Aufgabe 1: Laubmaschen

Teilnahme-ID: 69082

Bearbeiter dieser Aufgabe: Vincent Xigu Liu

05. März 2024

**Inhaltsverzeichnis**

1. [Beweis zur Schwierigkeit des Problems 2](#_bookmark4)
2. [Algorithmische Ansätze 4](#_bookmark9)
   1. [Greedy Algorithmus mit iterativer Suche 4](#_bookmark10)
      1. [Generierung von Suchpfaden 4](#_bookmark11)
      2. [Evaluation von Suchpfaden und überprüfen auf Gültigkeit 6](#_bookmark12)
      3. [Algorithmisches Konzept 7](#_bookmark15)
   2. [Optimierungsverfahren 7](#_bookmark17)
   3. [Greedy Optimierung 9](#_bookmark19)
      1. [Optimierung kurzer gültiger Pfade 9](#_bookmark20)
      2. [Optimierung aller gültigen Pfade 9](#_bookmark21)
   4. [Lineare Optimierung 9](#_bookmark22)
      1. [Reduktion 10](#_bookmark29)
      2. [Untere Schranke 12](#_bookmark41)
3. [Mögliche Erweiterungen 12](#_bookmark42)
   1. [Festlegung des Start- und Zielknotens 12](#_bookmark43)
   2. [Mehrdimensionale Routen 13](#_bookmark46)
   3. [Finden eines Weges mit möglichst wenig ungültigen Winkeln 14](#_bookmark49)
4. [Implementierung 15](#_bookmark52)
5. [Beispiele 15](#_bookmark53)
6. [Quellcode 20](#_bookmark54)

Im Laubblaseproblem geht es darum, wie der Hausmeister im quadratischen Feld das Laub möglichst auf ein Feld versammelt. Da wir allerdings feststellen, dass eine Verallgemeinerung zum rechteckigen Feld als Erweiterung gut mit den bisherigen Lösungsansätzen vereinbar ist, behandeln wir im Folgenden den allgemeineren Ansatz mit rechteckigen Feldern. Da quadratische Felder auch rechteckig sind, lassen sich also alle im Folgenden verwendeten Ansätze auch als Lösungen für den quadratischen Schulhof verstehen.

Wenn wir das Feld als eine zweidimensionale Liste modellieren, können wir den Blasevorgang als Umverteilung ansehen, wobei man zwei Felder der Liste auswählt und dann den Blasevorgang durchführt. Es sei dazu der gegebene Schulhof S und sei dabei das i-te Feld in der j-ten Spalte des Schulhofes und der Wert die Anzahl der Blätter auf diesem Feld. Der Blasevorgang über ein Feld, der nun einen Algorithmus zur Veränderung von S darstellt und dabei ein Feld und eine Richtung (oben, unten, links, rechts) als Eingabe nimmt, wird durch die Koordinaten (i,j) und die Richtungsvektoren ((0,-1), (0,1), (1,0), (-1,0)) dargestellt. Das Ziel ist es nun, die Laubanzahl in S für ein gegebenes Feld zu maximieren oder zumindest möglichst viel Laub in die Nähe des Feldes zu bringen, um dem Hausmeister seine Arbeit beim Zusammenblasen des Laubes zu vereinfachen.

# Überlegungen zum Simulationsprogramm

Zunächst wollen wir uns Gedanken darüber machen, wie es möglich ist, ein Simualtionsprogramm entsprechend der Aufgabenstellung und zu Implementieren. Dazu untersuchen wir zunächst, welche Umsetzung für die vorgegebenen Regeln sinnvoll sind und beziehen uns dann auf Fälle, welche den Rand und die Ecken betreffen, um uns dort realistische Ansätze zu überlegen. Da der Hausmeister auf einem Feld stehen muss, um auf ein anderes zu pusten, nennen wir einen Zug Legitim, wenn das Feld, worauf er steht sowie das Feld, worauf er pustet beide Teile des Schulhofes sind. Daher nimmt das Programm zur Simulation des Blasevorgangs auch zunächst die Eingaben des Feldes , auf den der Hausmeister steht, und die Pusterichtung , in die geblasen werden soll, um daraufhin den Blasevorgang zu simulieren.

## Ansätze für die vorgegebenen Regeln

Ein Bild, das Text, Screenshot, Diagramm, Schrift enthält.

Automatisch generierte BeschreibungWenn die beiden Eingaben für das Programm getätigt wurden, ist nun das Feld, von dem Laub geblasen wird, , das Feld, auf das das Laub landet, und das entsprechende Feld dahinter . Da durch das Vertauschen der x- und y-Koordinate alle von uns betrachteten Vektoren um 90° auf der Ebene gedreht werden, stellt ein Seitenfeld dar, wenn der gedrehte Vektor ist. Die Drehung können wir dadurch erzielen, dass wir die x-und y-Koordinaten des Vektors miteinander vertauschen. Da die Seitenfelder auf den gegenüberliegenden Seiten des geblasenden Feldes liegen, ist das andere Seitenfeld entsprechend (siehe Abb.1)[[1]](#footnote-1).

Um die Laubblätter einzeln zu simulieren, nutzen wir hierfür einen Pseudo-Zufallszahlengenerator aus der Python-Bibliothek random. Für jedes Blatt aus generieren wir also zunächst eine natürliche Zufallszahl zwischen 0 und einschließlich 9 und subtrahieren genau dann 1 von und addieren 1 auf , wenn diese Zahl 0 ist. Diesen Prozess wiederholen wir für jedes Blatt genau ein mal. Analog generieren wir dann für jedes Blatt aus zwei natürliche Zufallszahlen zwischen 0 und einschließlich 9, und addieren 1 zu in , wenn die erste Zahl 0 beträgt beziehungsweise addieren 1 zu und ziehen entsprechend vom Feld die Blätter ab. Der Rest der Blätter wird dann auf das Feld gepustet. Dadurch kann das Laubfeld entsprechend der Regeln verändert werden und die Simulation für jedes Blatt auf den betreffenden Feldern einzeln durchgeführt werden.

Abb. 1: Felder nach Beschreibung durch dem Blasefeld und dem Richtungsvektor .

## Randregeln

Dass alle Felder aus Abb.1 nun auch auf dem Schulhof liegen, muss nicht immer der Fall sein. In solchen Fällen ist es daher sinnvoll, sich realistische Regeln selbst zu überlegen. Im Folgenden gehen wir davon aus, dass der Schulhof innerhalb des Schulgebäudes liegt, also überall am Rande des Hofes Wände des Schulgebäudes stehen. Dadurch ist es nicht möglich, dass durch Pusten Laub vom Rand des Schulhofes verloren geht. Im Folgenden versuchen wir modellierend Regeln für ein solches Phänomen darzustellen:

**Fall 1:**  liegt außerhalb des Schulhofes.

In einem solchen Fall versucht der Hausmeister prinzipiell, gegen eine Wand zu pusten. Da das Laub, welches verlässt, nun keinen Platz mehr auf hat, wegen dem Laubdruck jedoch woanders hinmuss, nehmen wir modellierend an, dass jedes Laubblatt mit Wahrscheinlichkeit 50% beschließt, jeweils auf beziehungsweise auf zu landen.

**Fall 2:**  liegt außerhalb des Schulhofes.

Da hier die Blätter auf analog zu Fall 1 nicht weiter nach vorne gepustet werden können, jedoch die „Blaskraft“ des Laubbläsers schwächer ist, verbleiben diese einfach auf das Feld

**Fall 3:** oder liegen außerhalb des Schulhofes.

In einem solchen Fall wir das Laubblatt zwar von geblasen, kann jedoch nicht auf die Seitenfelder landen. Wir nehmen daher modellierend an, dass die Wand die Laubblätter einfach zurückhält und alle Blätter, welche auf ein Seitenfeld hätten landen sollen, stattdessen auf landen.

Wir beachten dabei insbesondere, dass die Fälle nicht exklusiv eintreten: Wenn Fall 1 eintritt, so tritt Fall 2 automatisch ebenfalls ein. Wenn Fall 2 eintritt, kann Fall 3 möglicherweise auch eintreten. In solchen Fällen ist es wichtig, die betrachteten Felder von denen Laub gepustet werden soll, falls überhaupt im Schulhof liegend, einzeln zu simulieren und dabei die Reihenfolge der Fallnummern einzuhalten.

