Aufgabe 1: Weniger krumme Touren

Finn Rudolph

Teilnahme-ID: 67571

16. April 2023

Inhaltsverzeichnis

1 Lösungsidee

Das Problem wird durch einen Graphen modelliert. Jeder Punkt wird einem Knoten zugeordnet und zwischen jedem Knotenpaar existiert eine ungerichtete Kante, deren Gewicht die euklidsche Distanz zwischen den zugehörigen Punkten ist. Das Ziel ist es, einen möglichst kurzen Hamiltonpfad durch diesen Graphen zu finden. Die Bedingung, dass kein Abbiegewinkel von mehr als $\pi/2$ vorkommen darf, bedeutet, dass bestimmte Tripel an Knoten nicht direkt aufeinander folgen dürfen.

Es wird angenommen, dass das Problem, einen Hamiltonpfad mit Abbiegewinkeln kleiner gleich $\pi/2$ zu finden, NP-schwer ist. Dafür konnte kein Beweis gefunden werden, es ist aber plausibel, da nah verwandte Probleme NP-schwer sind, wie beispielsweise das Finden eines Hamiltonpfads in einem allgemeinen Graphen. Insbesondere ist das Finden eines Hamiltonpfads durch eine Menge gegebener Punkte mit Einschränkung des Abbiegewinkels im Allgemeinen NP-schwer (d. h. für allgemeine maximale Abbiegewinkel). Denn von dem Problem, einen Hamiltonzyklus mit einer Einschränkung des Abbiegewinkels zu finden, wurde die NP-Schwere von Dumitrescu und Jiang [4] bereits bewiesen. Der dort vorgestellte Beweis lässt sich einfach auf Hamiltonpfade übertragen. Das zeigt zwar nicht, dass auch der Spezialfall mit $\pi/2$ NP-schwer ist, unterstützt allerdings die Annahme.

1.1 Formulierung als ganzzahliges lineares Programm

Im Folgenden wird mit d_{ij} die euklidsche Distanz zwischen Punkt i und Punkt j, für $0 \leq i, j \leq n-1$ bezeichnet, wobei n die Anzahl an Punkten ist. Daneben bezeichnet $\vec{z_i}$ den zweidimensionalen Ortsvektor zu Punkt i. $\vec{z_{ij}}$ bezeichnet den Vektor $\vec{z_j} - \vec{z_i}$. Der beschriebene Graph heißt $G = (V, E)$, mit Knotenmenge V und Kantenmenge E. Für die IP-Formulierung definieren wir die binären Variablen x_{ij} für $0 \leq i, j \leq n-1, i < j$, sodass $x_{ij} = 1$, wenn die Kante zwischen Punkt i und Punkt j in der optimalen Lösung verwendet wird, andernfalls $x_{ij} = 0$. Die Einschränkung $i < j$ ist sinnvoll, um die Anzahl nötiger Variablen zu halbieren. Im Folgenden wird der Einfachheit halber x_{ij} auch für $i > j$ geschrieben, gemeint ist dann immer x_{ji}. Das ganzzahlige lineare Programm sieht wie folgt aus.

$$
\begin{aligned}
\text{minimiere} \quad & \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} x_{ij} d_{ij} && && (1) \\
\text{sodass} \quad & \sum_{j=0, j \neq i}^{n-1} x_{ij} \geq 1 && 0 \leq i \leq n-1 && (2) \\
& \sum_{j=0, j \neq i}^{n-1} x_{ij} \leq 2 && 0 \leq i \leq n-1 && (3) \\
& \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} x_{ij} = n-1 && && (4) \\
& \sum_{i \in S} \sum_{j \in S, i<j} x_{ij} \leq |S| - 1 && S \subseteq V, S \neq \emptyset && (5) \\
& x_{ij} + x_{jk} \leq 1 && 0 \leq i, j, k \leq n-1, i \neq j \neq k, i < k, \vec{z_{ij}} \cdot \vec{z_{jk}} < 0 && (6) \\
& x_{ij} \in \{0, 1\} && 0 \leq i, j \leq n-1, i < j && (7)
\end{aligned}
$$

In der Zielfunktion (1) werden die Längen aller verwendeten Kanten summiert. Ungleichungen (2) und (3) beschränken den Grad jedes Knoten auf 1 oder 2. Gleichung (4) sorgt dafür, dass insgesamt $n-1$ Kanten verwendet werden. Ungleichung (5) ist der *Subtour Elimination Constraint (SEC)* in der Formulierung von Dantzig, Fulkerson und Johnson (DFJ)

[1], durch den die Konnektivität des Pfads sichergestellt wird. Für jede nichtleere Teilmenge S von V wird die Anzahl an Kanten, die innerhalb dieser verlaufen, auf $|S| - 1$ beschränkt. Ungleichung (6) setzt die Einschränkung des Abbiegewinkels um, wobei $\cdot$ das Skalarprodukt bezeichnet. Es wird nun gezeigt, dass die Kanten in der Lösung des IPs in (1) - (7) einen kürzesten Hamiltonpfad mit Abbiegewinkeln kleiner gleich $\pi/2$ bilden.

Wir nennen einen Hamiltonpfad mit Abbiegewinkeln kleiner gleich $\pi/2$ auch einen *zulässigen Pfad*. Damit von dem linearen Programm der kürzeste zulässige Pfad gefunden wird, muss die Menge zulässiger Pfade genau mit der Menge an Pfaden, die durch Gleichungen (2) - (7) erlaubt sind, übereinstimmen. Denn dann wird durch die Zielfunktion in (1) der kürzeste von ihnen gewählt. Zunächst wird gezeigt, dass jeder Pfad, der die Gleichungen (2) - (7) erfüllt, zulässig ist. Dafür sei $X = \{\{i, j\} : x_{ij} = 1\}$ die Menge an Kanten, die in der Lösung des ganzzahligen linearen Programms verwendet werden.

Lemma 1. *Der Graph $G' = (V, X)$ enthält keine Zyklen.*

Beweis. Für einen Widerspruch nehme man an, dass sich ein Zyklus in G' befindet, der genau die Knoten der Menge $T \subseteq V$ enthält. Dann gilt

$$\sum_{i \in T} \sum_{j \in T, i < j} x_{ij} \geq |T|$$

da ein Zyklus aus $|T|$ Knoten genau $|T|$ Kanten enthält. Das ist ein Widerspruch zu (5) mit $S = T$, folglich war die Annahme, dass G' einen Zyklus enthält, falsch. □

Lemma 2. *Die Kanten in X bilden einen Hamiltonpfad in G.*

Beweis. Wegen (4) enthält $G' = (V, X)$ genau $n - 1$ Kanten und wegen Lemma 1 ist G' azyklisch. Ein azyklischer Graph mit $n - 1$ Kanten ist ein Baum mit n Knoten, und damit ist G' ein Spannbaum von G. Da der Grad jedes Knoten von (3) auf maximal 2 begrenzt wird, hat G' genau zwei Blätter.

Denn gäbe es drei Blätter, genannt a, b, c, muss mindestens ein Knoten mindestens Grad 3 haben. Man betrachte beispielsweise den Knoten d, der auf dem eindeutigen Pfad von a zu b in G' liegt und die kürzeste Distanz zu c hat. Die erste Kante auf den Pfaden von d zu a, d zu b und d zu c muss jeweils unterschiedlich sein, d hat also mindestens Grad 3.

Der Pfad zwischen den zwei einzigen Blättern von G' muss also ein Hamiltonpfad sein. □

Ungleichung (2) ist zur Sicherstellung der gewünschten Eigenschaften von X nicht nötig, verkürzte jedoch praktisch die Laufzeit, weshalb sie mit aufgeführt ist.

Lemma 3. *Jeder Abbiegewinkel zwischen zwei aneinanderliegenden Kanten in X ist kleiner oder gleich $\pi/2$.*

Beweis. Der Abbiegewinkel α zweier Kanten $\{i, j\}$ und $\{j, k\}$ ist der Betrag des Winkels γ zwischen den Vektoren $\vec{z}_{ij}$ und $\vec{z}_{jk}$. Der Winkel zwischen zwei Vektoren $\vec{u}$ und $\vec{v}$ ist mit dem Skalarprodukt durch folgende Identität verknüpft.

$$\cos(\gamma) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}||\vec{v}|}$$

Da $|\gamma| \leq \pi/2 \iff \cos(\gamma) \geq 0$ und $|\vec{u}||\vec{v}| \geq 0$, ist $|\gamma| \leq \pi/2$ äquivalent zu $\vec{u} \cdot \vec{v} \geq 0$. Für einen Widerspruch nehme man an, dass der Abbiegewinkel zwischen zwei unterschiedlichen Kanten $\{i, j\} \in X$ und $\{j, k\} \in X$ größer als $\pi/2$ ist. Da die Kanten unterschiedlich sind, gilt $i \neq j \neq k$. Es wird außerdem angenommen, dass $i < k$, was durch Tauschen von i und k immer erreicht werden kann. Dann gilt $x_{ij} + x_{jk} = 2$ und $\vec{z_{ij}} \cdot \vec{z_{jk}} < 0$, ein Widerspruch zu (6). □

Die Einschränkung $i < k$ in Ungleichung (6) ist nicht notwendig, sie könnte auch durch $i \neq k$ ersetzt werden, wodurch jedoch Dopplungen entstehen würden. Aus Lemma 2 und 3 folgt direkt, dass jeder Pfad, der von (2) - (7) erlaubt wird, zulässig ist. Nun gilt es noch, die Umkehrung zu zeigen, dass jeder zulässige Pfad (2) - (7) erfüllt. Das ist nötig, da es sonst einen kürzesten zulässigen Pfad geben könnte, der durch (2) - (7) fälschlicherweise ausgeschlossen wird. Ungleichung (7) muss gelten, da eine Kante entweder im Pfad enthalten oder nicht enthalten ist.

Lemma 4. *Jeder zulässige Pfad erfüllt die Gleichungen bzw. Ungleichungen (2) - (4).*

Beweis. In einem Hamiltonpfad muss jeder Knoten mindestens eine angrenzende Kante haben, da der Pfad sonst nicht jeden Knoten besuchen würde. Außerdem kann kein Knoten mehr als zwei angrenzende Kanten haben, da in einem Hamiltonpfad kein Knoten zweimal besucht wird. Gleichung (4) ist erfüllt, da ein Hamiltonpfad aus n Knoten besteht und ein einfacher Pfad mit n Knoten $n-1$ Kanten besitzt. $\square$

Lemma 5. *Jeder zulässige Pfad erfüllt Ungleichung (5).*

Beweis. Wäre Ungleichung (5) nicht erfüllt, könnte man eine Menge an Knoten $T \subseteq V$ wählen, zwischen denen mindestens $|T|$ Kanten des Pfades verlaufen. Dadurch wird zwingend ein Zyklus in T geschaffen, ein Widerspruch dazu, dass die betrachteten Kanten Teil eines Hamiltonpfades sind. $\square$

Lemma 6. *Jeder zulässige Pfad erfüllt Ungleichung (6).*

Beweis. Wenn Ungleichung (6) nicht erfüllt ist, gibt es drei Knoten i, j, k, sodass $i \neq j \neq k$, $i < k$, $\vec{z_{ij}} \cdot \vec{z_{jk}} < 0$ und sowohl $x_{ij} = 1$ als auch $x_{jk} = 1$. Das heißt, sowohl die Kante $\{i, j\}$ als auch die Kante $\{j, k\}$ liegt auf dem Pfad (und $\{i, j\} \neq \{j, k\}$) und das Skalarprodukt ihrer zugehörigen Vektoren ist negativ. Wie in Lemma 3 aber bereits gezeigt wurde, ist ein negatives Skalarprodukt äquivalent zu einem Abbiegewinkel größer $\pi/2$, was der Annahme widerspricht, dass der betrachtete Pfad zulässig ist. $\square$

Satz 1. *Die Lösung des ganzzahligen linearen Programms in (1) - (7) ist ein kürzester zulässiger Pfad.*

Beweis. Folgt direkt aus den Lemmata 1 - 6 und der Minimierung der Gesamtlänge des Pfads durch (1). $\square$

Anmerkungen.

- Im Allgemeinen muss das IP in (1) - (7) keine Lösung besitzen. Ein Gegenbeispiel ist jedes Dreieck, dass keinen Innenwinkel größer oder gleich $\pi/2$ besitzt.
- Für jedes ungeordnete Tripel an Punkten $\{i, j, k\}$ werden von (6) drei Bedingungen hinzugefügt, wenn das Dreieck ijk spitzwinklig ist, andernfalls zwei Bedingungen. Im Fall eines spitzwinkligen Dreiecks könnte man die drei Bedingungen auch durch eine ersetzen, nämlich $x_{ij} + x_{jk} + x_{ki} \leq 1$. Obwohl diese Bedingung stärker als die drei einzelnen ist, verschlechterte eine Ersetzung der drei Bedingungen durch eine die Laufzeit.

1.2 Allmähliches Hinzufügen von Subtour Elimination Constraints

Die Anzahl an Bedingungen aus (5) beträgt $2^n - 1$, da für jede nichtleere Teilmenge von V ein SEC hinzugefügt wird. Das ist bereits für moderate Eingabegrößen nicht mehr praktikabel, weshalb folgendes Verfahren verwendet wird. Zu Beginn werden die SECs weggelassen und das IP optimal gelöst. Befinden sich in der Lösung Zyklen, wird für jede Knotenmenge, die in der Lösung einen Zyklus bildet, ein SEC hinzugefügt und das Verfahren wiederholt, bis die Lösung keine Zyklen mehr enthält. Das ist der übliche Weg, wie die DFJ-Formulierung von Subtour Elimination Constraints verwendet wird [1]. Die DFJ-Formulierung wurde ursprünglich für das Problem des Handlungsreisenden (TSP) entwickelt.

In [1] finden sich noch viele andere Möglichkeiten zur Formulierung von Subtour Elimination Constraints für das TSP, von denen die meisten auch auf dieses Problem übertragbar sind. Da viele von diesen für das TSP sowohl theoretisch als auch praktisch effizienter als die DFJ-Formulierung sind, soll begründet werden, warum sie dennoch gewählt wurde. Durch die Einschränkung des Abbiegewinkels werden bereits einige Subtouren eliminiert, z. B. alle mit 3 Knoten, alle mit 4 Knoten, die kein Rechteck bilden, und allgemein alle, die mindestens einen Abbiegewinkel größer $\pi/2$ haben. Für diese Subtouren sind SECs redundant, weshalb bei diesem Problem meistens schon nach wenigen Iterationen des beschriebenen Verfahrens genügend SECs vorhanden sind. Bei anderen Formulierungen wird dagegen die Komplexität des IP durch zusätzliche Variablen erhöht, da für jede andere in [1] genannte Formulierung mindestens $\Theta(n^2)$ zusätzliche Variablen nötig sind. Dafür ist bei diesen nur eine einzige Lösung des IPs nötig. Mit der DFJ-Formulierung kann die Laufzeit aber insgesamt geringer sein, da in jeder Iteration ein deutlich kleineres IP gelöst werden muss, und aufgrund der Winkelbeschränkungen nur wenige Iterationen nötig sind. Diese Begründung konnte experimentell unterstützt werden (siehe Abschnitt Beispiele).

1.3 Heuristische Lösung für große Instanzen

Mithilfe eines IP-Solvers und der obigen Formulierung konnten alle Instanzen von BWINF optimal gelöst werden. Bei ca. 300 Knoten sind aufgrund der hohen Laufzeit und des hohen Speicherverbrauchs aber die Grenzen dieses Verfahrens erreicht. Daher soll noch ein heuristisches Verfahren vorgestellt werden, das auf dem randomisierten Algorithmus zum Finden von Hamiltonzyklen von Posá [5] basiert.

Der Algorithmus funktioniert wie folgt: Zuerst wird ein zufälliger Startknoten ausgewählt. Anschließend wird ein zufälliger Knoten gewählt, der noch nicht im Pfad ist und vom letzten Knoten erreichbar ist, ohne dass ein Abbiegewinkel größer $\pi/2$ entsteht. Existiert kein solcher Knoten, wird das Ende des Pfads als Sackgasse markiert und versucht, das andere Ende auf die gleiche Weise zu erweitern. Sind beide Enden des Pfads als Sackgasse markiert, wird eine zufällige Kante aus dem Pfad ausgewählt und entfernt. Anschließend wird der Knoten am hinteren Ende des Pfads so mit einem der Endpunkte der entfernten Kante verbunden, dass der Pfad wieder verbunden ist. Natürlich werden bei der zufälligen Wahl der Kante nur solche Kanten in Betracht gezogen, dass die neu eingefügte Kante vom Knoten am Ende des Pfads nicht die Beschränkung des Abbiegewinkels verletzt. Existiert keine solche Kante, wird das gleiche mit dem Knoten am vorderen Ende des Pfads versucht. Durch Aufbrechen des Pfads und neues Verbinden des ersten oder letzten Knotens kann der Pfad häufig wieder erweitert werden. Die beschriebene Operation wird im Folgenden *Neuverbindung* genannt. Es kann aber vorkommen, dass weder eine Neuverbindung des ersten noch des letzten Knotens möglich ist, oder dass sich eine Folge an Neuverbindungen zyklisch wiederholt. Durch eine zufällige Wahl der entfernten Kante kann das meistens verhindert werden, aber nicht immer. Für diesen Fall wird eine Oberschranke an aufeinanderfolgenden Neuverbindungen festgelegt. Wird diese überschritten, wird die gesamte Suche von neuem gestartet. $\sqrt{n}$ hat sich experimentell als geeigneter Wert dafür herausgestellt (n ist immer noch die Anzahl an Punkten). Ein zu großes Limit sorgt für eine größere Laufzeit, wenn ein ohnehin aussichtsloser Pfad noch lange versucht wird zu erweitern, mit einem zu kleinen Limit wird die Suche dagegen sehr schnell abgebrochen, obwohl eine Erweiterung noch möglich gewesen wäre. Denn manchmal ist eine längere Folge an Neuverbindungen nötig, um den Pfad wieder erweiterbar zu machen.

