %d %.2f\n",
147                                         acuteAngles(), tourLength);
148                     break;
149                 }
150             }
151             // Weiterverschieben der Variablen zu den nächsten Punkten
152             int e = next(c);
153             if (e == a)
154                 continue;
155
156             // zweite Möglichkeit die Knoten wiederzuverbinden - analog zur ersten
157             int acuteBeforeB = acuteBeforeA +
158                 (isAcute(c, e, next(e)) ? 1 : 0);
159             acuteAfter =
160                 (isAcute(prev(a), a, c) ? 1 : 0) +
161                 (isAcute(c, b, next(b)) ? 1 : 0) +
162                 (isAcute(b, c, a) ? 1 : 0) +
163                 (isAcute(prev(d), d, e) ? 1 : 0) +
164                 (isAcute(d, e, next(e)) ? 1 : 0);
165             if (acuteAfter <= acuteBeforeB) {
166                 double gain = length(a, b) + length(d, c) + length(c, e) -
167                     length(a, c) - length(b, c) - length(d, e);
168                 if (acuteAfter < acuteBeforeB ||

169                     (acuteAfter == acuteBeforeB &&
170                      tourLength - gain < tourLength)) {
171                     flip(b, d);
172                     flip(e, a);
173                     tourLength -= gain;
174                     dontLook[a] = dontLook[b] = dontLook[c] =
175                         dontLook[d] = dontLook[e] = false;
176                     failures = 0;
177                     if (!false)
178                         System.out.printf("B: %d %.2f\n",
179                                         acuteAngles(), tourLength);
180                     break;
181                 }
182             }
183
184             // zweite Möglichkeit die Knoten wiederzuverbinden - analog zur ersten
185             e = prev(a);
186             if (e == next(c))
187                 continue;
188             int acuteBeforeC = acuteBeforeA +
189                 (isAcute(prev(e), e, a) ? 1 : 0);
190             acuteAfter =
191                 (isAcute(d, a, c) ? 1 : 0) +
192                 (isAcute(e, b, next(b)) ? 1 : 0) +
193                 (isAcute(a, c, next(c)) ? 1 : 0) +
194                 (isAcute(a, d, prev(d)) ? 1 : 0) +
195                 (isAcute(prev(e), e, b) ? 1 : 0);
196             if (acuteAfter <= acuteBeforeC) {
197                 double gain = length(a, b) + length(d, c) + length(e, a) -
198                     length(d, a) - length(a, c) - length(e, b);
199                 if (acuteAfter < acuteBeforeC ||

200                     (acuteAfter == acuteBeforeC &&
201                      tourLength - gain < tourLength)) {
202                     flip(b, d);
203                     flip(c, e);
204                     tourLength -= gain;
205                     dontLook[a] = dontLook[b] = dontLook[c] =
206                         dontLook[d] = dontLook[e] = false;
207                     failures = 0;
208                     if (!false)
209                         System.out.printf("C: %d %.2f\n",
210                                         acuteAngles(), tourLength);
211
212                     break;
213                 }
214             }
215         }
216     }
217 }
```

```
210 // Lösungsfunktion, die den 2-opt Algorithmus implementiert - analog zu 2h-opt nur, dass auschließlich eine Möglichkeit zur Wiederverbindung vorliegt
211 static void twoOpt() {
212     tourLength = tourLength();
213     int failures = 0, b = -1;
214     while (failures < n) {
215         if (-a == n)
216             b = 0;
217         if (dontLook[b])
218             continue;
219         dontLook[b] = true;
220         int a = prev(b), c = next(b), d;
221         while ((d = c) != prev(a)) {
222             c = next(d);
223             int acuteBefore =
224                 (isAcute(prev(a), a, b) ? 1 : 0) +
225                 (isAcute(a, b, next(b)) ? 1 : 0) +