## Laufzeit des Simulationprogramms

Wenn wir von einer konstanten Laufzeit zur Generierung einer Zufallszahl ausgehen, ist die Laufzeit eines derartigen Simulationsprogramms nicht höher als , wobei n die Anzahl der Laubblätter in den Feldern auf Abb.1 außer ist und k eine Konstante ist, um zu ermitteln, welche der beachteten Felder nicht im Schulhof liegen, da das Programm jedes Laubblatt betrachet, eine Zufallszahl für diese generiert und das Blatt entsprechend der Zufallszahl im Schulhof umverlegt. Wenn wir annähernd von einer festen benötigten Anzahl an Zügen z(g), welche von der Feldgröße abhängt, ausgehen (was aber nicht der Fall hier ist), können wir annähernd sagen, dass damit die Gesamtlaufzeit zur Simulation einer kompletten Blasesequenz nicht übersteigt, wobei wir die Feldgröße g als die Anzahl der Planquadrate im Schulhof definieren und n die durchschnittliche Anzahl der Blätter auf den Feldern darstellt.

# Heuristische Untersuchungen zum Lösungsansatz

Im folgenden Abschnitt wollen wir verschiedene Lösungsansätze untersuchen und diskutieren, welche zum Blasen auf ein Feld sinnvoll sind. Da der Simulationsprozess allerdings mit Zufallsvariablen bestückt ist, muss man in der Regel dynamische Strategien wählen, um zufällige Abweichungen vom Erwartungszustand des Schulhofes mit berücksichtigen zu können. Bevor wir allerdings auf allgemeine Blasestrategien zurückkommen, untersuchen wir zunächst das Blasen der Blätter aus zwei Feldern Entfernung sowie das Blasen der Blätter vom Rand, welche leichter zu lösende Spezialfälle des Laubblaseproblems darstellen.

## 2.1 Sonderfall: Blasen von zwei Feldern Entfernung

Wenn der Hausmeister ein Planquadrat bepustet, welches zwei Felder von ihm entfernt ist, ist es dem Hausmeister möglich, sämtliche Blätter vom bepusteten Planquadrat in das Zielquadrat zu befördern. Ein solcher Sonderfall ist dann möglich, wenn das betrachtete Planquadrat als ausgedrückt werden kann, wobei Teil des Schulhofes ist. Das Quadrat, auf dem dann Laub gepustet wird, ist entsprechend . Der Hausmeister steht dann zunächst auf und pustet wiederholt auf das Feld . Dadurch wird jedes Blatt aus mit einer Wahrscheinlichkeit von 10% auf das Feld gepustet. Die erwartete Anzahl[[2]](#footnote-2) der Blätter, die also in gepustet wird, beträgt . Da das Feld nach konsekutivem Pusten erst 0 wird, wenn alle Blätter auf das Feld gelandet sind, ist die erwartungswert der Blätter immer positiv. Da zudem keine Blätter vom Feld gepustet werden, steigt die Anzahl der Blätter auf diesem Feld während des Blasevorgangs immer. Dies führt dazu, dass nach endlich vielen Wiederholungen des Pustens im Erwartungsfall alle Blätter irgendwann auf landen.

Wenn wir die Anzahl der Pustwiederholungen untersuchen, die erwartet nötig sind, um alle Blätter in das Feld zu bringen, nehmen wir modellhaft an, dass ab einer erwarteten Anzahl von 0.5 Blättern keine Blätter mehr auf dem Feld liegen. Nun untersuchen wir, wie viele Wiederholungen nötig sind, um alle bis auf theoretische 0.5 Blätter auf zu pusten. Da nach jeder Pustewiederholung erwartungsgemäß nur noch 90% der Blätter in überbleibt, gilt die Gleichung:

Wobei x die Anzahl der Blätter und y die Anzahl der Pustewiederholungen ist. Nach umstellen erhalten wir:

(1.1.4)

Welches die Anzahl der erwarteten Wiederholungen beschreibt, die nötig sind, um alle Blätter bis auf eins in wegzupusten. Diese theoretische Modellierung entspricht den mithilfe eines Python Programms berechneten Simulationsergebnissen, die anhand des folgenden Pseudocodes durchgeführt wurden (siehe Abb. 4):

**Algorithmus 1. Laubblatt-Simulation**

Blätter x

Pusteanzahl 0

**While** Blätter > 0 **do**:

Pusteanzahl

**For** i = 0, i < Blätter, i **do:**

rand Zufallszahl zwischen einschl. 0 und 9

**If** rand = 0 **do:**

Blätter Blätter

**Return** Pusteanzahl

Ein Bild, das Text, Screenshot, Reihe, Diagramm enthält.

Automatisch generierte BeschreibungAnhand dieser Untersuchungen erkennen wir, dass die Anzahl der Pustezügen einen Logarithmischen Zusammenhang mit der Anzahl der zu pustenden Blätter auf ein Feld aufweist. Da die Laufzeit zum Bepusten der Blätter als konstant angenommen werden kann, lässt sich daraus beim Design der Algorithmen auf die Laufzeit schließen.

## 2.2 Sonderfall: Bepustung der Ränder

Wenn wir die Ränder bepusten möchten, fällt auf, dass diese nicht sehr schnell vom Rand des Schulhofes gepustet werden können, weil analog zum Abschnitt 2.1 immer nur ein Teil der Blätter durch die Seitenverluste vom Rand geblasen werden kann und dieser Vorgang entsprechend lange dauert. Insbesondere in der Implementation eines eher schnellen Algorithmus mit wenigen Pustevorgängen müsste der Hausmeister also sehr viel und sehr lange pusten. Da der Hausmeister aber auch Besen und Tüte hat, wollen wir für einen schnellen Ansatz möglichst schnell möglichst viele Blätter in eine Ecke des Hofes pusten, von wo der Hausmeister anschließend diese auffegen kann. Dazu startet der Hausmeister an einer Ecke und entfernt dort alle Blätter, welche in der Ecke liegen, durch wiederholtes Pusten in die Ecke. Anschließend

# Lösungsansätze

Im Folgenden präsentieren wir nun verschiedene Lösungsansätze für das Laubblasen und Diskutieren diese anhand der Güte der Lösungen sowie anhand ihrer Effizienz.

## 3.1 Greedy Ansatz

Im folgenden Greedy Ansatz möchten wir einen Greedy Algorithmus schreiben, welchen den Schulhof analysiert, mögliche Pustezüge des Hausmeisters berechnet und bewertet, um anschließend den besten Zug herauszugeben. Dabei wird ein Greedy Schritt so oft ausgeführt, bis die Bewertung des Schulhofes nicht mehr verbessert werden kann. Ein Ansatz mit einem Greedy Algorithmus eignet sich hierbei insbesondere für die Aufgabe, da das Sammeln des Laubes durch Pustevorgänge eine stetige Annäherung von allen Laubblättern zum Zielquadrat darstellt und daher insbesondere für große Felder eine gute Bewertungsfunktion mit nicht sehr tiefer Suche auch effizient und schnell zum optimalen Ergebnis führen kann. Der Greedy-Algorithmus in drei Teile gegliedert, welche im Folgenden erläutert werden:

1. Ermitteln aller legitimen Züge (wird nur einmal ausgeführt)

Wiederhole bis Bewertung nicht mehr optimierbar:

1. Errechnen der Schulhofbewertung anhand des Pusteverhaltens des Hausmeisters
2. Ermitteln des besten Zuges
3. Ermitteln aller legitimen Züge

Um zunächst alle möglichen Pustezüge in einem Schulhof ermitteln zu können, suchen wir alle Kombinationen heraus und speichern diese in eine Liste. Da die möglichen Züge sich nicht durch das Umverteilen der Blätter ändern, müssen wir diesen Schritt nur einmal durchführen.

1. Errechnen der Schulhofbewertung nach den Zügen

Da in den meisten Fällen aufgrund der Varianzen durch den zufälligen Pustevorgang genaue Blätterverteilungen nicht vorhersehbar sind (oder nach vielen vorausberechneten Pustezügen vom erwartungswert stark abweichen), berechnen wir die Güte des Zuges direkt anhand einer heuristischen Bewertung. Dafür seien die Koordinaten des Zielfeldes. Die *Entfernung zu Zielfeld* eines Feldes S = definieren wir als die Anzahl der Felder, über die man von S mindestens gehen muss, um Z zu gelangen. Formal ist diese Entfernung also . Die Bewertung, die ein Schulhof erhält, richtet sich zusätzlich daran, ob ein Blatt auf einem Randquadrat oder gar einem Eckquadrat liegt.

### Greedy Ansatz mit Blasen von zwei Feldern Entfernung

vernachlässigende Seitenverlust, weil auf dem Zielfeld zum Ein Bild, das Screenshot, Schrift, Zahl, Typografie enthält.