Das Verfahren ist im Algorithmus RandomizedObtusePath zusammengefasst. z enthält die Ortsvektoren aller Punkte, sodass $\vec{z}_i$ der Ortsvektor des i-ten Punkts ist, wie oben definiert. *path* enthält die Knoten des aktuellen Pfads in Reihenfolge, *noAddedNode* ist der Zähler für die Anzahl konsekutiver Neuverbindungen (oder, äquivalent, der Anzahl an Ausführungen der while-Schleife, in denen kein Knoten zu *path* hinzugefügt wurde). *frontIs-*

DeadEnd und *backIsDeadEnd* sind boolesche Variablen, die angeben, ob der Anfang bzw. das Ende des Pfads eine Sackgasse ist. *extendingBack* gibt an, ob der Pfad gerade hinten oder vorne erweitert wird. In der while-Schleife wird zunächst geprüft, ob das Limit von *noAddedNode* überschritten wurde. Im Pseudocode ist der Einfachheit halber nur der Fall dargestellt, wenn der Pfad gerade hinten erweitert wird, d. h. *extendingBack* = **true**, vorne funktioniert es aber ähnlich. w bezeichnet den Knoten, der als nächstes an den Pfad angehängt werden soll. Anschließend wird über jeden noch nicht besuchten Knoten x iteriert. Wenn das Anhängen von x einen Abbiegewinkel kleiner gleich $\pi/2$ erzeugt, d. h. $\vec{z}_{vu} \cdot \vec{z}_{ux} \geq 0$, wird x als Kandidat zur Erweiterung in Betracht gezogen. Dazu wird zunächst die Anzahl an Kandidaten erhöht, und anschließend w mit Wahrscheinlichkeit $1/\textit{candidates}$ auf x gesetzt. Dass damit jeder mögliche Kandidat mit gleicher Wahrscheinlichkeit gewählt wird, lässt sich durch einen simplen Beweis durch Induktion zeigen: Für einen Kandidaten stimmt die Aussage trivialerweise, und wenn der k-te Kandidat betrachtet wird, hat er eine Wahrscheinlichkeit von $1/k$, w zu werden, und jeder andere $1/(k-1) \cdot (k-1)/k = 1/k$. Allerdings wird nicht jeder mögliche Kandidat in Betracht gezogen, sondern nur die ersten $\sqrt{n}$. Das zufällige Wählen eines Kandidaten dient dazu, dass der Pfad nicht immer auf die gleiche Weise erweitert wird und so nicht immer wieder in den gleichen Sackgassen endet - um diesen Effekt zu erhalten, reicht die Betrachtung einiger Kandidaten aus. Indem nur $\sqrt{n}$ Kandidaten betrachtet werden, wird die Laufzeit verringert, und der Wert $\sqrt{n}$ hat sich experimentell als geeignet herausgestellt. Wenn w nach der for-Schleife nicht nil ist, wird der Pfad erweitert und die nächste Iteration der while-Schleife gestartet.

Andernfalls wird das hintere Ende als Sackgasse markiert. Wenn nicht beide Enden Sackgassen sind, wird *extendingBack* auf **false** gesetzt um eine Erweiterung vom anderen Ende zu versuchen (unter dem vorletzten **else**). Ansonsten wird versucht, eine Kante im Pfad zu entfernen und u mit einem ihrer Endpunkte zu verbinden. Dazu wird über alle Knoten außer den zwei letzten in *path* iteriert und geprüft, ob die Kante von u zum i-ten Knoten in *path* unter Beachtung der Einschränkung des Abbiegewinkels eingefügt werden kann. Aus den ersten $\sqrt{n}$ Knoten, bei denen das möglich ist, wird einer mit dem gleichen Verfahren wie oben zufällig ausgewählt (wieder w genannt). Es werden mit der gleichen Begründung wie oben nur die ersten $\sqrt{n}$ Kandidaten betrachtet. Wenn w nach Ende der for-Schleife nicht nil ist, wird u mit w „verbunden", indem das Suffix von *path* nach w umgekehrt wird. So sind u und w in *path* benachbart, und die Kante von w zu seinem Nachfolger wurde aus dem Pfad entfernt, da der Nachfolger von w nun am Ende des Pfads ist. Existiert kein solches w, wird das gleiche am vorderen Ende des Pfads versucht, indem *extendingBack* auf **false** gesetzt wird.

RandomizedObtusePath ist zwar in der Lage, für große Instanzen zulässige Pfade zu finden, achtet aber in keiner Weise auf deren Länge. Daher wird nach der Ausführung von RandomizedObtusePath noch die 2-opt-Heuristik verwendet, um die Länge des Pfads zu reduzieren. Die Idee der 2-opt Heuristik ist die Folgende: Solange es ein Kantenpaar gibt, sodass eine Vertauschung zweier Endknoten von diesem (also z. B. $\{u, v\}, \{x, y\}$ wird durch $\{u, x\}, \{v, y\}$ ersetzt) zu einem kürzeren Pfad führt, vertausche die Endknoten. Dabei wird natürlich darauf geachtet, keinen Zyklus und keine Abbiegewinkel größer $\pi/2$ zu kreieren.

2 Laufzeitanalyse

2.1 Analyse der Größe des ganzzahligen linearen Programms

Um die Laufzeit des Ansatzes zu charakterisieren, soll die Größe des IP in Abhängigkeit von n, der Anzahl an Knoten analysiert werden. Da es für jedes unterschiedliche Paar an Knoten eine Variable x_{ij} gibt, ist die Anzahl an Variablen $\binom{n}{2} = n(n-1)/2 = \Theta(n^2)$. Im Folgenden wird die Anzahl an Bedingungen in den Gleichungen (2) - (6) analysiert.

Von (2) und (3) werden jeweils n Bedingungen in das IP eingefügt. Gleichung (4) trägt eine Bedingung bei, sodass Gleichungen (2) - (4) insgesamt $\Theta(n)$ Bedingungen beitragen.

Algorithmus 1 : RANDOMIZEDOBTUSEPATH(z)

```
path ← [random integer between 0 and n − 1]
noAddedNode ← 0
frontIsDeadEnd ← false, backIsDeadEnd ← false
extendingBack ← false
while path.length < n do
    if noAddedNode > √n then
        reset path, noAddedNode, frontIsDeadEnd, backIsDeadEnd and restart
    if extendingBack then
        u ← last element of path, v ← second last element of path
        w ← nil
        candidates ← 0
        for node x ∉ path do
            if z⃗_vu · z⃗_ux ≥ 0 then
                candidates ← candidates + 1
                w ← x with probability 1/candidates
                if candidates > √n then
                    break
        if w ≠ nil then
            append v to the back of path
            noAddedNode = 0
            continue
        noAddedNode ← noAddedNode + 1
        backIsDeadEnd ← true
        if frontIsDeadEnd and backIsDeadEnd then
            candidates ← 0
            w ← nil
            for i ← path.length − 3 to 0 do
                if z⃗_vu · z⃗_{u,path[i]} ≥ 0 and z⃗_{u,path[i]} · z⃗_{path[i],path[i−1]} ≥ 0 then
                    candidates ← candidates + 1
                    w ← path[i] with probability 1/candidates
                    if candidates > √n then
                        break
            if w ≠ nil then
                reverse the suffix of path starting at the node after w
                backIsDeadEnd ← false
            else
                extendingBack = false
        else
            extendingBack = false
    else
        do the same for the front of path
return points in z in the permutation stored in path
```

Von Ungleichung (5) wird im schlechtesten Fall eine Bedingung für jede nichtleere Teilmenge von V eingefügt, was zu $O(2^n)$ Bedingungen führt. Die genaue Anzahl an Bedingungen von (6) hängt von der gegebenen Instanz ab. Wie bereits angemerkt wurde, werden für jedes ungeordnete Tripel an Punkten, von denen es $\Theta(n^3)$ gibt, 2 oder 3 Bedingungen eingefügt, insgesamt also $\Theta(n^3)$ Bedingungen. Das ganzzahlige lineare Programm in (1) - (7) hat also $\Omega(n^3)$ und $O(2^n + n^3)$ Bedingungen.

Ein ganzzahliges lineares Programm zu lösen ist NP-schwer und benötigt im schlechtesten Fall exponentielle Zeit in der Anzahl an Variablen. Des Weiteren wird von dem verwendeten IP-Solver der Simplex-Algorithms verwendet, dessen Laufzeit im schlechtesten Fall exponentiell in der Anzahl an Variablen + Bedingungen ist. Die Anzahl an Bedingungen ist im schlechtesten Fall ebenfalls exponentiell. Theoretisch ist die Laufzeit damit im schlechtesten Fall nicht mehr exponentiell - sondern schlechter - da eine Exponentialfunktion verkettet mit einer Exponentialfunktion $a^{(b^x)}$ keine Exponentialfunktion ist. Durch das allmähliche Hinzufügen von SECs wird das IP mehrmals gelöst, im schlechtesten Fall 2^n mal. Eine genauere Charakterisierung durch Angabe eines Terms ist nicht sinnvoll, da Details des IP-Solvers in Betracht gezogen werden müssten. Daneben zielt die Verwendung ganzzahliger linearer Programmierung nicht auf eine gute theoretische Laufzeit ab, sondern auf eine gute praktische Laufzeit. Die genannten Oberschranken sind weit von der tatsächlichen Laufzeit entfernt, wie bei den Beispielen zu sehen ist.

Der Speicherverbrauch des Verfahrens beträgt $\Omega(n^3)$ und $O(2^n n^2)$, da die beteiligten Variablen aller eingefügten Bedingungen gespeichert werden müssen. Im besten Fall sind das pro Bedingung von (6) nur konstant viele, im schlechtesten Fall pro Bedingung von (5) $O(n^2)$. Es liegt die Annahme zugrunde, dass der IP-Solver nur um einen konstanten Faktor mehr Speicher benötigt.

2.2 Analyse der Heuristik

Die Laufzeit von RandomizedObtusePath kann nicht nach oben beschränkt werden, denn es ist nicht garantiert, dass RandomizedObtusePath terminiert. Bei hinreichend unglücklicher Wahl der Knoten zur Erweiterung oder Neuverbindung kann es sein, dass nie alle Knoten zum Pfad hinzugefügt werden. Insbesondere terminiert RandomizedObtusePath nicht, wenn es keinen zulässigen Pfad in der gegebenen Instanz gibt, denn dann kann die Abbruchbedingung der while-Schleife (*path.length* $\geq n$) niemals wahr sein. Nach unten kann die Laufzeit mit $\Omega(n\sqrt{n})$ beschränkt werden. Es sind mindestens $n-1$ Iterationen der while-Schleife nötig, um alle Knoten zum Pfad hinzuzufügen. In jeder Iteration werden alle Knoten, die noch nicht im Pfad sind, für eine Erweiterung in Betracht gezogen. Nach $\sqrt{n}$ Knoten kann frühzeitig abgebrochen werden, wenn bereits $\sqrt{n}$ Kandidaten für eine Erweiterung betrachtet wurden. Alle Operationen, die für die Überprüfung nötig sind, haben konstante Laufzeit. Für das Anfügen an *path* vorne und hinten wird ebenfalls eine konstante Laufzeit angenommen (z. B. durch Verwendung einer Liste). Die Anzahl an nötigen Schritten ist also

$$\begin{aligned}
\sum_{i=1}^{n} \min(n-i, \sqrt{n}) &= \sum_{i=1}^{\lfloor\sqrt{n}\rfloor} i + \sum_{i=\lfloor\sqrt{n}\rfloor+1}^{n} \sqrt{n} \\
&= \frac{\lfloor\sqrt{n}\rfloor(\lfloor\sqrt{n}\rfloor+1)}{2} + (n - \lfloor\sqrt{n}\rfloor - 1)\sqrt{n} \\
&\geq \frac{\lfloor\sqrt{n}\rfloor^2}{2} + n\sqrt{n} - \lfloor\sqrt{n}\rfloor\sqrt{n} - \sqrt{n} \\
&= \Omega(n\sqrt{n})
\end{aligned}$$

2-opt benötigt pro Kante, die für eine Vertauschung in Betracht gezogen wird, $O(n)$ Zeit, da jede andere Kante im Pfad überprüft werden muss und der Pfad $n-1$ Kanten hat. Im Gegensatz zu RandomizedObtusePath terminiert 2-opt sicher, da die Länge des Pfads mit jeder Vertauschung streng monoton fällt. Da es eine minimale Pfadlänge gibt, die nicht unterschritten werden kann (die optimale Länge), muss 2-opt terminieren. Da die Pfadlänge eine reelle Zahl ist, sollte noch gesagt werden, dass es maximal $n!$ Pfadlängen gibt (jede Permutation der Knoten kann genau einem Hamiltonpfad zugeordnet werden), denn eine reelle Zahl könnte beliebig oft verringert werden, ohne eine bestimmte Unterschranke zu unterschreiten. Damit kann die Laufzeit von 2-opt auf $O(n! \cdot n)$ beschränkt werden. Eine bessere Schranke für die Laufzeit von 2-opt zu finden, erwies sich als schwierig. Das ist allerdings nicht verwunderlich, denn die 2-opt-Heursitik für das TSP hat im schlechtesten Fall ebenfalls eine exponentielle Laufzeit.

3 Implementierung

Die IP-Lösung und die heuristische Lösung werden in zwei getrennten Programmen implementiert, beide in C++. Sie wurden auf Linux-Systemen getestet. Als IP-Solver wird der Open-Source Solver HiGHS verwendet [2]. Eine Installationsanleitung für diesen findet sich in der Dokumentation von HiGHS. Zum Kompilieren der IP-Lösung kann im Order `aufgabe1` einfach `make ip` im Terminal ausgeführt werden, für die heuristische Lösung `make randomized`. Mit nur `make` werden beide Programme kompiliert. Die auführbaren Dateien heißen `aufgabe1_ip` und `aufgabe1_randomized`. Um z. B. die IP-Lösung auf `wenigerkrumm1` auszuführen, kann im Ordner `aufgabe1`

```
./aufgabe1_ip < beispiele/wenigerkrumm1.txt
```

ausgeführt werden. Nach `stdout` werden die Tourlänge sowie die Punkte in der Reihenfolge der gefundenen Lösung ausgegeben. Von dem Programm `aufgabe1_randomized` werden daneben noch Informationen zum Zustand der Lösung nach `stderr` ausgegeben (z. B. wann RandomizedObtusePath fertig ist und 2-opt gestartet wird). Da die 2-opt-Heuristik auf großen Instanzen sehr lange benötigt, kann mit der Kommandozeilenoption `--2-opt-time-limit [Zeit in s]` ein Zeitlimit für 2-opt festgelegt werden. Um beispielsweise die Instanz `pla85900` (selbst hinzugefügt) mit einem Zeitlimit für 2-opt von 20 s zu lösen, kann im Ordner `aufgabe1`

```
./aufgabe1_randomized < beispiele/pla85900.txt --2-opt-time-limit 20
```

ausgeführt werden. Das Zeitlimit gilt nur für 2-opt, die Laufzeit von RandomizedObtusePath wird nicht mitgezählt.

Bei der Installation von HiGHS trat auf meinem PC ein Problem auf, dessen Lösung kurz beschrieben wird. Bei der Installation nach der Anleitung in der Dokumentation wurden die Headerdateien von HiGHS in `/usr/local/include/highs/` gespeichert. Damit der Compiler eine Headerdatei findet, muss sie aber in `/usr/local/include/` liegen. Damit der Include `#include "Highs.h"` funktioniert, musste ich alle Dateien in `/usr/local/include/highs/` nach `/usr/local/include/` verschieben. Möglicherweise ist das auf anderen PCs auch nötig.

Einen Überblick über HiGHS verschafft das C++-Beispiel im GitHub-Repository von HiGHS im Ordner `examples`. Da aber noch keine ausführliche Dokumentation verfügbar ist, wird das Wichtigste hier kurz erklärt. Das ganzzahlige lineare Programm wird in ein `HighsModel` geschrieben, das dann an ein Objekt der Klasse `Highs` gegeben wird. `HighsModel` besitzt das Attribut `lp_`, in dem sich alle relevanten Datenstrukturen zur Spezifizierung des IP befinden. Die folgende Beschreibung bezieht sich auf die Attribute von `lp_`. Die Bedingungsmatrix `a_matrix_` ist eine dünnbesetzte Matrix, in der alle Einträge, die nicht 0 sind, in einem einzigen Vektor `a_matrix_.value_` nach Zeile geordnet gespeichert werden. Die Spaltenindizes der Einträge stehen in `a_matrix_.index_`. Die Längen von `a_matrix_.value_` und `a_matrix_.index_` sind also gleich. Eine Zeile nimmt immer einen kontinuierlichen Bereich in `a_matrix_.value_` bzw. `a_matrix_.index_` ein. Der Index des ersten Eintrags jeder Zeile steht in `a_matrix_.start_`, zusätzlich enthält `a_matrix_.start` als letztes Element die Länge von `a_matrix_.value_`. `a_matrix_.start_` enthält also $r + 1$ Elemente, wenn r die Anahl an Zeilen ist. Das Hinzufügen einer Zeile funktioniert wie folgt: Zu Beginn enthält `a_matrix_.start_` 0, da die aktuelle Länge von `a_matrix_.value_` 0 ist. Eine neue Zeile wird hinzugefügt, indem ihre Indizes und Koeffizienten an `a_matrix_.index_` und `a_matrix_.value_` angefügt werden. Die neue Länge von `a_matrix_.value_` wird anschließend an `a_matrix_.start_` angefügt, um die Invariante zu erhalten, dass das letzte Element in `a_matrix_.start_` die Länge von `a_matrix_.value_` ist. `row_lower_` und `row_upper_` speichern die Unter- und Oberschranke für den Wert jeder Zeile, `col_lower_` und `col_upper_` für jede Spalte. Beim Einfügen einer neuen Zeile werden Unter- und Oberschranke der Zeile jeweils an `row_lower_` bzw. `row_upper_` angefügt. `col_cost_` speichert die Koeffizienten der Kostenfunktion. Der Vektor `integrality_` enthält den Typ jeder Variablen (ob ganzzahlig oder reell).

