Aufgabe 1: Weniger krumme Touren

Teilnahme-ID: 64650

Team-Name: Team

Bearbeiter dieser Aufgabe:
Arne de Borman

17. April 2023

Inhaltsverzeichnis

1 Lösungsidee

Ein Pfad ist die Sequenz von Punkten, die der Pilot durchlaufen soll. Der Startpunkt ist der erste Punkt dieser Sequenz. Aus der Aufgabenstellung wird geschlossen, dass der Pilot nicht am Ende wieder am Startpunkt ankommen soll. Alle möglichen Pfade auszuprobieren ist keine Möglichkeit, da dies eine Laufzeit von $n!$ erfordern würde, wobei n die Anzahl der Pfade ist, was bei hoher Punktezahl gigantische Laufzeiten zur folge hätte. Deshalb wird eine Heuristik verwendet und zwar eine modifizierte Version des nearest-neighbour Algorithmus, um trotzdem einigermaßen kurze Pfade zu erhalten. Der Pfad beginnt mit einem Punkt und wird Schritt für Schritt um den nächsten Punkt erweitert. Dabei wird darauf geachtet, dass der Winkel zwischen den benachbarten Punkten niemals größer als 90° ist. Wenn kein weiterer unbesetzter Punkt in der korrekten Richtung gefunden werden kann, wird ein Schritt zurückgegangen, um einen anderen Weg zu suchen. Wenn ein vollständiger Pfad gefunden wurde, wird er gespeichert, und der Algorithmus wird fortgesetzt, um einen kürzeren Weg zu finden. Nach einer bestimmten Anzahl von Schritten wird der beste Weg, der bis dahin gefunden wurde, zurückgegeben. Darüber hinaus wird der Branch-and-Bound Algorithmus eingesetzt, um mit genügend langer Laufzeit den kürzesten Pfad zu finden. Dabei wird versucht, den Suchraum durch Abschätzungen der minimalen Weglänge zu reduzieren (untere Grenze). Wenn an einem Punkt die untere Grenze höher ist als die Länge eines bereits gefundenen Weges, wird dieser Pfad abgebrochen. Durch das benutzen dieser Technik kann trotzdem bei höherer Punktezahl aufgrund zu hoher Laufzeit nicht der kürzeste Pfad gefunden werden, weshalb sie je nach Situation als exaktes Lösungsverfahren oder Heuristik fungieren kann, indem der bisher beste Pfad gespeichert wird. Zur Berechnung der unteren Grenze benutze ich eine modifizierte Version des Algorithmus den ich hier gefunden habe: `https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/`. Den Algorithmus muss modifiziert werden, da er davon ausgeht, dass der Pfad wieder am Startpunkt endet. Eine genauere Erklärung unter „Umsetzung/Branch and Bound“.

1.1 Erweiterung: Linienfindung

Um schneller zu einer Lösung zu gelangen, werden Linien erkannt und ihre Punkte zu Abkürzungen zusammengefasst, die dann beim Finden des Weges genutzt werden. Um die Linien zu finden, werden Punkte in einer Punktmenge identifiziert, die jeweils nur zwei andere Punkte haben die sehr dicht liegen und zusammen eine Linie bilden. Diese Punkte und die beiden dichtesten Nachbarn bilden dann ein Liniensegment aus drei Punkten. Anschließend werden mehrere Liniensegmente zu vollständigen Linien zusammengefasst, indem mit einem Segment begonnen wird und dann immer wieder Segmente gesucht und angefügt werden, die an die Enden dieses Segmentes passen, bis keine neuen Segmente gefunden werden können. Eine genauere Erklärung unter „Umsetzung/Linienfindung“. Durch das Benutzten der Linien als Abkürzung kann viel Zeit gespart werden, da beim Zurückgehen über die Linie mehrere Punkte übersprungen werden, für die sonst separat jeweils jeder mögliche andere Pfad hätte ausprobiert werden müssen. Außerdem reduzieren die Linien die Anzahl der Punkte die betrachtet werden müssen.

2 Umsetzung

Die Lösungsidee wird in der Programmiersprache Rust umgesetzt, um möglichst viele Pfade in kürzester Zeit erforschen zu können.

2.1 Werte vorberechnen

Es werden einige Werte vorberechnet, um Rechenzeit zu sparen. Dazu gehören die Abstände zwischen den Punkten, oder auch ob drei Punkte aus der Liste einen Winkel bilden der kleiner als 90° ist. Um möglichst schnell die nächsten Punkte eines Punkts zu finden wird auch eine Liste erzeugt, die für jeden Punkt die anderen Punkte sowie ihren Abstand zum Punkt nach Abstand sortiert speichert. Die Punkte werden nicht mithilfe ihrer Koordinaten gespeichert, sondern nur noch mithilfe ihrer Position in der Punkte-Liste.