Automatisch generierte BeschreibungSchluss viel mehr Blätter liegen als am Anfang, was bei der Pustereihenfolge die Frage stellt, wie die Anzahl der Blätter, die direkt neben dem Zielfeld liegen, am besten verteilt sind, um daraus die größtmögliche Blattanzahl für den Zielhaufen zu erzielen. Bei Gleichverteilung der vier zum Zielfeld benachbarten Felder ist es beispielsweise sinnvoll, zunächst die gegenüberliegenden Felder zum Zielfeld zu bepusten und anschließend die anderen zwei sich gegenüberliegenden Felder zu bepusten, anstatt diese in irgendeiner anderen Reihenfolge bepusten. Dies liegt daran, dass die Verluste nach hinten auf das Feld dadurch insgesamt geringer ausfallen, weil diese beim zweiten und vierten Mal Pusten ausfallen und nicht ansonsten beim dritten und vierten Mal. Da der Zielhaufen jedoch mit der Anzahl an Pustzügen weiter wächst und deswegen die Verluste nach hinten in das Feld größer werden, entstehen bei anderen Reihenfolgen größere Verluste. Daraus lässt sich die heuristische Strategie schließen, bei alle Eckfelder (also Felder mit Entfernung 1, die diagonal mit dem Zielfeld benachbart sind,) zunächst zu bepusten und anschließend diese paarweise gegenüberliegend zum Zielfeld zu blasen, um eine Abnahme der Blattanzahl zu vermeiden. (Eckfeldr??)

### Heuristischer Ansatz

Im Folgenden betrachten wir die „Kantenfelder“, die wir als die Felder definieren, die direkt an Feldern der Entfernung grenzen und eine Kante mit diesen gemeinsam haben. Um diese in die richtige Richtung zu pusten, orientieren wir uns einfach daran, in welche Richtung die benachbarten Felder mit Abstand von liegen. Wenn die Kantenfelder jedoch in eine Reihe liegen, sollten diese immer von außen zur Mitte gepustet werden, weil dadurch die Verluste nach hinten mehr in die Mitte (also näher zum Zielquadrat) konzentrieren. Dies liegt daran, dass die Seitenverluste anschließend weiter mitgepustet werden, sodass wir letztendlich mehr Blätter in der Mitte konzentriert haben, wenn wir von der Seite pusten. Da heuristisch gesehen die Menge an Blättern bei gleichen betrachteten Feldern höher ist, wenn die durchschnittliche euklidische Distanz der auf den weniger als entfernten Feldern verteilten Blättern kleiner ist, ist es also sinnvoll, in die Strategie mit einzuarbeiten, dass, wenn man Kanten bepusten will, diese abwechselnd von „Außen“ zur Mitte tut (vgl. Abb. 3).

## Betrachtung der Ränder

# Lösungsansätze

## 3.1 Greedy-Algorithmus

## 3.2 Optimierungen

## 3.2 Nichtdynamische Heuristik

# Theoretische Analyse

# Mögliche Erweiterungen

## Der Schulhof ist rechteckig

## Frei wählbares Zielfeld

## Der Schulhof ist mehrdimensional

## Der Laubbläser wurde ausgewechselt

# Implementierung

Für die laubblattsimulation 683.2202928066254 seconds wallclock time. Prozessor:Intel core i5 2430M. Für den Zufallsgenerator nutzen wir pseud-zufallszahlen.

# Beispiele

Die folgende Tabelle fasst die Ergebnisse der verschiedenen Ansätze zusammen. In den Spalten ist die kürzeste gefundene Tour eingetragen. Wir gehen dabei in allen approximativen Ansätzen von einer ungefähren Laufzeit von 300s aus. In dem ILP Verfahren ist die Laufzeit angegeben. Das gleiche gilt für die untere Schranke. In dem ILP Verfahren wurde in einigen Beispielen das Verhalten des Solvers auf aggressiv gesetzt. Dies ist allerdings gleichzusetzen mit der Verwendung eines schnelleren Solvers und bestimmt daher nicht die Qualität der Ergebnisse. Der benötigte Arbeitsspeicher ist in allen Fällen gering und übersteigt nie wenige Gigabyte. Eine Verwendung der unteren Schranke, setzt eine Installation des GLPK Solvers voraus.

Die betrachteten Ansätze sind die folgenden:

ILP unser optimaler Lösungsweg

G1 unser Greedy Algorithmus mit iterativer Suche

G2 unsere erste Kombination in der gültige Pfade optimiert werden, die einen neuen kürzesten gültigen Pfad formen

G3 unsere zweite Kombination, in der alle gültigen Pfade optimiert werden US unsere untere Schranke

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Eingabedatei | ILP | G1 | G2 | G3 | US |
| wenigerkrumm1 | 847.434 in 91s | 847.434 | 847.434 | 847.434 | 847.4342000 in 20.8s |
| wenigerkrumm2 | 2183.66 in 141s | 2183.66 | 2183.66 | 2183.66 | 2153.285600 in 4.2s |
| wenigerkrumm3 | 1848.05 in 548s | 1878.27 | 1878.27 | 1848.53 | 1815.649473 in 134.8s |
| wenigerkrumm4 | 1205.07 in 2s | 1205.07 | 1205.07 | 1205.07 | 1205.068140 in 0.1s |
| wenigerkrumm5 | 3257.92 in 53s | 3335.79 | 3317.75 | 3257.92 | 2729.794309 in 4.0s |
| wenigerkrumm6 | 3457.99 in 2871s | 3821.9 | 3757.81 | 3553.59 | 2992.783828 in 16.3s |
| wenigerkrumm7 | 4150.64 in 12173s | 4665.42 | 4660.58 | 4279.77 | 3663.550349 in 48.3s |

Die Programme liefen dabei auf einem Kern eines Intel i7-5960X.

Im Folgenden werden die kürzesten Ergebnispfade aller Dateien aufgeführt. Dabei bedeutet

-> , dass wir uns von einem Ort zum nächsten bewegen. Alle angegebenen Beispiele sind die Lösung des Solvers.

Wenigerkrumm1

(405|15) -> (400|0) -> (390|0) -> (380|0) -> (370|0) -> (360|0) -> (350|0) -> (340|0) -> (330|0) ->

(320|0) -> (310|0) -> (300|0) -> (290|0) -> (280|0) -> (270|0) -> (260|0) -> (250|0) -> (240|0) ->

(230|0) -> (220|0) -> (210|0) -> (200|0) -> (190|0) -> (180|0) -> (170|0) -> (160|0) -> (150|0) ->

(140|0) -> (130|0) -> (120|0) -> (110|0) -> (100|0) -> (90|0) -> (80|0) -> (70|0) -> (60|0) -> (50|0) ->

(40|0) -> (30|0) -> (20|0) -> (10|0) -> (0|0) -> (-5|15) -> (0|30) -> (10|30) -> (20|30) -> (30|30) ->

(40|30) -> (50|30) -> (60|30) -> (70|30) -> (80|30) -> (90|30) -> (100|30) -> (110|30) -> (120|30) -> (130|30) -> (140|30) -> (150|30) -> (160|30) -> (170|30) -> (180|30) -> (190|30) -> (200|30) -> (210| 30) -> (220|30) -> (230|30) -> (240|30) -> (250|30) -> (260|30) -> (270|30) -> (280|30) -> (290|30) -

> (300|30) -> (310|30) -> (320|30) -> (330|30) -> (340|30) -> (350|30) -> (360|30) -> (370|30) -> (380|30) -> (390|30) -> (400|30)

Laenge: 847.434

Wenigerkrumm2

(173.205|-100) -> (190.211|-61.8034) -> (198.904|-20.9057) -> (198.904|20.9057) -> (190.211| 61.8034) -> (173.205|100) -> (148.629|133.826) -> (117.557|161.803) -> (81.3473|182.709) -> (41.5823|195.63) -> (0|200) -> (-41.5823|195.63) -> (-81.3473|182.709) -> (-117.557|161.803) -> (- 148.629|133.826) -> (-173.205|100) -> (-190.211|61.8034) -> (-198.904|20.9057) -> (-198.904|-

20.9057) -> (-190.211|-61.8034) -> (-173.205|-100) -> (-148.629|-133.826) -> (-117.557|-161.803) -

> (-81.3473|-182.709) -> (-41.5823|-195.63) -> (0|-200) -> (41.5823|-195.63) -> (81.3473|-182.709)

-> (117.557|-161.803) -> (148.629|-133.826) -> (129.904|-75) -> (142.658|-46.3525) -> (149.178|- 15.6793) -> (149.178|15.6793) -> (142.658|46.3525) -> (129.904|75) -> (111.472|100.37) -> (88.1678|121.353) -> (61.0105|137.032) -> (31.1868|146.722) -> (0|150) -> (-31.1868|146.722) -> (-61.0105|137.032) -> (-88.1678|121.353) -> (-111.472|100.37) -> (-129.904|75) -> (-142.658|

46.3525) -> (-149.178|15.6793) -> (-149.178|-15.6793) -> (-142.658|-46.3525) -> (-129.904|-75) ->

(-111.472|-100.37) -> (-88.1678|-121.353) -> (-61.0105|-137.032) -> (-31.1868|-146.722) -> (0|-

150) -> (31.1868|-146.722) -> (61.0105|-137.032) -> (88.1678|-121.353) -> (111.472|-100.37)