Es erweist sich als praktisch, Punkte im zweidimensionalen Raum als komplexe Zahlen `complex<double>` zu repräsentieren, d. h. die x-Koordinate entspricht dem Realteil und y-Koordinate dem Imaginärteil.

3.1 util.hpp

3.1.1 nchoose2

```
template <typename T>
T nchoose2(T n)
```

Gibt $\binom{n}{2}$ zurück.

3.1.2 dot_product

```
template <typename T>
T dot_product(complex<T> const &a, complex<T> const &b)
```

Gibt das Skalarprodukt von a und b, als Vektoren interpretiert, zurück. Es entspricht dem Realteil von $a\bar{b}$, da $a\bar{b} = a_1b_1 + a_2b_2 - a_1b_2i + a_2b_1i$, wenn $a = a_1 + a_2i, b = b_1 + b_2i$.

3.2 aufgabe1_ip.cpp

3.2.1 edge_index

```
size_t edge_index(size_t n, size_t i, size_t j)
```

Gibt den Index der zur Kante $\{i, j\}$ zugehörigen Variable x_{ij} zurück. Die Variablen x_{ij} für $i < j$ werden lexikographisch nummeriert, das heißt x_{01} hat Index 0, x_{02} Index 1, ..., x_{12} Index $n-1$, x_{23} Index $n-1+n-2$ und so weiter. Der Term $\binom{n}{2} - \binom{n-\min(i,j)}{2}$ gibt den Anfang des „Blocks" von $\min(i, j)$ an, und $\max(i, j) - \min(i, j) - 1$ den Index innerhalb des Blocks.

3.2.2 add_angle_constraints

```
void add_angle_constraints(
    HighsModel &model, vector<complex<double>> const &z)
```

Fügt die in Ungleichung (6) bestimmten Bedingungen zu `model` hinzu. Dazu wird über jeden möglichen Scheitelpunkt (j) und alle unterschiedlichen Knotenpaare i, k mit $i \neq j \neq k$ und $i < k$ iteriert. Wenn der Winkel ijk spitz ist, werden die Variablen von $\{i, j\}$ und $\{j, k\}$ zu einer neuen Zeile hinzugefügt und ihre Koeffizienten auf 1 gesetzt. Die Unterschranke für die neue Zeile der Matrix ist 0, die Oberschranke 1.

3.2.3 add_degree_constraints

```
void add_degree_constraints(HighsModel &model, size_t n)
```

Fügt für jeden Knoten i eine Zeile in die Bedingungsmatrix ein, in der die Variablen aller Kanten aufsummiert werden, von denen i ein Endpunkt ist. Der Wert der Zeile wird auf $[1, 2]$ beschränkt, womit Ungleichungen (2) und (3) umgesetzt werden.

3.2.4 add_num_edges_constraint

```
void add_num_edges_constraint(HighsModel &model, size_t n)
```

Beschränkt die Anzahl verwendeter Kanten auf genau $n - 1$, indem alle $\binom{n}{2}$ Variablen in einer neuen Zeile aufsummiert werden, und der Wert der Zeile auf $n - 1$ fixiert wird.

3.2.5 add_subtour_elimination_constraint

```
void add_subtour_elimination_constraint(
    Highs &highs, size_t n, vector<size_t> const &tour)
```

Fügt für die Knoten in `tour` einen Subtour Elimination Constraint ein, wie in Ungleichung (5) beschrieben. Da diese Funktion nach Übergabe des `HighsModel` an das `Highs`-Objekt verwendet wird, wird die Funktion `Highs::addRow` zum Einfügen der Zeile verwendet (ansonsten müsste das `HighsModel` neu übergeben werden). Die Parameter von `Highs::addRow` sind die Unterschranke und Oberschranke der neuen Zeile, die Anzahl an Einträgen, die nicht 0 sind, und Zeiger zu den Indizes und Werten. Zuerst werden zwei Arrays `ind` und `val` angelegt. Da in der neuen Zeile die Variablen für jede Kante zwischen zwei Knoten in `tour` aufsummiert werden sollen, wird über jedes Paar an Knoten in `tour` iteriert. Der Index der Kante zwischen dem Paar wird in `ind` eingefügt und der Koeffizient in `val` auf 1 gesetzt. Schließlich wird `Highs::addRow` aufgerufen und die Oberschranke der neuen Zeile auf `tour.size() - 1` gesetzt.

3.2.6 build_graph

```
vector<vector<size_t>> build_graph(Highs const &highs, size_t n)
```

Gibt den Graphen in Form einer Adjazenzliste zurück, der genau die Kanten aus der in `highs` gespeicherten Lösung enthält. Dazu wird über jedes Knotenpaar i, j mit $i < j$ iteriert, und wenn der Wert der zu $\{i, j\}$ zugehörigen Variable 1 ist, wird die Kante $\{i, j\}$ in den Graphen eingefügt. Da die Variablen in der Lösung nur 0 oder 1 sein können, aber aufgrund von Rundungsfehlern möglicherweise nicht exakt 0 oder 1 sind, wird die Bedingung `> 0.5` anstatt `== 1` verwendet.

3.2.7 check_for_subtours

```
bool check_for_subtours(Highs &highs, size_t n)
```

Gibt zurück, ob in der in `highs` enthaltenen Lösung Subtouren existieren und eliminiert dies gegebenenfalls. Dafür werden alle Zyklen in dem von `build_graph` erstellten Graphen gefunden. Von jedem noch nicht besuchten Knoten aus wird eine Suche (ähnlich zur Tiefensuche) durchgeführt und festgestellt, ob der Ausgangsknoten `i` wieder erreicht wird. Jeder besuchte Knoten wird als besucht markiert und zu `subtour` hinzugefügt. Der Unterschied zur Tiefensuche ist, dass immer nur ein Nachbar betrachtet wird, was ausreicht, da jeder Knoten maximal Grad 2 hat. So lässt es sich iterativ implementieren und es ist keine separate, rekursive Funktion nötig. Wird `i` wieder erreicht, enthält `subtour` den gesamten Zyklus, von dem `i` Teil ist, und es wird ein Subtour Elimination Constraint für ihn hinzugefügt.

3.2.8 shortest_obtuse_path

```
vector<complex<double>
    shortest_obtuse_path(vector<complex<double>> const &z)
```

Gibt den kürzesten Hamiltonpfad mit Abbiegewinkeln von maximal $\pi/2$ als Permutation der in `z` gegebenen Punkte zurück, sowie dessen Länge. Falls keine Lösung existiert, wird ein leerer Vektor zurückgegeben.

Zunächst werden grundsätzliche Eigenschaften des `model` festgelegt, z. B., dass die Zielfunktion minimiert werden soll und die Anzahl an Spalten der Bedingungsmatrix $\binom{n}{2}$ ist. Darauf wird über jedes Knotenpaar i, j mit $i < j$ iteriert und der Koeffizient der Variable x_{ij} auf d_{ij} gesetzt. Daneben wird festgelegt, dass $x_{ij} \in \{0, 1\}$ sein muss. Anschließend werden die verschiedenen Bedingungen zu `model` hinzugefügt und dessen Anzahl an Zeilen aktualisiert.

Danach wird das Objekt `highs` angelegt und ihm `model` übergeben. In der folgenden while-Schleife wird das IP mit `Highs::run` gelöst, solange Subtouren existieren und nicht festgestellt wurde, dass es keine Lösung gibt, was durch `HighsModelStatus::kInfeasible` signalisiert wird. Nach der Lösung des IPs werden entsprechende SECs eingefügt.

Falls eine Lösung existiert, wird der Graph mit Kanten aus der Lösung mithilfe von `build_graph` erstellt. Anschließend wird ein Knoten mit Grad 1 als Startpunkt für eine Suche durch den Graphen ausgewählt. Die Suche zum Finden des Hamiltonpfads funktioniert wie in `check_for_subtours`, es wird immer ein aktueller Knoten `j` und dessen Vorgänger `last` verwaltet. Jeder besuchte Knoten wird zum Vektor `path` hinzugefügt, der am Ende zurückgegeben wird.

3.3 aufgabe1_randomized.cpp

3.3.1 randomized_obtuse_path

```
vector<complex<double>>
    randomized_obtuse_path(vector<complex<double>> const &z)
```

Gibt einen Hamiltonpfad mit Abbiegewinkeln kleiner gleich $\pi/2$ durch die in `z` gegebenen Punkte als Permutation der Punkte zurück. Die grundlegende Funktionsweise entsptricht genau der im Algorithms RandomizedObtusePath, dieser sollte zum Verständnis bekannt sein. Einige Implementierungsdetails müssen aber dennoch geklärt werden.

Der wichtigste Unterschied zum Pseudocode ist, dass hier beide Fälle, das Erweitern des Pfads vorne und hinten behandelt werden müssen. `path` wurde mit einer `deque<size_t>` umgesetzt, da eine `deque` das Anfügen vorne und hinten in amortisiert konstanter Zeit unterstützt. `path` enthält die Indizes der Punkte im aktuellen Pfad in Reihenfolge. Da alle Datenstrukturen sowohl am Anfang initialisiert als auch später möglicherweise zurückgesetzt werden müssen, wurde das Lambda `restart_search` eingefügt - das vermeidet Codewiederholung. `restart_search` setzt mit `srand(time(0))` auch einen Seed-Wert (abhängig vom aktuellen Zeitpunkt) für die Funktion `rand()` zum Generieren pseudozufälliger Zahlen. Um schnell über alle noch nicht besuchten Knoten iterieren zu können, wird eine verkettete Liste `unvisited` verwendet, in der stets alle Knoten enthalten sind, die nicht in `path` sind. `u` und `v` bezeichnen im Quellcode den letzten bzw. ersten und vorletzten bzw. zweiten Knoten im Pfad, je nachdem, ob gerade vorne oder hinten erweitert wird. Bei der Auswahl des Knotens zur Erweiterung des Pfads wird nicht dessen Index gespeichert, sondern ein Iterator zu seiner Position in `unvisited` - so kann er in $O(1)$ aus `unvisited` entfernt werden. Um `w` mit einer Wahrscheinlichkeit von `1 / candidates` zuzuweisen, wird die Funktion `rand()` aus der Standardbibliothek verwendet. Sie gibt eine Ganzzahl zwischen `0` und `RAND_MAX` (meist das gleiche wie `INT_MAX`) zurück. Wenn ihr zurückgegebener Wert $\equiv 0 \mod$ `candidates` ist, wird `w` aktualisiert. Beim Anfügen von `*w` an den Pfad muss unterschieden werden, ob gerade vorne oder hinten erweitert wird. Beim Neuverbinden des Pfads (zweite Hälfte der äußeren while-Schleife) wird abhängig vom Wert von `extendingBack` beim vorvorletzten

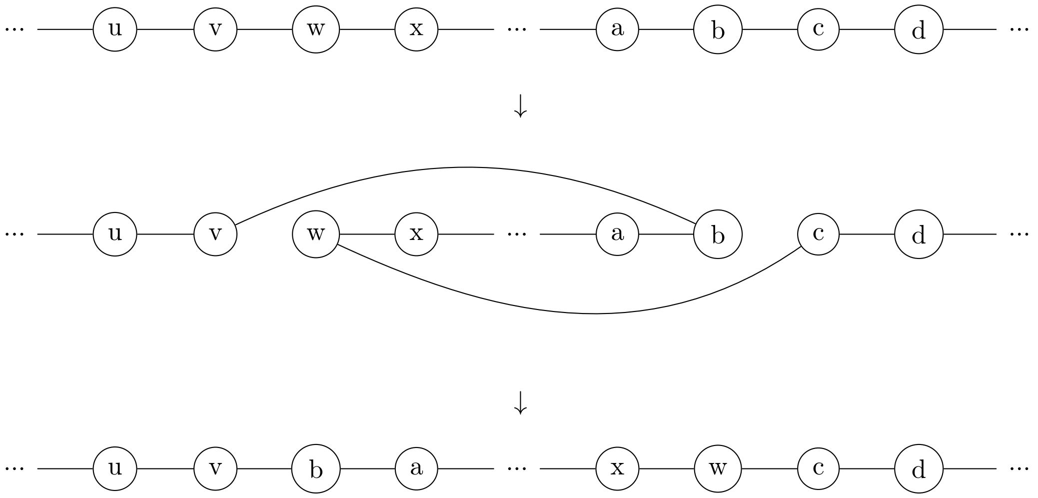


Abbildung 1: Ablauf einer Vertauschung von Endknoten zweier Kanten. Die Lage der Knoten entspricht nicht der Lage ihrer zugehörigen Punkte im zweidimensionalen Raum.

oder dritten Element von `path` gestartet. In der innerern while-Schleife ist `j` der Knoten vor bzw. nach `i` in `path`, der für die Überprüfung der Abbiegewinkel benötigt wird. Nachdem überprüft wurde, ob `i` als Kandidat in Frage kommt, wird `i` abhängig von `extendingBack` nach weiter vorne oder hinten in `path` bewegt. `w` enthält den Index des aktuell ausgewählten Knotens für die Neuverbindung. So kann mithilfe von `reverse` aus der Standardbibliothek einfach das Suffix ab `w` bzw. Präfix bis `w` des Pfades umgekehrt werden.

3.3.2 optimize_path

```
vector<complex<double>> optimize_path(
    vector<complex<double>> const &path, double time_limit)
```

Wendet die 2-opt-Heuristik auf `path` mit Beachtung von Abbiegewinkeln an. Zunächst wird die aktuelle Zeit in `start_time` gespeichert, um das `time_limit` beachten zu können. Um effizient Kanten aus dem Pfad entfernen und neu einfügen zu können, wird in dem Vektor `nodes` für jeden Knoten der Vorgänger (Index `0`) und Nachfolger (Index `1`) im Pfad verwaltet. Für die zwei Endknoten wird der nicht existente Vorgänger bzw. Nachfolger auf `SIZE_MAX` gesetzt. In einer Warteschlange `q` werden alle noch zu bearbeitenden Kanten (repräsentiert als `pair<size_t, size_t>`) verwaltet. In der for-Schleife am Anfang wird der Vorgänger und Nachfolger jedes Knoten auf den den direkt vorangehenden bzw. folgenden Index gesetzt, sodass der in `nodes` gespeicherte Pfad anfänglich genau dem in `path` entspricht. Daneben wird in der for-Schleife jede Kante in beiden Richtungen in `q` eingefügt.

In der folgenden while-Schleife wird in jeder Iteration genau eine Kante aus `q` bearbeitet. Ihre Knoten heißen `v` und `w`. Da die Kante durch vorherige Operationen entfernt worden sein könnte, wird zunächst überprüft, ob `v` `w` noch als Nachbarn hat. In einer Iteration wird nach möglichen Kanten zum Tauschen in dem Teil des Pfads gesucht, wenn man sich ausgehend von `w` in der Richtung `v` $\rightarrow$ `w` von der Kante wegbewegt. Nur eine Richtung pro Iteration zu bearbeiten, verkürzt die Implementierung deutlich, da der Pfad nur in eine Richtung abgelaufen werden muss. Die boolesche Variable `direction` gibt an, ob man vorwärts auf dem Pfad läuft. Sie wird als Index in `nodes` verwendet, um den Vorgänger bzw. Nachfolger in der aktuellen Laufrichtung zu erhalten (z. B. gibt `nodes[w][direction]` den Nachfolger von `w` in Laufrichtung). Für einen Überblick über den Rest der Variablen ist Abbildung 1 hilfreich. `{b, c}` ist immer der aktuell betrachtete Tauschpartner, die übrigen 4 Knoten müssen zur Überprüfung der neuen Abbiegewinkel verwaltet werden.

In jeder Iteration der inneren while-Schleife wird versucht, die Kanten `{v, w}` und `{b, c}` durch `{v, b}` und `{w, c}` zu ersetzen. Dafür müssen die 4 Abbiegewinkel `uvb, vba, xwc` und `wcd` überprüft werden (großes `if`-statement), wie in Abbildung 4 zu sehen. Da `v` und `c` Endpunkte des Pfads sein können, können `u` und `d` gleich `SIZE_MAX` sein - dann gibt es keinen Abbiegewinkel, der überprüft werden muss. Daneben muss die Vertauschung der Endpunkte auch eine Verkürzung des Pfads bringen. Sind all diese Bedingungen erfüllt, wird der Pfad zwischen `x` und `a` umgekehrt sowie der Vorgänger und Nachfolger aller beteiligter Knoten aktualisiert. Die Operationen können gut anhand von obigem Diagramm nachvollzogen werden. Anschließend werden die zwei neuen Kanten in beiden Richtungen in `q` eingefügt, da sie erneut zum Vertauschen von Endpunkten in Frage kommen. Ist das große `if`-statement in der inneren while-Schleife nicht erfüllt, wird zur nächsten Kante vorangeschritten, indem `a` und `b` auf den Nachbar in Richtung `direction` gesetzt werden.