2.1.1 Abstände

Die Abstände werden in dem CostList-Objekt gespeichtert. Um den Abstand für zwei Punkte zu bestimmen, wird die get Methode verwendet.
Beispiel:

```
//Abstände vorberechnen
let cost_list = CostList::new(&points);
//Abstand zwischen dem dritten und fünften Punkt erhalten
let dist: f64 = cost_list.get(0,5);
```

2.1.2 Winkel

Ob die Winkel in Ordnung sind, wird in dem AngleOkList-Objekt gespeichert. Um festzustellen, ob drei Punkte den richtigen Winkel haben, wird die is_ok Methode verwendet.
Beispiel:

```
//Vorberechnen
let angle_list = AngleOkList::new(&points);

//Ob der Winkel über p1 von p0, p1 und p2 in Ordnung ist
let angle_ok: bool = angle_list.is_ok(p0,p1,p2);
```

2.1.3 Dichteste Punkte berechnen

Die funktion find_shortest_paths berechnet die dichtesten Punkte für jeden Punkt. Sie nimmt ein CostList Objekt und eine Liste von Linien als Argument. Die Linien werden verwendet, um zu verhindern, dass ein Punk als nächster ausgewählt, der auf einer Linie liegt.
Beispiel:

```
let short_paths: Vec<Vec<(usize, f64)>> = find_shortest_paths(&costs, &lines);
//liest den nächsten Punkt zu pt, sowie den Abstand zu ihm aus
let (nearest, dist) = short_paths[pt][0];
//liest den dritt-nächsten Punkt zu pt aus
let (thrd_nearest, thrd_dist) = short_paths[pt][2];
```

Die Werte werden in der PathFinder-Klasse gespeichert. Diese Klasse ist für das Finden des Wegs zuständig. Da diese Klasse für viele Sachen zuständig ist, ist die Anzahl der Member-Variablen relativ groß. Ich habe jedoch festgestellt, dass das Trennen der Klasse in mehrere Verschiedene Einheiten zu mehr Komplexität führt, da verschiedene Werte oft an vielen Stellen benötigt werden. Deshalb hat die Klasse jetzt etwas mehr Klassenvariablen als ideal.

2.2 Erkundung der Pfade

Zuerst wird der Punkt, der den größten zweiten Abstand zu anderen Punkten hat als Startpunkt bestimmt. Dadurch muss dieser Abstand nicht im Weg verwendet werden. Es kann aber auch ein Startpunkt durch den Benutzer festgelegt werden (siehe „Startpunkt suchen“ unter „Beispiele“). Das Erkunden der Pfade läuft nicht rekursiv ab, da das Aufrufen von Funktionen relativ lange dauert, sondern in einer

Schleife. Um das zu ermöglichen wird in einer Liste für jeden Punkt im Weg gespeichert, der wievieldichteste Punkt als nächstes benutzt werden soll. Dieser Wert wird Index genannt. Anfangs ist der Index für jeden Punkt Null, sodass immer der dichteste Punkt als Nächstes benutzt wird.

Um den nächsten dichtesten Punkt zu finden wird zuerst der Index für den aktuellen Punkt geladen. Dann wird mithilfe des Indices der nächste Punkt aus der short_paths_liste ausgelesen.

```
//der wivielt-nähste punkt als nächstes genommen wird
let curr_idx = self.idxs.get_mut(level).unwrap();
//die liste der nähsten punkte
let next_pts = &self.nearest_pts[curr_pt];
/* ... */
let (next_pt, cost) = next_pts[*curr_idx];
```

Passt dieser nicht, zum Beispiel weil er schon Teil des Pfads ist oder weil der Winkel nicht stimmt, wird der Index um eins erhöht und der Vorgang wiederholt. Wenn schon alle möglichen Indices ausprobiert wurden, wird einen Schritt zurückgegangen. Dies geschieht in der PathFinder::find_next_nearest Methode. Falls kein nächster Punkt gefunden wurde, wird um einen Schritt zurückgegangen.

Bevor um einen Punkt weitergegangen wird, wird der Index um eins erhöht, sodass beim Rückgang auf den Punkt nicht wieder derselbe Punkt als Nächstes benutzt wird, sondern der nächst-dichteste Punkt. Außerdem wird der Liste für den neuen Punkt eine Null hinzugefügt.

Um einen Schritt zurückzugehen wird die backtrack-Methode verwendet. Hier wird der letzte Punkt aus der Pfad-Liste sowie der entsprechende Index aus der Index-Liste entfernt.