Laenge: 2183.66

Wenigerkrumm3

(20.4382|91.6377) -> (20.4382|108.362) -> (23.9155|124.721) -> (30.718|140) -> (40.5484|153.53)

-> (59.4516|153.53) -> (69.282|140) -> (76.0845|124.721) -> (79.5618|108.362) -> (79.5618| 91.6377) -> (67.4611|73.0836) -> (52.9772|64.7214) -> (47.0228|64.7214) -> (40.5484|53.5304) -> (40.5484|46.4696) -> (47.0228|35.2786) -> (52.9772|35.2786) -> (67.4611|26.9164) -> (79.5618| 8.36228) -> (79.5618|-8.36228) -> (76.0845|-24.7214) -> (69.282|-40) -> (59.4516|-53.5304) ->

(40.5484|-53.5304) -> (30.718|-40) -> (23.9155|-24.7214) -> (20.4382|-8.36228) -> (20.4382| 8.36228) -> (16.6329|21.7482) -> (23.9155|24.7214) -> (32.5389|26.9164) -> (30.718|40) -> (30.718|60) -> (32.5389|73.0836) -> (23.9155|75.2786) -> (16.6329|78.2518) -> (0|80) -> (- 16.6329|78.2518) -> (-32.5389|73.0836) -> (-47.0228|64.7214) -> (-59.4516|53.5304) -> (-69.282|

60) -> (-76.0845|75.2786) -> (-79.5618|91.6377) -> (-79.5618|108.362) -> (-76.0845|124.721) -> (-

69.282|140) -> (-59.4516|153.53) -> (-47.0228|164.721) -> (-32.5389|173.084) -> (-16.6329|

178.252) -> (0|180) -> (16.6329|178.252) -> (32.5389|173.084) -> (47.0228|164.721) -> (52.9772|

164.721) -> (67.4611|173.084) -> (83.3671|178.252) -> (100|180) -> (116.633|178.252) -> (132.539|173.084) -> (147.023|164.721) -> (159.452|153.53) -> (169.282|140) -> (176.085| 124.721) -> (179.562|108.362) -> (179.562|91.6377) -> (176.085|75.2786) -> (169.282|60) -> (159.452|53.5304) -> (147.023|64.7214) -> (132.539|73.0836) -> (116.633|78.2518) -> (100|80) -> (83.3671|78.2518) -> (76.0845|75.2786) -> (69.282|60) -> (59.4516|53.5304) -> (59.4516|46.4696)

-> (69.282|40) -> (76.0845|24.7214) -> (83.3671|21.7482) -> (100|20) -> (116.633|21.7482) -> (132.539|26.9164) -> (147.023|35.2786) -> (159.452|46.4696) -> (169.282|40) -> (176.085| 24.7214) -> (179.562|8.36228) -> (179.562|-8.36228) -> (176.085|-24.7214) -> (169.282|-40) -> (159.452|-53.5304) -> (147.023|-64.7214) -> (132.539|-73.0836) -> (116.633|-78.2518) -> (100|-80)

-> (83.3671|-78.2518) -> (67.4611|-73.0836) -> (52.9772|-64.7214) -> (47.0228|-64.7214) ->

(32.5389|-73.0836) -> (16.6329|-78.2518) -> (0|-80) -> (-16.6329|-78.2518) -> (-32.5389|-73.0836)

-> (-47.0228|-64.7214) -> (-59.4516|-53.5304) -> (-69.282|-40) -> (-76.0845|-24.7214) -> (-

79.5618|-8.36228) -> (-79.5618|8.36228) -> (-76.0845|24.7214) -> (-69.282|40) -> (-59.4516|

46.4696) -> (-47.0228|35.2786) -> (-32.5389|26.9164) -> (-16.6329|21.7482) -> (0|20)

Laenge: 1848.05

Wenigerkrumm4

(-129.104|-155.042) -> (-82.8641|-104.174) -> (-98.7604|-81.7706) -> (-137.318|-20.1469) -> (-

191.717|-28.3605) -> (-221.15|-32.8625) -> (-239.848|8.6714) -> (-239.414|40.4271) -> (-240.369|

57.4261) -> (-219.149|103.685) -> (-154.088|115.023) -> (-119.026|168.454) -> (-107.989|185.174)

-> (20.2122|156.013) -> (33.3797|100.161) -> (28.9137|58.6999) -> (-20.9712|-5.63711) -> (- 16.7231|-12.6895) -> (51.0081|5.7696) -> (101.499|33.4842) -> (139.447|0.233238) -> (153.13|-

20.3609) -> (144.833|-43.4763) -> (94.7899|-67.0877) -> (42.1378|-60.3199)

Laenge: 1205.07

Wenigerkrumm5

(267.846|127.627) -> (244.229|119.193) -> (171.596|135.521) -> (116.703|132.022) -> (63.5416| 55.1402) -> (62.3667|50.7139) -> (45.1237|31.7402) -> (25.0982|35.2055) -> (-8.93692|13.5439) -> (-70.1835|73.7383) -> (-86.4576|105.836) -> (-82.1735|119.466) -> (-68.4462|137.179) -> (27.7063| 169.284) -> (36.5998|147.885) -> (47.0405|141.207) -> (51.4177|146.418) -> (38.6547|188.609) -> (-30.9914|186.807) -> (-44.6699|170.088) -> (-49.4474|173.211) -> (-104.782|158.212) -> (- 177.686|158.266) -> (-223.04|171.558) -> (-258.869|166.669) -> (-280.008|11.6578) -> (-286.024|-

55.9552) -> (-284.548|-60.155) -> (-278.41|-111.914) -> (-281.679|-187.718) -> (-260.478|-

196.956) -> (-251.657|-195.19) -> (-214.362|-193.265) -> (-116.832|-191.381) -> (-41.263|-

144.118) -> (-57.2662|-115.738) -> (-58.6842|-76.9889) -> (-31.5486|-55.2239) -> (-74.6394|

25.5429) -> (-81.3844|32.3683) -> (-95.6218|77.4685) -> (-136.787|79.5017) -> (-235.099|47.8103)

-> (-247.341|-160.278) -> (-234.711|-162.775) -> (-202.218|-178.736) -> (-147.023|-166.22) -> (26.4511|-192.813) -> (30.3668|-167.573) -> (90.5846|-164.218) -> (142.766|-118.682) -> (162.493|-84.574) -> (141.513|2.82114) -> (92.6399|22.216) -> (106.033|69.7549) -> (209.545| 94.2671) -> (239.64|79.4911) -> (253.535|38.015) -> (283.99|-101.866) -> (263.237|-144.293)

Laenge: 3257.92

Wenigerkrumm6

(-187.485|-177.031) -> (-191.216|-162.689) -> (-154.226|-135.522) -> (-107.197|-77.7926) -> (-

90.1602|-25.2008) -> (-72.5653|-24.2818) -> (-51.3438|-57.6548) -> (-4.59066|-40.0672) -> (19.7653|-99.2364) -> (64.9434|-119.784) -> (92.2558|-93.5141) -> (155.405|-56.4379) -> (221.029|-139.435) -> (216.826|-152.024) -> (157.589|-144.201) -> (143.114|-135.867) -> (134.693|-102.153) -> (150.526|-88.2301) -> (187.669|-122.656) -> (210.186|-127.404) -> (238.583|-133.144) -> (245.021|-167.449) -> (242.81|-182.054) -> (202.347|-189.07) -> (152.103|-

193.381) -> (46.6743|-193.09) -> (21.1766|-165.422) -> (-19.31|-131.811) -> (-31.7454|-69.208) -> (-18.5074|-22.9053) -> (14.0056|-14.0153) -> (40.3276|19.216) -> (58.7166|32.8359) -> (58.0197| 49.9379) -> (56.7165|66.9596) -> (20.2183|88.0312) -> (21.0674|122.165) -> (49.0919|150.679) -> (96.7817|141.371) -> (120.906|131.799) -> (132.794|135.681) -> (151.432|121.427) -> (175.678| 98.9293) -> (228.929|82.625) -> (231.945|82.9611) -> (246.622|101.706) -> (255.135|115.595) -> (277.822|104.263) -> (289.299|56.0513) -> (245.415|44.7941) -> (172.837|59.1842) -> (155.342|- 20.2528) -> (138.137|-31.3488) -> (112.833|-38.0576) -> (76.6471|-7.2897) -> (81.7404|10.2763) -

> (121.393|56.6941) -> (100.007|76.5793) -> (85.0438|108.946) -> (73.6898|110.224) -> (64.559| 82.5676) -> (83.0059|64.6468) -> (102.909|60.1079) -> (121.661|85.2677) -> (154.871|140.328) -> (102.223|174.202) -> (-55.0915|198.827) -> (-89.4538|162.237) -> (-100.569|140.809) -> (-175.118| 77.8426) -> (-194.987|101.364) -> (-167.995|138.195) -> (-97.3917|124.121) -> (-102.7|95.6321) ->