Nach Ende der äußeren while-Schleife wird der neue Pfad abgelaufen und die neue Punktefolge in `new_path` gespeichert. Diese wird anschließend zurückgegeben.

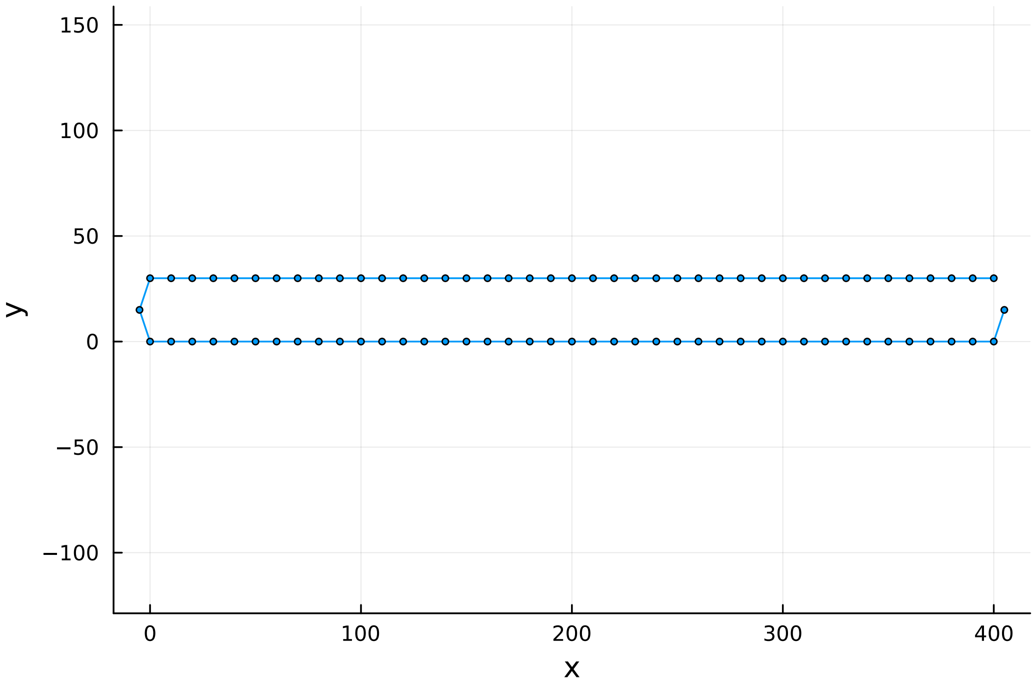
4 Beispiele

Die Instanzen von BWINF konnten alle optimal gelöst werden. Für die größte von ihnen, *wenigerkrumm3* mit 120 Punkten, wurde 1 min 4,14 s benötigt. Die Laufzeitmessungen wurden mit einem AMD Ryzen 5 3600 durchgeführt, es standen 15,6 GB Arbeitsspeicher zur Verfügung. Alle Beispiele nach *wenigerkrumm8* stammen von TSPLIB [3], allerdings wurden die Eingabedateien vom TSPLIB-Format in das Format der anderen Eingaben umgewandelt. Eine Ausnahme gibt es noch: Das Beispiel *world* stammt von der World TSP-Seite von der University of Waterloo [6].

Für die BWINF-Instanzen waren maximal 6 SECs nötig, was die obige Begründung für die DFJ-Formulierung unterstützt. Mit 29 SECs waren bei der TSPLIB-Instanz a280 die meisten SECs nötig, was im Verhältnis zu den 280 Punkten dieser Instanz aber immer noch wenig ist. Die Zahl an Bedingungen der ganzzahligen linearen Programme wurde stets von den $\Theta(n^3)$ Winkelbedingungen dominiert. Die größte optimal gelöste Instanz von TSPLIB (lin318) bestand aus 318 Punkten und benötigte ca. 30 min. Da die Programmausgaben viel Platz benötigen, sind die Ausgaben für die BWINF-Beispiele im Anhang zu finden. Die Lösungen werden hier graphisch präsentiert. Die Ausgaben der selbst hinzugefügten Beispiele sind nicht im Anhang, da dieser sonst sehr lang geworden wäre, die Ausgaben aller Beispiele finden sich aber im Ordner `aufgabe1/ausgaben`. Auch die Eingaben der selbst hinzugefügten Beispiele sind aus Platzgründen nicht in der Dokumentation abgedruckt, finden sich aber im Ordner `aufgabe1/besipiele`.

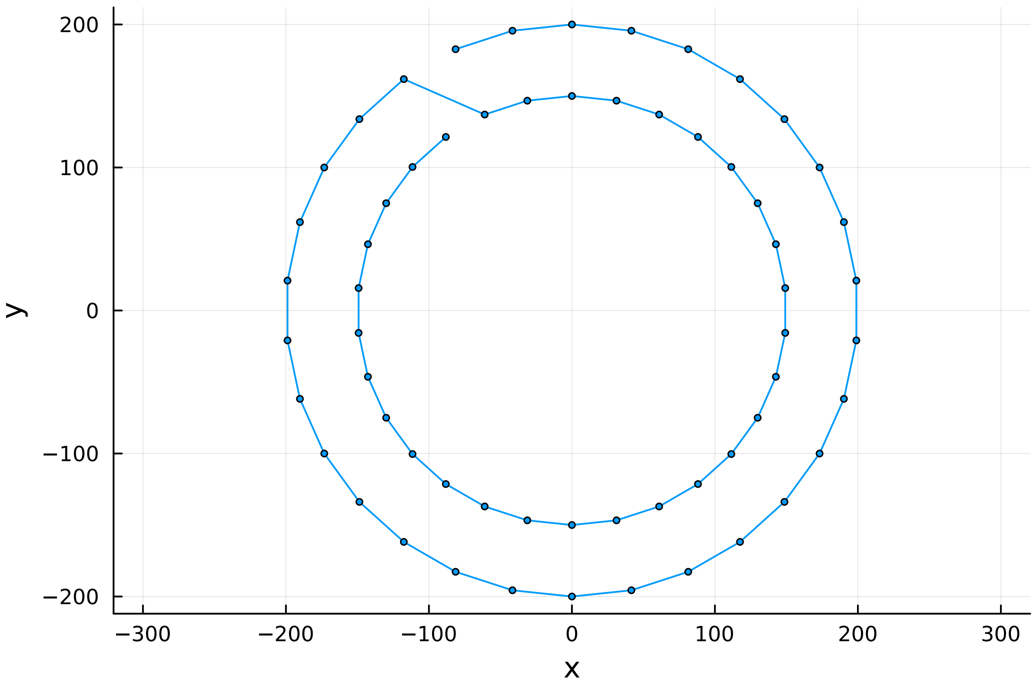
Für größere Beispiele als lin318 wurde der randomisierte Algorithmus in Kombination mit 2-opt verwendet. Die größte damit gelöste Instanz - das World TSP - hat 1904711 Knoten. Die mit dem Programm `aufgabe1_randomized` erzielten Ergebnisse sind wahrscheinlich nicht reproduzierbar - jedoch befinden sich Laufzeit und Ergebnisqualität bei mehreren Ausführungen meist in der gleichen Größenordnung.

4.1 wenigerkrumm1



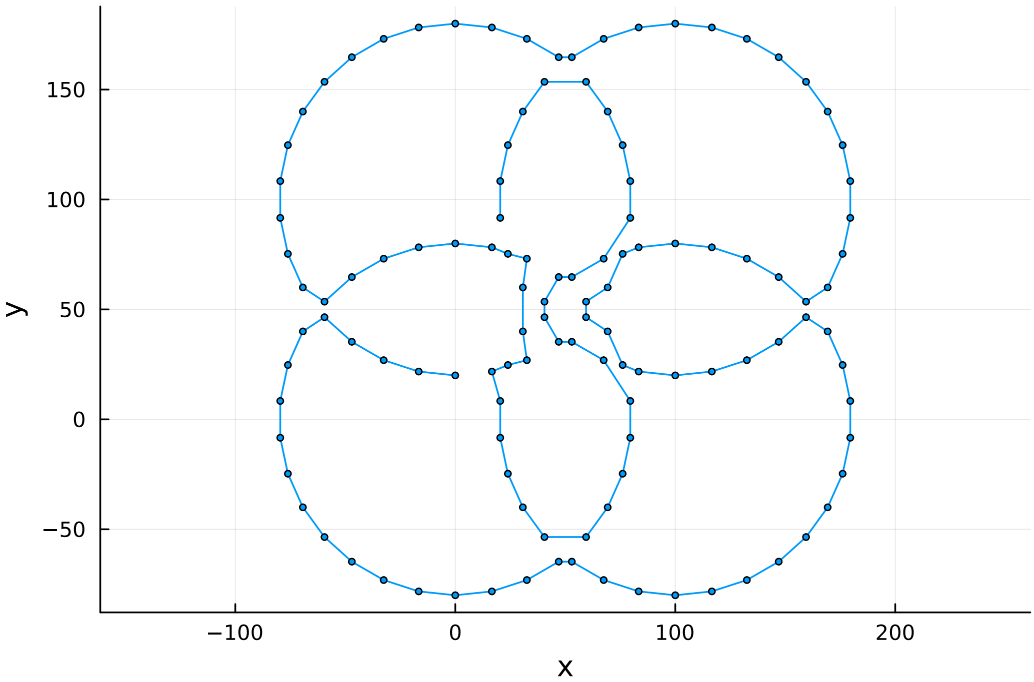
Tourlänge: 847.434165 Laufzeit: 0,607 s #SECs: 0

4.2 wenigerkrumm2



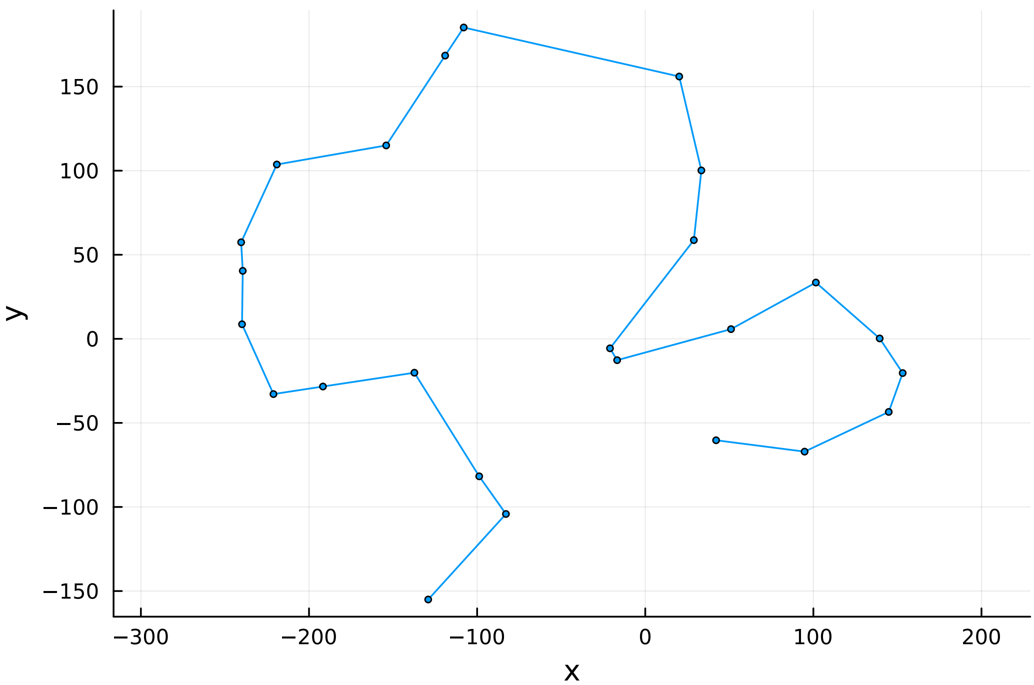
Tourlänge: 2183.662266 Laufzeit: 2,935 s #SECs: 2

4.3 wenigerkrumm3



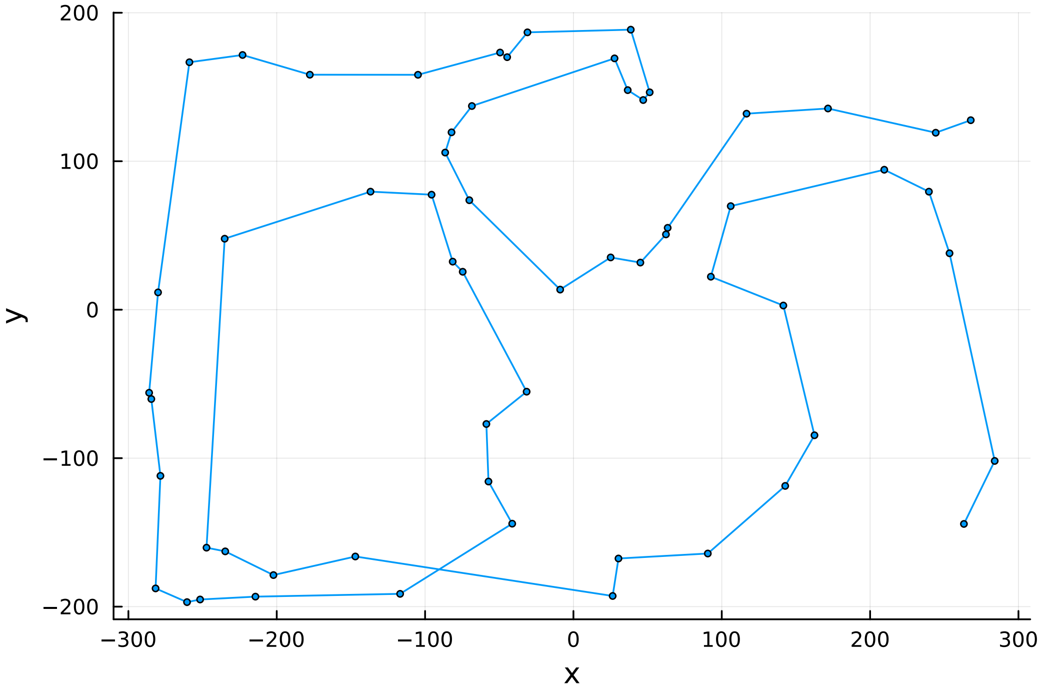
Tourlänge: 1848.046986 Laufzeit: 1 min 4,14 s #SECs: 6

4.4 wenigerkrumm4



Tourlänge: 1205.068555 Laufzeit: 0,024 s #SECs: 0

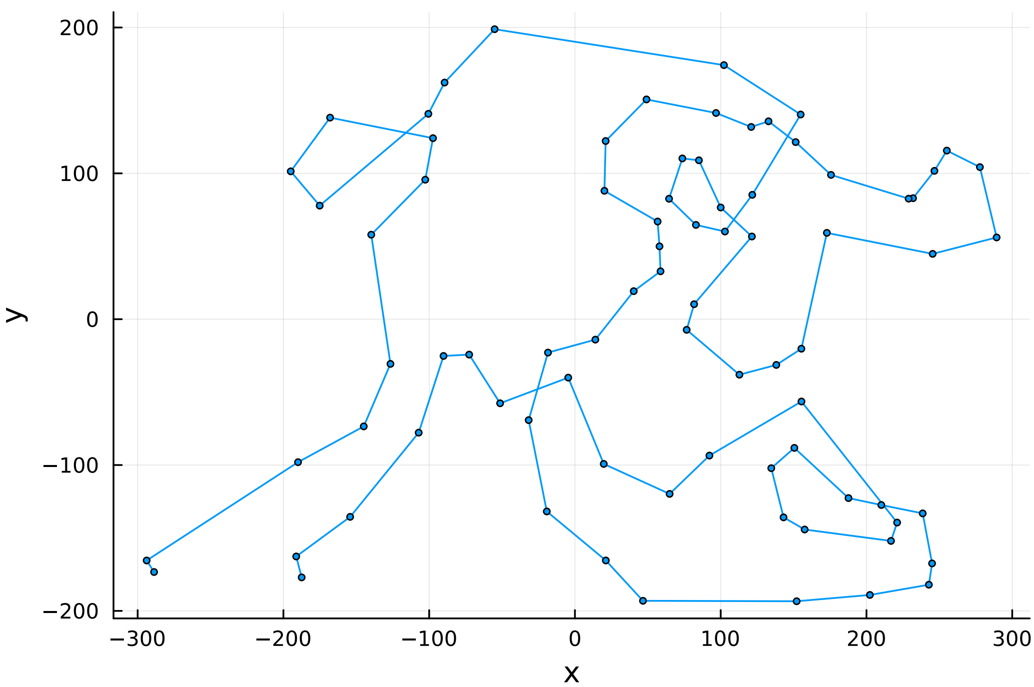
4.5 wenigerkrumm5



Tourlänge: 3257.920434 Laufzeit: 0,375 s #SECs: 0

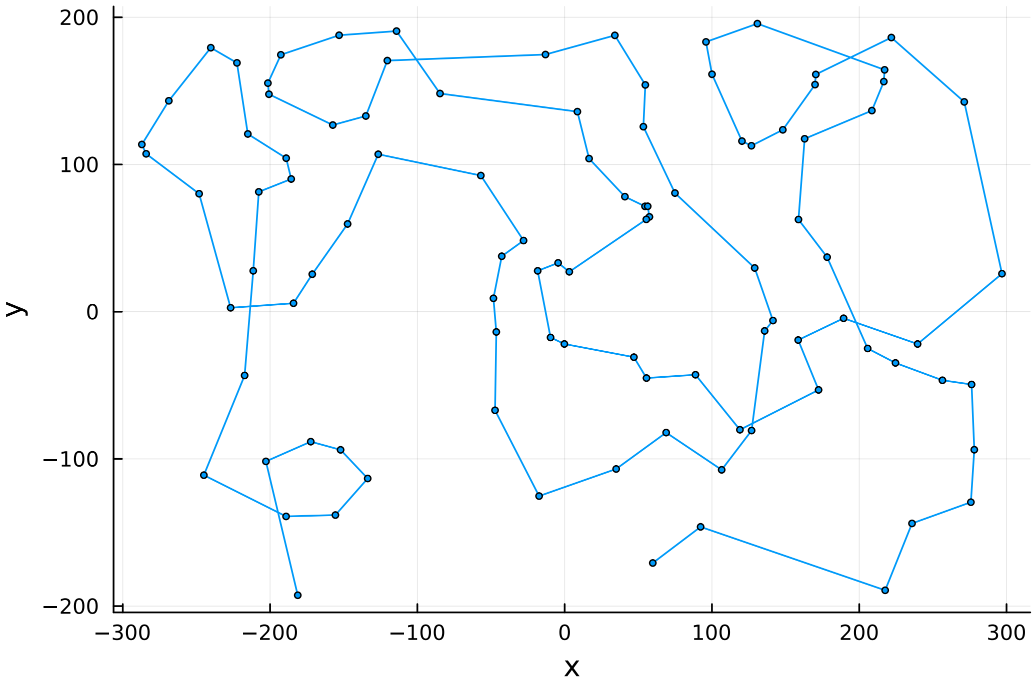
Eine bemerkenswerte Eigenschaft der optimalen Lösungen ist, dass sie sich selbst schneiden können. Bei optimalen Lösungen des TSP auf euklidschen Graphen ist das nicht möglich.