226                 (isAcute(d, c, next(c)) ? 1 : 0) +
227                 (isAcute(prev(d), d, c) ? 1 : 0);
228             int acuteAfter =
229                 (isAcute(d, a, prev(a)) ? 1 : 0) +
230                 (isAcute(c, b, next(b)) ? 1 : 0) +
231                 (isAcute(b, c, next(c)) ? 1 : 0) +
232                 (isAcute(a, d, prev(d)) ? 1 : 0);
233             if (acuteAfter > acuteBefore)
234                 continue;
235             double gain = length(a, b) + length(c, d) -
236                         length(a, d) - length(b, c);
237             if (acuteAfter < acuteBefore ||
238                 (acuteAfter == acuteBefore &&
239                  tourLength - gain < tourLength)) {
240                 flip(b, d);
241                 tourLength -= gain;
242                 dontLook[a] = dontLook[b] = false;
243                 dontLook[c] = dontLook[d] = false;
244                 failures = 0;
245                 if (false)
246                     System.out.printf("%d %.1f\n", acuteAngles(), tourLength);
247                 break;
248             }
249         }
250     }
251 }
252
253 // Funktion zum Einlesen der Daten
254 static void readInstance(String fileName) throws Exception {
255     Scanner scan = new Scanner(new File(fileName));
256     n = scan.nextInt();
257     x = new double[n];
258     y = new double[n];
259     for (int i = 0; i < n; i++) {
260         x[i] = scan.nextDouble();
261         y[i] = scan.nextDouble();
262     }
263 }
264
265 // main-Funktion
266 public static void main(String[] args) throws Exception {
267     long startTime = System.currentTimeMillis();
268     readInstance(args[0]);
269     // Berechnung der Kostenmatrix
270     if (n <= 20000) {
271         dist = new double[n][n];
272         for (int i = 0; i < n - 1; i++)
273             for (int j = i + 1; j < n - 1; j++)
274                 dist[i][j] = dist[j][i] =
275                     edgeLength(x[i], y[i], x[j], y[j]);
276     }
277     tour = new int[n];
278     bestTour = new int[n];
279     index = new int[n];
280     for (int run = 1; run <= RUNS; run++) { // Schleife, die die Anzahl an Durchläufen mit neuen Starttouren angibt
281         createRandomTour();
282         dontLook = new boolean[n];
283         twoOpt(); // Aufrufen der Lösungsfunktion
284
285         // Aktualisierung des besten rundwegs falls notwendig
286         int acuteAngles = acuteAngles();
287         if (acuteAngles < bestAcuteAngles ||
288             (acuteAngles == bestAcuteAngles &&
289              tourLength < bestTourLength)) {
290             bestAcuteAngles = acuteAngles;
291             bestTourLength = tourLength;
292             System.arraycopy(tour, 0, bestTour, 0, n);
293             System.out.printf("%d: Acute angles = %d, Length = %.1f\n",
294                               run, bestAcuteAngles, bestTourLength);
295         }
296     }
297     // Ausgabe der besten Lösung ohne den Dummy Node
298     System.out.print("Path: ");
299     int dummyIndex;
300     for (dummyIndex = 0; dummyIndex < n; dummyIndex++)
301         if (bestTour[dummyIndex] == n - 1)
302             break;
303     for (int i = dummyIndex + 1; i < n; i++)
304         System.out.print(bestTour[i] + " ");
305     for (int i = 0; i < dummyIndex; i++)
306         System.out.print(bestTour[i] + " ");
307     System.out.println();
308
309     System.out.printf("Length = %.2f\n", bestTourLength);
310     System.out.printf("Acute angles = %d\n", bestAcuteAngles);
311     System.out.printf("Time used = %.2f seconds\n",
312                      (System.currentTimeMillis() - startTime) / 1000.0);
313 }
```

basicLinKernighan.py

```
1 # Umgesetzt in Python 3.8; Grundaufbau gleich zu Implementation von Arthur Matheo
2
3 # import der notwendigen externen Pakete
4 from abc import ABCMeta, abstractmethod
5 from copy import deepcopy
6 import math
7 import numpy as np
8 import random