(-139.742|57.9367) -> (-126.57|-30.6452) -> (-144.888|-73.4954) -> (-189.988|-98.0439) -> (- 293.833|-165.44) -> (-288.744|-173.35)

Laenge: 3457.99

Wenigerkrumm7

(59.8272|-170.714) -> (92.298|-146.169) -> (217.599|-189.258) -> (235.827|-143.839) -> (275.793|- 129.415) -> (278.105|-93.7718) -> (276.277|-49.4487) -> (256.476|-46.5914) -> (224.599|-34.7985)

-> (205.718|-24.9765) -> (178.198|37.032) -> (158.742|62.6188) -> (162.923|117.466) -> (208.593| 136.618) -> (216.691|156.314) -> (217.219|164.295) -> (130.855|195.695) -> (95.9472|183.278) -> (100.044|161.295) -> (120.375|115.89) -> (126.8|112.727) -> (148.108|123.558) -> (169.99|154.26)

-> (170.514|161.17) -> (221.808|186.241) -> (271.301|142.524) -> (296.912|25.8116) -> (239.617|- 21.9442) -> (189.387|-4.46523) -> (158.552|-19.2544) -> (172.389|-53.1333) -> (118.99|-80.2036) -

> (88.8189|-42.8345) -> (55.5509|-45.09) -> (47.0114|-30.8872) -> (-0.200936|-21.9277) -> (- 9.58087|-17.5166) -> (-18.3161|27.7559) -> (-4.43492|33.1649) -> (3.15211|27.1039) -> (55.4348|

62.7292) -> (57.5554|64.4173) -> (56.3898|71.6185) -> (54.5519|71.5671) -> (40.8971|78.1523) -> (16.5732|104.021) -> (8.64366|135.907) -> (-84.6269|148.216) -> (-114.146|190.615) -> (-153.13| 187.817) -> (-192.681|174.523) -> (-201.485|155.275) -> (-200.771|147.741) -> (-157.423|126.8) -> (-134.985|132.945) -> (-120.387|170.589) -> (-13.0303|174.698) -> (34.079|187.731) -> (54.7665| 154.054) -> (53.4365|125.683) -> (74.8875|80.5865) -> (129.024|29.7017) -> (141.433|-6.0231) -> (135.781|-13.0534) -> (126.904|-80.7333) -> (106.599|-107.434) -> (68.9109|-82.1233) -> (34.9591|-106.842) -> (-17.3566|-125.254) -> (-47.2666|-66.984) -> (-46.4031|-13.7558) -> (- 48.3544|9.09141) -> (-42.7047|37.6795) -> (-27.912|48.3267) -> (-56.9145|92.5012) -> (-126.569|

106.965) -> (-147.363|59.6082) -> (-171.355|25.4631) -> (-184.093|5.73728) -> (-226.788|2.65886)

-> (-248.169|80.1322) -> (-284.129|107.253) -> (-287.058|113.6) -> (-268.739|143.276) -> (- 240.249|179.335) -> (-222.492|169.033) -> (-215.114|120.741) -> (-189.062|104.286) -> (-185.649|

90.1445) -> (-207.665|81.4104) -> (-211.429|27.7704) -> (-217.282|-43.3166) -> (-244.96|-111.047)

-> (-189.135|-139.079) -> (-155.652|-138.152) -> (-133.731|-113.306) -> (-152.13|-93.8443) -> (- 172.378|-88.2982) -> (-202.829|-101.7) -> (-181.209|-192.623)

Laenge: 4150.64

#### Die hier ausgegebenen Lösungen befinden sich auch in einem Ordner in der Abgabe. Darin finden sich auch weitere Beispiele, die die Funktionalität der Erweiterungen beweisen.

# Quellcode

In dem tatsächlichen Quellcode wird vor jeder Funktion nochmal ausführlich auf deren Funktionalitäten und Parameter eingegangen. Dies hätte allerdings komplett den Rahmen von 10 Seiten Quellcode gesprengt und wird daher weggelassen. Das gleiche gilt für den Quellcode der Erweiterungen, der in dem C++ Header zu finden ist.

**\* @attention Mithilfe dieser Struktur können wir den Index eines Knotens zusammen mit der Entfernung zu einem anderen Knoten speichern**

**struct ENTFERNUNG\_ZUM\_KNOTEN**

**{**

**long double Entfernung;**

**signed long long int Index\_des\_Knotens;**

**};**

**\* @attention Dies ist die zentrale Struktur zum abspeichern eines Knotens struct KNOTEN**

**{**

**long double x, y;**

**unsigned long long int Index\_in\_Koordinaten; std::vector<ENTFERNUNG\_ZUM\_KNOTEN> Entfernung\_zu\_anderen\_Knoten; vector<vector<ENTFERNUNG\_ZUM\_KNOTEN>> Folgeknoten;**

**signed long long int Verbindung\_1, Verbindung\_2; bool bereits\_abgeflogen = false;**

**};**

**// Variablen zum abspeichern des kürzesten gefundenen gültigen Pfades long double Länge\_des\_kürzesten\_gefundenen\_Pfades; std::vector<signed long long int> kürzester\_gefundener\_Pfad;**

**// Vektor zum Abspeichern aller Knoten std::vector<KNOTEN> Koordinaten;**

**\* @attention Methode zur Berechnung der Entfernung zwischen zwei Knoten long double Entfernung(KNOTEN &K1, KNOTEN &K2)**

**{**

**return sqrt(pow(K1.x - K2.x, 2.0) + pow(K1.y - K2.y, 2.0));**

**}**

**\* @attention Diese Methode überprüft, ob die drei gegebenen Knoten nacheinander traversiert werden können, ohne mehr als 90° abbiegen zu müssen.**

**bool gültiger\_Winkel(KNOTEN &K1, KNOTEN &K2, KNOTEN &K3)**

**{**

**return ((K1.x - K2.x)/\*w[0]\*/ \* /\*v[0]\*/ (K3.x - K2.x) + (K1.y - K2.y)/\*w[1]\*/ \* /\*v[1]\*/ (K3.y - K2.y)) / (Entfernung(K1,K2)/\*|w|\*/ \* /\*|v|\*/ Entfernung(K2,K3)) <= 0;**

**}**

**\* @attention Methode zur Vorberechnung von Folgeknoten und der Entfernung zu anderen Knoten void Vorberechnung()**

**{**

**for (KNOTEN K1 : Koordinaten)**

**{**

**for (KNOTEN K2 : Koordinaten)**

**{**

**ENTFERNUNG\_ZUM\_KNOTEN Entfernung\_zum\_Knoten; Entfernung\_zum\_Knoten.Index\_des\_Knotens = K2.Index\_in\_Koordinaten; // Index von K2 in**

**Koordinaten**

**if (K1.Index\_in\_Koordinaten != K2.Index\_in\_Koordinaten)Entfernung\_zum\_Knoten.Entfernung = Entfernung(K1, K2); // Abstand von K1 und K2**

**else Entfernung\_zum\_Knoten.Entfernung = INFINITY; // Entfernung Infinity, da K1 = K2 und wir keine Selbstschleifen zulassen**

**Koordinaten[K1.Index\_in\_Koordinaten].Entfernung\_zu\_anderen\_Knoten.push\_back(Entfernung\_zum\_K noten); // Hinzufügen des Objektes zu dem Vektor von K1**

**vector<ENTFERNUNG\_ZUM\_KNOTEN> spezifische\_Folgeknoten; for (KNOTEN K3 : Koordinaten)**

**{**

**if (K1.Index\_in\_Koordinaten != K2.Index\_in\_Koordinaten && K1.Index\_in\_Koordinaten != K3.Index\_in\_Koordinaten && K2.Index\_in\_Koordinaten != K3.Index\_in\_Koordinaten && gültiger\_Winkel(K1, K2, K3))// möglicher Folgeknoten**

**{**

**Folgeknotens besucht wird**

**}**

**}**

**ENTFERNUNG\_ZUM\_KNOTEN Folgeknoten;**

**Folgeknoten.Index\_des\_Knotens = K3.Index\_in\_Koordinaten; // Index des möglichen Folgeknoten.Entfernung = Entfernung(K2, K3); // Kosten wenn möglicher Folgeknoten spezifische\_Folgeknoten.push\_back(Folgeknoten);**

**sort(spezifische\_Folgeknoten.begin(), spezifische\_Folgeknoten.end(), vergleiche\_Entfernungen); // sotieren alle möglichen Folgeknoten der Entfernung nach**

**Koordinaten[K1.Index\_in\_Koordinaten].Folgeknoten.push\_back(spezifische\_Folgeknoten); //**

**Hinzufügen zu K1**

**}**

**// sortieren des Vektors von K1 anhand der Entfernung sort(Koordinaten[K1.Index\_in\_Koordinaten].Entfernung\_zu\_anderen\_Knoten.begin(),**