4.6 wenigerkrumm6



Tourlänge: 3457.994092 Laufzeit: 22,35 s #SECs: 2

4.7 wenigerkrumm7



Tourlänge: 4150.643872 Laufzeit: 17,83 s #SECs: 3

4.8 wenigerkrumm8

Dieses Beispiel (ein Dreieck mit keinem Winkel größer gleich $\pi/2$) zeigt, dass das Programm `aufgabe1_ip` erkennt, wenn keine Tour möglich ist. Führt man das Programm `aufgabe1_randomized` auf dieser Instanz aus, terminiert es nicht.

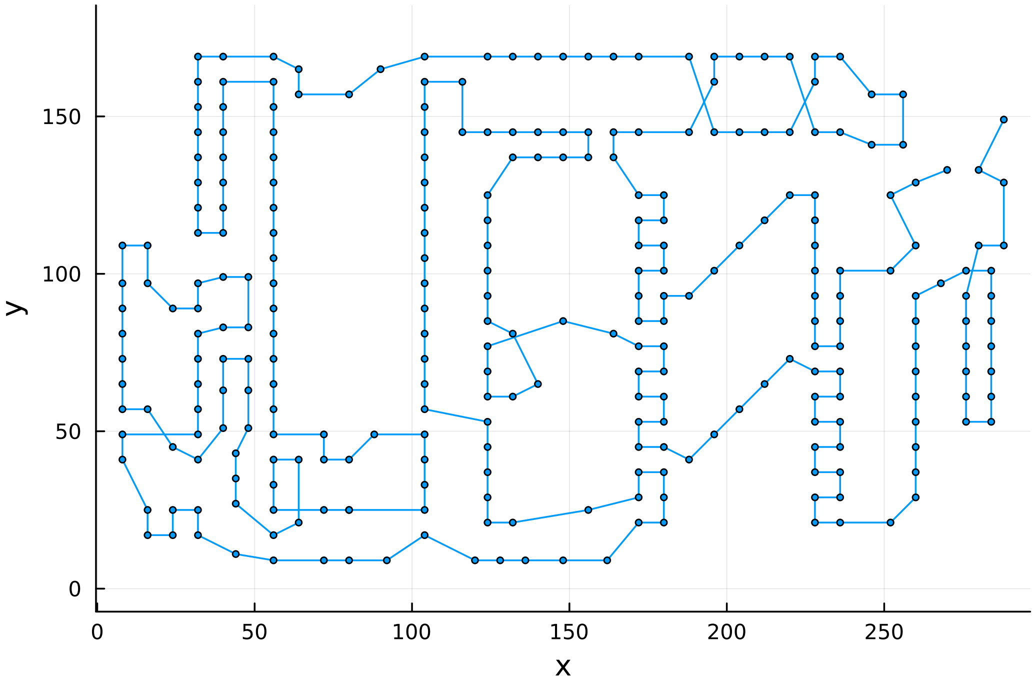
Eingabe:

```
0.0 0.0
6.0 0.0
3.0 5.0
```

Ausgabe (von `aufgabe1_ip`):

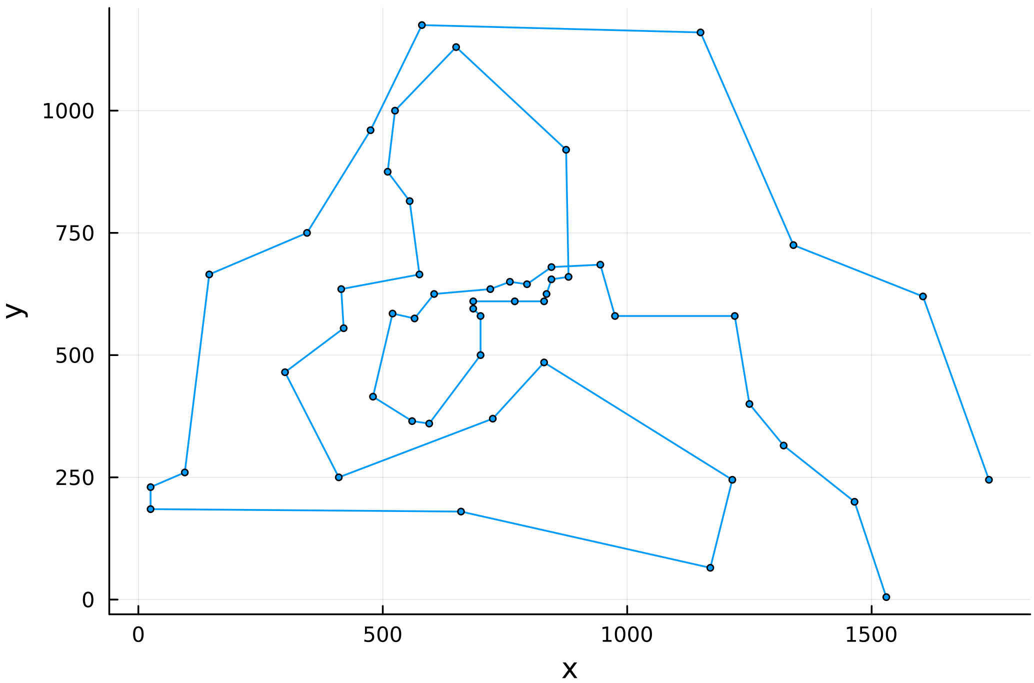
```
Keine Tour mit maximalem Abbiegewinkel von 90 Grad möglich.
```

4.9 a280



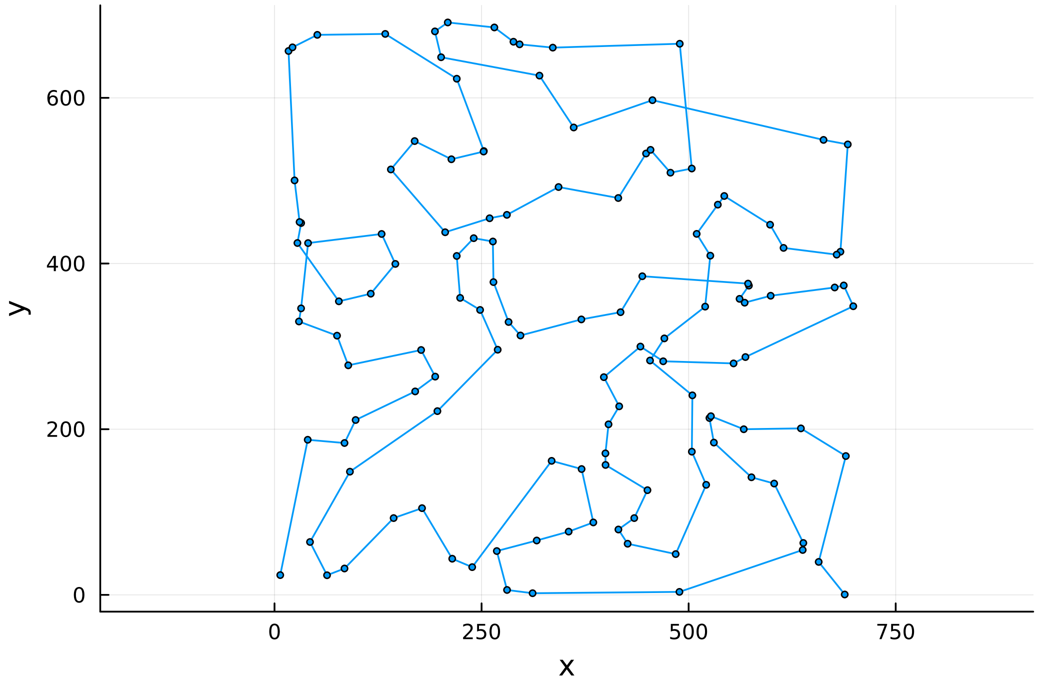
Tourlänge: 2753.696953 Laufzeit: 26 m 3 s #SECs: 29

4.10 berlin52



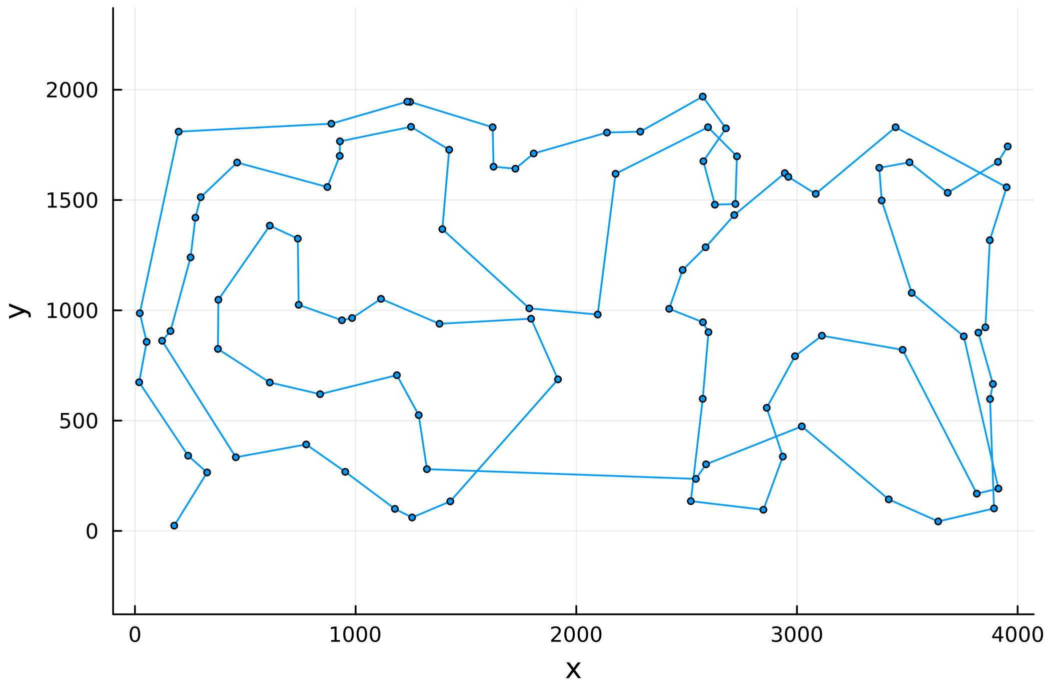
Tourlänge: 9311.526799 Laufzeit: 3,908 s #SECs: 0

4.11 ch130



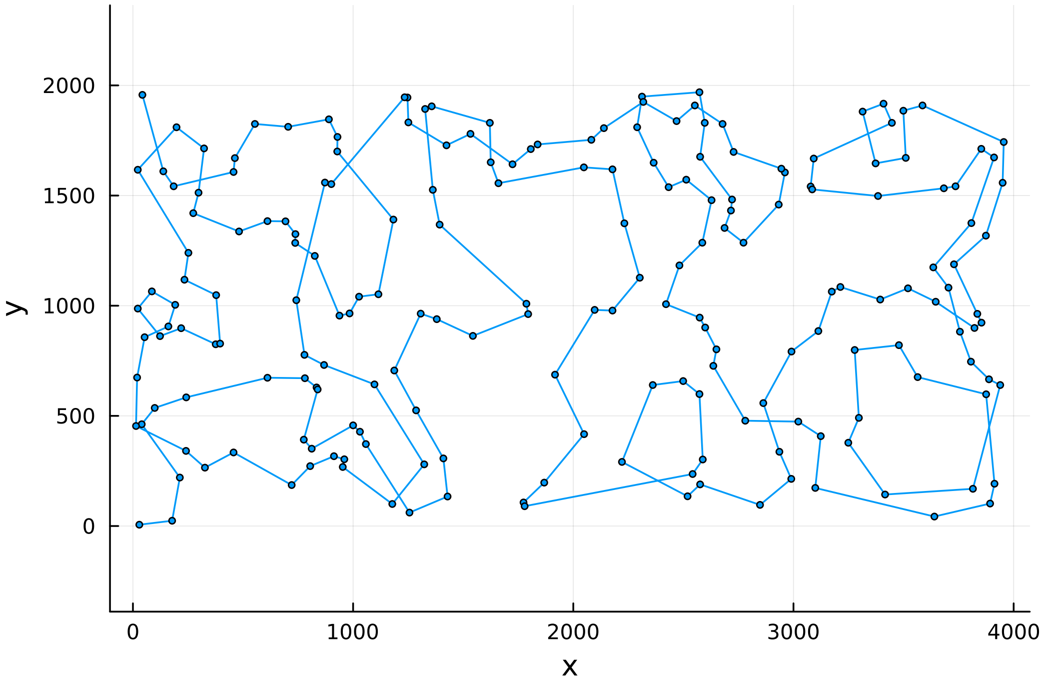
Tourlänge: 7365.270508 Laufzeit: 1 min 51 s #SECs: 5

4.12 kroA100



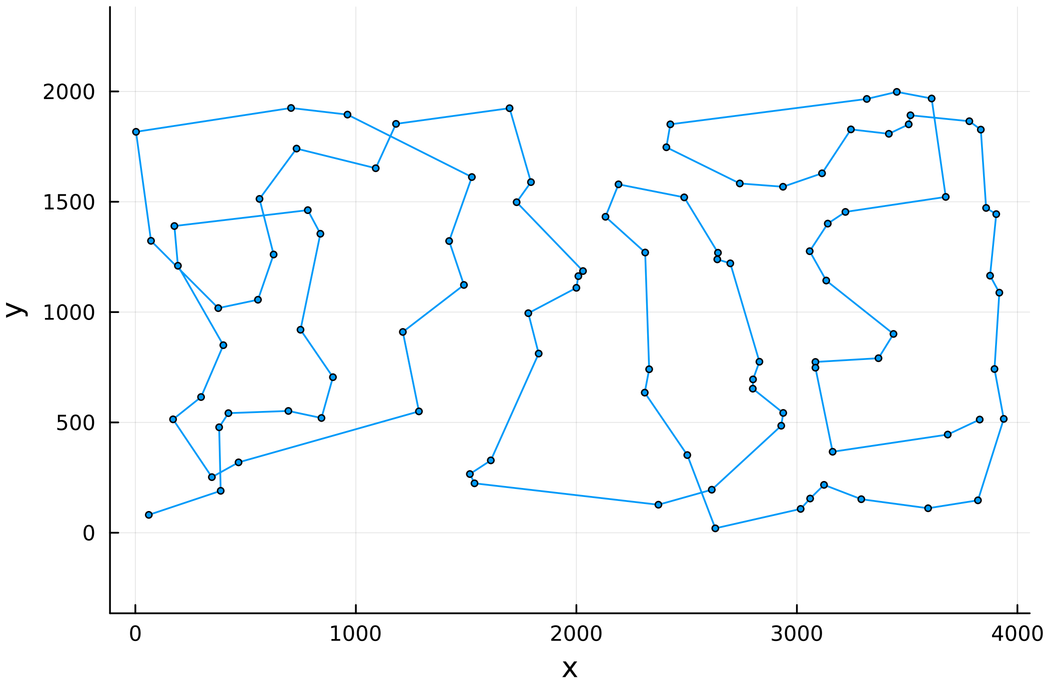
Tourlänge: 28421.204198 Laufzeit: 25,35 s #SECs: 2