9
10 # Definierung der TSP Klasse - speichert ein TSP Instanz
11 class TSP():
12     __metaclass__ = ABCMeta
13
14     angles = [[[1 for i in range(101)] for j in range(101)] for k in range(101)]
15
16     edges = {} # globale Kostenmatrix
17     routes = {} # globale Kosten der abgespeicherten Rundwege
18
19     # Initialisierung einer TSP Instanz anhand der gegebenen Punkte
20     def __init__(self, nodes, fast=False):
21
22         self.nodes = nodes
23         self.fast = fast
24
25         self.initial_path = list(nodes)
26         # random.shuffle(self.initial_path)
27         self.initial_cost = self.pathCost(self.initial_path)
28         self.heuristic_path = self.initial_path
29         self.heuristic_cost = self.initial_cost
30
31     # Funktion, die einen gegebenen Weg mit seinen Kosten abspeichert
32     def save(self, path, cost):
33
34         self.heuristic_path = path
35         self.heuristic_cost = cost
36
37         self.routes[str(sorted(path))] = {"path": path, "cost": cost}
38
39     # Funktion, die die strukturierte Ausgabe einer Tour ermöglicht
40     def __str__(self):
41         out = "Route with {} nodes {}\n".format(
42             len(self.heuristic_path), self.heuristic_cost)
43
44         if self.heuristic_cost > 0:
45             out += " -> ".join(map(str, self.heuristic_path))
46             out += " -> {}".format(self.heuristic_path[0])
47         else:
48             out += "No current route."
49
50         return out
51
52     # Funktion die aus der Kostenmatrix die Entfernung zweier Punkte nimmt
53     @staticmethod
54     def dist(i, j):
55         return TSP.edges[i][j]
56
57     # Funktion, die die Kosten eines Rundweges, inklusiv des Penalties berechnet
58     @staticmethod
59     def pathCost(path):
60
61         cost = 0
62
63         for i in range(len(path)):
64             if i > 2:
65                 a = calcAngle(coords[path[i-2]], coords[path[i-1]], coords[path[i]]) # Berechnung des Innenwinkels
66                 if a < 90 and a != 0:
67                     cost += 100 * (90 - a) # Verhundertfachung der Differenz zu 90 - hat sich als gute Verrechnung herausgestellt
68                 cost += TSP.dist(path[i-1], path[i])
69
70         return cost
71
72     # Initialisierungsfunktion für die Wegkanten
73     @staticmethod
74     def setEdges(edges):
75         TSP.edges = edges
76
77     # Funktion, die die Lösungsfunktion aufruft, welche einen gegebenen Weg optimiert
78     def optimise(self):
79
80         route = str(sorted(self.heuristic_path))
81
82         if route in self.routes:
83             saved = TSP.routes[route]
84             self.heuristic_path = saved["path"]
85             self.heuristic_cost = saved["cost"]
86         else:
87             self._optimise()
88
89         print(self.initial_cost)
90         print(list(self.initial_path))
91
92         return self.heuristic_path, self.heuristic_cost
93
94     @abstractmethod
95     def _optimise(self):
96         pass
97
98     # Funktion, die die Knotenreihenfolge für eine Kante wiedergibt
99     def makePair(i, j):
100         if i > j:
101             return (j, i)
102         else:
103             return (i, j)
104
105     # Funktion, die mithilfe des Dotproducts und des Arkus Kosinus den Innenwinkel berechnet
106     def calcAngle(a1, b1, c1):
107         a = np.array(a1)
108         b = np.array(b1)
109         c = np.array(c1)
110
111         ba = a - b
112         bc = c - b
113
114         cosine_angle = np.dot(ba, bc) / (np.linalg.norm(ba) * np.linalg.norm(bc)) # Dotproduct
115         angle = np.arccos(cosine_angle) # Arkus Kosinus
116
117         res = np.degrees(angle)
118
119         return res
120
```

```
121 # Klasse, die einen Rundweg repräsentiert
122 class Tour():
123
124     # Initialisierung eines Rundweges
125     def __init__(self, tour):
126         self.tour = tour
127         self.size = len(tour)