**Koordinaten[K1.Index\_in\_Koordinaten].Entfernung\_zu\_anderen\_Knoten.end(),vergleiche\_Entfernungen);**

**}**

**}**

**\* @attention Methode zum generieren eines neuen Suchpfades**

**void generiere\_neuen\_Suchpfad(vector<signed int> &Suchpfad, signed int &Quersumme)**

**{**

**if (Suchpfad[Suchpfad.size()-1] == Quersumme) // alle Suchpfade mit dieser Quersumme wurden berechnet**

**{**

**Quersumme++; // erhöhen der Quersumme um eins Suchpfad[Suchpfad.size()-1] = 0;**

**Suchpfad[0] += Quersumme; // Platzieren der gleichen Zahl Bits auf den ersten Stapel**

**}**

**else**

**{**

**for (signed int i = Suchpfad.size()-2; i >= 0; i--)**

**{**

**if (Suchpfad[i] > 0) // letze Bit das nicht auf dem letzen Stapel liegt**

**{**

**Suchpfad[i]--;**

**Suchpfad[i+1]++; // verschieben um eins nach rechts**

**if (i != Suchpfad.size()-2) // wenn nicht das Bit auf den letzen Stapel gelegt wurde**

**{**

**Suchpfad[i+1] += Suchpfad[Suchpfad.size()-1];// Platzieren aller Bits des letzen Stapels auf den selben Stapel**

**Suchpfad[Suchpfad.size()-1] = 0;**

**}**

**break;**

**}**

**}**

**}**

**}**

**\* @attention Methode zum rotieren von Pfadsegmenten**

**void optimiere\_Pfad(signed long long int &vor\_Schnitt\_1, signed long long int &nach\_Schnitt\_1, signed long long int &vor\_Schnitt\_2, signed long long int &nach\_Schnitt\_2, signed long long int &Schnitt\_1,**

**signed long long int &Schnitt\_2, signed long long int &erster\_Knoten, long double &Länge\_des\_Pfades,**

**bool &verbessert)**

**{**

**// Berechnung ob eine Rotation den Pfad verkürzt long double Unterschied\_Pfadlänge = 0;**

**// die Kanten (vor\_Schnitt\_1, nach\_Schnitt\_1) und (vor\_Schnitt\_2, nach\_Schnitt\_2), würden ersetzt durch (vor\_Schnitt\_1, vor\_Schnitt\_2) und (nach\_Schnitt\_1, nach\_Schnitt\_2)**

**if (Schnitt\_1 > 0)Unterschied\_Pfadlänge += Entfernung(Koordinaten[vor\_Schnitt\_1],**

**Koordinaten[nach\_Schnitt\_1]) - Entfernung(Koordinaten[vor\_Schnitt\_1], Koordinaten[vor\_Schnitt\_2]); if (Schnitt\_2 < Koordinaten.size())Unterschied\_Pfadlänge +=**

**Entfernung(Koordinaten[vor\_Schnitt\_2], Koordinaten[nach\_Schnitt\_2]) - Entfernung(Koordinaten[nach\_Schnitt\_1], Koordinaten[nach\_Schnitt\_2]); if (Unterschied\_Pfadlänge > 0)// eine Rotation verkürzt den Pfad**

**{**

**// überprüfen ob nach der Rotation ein gültiger Pfad besteht**

**if (Schnitt\_1 > 0 && !gültiger\_Winkel(Koordinaten[vor\_Schnitt\_1], Koordinaten[vor\_Schnitt\_2], Koordinaten[finde\_verbundenen\_Knoten(vor\_Schnitt\_2, nach\_Schnitt\_2, 1)]))return;**

**if (Schnitt\_1 > 1 && !gültiger\_Winkel(Koordinaten[finde\_verbundenen\_Knoten(vor\_Schnitt\_1,**

**nach\_Schnitt\_1, 1)], Koordinaten[vor\_Schnitt\_1], Koordinaten[vor\_Schnitt\_2]))return;**

**if (Schnitt\_2 < Koordinaten.size() && ! gültiger\_Winkel(Koordinaten[finde\_verbundenen\_Knoten(nach\_Schnitt\_1, vor\_Schnitt\_1, 1)], Koordinaten[nach\_Schnitt\_1], Koordinaten[nach\_Schnitt\_2]))return;**

**if (Schnitt\_2 < Koordinaten.size() - 1 && !gültiger\_Winkel(Koordinaten[nach\_Schnitt\_1], Koordinaten[nach\_Schnitt\_2], Koordinaten[finde\_verbundenen\_Knoten(nach\_Schnitt\_2, vor\_Schnitt\_2, 1)]))return;**

**// die Rotation führt zu einem verkürzten gültigen Pfad**

**//abspeichern der neuen Verbindungen**

**if (Schnitt\_1 == 0)erster\_Knoten = vor\_Schnitt\_2; // neuer erster Knoten**

**else lege\_neue\_Verbindung\_fest(vor\_Schnitt\_1, nach\_Schnitt\_1, vor\_Schnitt\_2); lege\_neue\_Verbindung\_fest(vor\_Schnitt\_2, nach\_Schnitt\_2, vor\_Schnitt\_1);**

**if (Schnitt\_2 < Koordinaten.size())lege\_neue\_Verbindung\_fest(nach\_Schnitt\_2, vor\_Schnitt\_2, nach\_Schnitt\_1);**

**lege\_neue\_Verbindung\_fest(nach\_Schnitt\_1, vor\_Schnitt\_1, nach\_Schnitt\_2);**

**// updaten der Schnittvariablen**

**signed long long int Zwischenspeicher = nach\_Schnitt\_1; nach\_Schnitt\_1 = vor\_Schnitt\_2;**

**vor\_Schnitt\_2 = Zwischenspeicher;**

**Länge\_des\_Pfades -= Unterschied\_Pfadlänge; //Pfad ist kürzer als zuvor verbessert = true; // Verbesserung gefunden**

**}**

**}**

**\* @attention Methode zum Optimieren eines bereits gefundenen Pfades**

**void Optimierungsverfahren(vector<signed long long int> &gültiger\_Pfad, long double &Länge\_des\_Pfades, const unsigned long long int &Iterationen)**

**{**

**// erzeuge die Datenstruktur**

**signed long long int erster\_Knoten = verbinde\_Knoten(gültiger\_Pfad);**

**for (signed long long int i = 0; i < Iterationen; i++) // Wiederhole Anzahl Iterationen mal**

**{**

**wurde.**

**bool verbessert = false; // Variable zum überprüfen ob in der Iteration eine Verbesserung gefunden**

**// wir wollen die Reihenfolge aller Knoten zwischen Schnitt 1 und 2 umkehren**

**signed long long int vor\_Schnitt\_1 = -1, nach\_Schnitt\_1 = erster\_Knoten; // Begrenzung des ersten**

**Schnitts, setzten ihn zunächst vor den ersten Knoten**

**signed long long int vor\_Schnitt\_2, nach\_Schnitt\_2; // Begrenzung des zweiten Schnitts signed long long int Zwischenspeicher;**

**for (signed long long int Schnitt\_1 = 0; Schnitt\_1 < gültiger\_Pfad.size()-1; Schnitt\_1++)// wir verschieben Schnitt 1 gültiger\_Pfad.size()-2 mal**

**{**

**if (Schnitt\_1 > 0)**

**{**

**// verschieben des ersten Schnitts um 1 nach rechts Zwischenspeicher = nach\_Schnitt\_1;**

**nach\_Schnitt\_1 = finde\_verbundenen\_Knoten(nach\_Schnitt\_1, vor\_Schnitt\_1, 1); vor\_Schnitt\_1 = Zwischenspeicher;**

**}**

**vor\_Schnitt\_2 = finde\_verbundenen\_Knoten(nach\_Schnitt\_1, vor\_Schnitt\_1, 1); // wir starten mit Schnitt 2 zwei Positionen rechts von Schnitt 1**

**// durch das Starten weiter links würden keine neuen Kombinationen von Schnitten entstehen,**

**die Positionen von Schnitt 1 und 2 wären nur getauscht**

**nach\_Schnitt\_2 = finde\_verbundenen\_Knoten(vor\_Schnitt\_2, nach\_Schnitt\_1, 1); // eine Position weiter rechts als vor\_Schnitt\_2**

**for (signed long long int Schnitt\_2 = Schnitt\_1 + 2; Schnitt\_2 <= gültiger\_Pfad.size(); Schnitt\_2+**

**+) // zweiter Schnitt ist, zum Schluss, nach letztem Element**

**{**

**if (Schnitt\_2 > Schnitt\_1+2)**

**{**

**// verschiebe des zweiten Schnitts um 1 nach rechts Zwischenspeicher = nach\_Schnitt\_2;**

**nach\_Schnitt\_2 = finde\_verbundenen\_Knoten(nach\_Schnitt\_2, vor\_Schnitt\_2, 1); vor\_Schnitt\_2 = Zwischenspeicher;**