4.13 kroA200



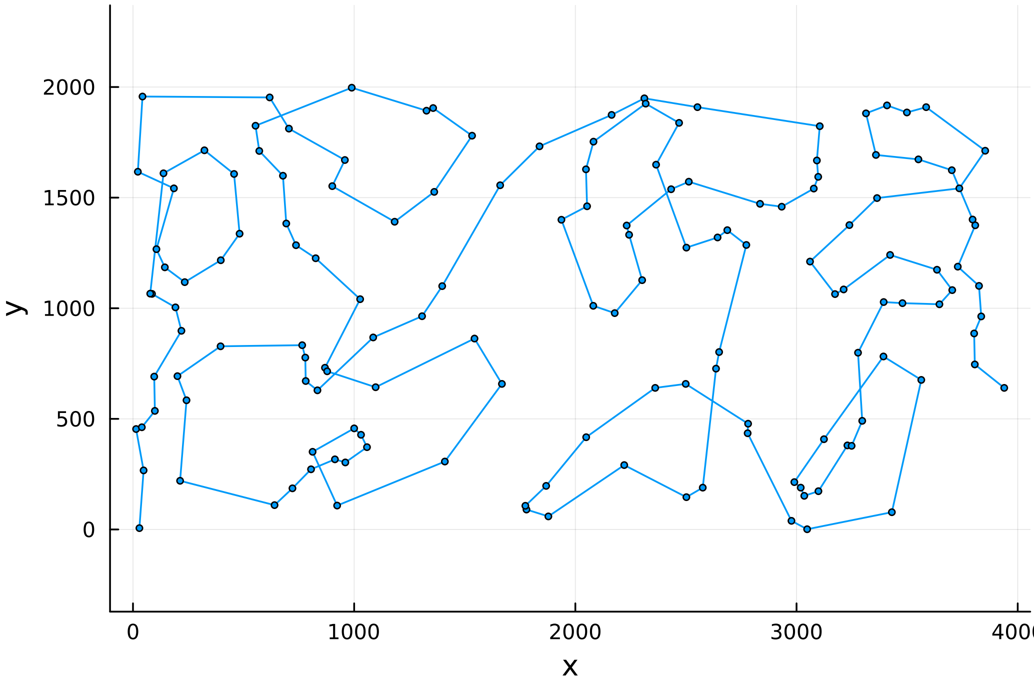
Tourlänge: 37365.104952 Laufzeit: 17 m 4 s #SECs: 5

4.14 kroB100



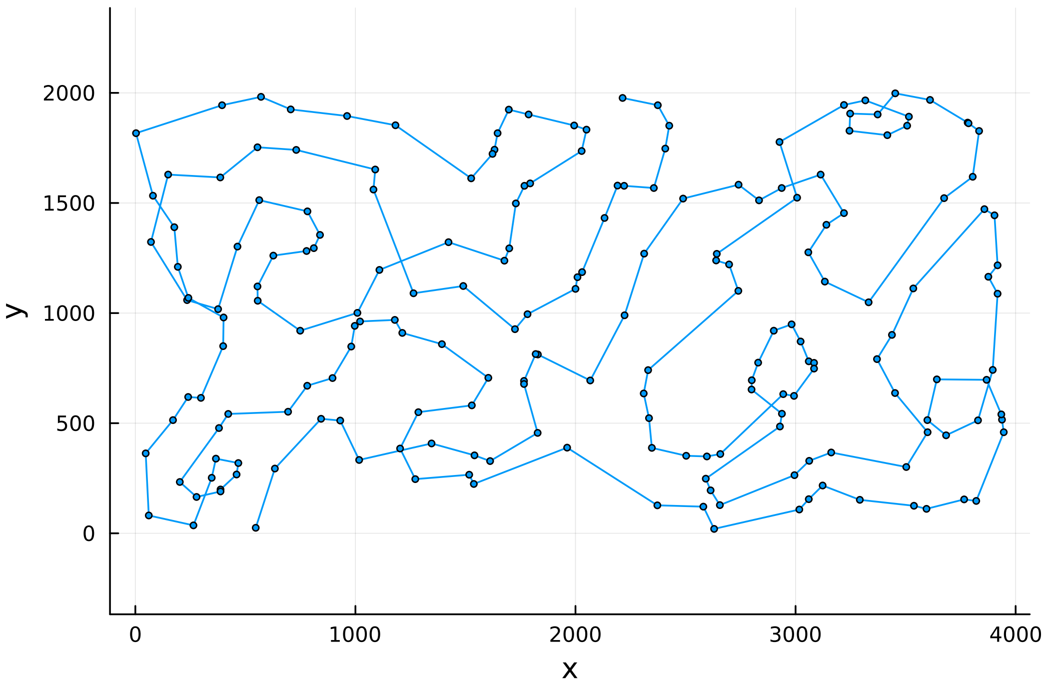
Tourlänge: 26943.309447 Laufzeit: 23,48 s #SECs: 3

4.15 kroB150



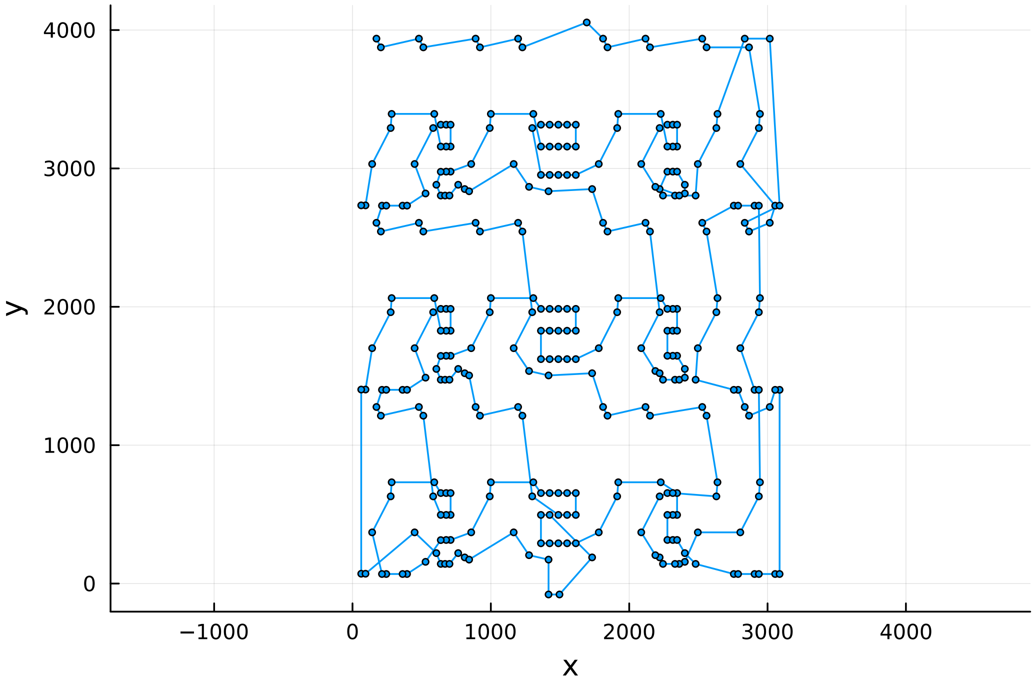
Tourlänge: 31214.645516 Laufzeit: 3 m 33 s #SECs: 8

4.16 kroB200



Tourlänge: 35061.000654 Laufzeit: 23 m 35 s #SECs: 4

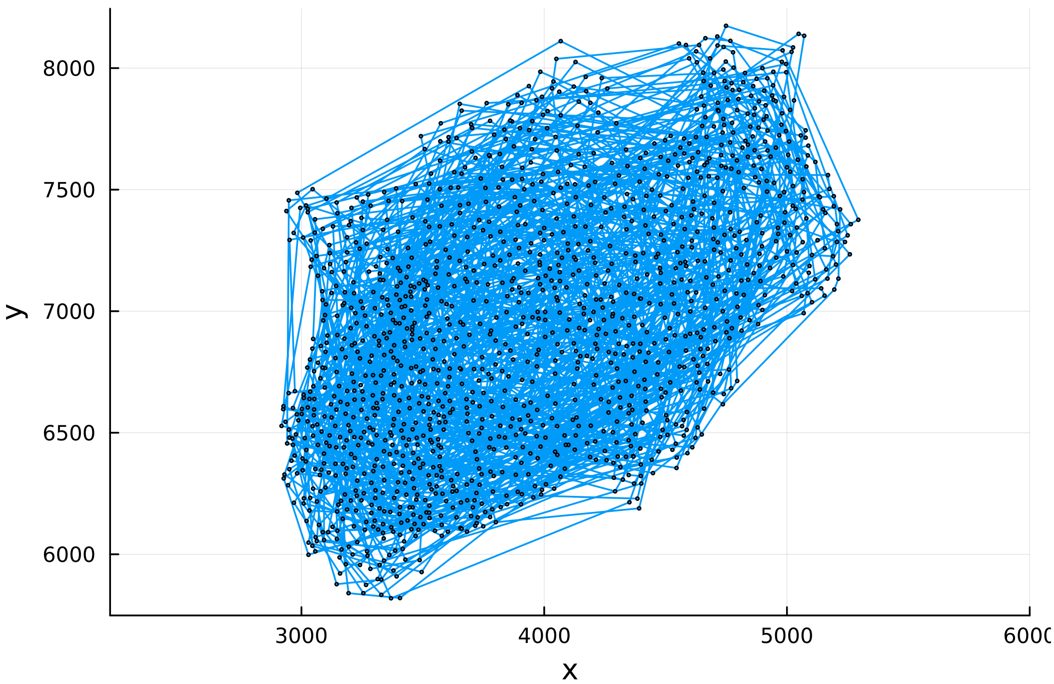
4.17 lin318

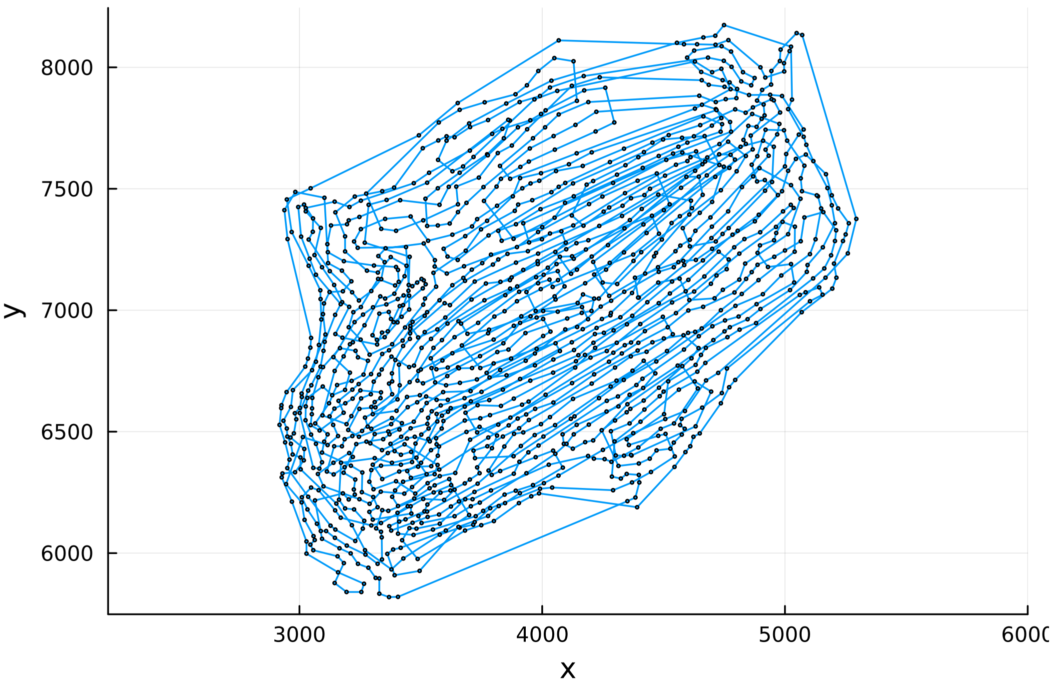


Tourlänge: 51648.008358 Laufzeit: 30 min 29 s #SECs: 4

4.18 nrw1379

Für dieses Beispiel wurde der randomisierte Algorithmus verwendet. Die Laufzeit betrug ca. 2 s (inklusive 2-opt ohne Zeitlimit). Die Länge der Lösung betrug vor 2-opt 846360, und danach 140123. In folgender Abbildung ist oben der Pfad vor 2-opt und unten nach 2-opt dargestellt, woran man gut die starke Verkürzung durch 2-opt nachvollziehen kann.

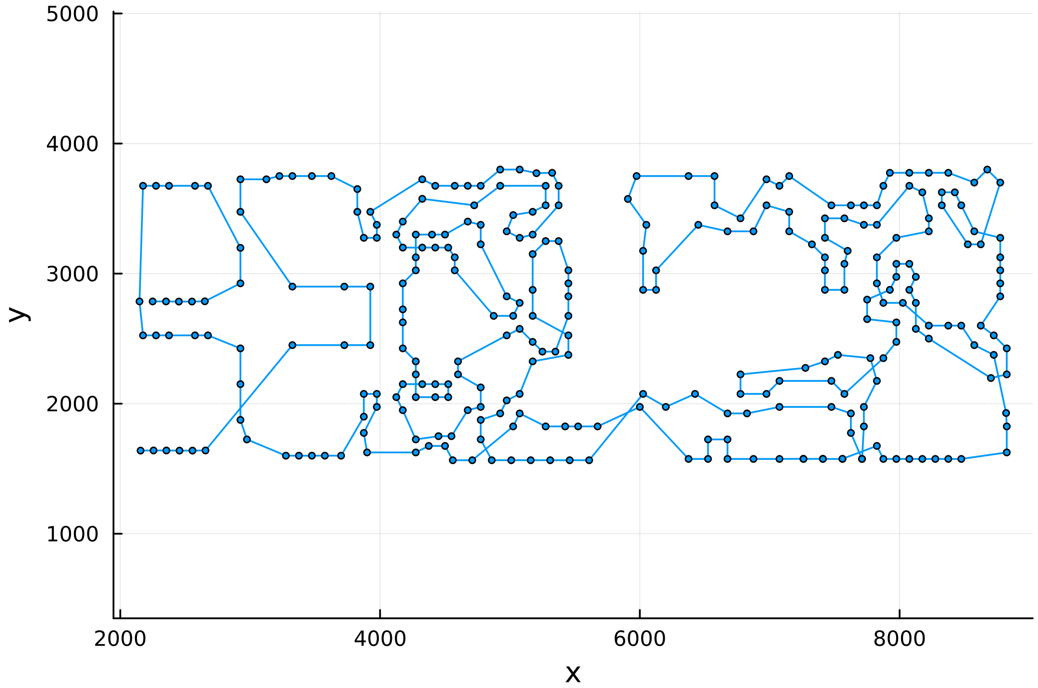




4.19 pla85900

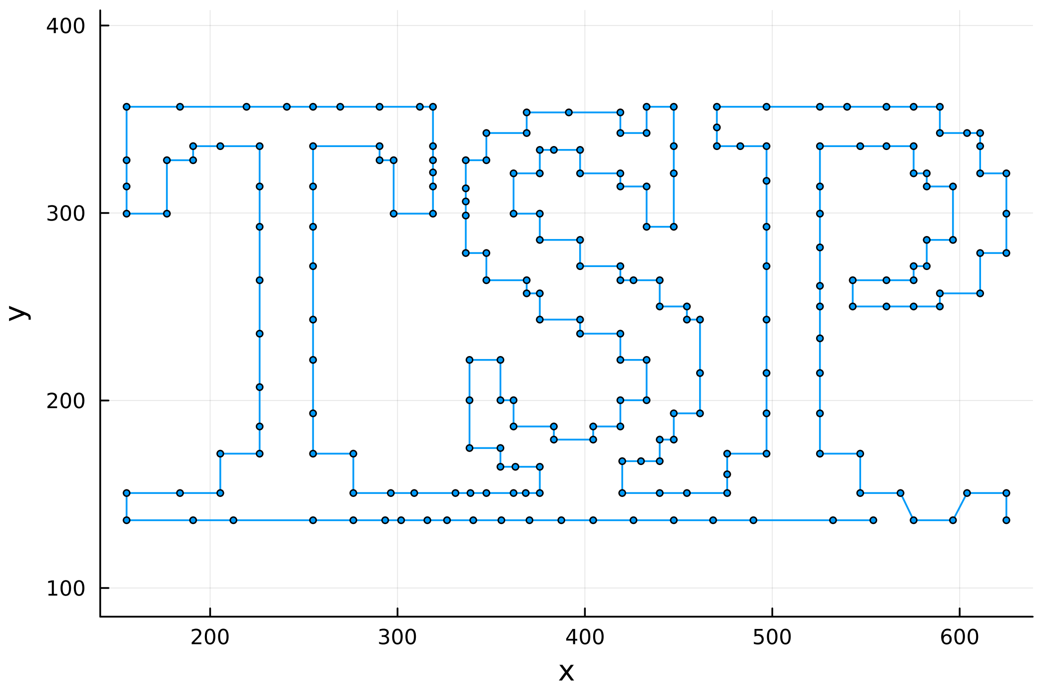
Wegen der Größe dieser Instanz ist eine graphische Darstellung nicht mehr sinnvoll, genauso wie das Abdrucken der Punktefolge. Die Lösung befindet sich aber im Ordner `ausgaben` in der Datei `pla85900.out`. Zum Finden einer zulässigen Tour wurden ca. 4 s benötigt. 2-opt terminert auf dieser Instanz meistens nicht in annehmbarer Zeit, weshalb ein Zeitlimit von 5 min durch Angabe der Kommandozeilenoption `--2-opt-time-limit 300` gesetzt wurde. Die Länge der von RandomizedObtusePath gefundenen Tour betrug 8783948152.162382, sie konnte von der 2-opt-Heuristik auf 1615519447.843837 verkürzt werden.

4.20 pr299



Tourlänge: 56257.537596 Laufzeit: 50 m 59 s #SECs: 18

4.21 tsp225



Tourlänge: 3805.926070 Laufzeit: 54,18 s #SECs: 1

Die Form der Tour lässt vermuten, dass die Lösung nahezu mit der optimalen Lösung für das TSP übereinstimmt.