128         self.length = TSP.pathCost(tour)
129         self._makeEdges()
130
131     # Funktion, die die Liste an Kanten für den aktuellen Rundweg initialisiert
132     def _makeEdges(self):
133         self.edges = set()
134
135         for i in range(self.size):
136             self.edges.add(makePair(self.tour[i - 1], self.tour[i]))
137
138     # Funktion, die den Knoten an einem Index wiedergibt
139     def at(self, i):
140         return self.tour[i]
141
142     # Funktion, die prüft, ob eine Kante teil eines Rundwegs ist
143     def contains(self, edge):
144         return edge in self.edges
145
146     # Funktion, die mit Hilfe der calcAngle Funktion und der oben genannten Verrechnungsvorschrift den Penalty von drei Punkten berechnet
147     def penalty(self, p, i, s):
148
149         if p == 0 or i == 0 or s == 0: return 0
150
151         if TSP.angles[p][i][s] != -1:
152             angle = TSP.angles[p][i][s]
153         else:
154             angle = calcAngle(coords[p], coords[i], coords[s])
155             TSP.angles[p][i][s] = angle
156
157         if angle < 90 and angle != 0:
158             return 100 * (90 - angle)
159
160         return 0
161
162     # Funktion, die den Index eines Punktes in dem Rundweg wiedergibt
163     def index(self, i):
164         return self.tour.index(i)
165
166     # Funktion, die die beiden Knoten vor und nach einem Punkt wiedergibt
167     def around(self, node):
168         index = self.tour.index(node)
169
170         pred = index - 1
171         succ = index + 1
172
173         if succ == self.size:
174             succ = 0
175
176         return (self.tour[pred], self.tour[succ])
177
178     # Funktion, die abhängig von dem prev Parameter entweder Vorgänger oder Nachfolger eines Punktes in dem Rundweg wiedergibt
179     def pred(self, index, prev):
180         return self.tour[index - 1] if prev else self.tour[index + 1]
181
182     # Funktion, die die Veränderung an dem Rundweg anhand dem derzeitigen alternierenden Spaziergang vornimmt
183     def generate(self, broken, joined):
184
185         edges = (self.edges - broken) | joined
186
187         # Wenn zu wenige Kanten vorliegen kann kein Rundweg geformt werden
188         if len(edges) < self.size:
189             return False, []
190
191         successors = {}
192         node = 0
193
194         # Aktualisierung der Nachfolger
195         while len(edges) > 0:
196             for i, j in edges:
197                 if i == node:
198                     successors[node] = j
199                     node = j
200                     break
201                 elif j == node:
202                     successors[node] = i
203                     node = i
204                     break
205
206             edges.remove((i, j))
207
208             # jeder Punkt muss einen Nachfolger haben
209             if len(successors) < self.size:
210                 return False, []
211
212             succ = successors[0]
213             new_tour = [0]
214             visited = set(new_tour)
215
216             while succ not in visited:
217                 visited.add(succ)
218                 new_tour.append(succ)
219                 succ = successors[succ]
220
221             # Der rundweg darf keine Schleife in sich haben
222             if len(new_tour) == self.size, new_tour
223
224
```



```
352     # Funktion, die eine (weitere) Kante wählt, die entfernt werden soll
353     def chooseX(self, tour, t1, t2, last, bef, gain, broken, joined):
354         around = tour.around(last)
355
356         NOTt2iIdx = 0
357
358         for t2i in around:
359             xi = makePair(last, t2i)
360
361             NOTt2i = around[1 - NOTt2iIdx]
362             NOTt2iIdx = 1
363
364             NOTlast, option2 = tour.around(t2i)
365             if NOTlast == last: NOTlast = option2
366
367             # Gain bei aktuellen Iteration