**}**

**// überprüfe die Kombination der Schnitte**

**optimiere\_Pfad(vor\_Schnitt\_1, nach\_Schnitt\_1, vor\_Schnitt\_2, nach\_Schnitt\_2, Schnitt\_1, Schnitt\_2, erster\_Knoten, Länge\_des\_Pfades, verbessert);**

**}**

**}**

**if (!verbessert)break; // keine der folgenden Iterationen werden keine Verbesserung finden, da sie das gleiche berechnen wie die derzeitige Iteration**

**}**

**// wandle Datenstruktur zu einem Vektor um rekonstruiere\_Pfad\_aus\_Datenstruktur(gültiger\_Pfad, erster\_Knoten);**

**}**

**\* @attention Methode zum erzeugen eines gültigen Pfades aus einem Suchpfad**

**void erzeuge\_gültigen\_Pfad(const vector<signed int> &Suchpfad, signed long long int &Startknoten, const bool &volle\_Optimierung = false, const bool &beschränkte\_Optimierung = false)**

**{**

**for (signed long long int i = 0; i < Koordinaten.size(); i++)Koordinaten[i].bereits\_abgeflogen = false; // Zurücksetzten zuvor berechneter Ergebnisse**

**vector<signed long long int> bisheriger\_Pfad; // wir speichern besuchte Knoten in dem Vektor P (->**

**Dokumentation)**

**bisheriger\_Pfad.reserve(Koordinaten.size());**

**bisheriger\_Pfad.push\_back(Startknoten);// wir starten am Startknoten und speichern diesen ab Koordinaten[bisheriger\_Pfad.back()].bereits\_abgeflogen = true; // und haben ihn somit besucht**

**//wir wählen den nächsten Knoten, der die Suchpfad[0]-beste Wahl darstellt. ( Es wurden noch keine anderen Knoten abgeflogen und zwei Knoten besitzten keinen Abbiegewinkel )**

**long double bisherige\_Länge =**

**Koordinaten[bisheriger\_Pfad.back()].Entfernung\_zu\_anderen\_Knoten[Suchpfad[0]].Entfernung; // abspeichern der Entfernung der ersten beiden Knoten**

**if (Suchpfad[0] > Koordinaten.size()-3)return; // Erstellen eines gültigen Pfades mit dem Suchpfad und Startknoten nicht möglich bisheriger\_Pfad.push\_back(Koordinaten[bisheriger\_Pfad.back()].Entfernung\_zu\_anderen\_Knoten[Suchpf ad[0]].Index\_des\_Knotens); // abspeichern des Index des neuen Knotens**

**Koordinaten[bisheriger\_Pfad.back()].bereits\_abgeflogen = true;// wir sind den Knoten nun abgeflogen for (signed long long int i = 1; i < Koordinaten.size()-1; i++)// wiederhole den Prozess**

**Koordinaten.size()-2**

**{**

**signed long long int gültige\_Knoten = 0; // Variable die beim Iterieren die Zahl der zu dem Zeitpunkt gefundenen Knoten, die zum Pfad hinzugefügt werden können, ohne ihn ungültig zu machen, repräsentiert.**

**for (ENTFERNUNG\_ZUM\_KNOTEN möglicher\_Folgeknoten : Koordinaten[bisheriger\_Pfad[bisheriger\_Pfad.size()-2]].Folgeknoten[bisheriger\_Pfad.back()])//Iterieren durch Liste an möglichen Folgeknoten**

**{**

**if (!**

**Koordinaten[möglicher\_Folgeknoten.Index\_des\_Knotens].bereits\_abgeflogen)gültige\_Knoten++; // Knoten verletzt, falls hinzugefügt zu bisheriger\_Pfad, keine Regeln**

**if (gültige\_Knoten == Suchpfad[i]+1)// wir haben den Knoten gefunden, den wir als nächstes abfliegen wollen**

**{**

**bisherige\_Länge += möglicher\_Folgeknoten.Entfernung; // updaten der Länge des bisherigen\_Pfades**

**bisheriger\_Pfad.push\_back(möglicher\_Folgeknoten.Index\_des\_Knotens); // abspeichern des**

**Index des neuen Knotens**

**Koordinaten[bisheriger\_Pfad.back()].bereits\_abgeflogen = true; // wir sind den Knoten nun**

**abgeflogen**

**}**

**}**

**break;**

**if (gültige\_Knoten != Suchpfad[i]+1)return; // Erstellen eines gültigen Pfades mit diesem Suchpfad und Startknoten nicht möglich**

**}**

**// wir haben nun einen gültigen Pfad passend zu dem Suchpfad vom Startknoten aus erstellt if (volle\_Optimierung)Optimierungsverfahren(bisheriger\_Pfad, bisherige\_Länge,**

**Koordinaten.size());//und optimieren ihn**

**if (bisherige\_Länge < Länge\_des\_kürzesten\_gefundenen\_Pfades)// der Pfad ist kürzer als alle zuvor gefundenen Pfade**

**{**

**if (beschränkte\_Optimierung)Optimierungsverfahren(bisheriger\_Pfad, bisherige\_Länge, Koordinaten.size());// wir optimieren nur beim finden eines neuen küresten Pfades diesen**

**Länge\_des\_kürzesten\_gefundenen\_Pfades = bisherige\_Länge; //Pfad mit kürzerer Länge**

**kürzester\_gefundener\_Pfad.swap(bisheriger\_Pfad); // neurer kürzester Pfad**

**}**

**}**

**\* @attention Methode zum schreiben der Instanz im MPS Format void schreibe\_als\_MPS(const string Textfile)**

**{**

**ofstream Outputfile (Textfile); Outputfile << "NAME FSP\n";**

**// definieren von Beschränkungen Outputfile << "ROWS\n";**

**Outputfile << " N COST\n";// unsere Kostenfunktion, die wir optimieren wollen**

**Outputfile << " E LENGTH\n"; // Beschränkung um die Anzahl der Kanten festzulegen, die abgeflogen werden sollen**

**for (KNOTEN Knoten : Koordinaten) // Beschränkungen für jeden Knoten**

**{**

**Outputfile << " L IN\_" << Knoten.Index\_in\_Koordinaten << '\n'; // wir dürfen zu jedem Knoten über maximal eine Kante gelangen**

**Outputfile << " L OUT\_" << Knoten.Index\_in\_Koordinaten << '\n';// wir dürfen jeden Knoten**

**über maximal eine Kante verlassen**

**}**

**vector<vector<signed long long int>> Kanten; // wir wollen für jede mögliche Kante einen Vektor erstellen, in dem wir abspeichern, in welchen Zeilen sie mitwirkt**

**Kanten.reserve(Koordinaten.size()\*Koordinaten.size()); // es gibt jeder Knoten ist mit jedem anderen**

**Knoten verbunden**

**for (signed long long int i = 0; i < Koordinaten.size()\*Koordinaten.size(); i++)**

**{**

**if (floor(i / Koordinaten.size()) != i % Koordinaten.size())Outputfile << " L M\_" << i << '\n'; // Wir fügen unsere Abgewandelte MTZ-Formulierung für jedes paar verschiedener Knoten ein**

**vector<signed long long int> Vektor; // wir fügen für jede Kante einen leeren Vektor zu Kanten**

**hinzu**

**}**

**Kanten.push\_back(Vektor);**

**signed long long int Anzahl\_ungültiger\_Winkel = 0;**

**for (signed long long int i = 0; i < Koordinaten.size(); i++)**

**{**

**for (signed long long int j = 0; j < Koordinaten.size(); j++)**

**{**

**for (signed long long int k = i + 1; k < Koordinaten.size(); k++)**

**{**

**if (i != j && k != i && k != j && !gültiger\_Winkel(Koordinaten[i], Koordinaten[j], Koordinaten[k])) // überprüfe für jedes Trippel an ungleichen Knoten**

**{**

**// füge eine Beschränkung hinzu, die verhindert, dass Kanten (i, j) und (j, k) bzw. (k, j) und (j, i) abgeflogen werden wenn sie einen ungültigen Winkel bilden**

**Kanten[i\*Koordinaten.size()+j].push\_back(Anzahl\_ungültiger\_Winkel);// dafür vermerken**

**wir in jedem Vektor der Kanten, dass sie später**

**Kanten[j\*Koordinaten.size()+i].push\_back(Anzahl\_ungültiger\_Winkel);// zur Beschränkung hinzugefügt werden**

**Kanten[j\*Koordinaten.size()+k].push\_back(Anzahl\_ungültiger\_Winkel);**

**Kanten[k\*Koordinaten.size()+j].push\_back(Anzahl\_ungültiger\_Winkel); Outputfile << " L A\_" << Anzahl\_ungültiger\_Winkel++ << '\n'; // erstellen der**