4.22 world

Dieses Beispiel ist die größte Instanz, die von dem Programm `aufgabe1_randomized` gelöst wurde (Die Lösung steht in der Datei `world.out` im Ordner `ausgaben`). Sie besteht aus 1 904 711 Punkten auf der Erde, gegeben als Koordinaten (Längengrad und Breitengrad). Das wird jedoch ignoriert, die Punkte werden einfach als Punkte in der xy-Ebene mit den gegebenen Koordinaten behandelt. Ein zulässiger Pfad konnte in ca. 20 min gefunden werden, er hatte die Länge 26896725.858146. Für 2-opt wurde ein Zeitlimit von 15 min mit der Kommandozeilenoption `--2-opt-time-limit 900` festgelegt. In dieser Zeit konnte der Pfad auf 26123454.502370 verkürzt werden.

5 Quellcode

5.1 util.hpp

```
#include <bits/stdc++.h>
using namespace std;

template <typename T>
T nchoose2(T n) { return n * (n - 1) / 2; }

template <typename T>
T dot_product(complex<T> const &a, complex<T> const &b)
{
    return (a * conj(b)).real();
}

double path_length(vector<complex<double>> const &z)
{
    double length = 0.0;
    for (size_t i = 1; i < z.size(); i++)
        length += abs(z[i] - z[i - 1]);
    return length;
}
```

5.2 aufgabe1_ip.cpp

```
#include <bits/stdc++.h>
#include "util.hpp"
#include "Highs.h"
using namespace std;

// Gibt den Index der zur Kante {i, j} zugehörigen Variable zurück.
size_t edge_index(size_t n, size_t i, size_t j)
{
    return nchoose2(n) - nchoose2(n - min(i, j)) + max(i, j) - min(i, j) - 1;
}

// Fügt für jedes Tripel i, j, k (i != j != k, i < k) die Bedingung hinzu,
// dass die Kanten ij und jk nicht gleichzeitig verwendet werden dürfen, wenn
// der Betrag ihres Außenwinkels > pi / 2 ist.
void add_angle_constraints(HighsModel &model, vector<complex<double>> const &z)
{
    for (size_t j = 0; j < z.size(); j++)
```

```
    {
        for (size_t i = 0; i < z.size(); i++)
        {
            if (i == j)
                continue;
            for (size_t k = i + 1; k < z.size(); k++)
            {
                if (k != j && dot_product(z[j] - z[i], z[k] - z[j]) < 0.0)
                {
                    HighsLp &lp = model.lp_;
                    lp.a_matrix_.index_.push_back(edge_index(z.size(), i, j));
                    lp.a_matrix_.index_.push_back(edge_index(z.size(), j, k));
                    lp.a_matrix_.value_.push_back(1); // Der Koeffizient jeder
                    lp.a_matrix_.value_.push_back(1); // Kante ist 1.
                    lp.row_lower_.push_back(0);
                    lp.row_upper_.push_back(1);
                    //
                    lp.a_matrix_.start_.push_back(lp.a_matrix_.index_.size());
                }
            }
        }
    }
}

// Schränkt den Grad jedes Knoten auf 1 oder 2 ein.
void add_degree_constraints(HighsModel &model, size_t n)
{
    for (size_t i = 0; i < n; i++)
    {
        for (size_t j = 0; j < n; j++)
        {
            if (i != j) // Die Kante zu jedem Knoten != i wird mit Koeffizient
            {           // 1 zur aktuellen Zeile hinzugefügt.
                model.lp_.a_matrix_.index_.push_back(edge_index(n, i, j));
                model.lp_.a_matrix_.value_.push_back(1);
            }
        }
        model.lp_.row_lower_.push_back(1); // Setze Unter- und Oberschranke der
        model.lp_.row_upper_.push_back(2); // Zeile.
        model.lp_.a_matrix_.start_.push_back(model.lp_.a_matrix_.index_.size());
    }
}

// Schränkt die Anzahl verwendeter Kanten auf genau n - 1 ein.
void add_num_edges_constraint(HighsModel &model, size_t n)
{
    for (size_t i = 0; i < nchoose2(n); i++) // Iteriere über alle Kanten.
    {
        model.lp_.a_matrix_.index_.push_back(i);
        model.lp_.a_matrix_.value_.push_back(1);
    }
    model.lp_.row_lower_.push_back(n - 1); // Fixiere den Wert der neuen Zeile
    model.lp_.row_upper_.push_back(n - 1); // auf genau n - 1.
```

```
    model.lp_.a_matrix_.start_.push_back(model.lp_.a_matrix_.index_.size());
}

// Fügt einen SEC für die Knoten in tour ein.
void add_subtour_elimination_constraint(
    Highs &highs, size_t n, vector<size_t> const &tour)
{
    // Arrays für die neuen Spaltenindizes und Werte.
    HighsInt *ind = (HighsInt *)malloc(nchoose2(tour.size()) * sizeof *ind);
    double *val = (double *)malloc(nchoose2(tour.size()) * sizeof *val);
    size_t k = 0; // Anzahl bereits in ind bzw. val eingefügter Elemente

    // Setze den Koeffizienten jeder Kante zwischen Knoten der Subtour auf 1.
    for (size_t i = 0; i < tour.size(); i++)
    {
        for (size_t j = i + 1; j < tour.size(); j++)
        {
            ind[k] = edge_index(n, tour[i], tour[j]);
            val[k] = 1;
            k++;
        }
    }

    // Verwendet Highs::addRow(double lower, double upper, HighsInt num_new_nz,
    //                         const HighsInt *indices, const double *values)
    HighsStatus status;
    status = highs.addRow(0, tour.size() - 1, nchoose2(tour.size()), ind, val);
    assert(status == HighsStatus::kOk);
    free(ind);
    free(val);
}

// Gibt den als Lösung gefundenen Graphen in Form einer Adjazenzliste zurück.
vector<vector<size_t>> build_graph(Highs const &highs, size_t n)
{
    HighsSolution const &solution = highs.getSolution();
    vector<vector<size_t>> graph(n); // Adjazenzliste

    for (size_t i = 0; i < n; i++)          // Überprüfe für jede Kante, ob der
        for (size_t j = i + 1; j < n; j++) // Wert ihrer Variablen 1 ist.
            if (solution.col_value[edge_index(n, i, j)] > 0.5)
                graph[i].push_back(j), graph[j].push_back(i);

    return graph;
}

// Gibt zurück, ob in der Lösung Subtouren existieren.
bool check_for_subtours(Highs &highs, size_t n)
{
    vector<vector<size_t>> graph = build_graph(highs, n);
    vector<bool> visited(n, 0);
    bool has_subtours = 0;
```

```
    for (size_t i = 0; i < n; i++)
    {
        if (!visited[i])
        {
            vector<size_t> subtour;
            size_t j = i, last = SIZE_MAX; // aktueller und vorheriger Knoten

            do
            {
                visited[j] = 1;
                subtour.push_back(j);
                size_t next = SIZE_MAX;
                for (size_t k : graph[j]) // Da der Grad jedes Knoten <= 2 ist,
                    if (k != last)        // wurde jeder Knoten unterschiedlich
                        next = k;         // zum letzten noch nicht besucht oder
                last = j;                 // ein Zyklus gefunden.
                j = next;
            } while (j != i && j != SIZE_MAX);

            if (j == i) // Zyklus gefunden.
            {
                has_subtours = 1;
                add_subtour_elimination_constraint(highs, n, subtour);
            }
        }
    }

    return has_subtours;
}

// Gibt den kürzesten Hamiltonpfad zurück, auf dem der Abbiegewinkel jedes
// benachbarten Kantenpaares <= pi / 2 ist. Existiert kein solcher Pfad, ist der
// zurückgegebene Vektor leer.
vector<complex<double>> shortest_obtuse_path(vector<complex<double>> const &z)
{
    size_t const n = z.size();

    HighsModel model; // Objekt, in dem das IP spezifiziert wird.
    model.lp_.sense_ = ObjSense::kMinimize;
    model.lp_.a_matrix_.format_ = MatrixFormat::kRowwise;
    model.lp_.a_matrix_.start_ = {0}; // Die erste Zeile beginnt bei Index 0.
    model.lp_.num_col_ = nchoose2(n);

    // Füge die Länge jeder Kante als ihren Koeffizienten zur Kostenfunktion
    // hinzu und beschränke ihre Variable auf 0 oder 1.
    for (size_t i = 0; i < n; i++)
        for (size_t j = i + 1; j < n; j++)
        {
            model.lp_.col_cost_.push_back(abs(z[i] - z[j]));
            model.lp_.integrality_.push_back(HighsVarType::kInteger);
            model.lp_.col_lower_.push_back(0);
            model.lp_.col_upper_.push_back(1);
        }
```

```
    add_angle_constraints(model, z);
    add_degree_constraints(model, n);
    add_num_edges_constraint(model, n);
    model.lp_.num_row_ = model.lp_.row_lower_.size();

    Highs highs;
    HighsStatus status;
    HighsModelStatus model_status = HighsModelStatus::kNotset;
    status = highs.passModel(model); // Übergebe model an highs.
    assert(status == HighsStatus::kOk);
    bool has_subtours = 1;

    while (has_subtours && model_status != HighsModelStatus::kInfeasible)
    {
        status = highs.run(); // Löse das ganzzahlige lineare Programm.
        assert(status == HighsStatus::kOk);
        model_status = highs.getModelStatus();
        has_subtours = check_for_subtours(highs, n); // Füge SECs ein.
    }

    if (model_status == HighsModelStatus::kInfeasible) // Keine Tour möglich.
        return vector<complex<double>>();

    vector<vector<size_t>> graph = build_graph(highs, n);
    vector<complex<double>> path;

    size_t j = SIZE_MAX, last = SIZE_MAX; // aktueller und vorheriger Knoten
    for (size_t i = 0; i < n; i++)
        if (graph[i].size() == 1) // Ein Knoten mit Grad 1 muss Anfang des Pfads
            j = i;                // sein.

    while (j != SIZE_MAX)
    {
        path.push_back(z[j]);
        size_t next = SIZE_MAX;
        for (size_t k : graph[j]) // Wähle den der maximal zwei benachbarten
            if (k != last)        // Knoten als Nachfolger, der nicht Vorgänger
                next = k;         // ist.
        last = j;
        j = next;
    }

    return path;
}

int main()
{
    vector<complex<double>> z;
    double x, y;
    while (scanf("%lf %lf", &x, &y) == 2)
        z.emplace_back(x, y);
```

```
    vector<complex<double>> const path = shortest_obtuse_path(z);

    if (path.empty())
    {
        cout << "Keine Tour mit maximalem Abbiegewinkel von 90 Grad möglich.\n";
    }
    else
    {
        cout << setprecision(6) << fixed
             << "Tourlänge: " << path_length(path) << '\n';
        for (complex<double> const &u : path)
            cout << u.real() << ' ' << u.imag() << '\n';
    }
}
```

5.3 aufgabe1_randomized.cpp

```
#include <bits/stdc++.h>
#include "util.hpp"
using namespace std;

vector<complex<double>> randomized_obtuse_path(vector<complex<double>> const &z)
{
    size_t const n = z.size(), sqrtn = sqrt(n);
    deque<size_t> path;
    list<size_t> unvisited;
    bool front_is_dead_end, back_is_dead_end, extending_back;
    size_t no_added_node;

    auto restart_search = [&]()
    {
        front_is_dead_end = back_is_dead_end = 0; // Setze alle Datenstrukturen
        extending_back = 0;                       // zurück.
        unvisited.clear();
        path.clear();
        no_added_node = 0;
        srand(time(0));
        path.push_back(rand() % n);     // Wähle einen zufälligen Startknoten.
        for (size_t i = 0; i < n; i++) // Fülle unvisited mit allen Knoten
            if (i != path.front())     // außer dem Startknoten.
                unvisited.push_back(i);
    };

    restart_search();

    while (path.size() < n)
    {
        if (no_added_node > sqrtn) // Zu große Anzahl aufeinanderfolgener
            restart_search();      // Iterationen ohne Hinzufügen eines Knotens.

        // u: letzter / erster Knoten, v: vorletzter / zweiter Knoten
        // w: Iterator in unvisited zum neu hinzugefügten Knoten
        size_t u, v, candidates = 0;
        list<size_t>::iterator w = unvisited.end();
```

```
if (extending_back)
    u = path.back(), v = path.size() >= 2 ? *++path.rbegin() : SIZE_MAX;
else
    u = path.front(), v = path.size() >= 2 ? *++path.begin() : SIZE_MAX;

for (auto it = unvisited.begin(); it != unvisited.end(); it++)
    if (path.size() < 2 || dot_product(z[u] - z[v], z[*it] - z[u]) >= 0)
    {
        candidates++;
        if (!(rand() % candidates)) // wahr mit Wahrscheinlichkeit
            w = it;                 // 1 / candidates.
        if (candidates > sqrtn)
            break;
    }
if (candidates)
{
    if (extending_back)
        path.push_back(*w); // Erweitere den Pfad um w.
    else
        path.push_front(*w);
    unvisited.erase(w);
    no_added_node = 0;
    continue;
}

// Der Pfad kann von u aus nicht mehr erweitert werden. Das aktuell
// behandelte Ende des Pfads wird als Sackgasse markiert.
(extending_back ? back_is_dead_end : front_is_dead_end) = 1;
no_added_node++; // In dieser Iteration wurde kein Knoten hinzugefügt.

if (front_is_dead_end && back_is_dead_end)
{
    // i: Index des aktuell betrachteten Knotens
    // w: Index des Knotens, nach dem der Pfad aufgebrochen wird.
    size_t i = extending_back ? path.size() - 3 : 2, w = SIZE_MAX,
           candidates = 0;

    while (i < path.size())
    {
        // j: Nachbar von path[i], mit dem beim Einfügen der Kante
        //    {u, path[i]} ein Abbiegewinkel entstehen würde.
        size_t j = extending_back ? i - 1 : i + 1;
        if (dot_product(z[u] - z[v], z[path[i]] - z[u]) >= 0 &&
            (j >= path.size() ||
             dot_product(z[path[i]] - z[u], z[path[j]] - z[path[i]]) >= 0))
        {
            candidates++;
            if (!(rand() % candidates)) // wahr mit Wahrscheinlichkeit
                w = i;                  // 1 / candidates.
            if (candidates > sqrtn)
                break;
        }
```

```
                i += extending_back ? -1 : 1;
            }

            if (candidates) // Breche den Pfad nach w auf und verbinde u mit w.
            {
                if (extending_back) // Kehre das Suffix strikt nach w um.
                    reverse(path.begin() + w + 1, path.end());
                else // Kehre das Präfix strikt vor w um.
                    reverse(path.begin(), path.begin() + w);
                (extending_back ? back_is_dead_end : front_is_dead_end) = 0;
            }
            else
                extending_back = !extending_back;
        }
        else // Versuche den Pfad am anderen Ende zu erweitern.
            extending_back = !extending_back;
    }

    vector<complex<double>> point_order;
    for (size_t i = 0; i < n; i++)
        point_order.push_back(z[path[i]]);
    return point_order;
}

// Optimiert den gegebenen Pfad mit der 2-opt Heuristik, ohne die Beschränkung
// von Abbiegewinkeln zu verletzen. Mit time_limit wird ein Zeitlimit in s
// gesetzt.
vector<complex<double>> optimize_path(
    vector<complex<double>> const &path, double time_limit)
{
    auto start_time = chrono::system_clock::now();

    size_t const n = path.size();
    vector<array<size_t, 2>> nodes(n);
    queue<pair<size_t, size_t>> q;
    for (size_t i = 0; i + 1 < n; i++)
    { // Füge alle Knoten in die Warteschlange ein.
        q.emplace(i, i + 1), q.emplace(i + 1, i);
        nodes[i][1] = i + 1, nodes[i + 1][0] = i;
    }
    nodes[0][0] = nodes[n - 1][1] = SIZE_MAX;

    while (!q.empty() &&
           chrono::duration<double>(chrono::system_clock::now() - start_time)
                   .count() < time_limit)
    {
        auto const [v, w] = q.front();
        q.pop();
        if (nodes[v][0] != w && nodes[v][1] != w) // Überprüfe, ob es die Kante
            continue;                             // noch gibt.
        bool const direction = nodes[w][0] == v;

        // Die aktuell bearbeitete Kante ist {v, w}. u kommt vor v, x nach w.
```

```
        // Als mögliche Tauschpartner werden nur Kanten in Richtung von x in
        // Betracht gezogen.
        size_t const u = nodes[v][!direction], x = nodes[w][direction];
        size_t a = w, b = x;

        while (b != SIZE_MAX && nodes[b][direction] != SIZE_MAX)
        {
            size_t c = nodes[b][direction], d = nodes[c][direction];

            // Überprüfe, ob die Tour durch Erstetzen von {v, w}, {b, c} durch
            // {v, b}, {w, c} verkürzt wird und die 4 neuen Abbiegewinkel (uvb,
            // vba, xwc, wcd) alle <= pi / 2 sind.
            if ((u == SIZE_MAX ||
                 dot_product(path[v] - path[u], path[b] - path[v]) >= 0) &&
                dot_product(path[b] - path[v], path[a] - path[b]) >= 0 &&
                dot_product(path[w] - path[x], path[c] - path[w]) >= 0 &&
                (d == SIZE_MAX ||
                 dot_product(path[c] - path[w], path[d] - path[c]) >= 0) &&
                abs(path[v] - path[w]) + abs(path[b] - path[c]) >
                    abs(path[v] - path[b]) + abs(path[w] - path[c]))
            {
                // Kehre den Teil des Pfads von x bis a um.
                for (size_t i = x; i != b; i = nodes[i][!direction])
                    swap(nodes[i][0], nodes[i][1]);
                // Entferne {v, w}, {b, c} und füge {v, b}, {w, c} ein.
                nodes[v][direction] = b;
                nodes[b][direction] = a;
                nodes[b][!direction] = v;
                nodes[w][!direction] = x;
                nodes[w][direction] = c;
                nodes[c][!direction] = w;
                // Die neu eingefügten Kanten können erneut mit anderen
                // vertauscht werden, daher werden sie zu q hinzugefügt.
                q.emplace(v, b);
                q.emplace(b, v);
                q.emplace(w, c);
                q.emplace(c, w);
                break;
            }
            a = nodes[a][direction]; // Gehe zur nächsten Kante im Pfad.
            b = nodes[b][direction];
        }
    }

    vector<complex<double>> new_path; // nue
    size_t start = SIZE_MAX;
    bool direction = 0;
    for (size_t i = 0; i < n; i++) // Suche nach einem Knoten mit Grad 1.
        if (nodes[i][0] == SIZE_MAX || nodes[i][1] == SIZE_MAX)
        {
            start = i; // Startknoten gefunden.
            direction = nodes[i][1] != SIZE_MAX;
            break;
```

```
        }
    for (size_t i = start; i != SIZE_MAX; i = nodes[i][direction])
        new_path.push_back(path[i]);
    return new_path;
}

int main(int argc, char **argv)
{
    vector<complex<double>> z;
    double x, y;
    while (scanf("%lf %lf", &x, &y) == 2)
        z.emplace_back(x, y);

    auto start_time = chrono::system_clock::now();

    vector<complex<double>> path = randomized_obtuse_path(z);
    auto duration = chrono::duration_cast<chrono::duration<double>>(
        chrono::system_clock::now() - start_time);

    cerr << setprecision(6) << fixed << "Zulässige Tour mit Länge "
         << path_length(path) << " nach " << duration.count()
         << " s gefunden.\nStarte 2-opt..." << endl;

    double two_opt_time_limit = DBL_MAX;
    for (int i = 1; i + 1 < argc; i++)
        if (!strcmp(argv[i], "--2-opt-time-limit"))
            two_opt_time_limit = stod(argv[i + 1]);

    path = optimize_path(path, two_opt_time_limit);
    duration = chrono::duration_cast<chrono::duration<double>>(
        chrono::system_clock::now() - start_time);
    cerr << "Tourlänge nach Optimierung: " << path_length(path) << '\n'
         << "Laufzeit: " << duration.count() << " s" << endl;

    cout << setprecision(6) << fixed
         << "Tourlänge: " << path_length(path) << '\n';
    for (complex<double> const &u : path)
        cout << u.real() << ' ' << u.imag() << '\n';
}
```