368             Gi = gain + (TSP.dist(last, t2i) + tour.penalty(NOTt2i, last, t2i) + tour.penalty(NOTlast, t2i, last))
369
370             if xi not in joined and xi not in broken:
371                 added = deepcopy(joined)
372                 removed = deepcopy(broken)
373
374                 # Prüfen, ob der Rundweg nun wieder mit dem Spaziergang aktualisiert werden kann, d.h. dieser geschlossen ist
375                 removed.add(xi)
376                 added.add(makePair(t2i, t1))
377
378                 NOTt2, option22 = tour.around(t1)
379                 if NOTt2 == t2: NOTt2 = option22
380
381                 if t2i == t1:
382                     continue
383
384                 is_tour, new_tour = tour.generate(removed, added)
385
386                 # Prüfen, ob ein valider Rundweg vorliegt
387                 if not is_tour and len(added) > 2:
388                     continue
389
390                 # Abbruch, falls es die Lösung schonmal gab
391                 if str(new_tour) in self.solutions:
392                     return False
393
394                 relink = Gi - (TSP.dist(t2i, t1) + tour.penalty(t1, t2i, NOTlast) + tour.penalty(NOTt2, t1, t2i))
395
396                 newCost = TSP.pathCost(new_tour)
397
398                 # prüfen, dass dieser neue Rundweg ein Rundweg ist und besser ist als der alte
399                 if is_tour and relink > 0:
400
401                     # Aktualisieren der Werte
402                     self.heuristic_path = new_tour
403                     self.heuristic_cost = newCost
404
405                     return True
406
407                 else:
408                     # Sonst wird ohne den Wiederverbindungsteil der Spaziergang weiter verlängert
409                     choice = self.chooseY(tour, t1, t2, last, t2i, Gi, removed, joined)
410
411                     if len(broken) == 2 and choice:
412                         return True
413                     else:
414                         return choice
415
416
417             return False
418
419     # wählt analog zu chooseX eine Kante zum Hinzufügen zu dem Rundweg
420     def chooseY(self, tour, t1, t2, bef, t2i, gain, broken, joined):
421
422         ordered = self.closest(bef, t2i, tour, gain, broken, joined)
423
424         # wenn i = 2 ist, dann fünf vielversprechendsten, sonst nur den vielversprechendsten
425         if len(broken) == 2:
426             top = 5
427         else:
428             top = 1
429
430         for node, (_, Gi) in ordered:
431             yi = makePair(t2i, node)
432             added = deepcopy(joined)
433             added.add(yi)
434
435             # Sobald eine verbessende Bewegung gefunden wurde, aufhören
436             if self.chooseX(tour, t1, t2, node, t2i, Gi, broken, added):
437                 return True
438
439             top -= 1
440             if top == 0:
441                 return False
442
443     return False
444
```

```
445     # Funktion, die den Abstand zweier Punkte berechnet
446     def distBetween(i, j):
447         return math.sqrt((pow(abs(j[0] - i[0]), 2) + pow(abs(j[1] - i[1]), 2)) * 1.0);
448
449     # Funktion, die ursprünglich die Kostenmatrix erstellt
450     def constructMatrix(coords):
451         matrix = [[-1 for i in range(len(coords))] for j in range(len(coords))] # Initialisierung der Matrix
452
453         for i in range(len(coords)):
454             for j in range(len(coords)):
455
456                 if i == j: continue
457                 if i > j: continue
458
459                 d = distBetween(coords[i], coords[j]) # Berechnung der Abstände
460
461                 # Speicherung der Abstände
462                 matrix[i][j] = d
463                 matrix[j][i] = d
464
465         for arr in matrix:
466             arr.insert(0, 0)
467
468         matrix.insert(0, [0 for i in range(len(coords)+1)])
469         matrix[0][0] = -1
470
471     return matrix
472