**Beschränkung**

**}**

**}**

**}**

**}**

**Outputfile << "COLUMNS\n"; // wir definieren nun alle Variablen**

**for (signed long long int i = 0; i < Koordinaten.size(); i++) // wir definieren nun die Hilfsvariable U der MTZ Formulierung**

**{**

**for (signed long long int j = 0; j < Koordinaten.size(); j++) // für jeden anderen Knoten**

**{**

**if (i == j)continue; // wir vermeiden Selbstschleifen**

**Outputfile << " U\_" << i << " M\_" << i\*Koordinaten.size()+j << " 1\n"; // in diesen Beschränkungen muss das U kleiner sein als das andere U der Beschränkung**

**Outputfile << " U\_" << i << " M\_" << j\*Koordinaten.size()+i << " -1\n"; // in diesen**

**Beschränkungen muss das U größer sein als das andere U der Beschränkung**

**}**

**}**

**for (signed long long int i = 0; i < Koordinaten.size()\*Koordinaten.size(); i++) // wir erzeugen nun für jede Variable eine Kante**

**{**

**if (floor(i / Koordinaten.size()) == i % Koordinaten.size())continue; // wir vermeiden Selbstschleifen Outputfile << " E\_" << i << " LENGTH " << 1; // wenn diese Kante abgeflogen wird, leistet sie**

**einen Beitrag von 1 zu Anzahl aller Kanten**

**Outputfile<< " COST " << Entfernung(Koordinaten[floor(i / Koordinaten.size())], Koordinaten[i % Koordinaten.size()]) << '\n'; // hinzugügen der Länge / Kosten der Kante zur Kostenfunktion**

**Outputfile << " E\_" << i << " M\_" << i << " " << Koordinaten.size() << '\n'; // jede Kante wird ihrer MTZ Formulierung zugeordnet**

**Outputfile << " E\_" << i << " OUT\_" << floor(i / Koordinaten.size()) << " 1\n"; // jede Kante**

**verlässt einen speziellen Knoten**

**Outputfile << " E\_" << i << " IN\_" << i % Koordinaten.size() << " 1\n"; // jede Kante betritt einen speziellen Knoten**

**for (signed long long int j : Kanten[i])**

**{**

**Outputfile << " E\_" << i << " A\_" << j << " 1\n"; // jede Kante wird in die zuvor berechneten Konstellationen an ungültigen Winkeln hinzugefügt**

**}**

**}**

**Outputfile << "RHS\n"; // festlegen der Beschränkungen der Ungleichungen**

**Outputfile << " RHS1 LENGTH " << Koordinaten.size()-1 << '\n'; // es sollen genau Koordinaten.size()-1 Kanten abgeflogen werden**

**for (KNOTEN Knoten : Koordinaten)**

**{**

**Outputfile << " RHS1 IN\_" << Knoten.Index\_in\_Koordinaten << " 1\n"; // jeden Knoten soll maximal eine Kante betreten**

**Outputfile << " RHS1 OUT\_" << Knoten.Index\_in\_Koordinaten << " 1\n"; // jeden Knoten**

**soll maximal eine Kante verlassen**

**}**

**for (unsigned long long int i = 0; i < Anzahl\_ungültiger\_Winkel; i++)**

**{**

**Outputfile << " RHS1 A\_" << i << " 1\n"; // (i, j) und (j, k) bzw. (k, j) und (j, i) bilden einen ungültigen Winkel, daher sind alle Kombinationen der 4 Kanten ungültig und es darf maximal 1 abgeflogen werden**

**}**

**for (unsigned long long int i = 0; i < Koordinaten.size()\*Koordinaten.size(); i++)**

**{**

**if (floor(i / Koordinaten.size()) != i % Koordinaten.size())Outputfile << " RHS1 M\_" << i << " " << Koordinaten.size()-1 << '\n'; //jede Beschränkung der MTZ-Formulierung dar nicht größer als Koordinaten.size()-1 sein**

**}**

**Outputfile << "BOUNDS\n"; // festlegen der Beschränkungen der Variablen**

**for (unsigned long long int i = 0; i < Koordinaten.size(); i++)// für jede Variable U**

**{**

**Outputfile << " UP BND1 U\_" << i << " " << Koordinaten.size() << '\n'; // U muss größer gleich 1 und kleiner gleich Koordinaten.size() sein. Da Koordinaten.size() Variablen existieren und eine**

**Outputfile << " LO BND1 U\_" << i << " 1\n"; // Variable mindestens einen Unterschied von 1**

**zu allen anderen Variablen besitzten muss, werden alle Ganzzahlen in dem Intervall von einer Variable beansprucht**

**}**

**for (unsigned long long int i = 0; i < Koordinaten.size()\*Koordinaten.size(); i++)**

**{**

**if (floor(i / Koordinaten.size()) == i % Koordinaten.size())continue; // es gibt keine Selbstschleifen Outputfile << " BV BND1 E\_" << i << '\n'; // jede Kante ist eine binäre Variable. 0 steht dabei**

**für "wir fliegen sie nicht ab", während 1 für "wir fliegen sie ab" steht**

**}**

**Outputfile << "ENDATA" << '\n'; // Ende des Files**

**}**

**\* @attention Methode zum Auslesen der Ergebnisse des SCIP ILP-Solvers void lese\_gelöstes\_MPS(string Textfile)**

**{**

**ifstream Inputfile (Textfile); kürzester\_gefundener\_Pfad.resize(Koordinaten.size()); // löschen alter Daten string Zeile, \_, Name;**

**getline(Inputfile, Zeile); getline(Inputfile,Zeile);**

**stringstream(Zeile) >> \_ >> \_ >> Länge\_des\_kürzesten\_gefundenen\_Pfades; // auslesen der Länge des Pfades**

**while (Zeile[0] != 'U' && Inputfile.good())getline(Inputfile, Zeile); // Überspringen bis**

**Variablenbelegung der Us, die die Reihenfolge der Knoten beschreibt if (! Inputfile.good())return;**

**long double Index;**

**stringstream(Zeile) >> Name >> Index; // Auslesen des Indexes des Knotens und der Stelle an der dieser abgeflogen wird**

**Name.erase(Name.begin());// Name hat U\_ am Anfang, wir wollen nur die Zahl im Anschluss Name.erase(Name.begin());**

**kürzester\_gefundener\_Pfad[round(Index)-1] = stoll(Name);// runde Index zwecks kleiner Schwankungen, konvertiere Name zu Index**

**for (signed long long int i = 1; i < Koordinaten.size(); i++) // wiederhole dies für die übrigen Knoten**

**{**

**getline(Inputfile, Zeile); stringstream(Zeile) >> Name >> Index; Name.erase(Name.begin()); Name.erase(Name.begin());**

**kürzester\_gefundener\_Pfad[round(Index)-1] = stoll(Name);**

**}**

**}**

**\* @attention Methode zum finden eines möglichst kurzen Pfades, der den Kriterien der 1. Aufgabe des**

**41.BWINF entspricht.**

**void approximiere(const signed long long int &Iterationen, const bool &volle\_Optimierung = false, const bool &beschränkte\_Optimierung = false)**

**{**

**if (Koordinaten.size() < 2)throw invalid\_argument("Es müssen mindestens zwei Knoten definiert sein um einen Pfad zu berechnen!");**

**Vorberechnung(); // erleichtert später das finden von möglichen Folgeknoten (-> Dokumentation)**

**// löschen zuvor berechneter Daten kürzester\_gefundener\_Pfad.clear(); Länge\_des\_kürzesten\_gefundenen\_Pfades = INFINITY;**

**// festlegung des ersten Suchpfades, wir treffen in diesem Fall lokal immer die optimale Wahl vector<signed int> Suchpfad;**

**for (unsigned long long int i = 0; i < Koordinaten.size()-1; i++)Suchpfad.push\_back(0); signed int Quersumme = 0;**

**for (signed long long int i = 0; i < Iterationen; i++) // generiere Suchpfade und suche für jeden nach gültigen Pfaden**

**{**

**for (signed long long int j = 0; j < Koordinaten.size(); j++)erzeuge\_gültigen\_Pfad(Suchpfad, j, volle\_Optimierung, beschränkte\_Optimierung); // Suche von jedem Knoten aus nach neuem Suchpfad**

**generiere\_neuen\_Suchpfad(Suchpfad, Quersumme);// generiere neuen Suchpfad**

**}**

**}**

1. Abbildung nach https://bwinf.de/fileadmin/bundeswettbewerb/42/aufgaben422.pdf [↑](#footnote-ref-1)
2. https://de.wikipedia.org/wiki/Erwartungswert [↑](#footnote-ref-2)