Literatur

[1] Öncan, T., Altinel, I. K., Laporte, G. (2009). A comparative analysis of several asymmetric traveling salesman problem formulations.
https://mate.unipv.it/~gualandi/famo2conti/papers/tsp_formulations.pdf

[2] Huangfu, Q., Hall, J. A. J. (2018). Parallelizing the dual revised simplex method.
https://www.maths.ed.ac.uk/hall/HuHa13/
https://github.com/ERGO-Code/HiGHS

[3] Ruprecht-Karls Universität Heidelberg. TSPLIB
http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[4] Dumitrescu, A., Jiang, M. (2018). On the approximability of covering points by lines and related problems.
https://arxiv.org/pdf/1312.2549.pdf

[5] Johnsonbaugh, R. (2017). Discrete Mathematics (8. Auflage). Pearson Verlag

[6] University of Waterloo. World TSP
https://www.math.uwaterloo.ca/tsp/world/index.html

Anhang

wenigerkrumm1

```
Tourlänge: 847.434165
400.000000 30.000000
390.000000 30.000000
380.000000 30.000000
370.000000 30.000000
360.000000 30.000000
350.000000 30.000000
340.000000 30.000000
330.000000 30.000000
320.000000 30.000000
310.000000 30.000000
300.000000 30.000000
290.000000 30.000000
280.000000 30.000000
270.000000 30.000000
260.000000 30.000000
250.000000 30.000000
240.000000 30.000000
230.000000 30.000000
220.000000 30.000000
210.000000 30.000000
200.000000 30.000000
190.000000 30.000000
180.000000 30.000000
170.000000 30.000000
160.000000 30.000000
150.000000 30.000000
140.000000 30.000000
130.000000 30.000000
120.000000 30.000000
110.000000 30.000000
100.000000 30.000000
90.000000 30.000000
80.000000 30.000000
70.000000 30.000000
60.000000 30.000000
50.000000 30.000000
40.000000 30.000000
30.000000 30.000000
20.000000 30.000000
10.000000 30.000000
```

```
0.000000 30.000000
-5.000000 15.000000
0.000000 0.000000
10.000000 0.000000
20.000000 0.000000
30.000000 0.000000
40.000000 0.000000
50.000000 0.000000
60.000000 0.000000
70.000000 0.000000
80.000000 0.000000
90.000000 0.000000
100.000000 0.000000
110.000000 0.000000
120.000000 0.000000
130.000000 0.000000
140.000000 0.000000
150.000000 0.000000
160.000000 0.000000
170.000000 0.000000
180.000000 0.000000
190.000000 0.000000
200.000000 0.000000
210.000000 0.000000
220.000000 0.000000
230.000000 0.000000
240.000000 0.000000
250.000000 0.000000
260.000000 0.000000
270.000000 0.000000
280.000000 0.000000
290.000000 0.000000
300.000000 0.000000
310.000000 0.000000
320.000000 0.000000
330.000000 0.000000
340.000000 0.000000
350.000000 0.000000
360.000000 0.000000
370.000000 0.000000
380.000000 0.000000
390.000000 0.000000
400.000000 0.000000
405.000000 15.000000
```

wenigerkrumm2

```
Tourlänge: 2183.662266
-88.167788 121.352549
-111.471724 100.369591
-129.903811 75.000000
-142.658477 46.352549
-149.178284 15.679269
-149.178284 -15.679269
```

```
-142.658477 -46.352549
-129.903811 -75.000000
-111.471724 -100.369591
-88.167788 -121.352549
-61.010496 -137.031819
-31.186754 -146.722140
0.000000 -150.000000
31.186754 -146.722140
61.010496 -137.031819
88.167788 -121.352549
111.471724 -100.369591
129.903811 -75.000000
142.658477 -46.352549
149.178284 -15.679269
149.178284 15.679269
142.658477 46.352549
129.903811 75.000000
111.471724 100.369591
88.167788 121.352549
61.010496 137.031819
31.186754 146.722140
0.000000 150.000000
-31.186754 146.722140
-61.010496 137.031819
-117.557050 161.803399
-148.628965 133.826121
-173.205081 100.000000
-190.211303 61.803399
-198.904379 20.905693
-198.904379 -20.905693
-190.211303 -61.803399
-173.205081 -100.000000
-148.628965 -133.826121
-117.557050 -161.803399
-81.347329 -182.709092
-41.582338 -195.629520
0.000000 -200.000000
41.582338 -195.629520
81.347329 -182.709092
117.557050 -161.803399
148.628965 -133.826121
173.205081 -100.000000
190.211303 -61.803399
198.904379 -20.905693
198.904379 20.905693
190.211303 61.803399
173.205081 100.000000
148.628965 133.826121
117.557050 161.803399
81.347329 182.709092
41.582338 195.629520
0.000000 200.000000
-41.582338 195.629520
```

```
-81.347329 182.709092
```

wenigerkrumm3

```
Tourlänge: 1848.046986
0.000000 20.000000
-16.632935 21.748192
-32.538931 26.916363
-47.022820 35.278640
-59.451586 46.469551
-69.282032 40.000000
-76.084521 24.721360
-79.561752 8.362277
-79.561752 -8.362277
-76.084521 -24.721360
-69.282032 -40.000000
-59.451586 -53.530449
-47.022820 -64.721360
-32.538931 -73.083637
-16.632935 -78.251808
0.000000 -80.000000
16.632935 -78.251808
32.538931 -73.083637
47.022820 -64.721360
52.977180 -64.721360
67.461069 -73.083637
83.367065 -78.251808
100.000000 -80.000000
116.632935 -78.251808
132.538931 -73.083637
147.022820 -64.721360
159.451586 -53.530449
169.282032 -40.000000
176.084521 -24.721360
179.561752 -8.362277
179.561752 8.362277
176.084521 24.721360
169.282032 40.000000
159.451586 46.469551
147.022820 35.278640
132.538931 26.916363
116.632935 21.748192
100.000000 20.000000
83.367065 21.748192
76.084521 24.721360
69.282032 40.000000
59.451586 46.469551
59.451586 53.530449
69.282032 60.000000
76.084521 75.278640
83.367065 78.251808
100.000000 80.000000
116.632935 78.251808
132.538931 73.083637
```

```
147.022820 64.721360
159.451586 53.530449
169.282032 60.000000
176.084521 75.278640
179.561752 91.637723
179.561752 108.362277
176.084521 124.721360
169.282032 140.000000
159.451586 153.530449
147.022820 164.721360
132.538931 173.083637
116.632935 178.251808
100.000000 180.000000
83.367065 178.251808
67.461069 173.083637
52.977180 164.721360
47.022820 164.721360
32.538931 173.083637
16.632935 178.251808
0.000000 180.000000
-16.632935 178.251808
-32.538931 173.083637
-47.022820 164.721360
-59.451586 153.530449
-69.282032 140.000000
-76.084521 124.721360
-79.561752 108.362277
-79.561752 91.637723
-76.084521 75.278640
-69.282032 60.000000
-59.451586 53.530449
-47.022820 64.721360
-32.538931 73.083637
-16.632935 78.251808
0.000000 80.000000
16.632935 78.251808
23.915479 75.278640
32.538931 73.083637
30.717968 60.000000
30.717968 40.000000
32.538931 26.916363
23.915479 24.721360
16.632935 21.748192
20.438248 8.362277
20.438248 -8.362277
23.915479 -24.721360
30.717968 -40.000000
40.548414 -53.530449
59.451586 -53.530449
69.282032 -40.000000
76.084521 -24.721360
79.561752 -8.362277
79.561752 8.362277
```

```
67.461069 26.916363
52.977180 35.278640
47.022820 35.278640
40.548414 46.469551
40.548414 53.530449
47.022820 64.721360
52.977180 64.721360
67.461069 73.083637
79.561752 91.637723
79.561752 108.362277
76.084521 124.721360
69.282032 140.000000
59.451586 153.530449
40.548414 153.530449
30.717968 140.000000
23.915479 124.721360
20.438248 108.362277
20.438248 91.637723
```

wenigerkrumm4

```
Tourlänge: 1205.068555
42.137753 -60.319863
94.789917 -67.087689
144.832862 -43.476284
153.130159 -20.360910
139.446709 0.233238
101.498782 33.484198
51.008140 5.769601
-16.723130 -12.689542
-20.971208 -5.637107
28.913721 58.699880
33.379688 100.161238
20.212169 156.013261
-107.988514 185.173669
-119.026308 168.453598
-154.088455 115.022553
-219.148505 103.685337
-240.369194 57.426131
-239.414022 40.427118
-239.848226 8.671399
-221.149792 -32.862538
-191.716829 -28.360492
-137.317503 -20.146939
-98.760442 -81.770618
-82.864121 -104.173600
-129.104485 -155.041640
```

wenigerkrumm5

```
Tourlänge: 3257.920434
267.845908 127.627482
244.228552 119.192512
171.595574 135.520994
```

```
116.702667 132.021991
63.541591 55.140221
62.366656 50.713913
45.123674 31.740242
25.098172 35.205544
-8.936916 13.543851
-70.183535 73.738342
-86.457580 105.836348
-82.173510 119.465553
-68.446198 137.178953
27.706327 169.284192
36.599805 147.885350
47.040512 141.206562
51.417675 146.417721
38.654730 188.608557
-30.991436 186.807059
-44.669924 170.088013
-49.447381 173.210759
-104.781549 158.212048
-177.685937 158.265884
-223.039999 171.558368
-258.868593 166.669198
-280.008136 11.657786
-286.024059 -55.955204
-284.547616 -60.154961
-278.409792 -111.914073
-281.678990 -187.717923
-260.477802 -196.955535
-251.656688 -195.189953
-214.362324 -193.265190
-116.831788 -191.380552
-41.263039 -144.118212
-57.266232 -115.737582
-58.684205 -76.988884
-31.548604 -55.223912
-74.639411 25.542881
-81.384378 32.368323
-95.621797 77.468533
-136.787038 79.501703
-235.099412 47.810306
-247.341131 -160.277639
-234.711279 -162.774591
-202.218443 -178.735864
-147.023475 -166.220130
26.451074 -192.813352
30.366828 -167.573232
90.584569 -164.218416
142.765554 -118.682439
162.493244 -84.574019
141.513053 2.821137
92.639946 22.216030
106.033430 69.754891
209.544977 94.267052
```

```
239.639550 79.491132
253.534863 38.014987
283.989938 -101.866465
263.236651 -144.293091
```

wenigerkrumm6

```
Tourlänge: 3457.994092
-288.744132 -173.349893
-293.833463 -165.440105
-189.988471 -98.043874
-144.887799 -73.495410
-126.569816 -30.645224
-139.741580 57.936680
-102.699992 95.632069
-97.391662 124.120512
-167.994506 138.195365
-194.986965 101.363745
-175.118239 77.842636
-100.569041 140.808607
-89.453831 162.237392
-55.091518 198.826966
102.223372 174.201904
154.870684 140.327660
121.661135 85.267672
102.909291 60.107868
83.005905 64.646774
64.559003 82.567627
73.689751 110.224271
85.043830 108.946389
100.006932 76.579303
121.392544 56.694081
81.740403 10.276251
76.647124 -7.289705
112.833346 -38.057607
138.136997 -31.348808
155.341949 -20.252779
172.836936 59.184233
245.415055 44.794067
289.298882 56.051342
277.821597 104.262606
255.134924 115.594915
246.621634 101.705861
231.944823 82.961057
228.929427 82.624982
175.677917 98.929343
151.432196 121.427337
132.794476 135.681392
120.906436 131.798810
96.781707 141.370805
49.091876 150.678826
21.067444 122.164599
20.218290 88.031173
56.716498 66.959624
```

```
58.019653 49.937906
58.716620 32.835930
40.327635 19.216022
14.005617 -14.015334
-18.507391 -22.905270
-31.745416 -69.207960
-19.310012 -131.810965
21.176627 -165.421555
46.674278 -193.090008
152.102728 -193.381252
202.346980 -189.069699
242.810288 -182.054289
245.020791 -167.448848
238.583388 -133.143524
210.186432 -127.403563
187.669263 -122.655771
150.526118 -88.230057
134.692592 -102.152826
143.114152 -135.866707
157.588994 -144.200765
216.825920 -152.024123
221.028639 -139.435079
155.405344 -56.437901
92.255820 -93.514104
64.943433 -119.784474
19.765322 -99.236400
-4.590656 -40.067226
-51.343758 -57.654823
-72.565291 -24.281820
-90.160190 -25.200829
-107.196865 -77.792599
-154.225945 -135.522059
-191.216327 -162.689024
-187.485329 -177.031237
```

wenigerkrumm7

```
Tourlänge: 4150.643872
-181.208895 -192.622935
-202.828627 -101.700050
-172.378071 -88.298187
-152.130365 -93.844349
-133.730932 -113.306155
-155.651746 -138.151811
-189.135201 -139.078513
-244.959501 -111.046573
-217.282470 -43.316616
-211.429137 27.770425
-207.665172 81.410371
-185.649161 90.144456
-189.062172 104.285631
-215.113949 120.740679
-222.492322 169.033315
-240.249363 179.334919
```

```
-268.739142 143.276483
-287.058297 113.599823
-284.129027 107.252583
-248.169463 80.132237
-226.787625 2.658862
-184.092700 5.737284
-171.354954 25.463068
-147.363185 59.608175
-126.568914 106.964962
-56.914543 92.501249
-27.911955 48.326745
-42.704691 37.679514
-48.354421 9.091412
-46.403062 -13.755804
-47.266557 -66.984045
-17.356579 -125.254131
34.959132 -106.842499
68.910854 -82.123346
106.599423 -107.433987
126.904044 -80.733297
135.781192 -13.053440
141.433472 -6.023095
129.024315 29.701695
74.887500 80.586458
53.436521 125.683201
54.766523 154.053847
34.079032 187.731112
-13.030259 174.698005
-120.386562 170.589454
-134.985023 132.944989
-157.423365 126.800331
-200.771246 147.741054
-201.485143 155.274830
-192.681053 174.522947
-153.130140 187.817274
-114.146166 190.615321
-84.626900 148.216494
8.643660 135.907430
16.573231 104.020979
40.897139 78.152317
54.551865 71.567133
56.389778 71.618509
57.555364 64.417343
55.434792 62.729160
3.152113 27.103890
-4.434919 33.164884
-18.316063 27.755860
-9.580869 -17.516639
-0.200936 -21.927663
47.011363 -30.887180
55.550895 -45.089968
88.818853 -42.834512
118.989764 -80.203583
```

```
172.389228 -53.133270
158.552316 -19.254407
189.387028 -4.465225
239.616628 -21.944160
296.911892 25.811569
271.301094 142.524086
221.808162 186.241012
170.514252 161.169850
169.990437 154.260412
148.108328 123.558283
126.799911 112.727280
120.375033 115.889661
100.043893 161.295125
95.947213 183.278211
130.854855 195.695082
217.218893 164.294928
216.691000 156.314370
208.592696 136.618460
162.923138 117.465744
158.742184 62.618834
178.198360 37.031984
205.717887 -24.976511
224.599361 -34.798485
256.475967 -46.591418
276.276517 -49.448662
278.105364 -93.771765
275.793495 -129.415477
235.827007 -143.838844
217.599278 -189.258062
92.298040 -146.169487
59.827200 -170.713714
```