473     # main-Funktion
474     if __name__ == "__main__":
475
476         # Einlesen der Daten
477         coords = []
478         with open('input.txt') as file:
479
480             for l in file:
481                 row = l.split()
482                 coords.append((float(row[0]), float(row[1])))
483
484         # Erstellen der Kostenmatrix
485         matrix = constructMatrix(coords)
486         coords.insert(0, (0, 0))
487
488         # TSP Instanz initialisieren
489         TSP.setEdges(matrix)
490         lk = Kopt(range(len(matrix)))
491
492         # Lösungsfunktion aufrufen und beste Lösung ausgeben
493         for _ in range(2):
494             path, cost = lk.optimise()
495             print("Best path has cost: {}".format(cost))
496             print([i - 1 for i in path[1:]])
497
```

Penalty_OpenTSP.c

```
1 // Umgesetzt in C
2
3 // includes für die Einbindung in LKH-3
4 #include "LKH.h"
5 #include "Segment.h"
6
7 // Funktion zur Prüfung, ob ein geformter Innenwinkel valide ist
8 static int IsAcute(Node * a, Node * b, Node * c) {
9     // schaut, ob es sich um den Dummy Node handelt
10    if (a->Id == Dimension || b->Id == Dimension || c->Id == Dimension)
11        return 0;
12    // schaut, ob das Dotproduct positiv ist
13    return (a->X - b->X) * (c->X - b->X) +
14        (a->Y - b->Y) * (c->Y - b->Y) > 0;
15 }
16
17 // Funktion, um die Anzahl an invaliden Punkten zu berechnen, die bei einer Bewegung entstehen
18 GainType Penalty_OpenTSP()
19 {
20     int i, j, AcuteBefore = 0, AcuteAfter = 0;
21     for (i = 0; i < Swaps; i++) {
22         for (j = 1; j <= 4; j++) {
23             // Auswahl der beteiligten Nodes an dem Winkel
24             Node *N = j == 1 ? SwapStack[i].t1 :
25                         j == 2 ? SwapStack[i].t2 :
26                         j == 3 ? SwapStack[i].t3 :
27                         j == 4 ? SwapStack[i].t4 : 0;
28             // Prüfen, ob die Winkel valide sind
29             AcuteBefore += IsAcute(N->OldPred, N, N->OldSuc);
30             AcuteAfter += IsAcute(PREDD(N), N, SUCC(N));
31         }
32     }
33     // Rückgabe entweder der Anzahl der dazugekommenen invaliden Winkel oder 0
34     return AcuteAfter > AcuteBefore ? AcuteAfter - AcuteBefore : 0;
35 }
36 }
```

Plotter.py (kurzes Programm, dass die Ausgaberundwege visualisiert)

```
1 # Umgesetzt in Python 3.8
2
3 # Import, um die Punktescharen auf dem Bildschirm plotten zu können
4 import matplotlib.pyplot as plt
5
6 # Definierung des Aussehens des auf dem Bildschirm erscheinenden Koordinatensystems
7 plt.xlim(-500, 600)
8 plt.ylim(-500, 400)
9 ax = plt.gca()
10 ax.set_aspect('equal', adjustable='box')
11 plt.draw()
12
13 x,y = [], [] # die beiden Arrays zur Speicherung der Koordinaten
14
15 with open('input.txt') as file:
16
17     # Einlesen der Koordinaten
18     for l in file:
19         row = l.split()
20         print(row)
21         x.append(float(row[0]))
22         y.append(float(row[1]))
23
24     # Ausgabe der Punkteschar ohne Weg
25     for i in range(0, len(x)):
26         plt.plot(x[i], y[i], 'ro-')
27
28 plt.show()
29
30 # Definierung des Aussehens des auf dem Bildschirm erscheinenden Koordinatensystems
31 plt.xlim(-500, 600)
32 plt.ylim(-500, 400)
33 ax = plt.gca()
34 ax.set_aspect('equal', adjustable='box')
35 plt.draw()
36
37 with open('output.txt') as file:
38
39     # Einlesen des auszugebenden Wegs
40     x1, y1 = [], []
41     for ele in file:
42         x1.append(int(ele)-1)
43         y1.append(y[int(ele)-1])
44
45     # Ausgabe des Wegs
46     plt.plot(x1, y1, 'ro-')
47
48 plt.show()
```

LITERATUR:

<https://www.cs.princeton.edu/~bwk/btl.mirror/tsp.pdf>
<http://webhotel4.ruc.dk/~keld/research/LKH-3/>
http://webhotel4.ruc.dk/~keld/research/LKH/LKH-2.0/DOC/LKH_REPORT.pdf
https://boelter.blog/pdf/seminararbeit_2013.pdf
<http://tsp-basics.blogspot.com/> (einige der Abbildungen stammen von dieser Quelle)