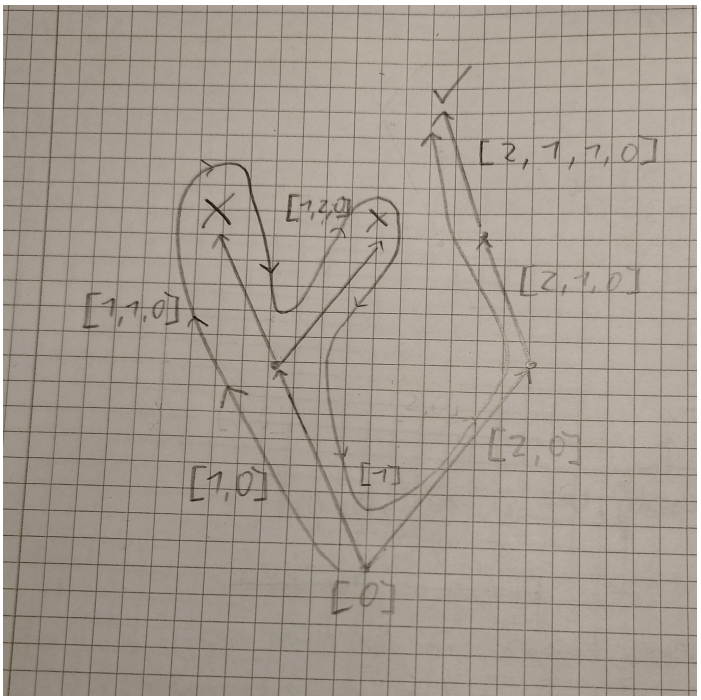

Abbildung 1: Ein Beispiel für die Index Werte während der Suche.

Wenn nach einer bestimmten Schrittanzahl kein Weg gefunden wurde, wird der ganze Prozess mit einem neuen Startpunkt abermals begonnen. Dadurch kann ein Weg gefunden werden, selbst wenn nicht jeder Startpunkt einen Weg ermöglicht.

2.3 Branch and Bound

Es wird für jeden Schritt eine untere Grenze berechnet, das heißt wie klein der vollständige Pfad mindestens ist. Wenn diese untere Grenze eines Pfades die Länge eines bereits gefundenen Pfades überschreitet, so wird wieder um einen Schritt zurückgegangen, da der vollständige Pfad niemals kürzer als der bereits gefundene Weg sein kann. Die Grundidee für die Berechnung stammt von hier: `https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/` und wird gleich Im

folgenden Abschnitt erklärt. Sie geht davon aus, das der Pfad mit dem Startpunkt abschließen soll, weshalb sie verändert werden muss.

2.3.1 Grundidee

Im vollständigen Pfad muss es für jeden Punkt zwei Verbindungen, eine zum vorherigen und eine zum nachfolgenden Punkt geben. Um die theoretisch kleinstmögliche Weglänge zu finden, werden nun für alle Punkte die zwei dichtesten Punkte gefunden und ihre Abstände gespeichert. Diese Abstände stellen jetzt die idealen Verbindungen für diesen Punkt dar. Addiert man nun alle diese Abstände und Teilt das Ergebnis durch zwei (sonst werden Verbindungen doppelt gezählt), so erhält man die kleinstmögliche Weglänge vom Startpunkt aus. Wenn man nun einen Punkt weitergeht, muss dieser Wert wieder erneuert werden. Da möglicherweise nicht die kürzeste Verbindung für den neuen und vorherigen Punkt verwendet wurde, werden die vorher angenommenen kürzesten Verbindungen für diese Punkte wieder vom Wert abgezogen, nachdem die Werte wieder durch zwei geteilt wurden. Für den vorherigen Punkt ist das die längere, abgehende Verbindung und für den neuen Punkt die kürzere zugehende Verbindung. Ob die abgehende oder Zugehende Verbindung die kürzere ist, ist egal, es muss nur immer das gleiche sein. Danach wird die tatsächliche neue Strecke vom vorherigen zum neuen Punkt addiert. Dadurch erhält man eine neue kürzere Weglänge.

2.3.2 Modifikation

Bei der Grundidee wird davon ausgegangen, dass der Pfad wieder am Startpunkt endet. Das ist jedoch bei dieser Aufgabe nicht der Fall, denn hier ist bei einem vollständigem Pfad der Startpunkt vom Endpunkt unterschiedlich. Deshalb muss man die Grundidee etwas verändern.

Für den Startpunkt wird nur noch die kürzere Verbindung berücksichtigt, da keine Verbindung zum Startpunkt geht, sondern nur eine Verbindung weg. Es wird auch ein Endpunkt bestimmt, bei dem auch nur die kürzere Verbindung gezählt wird, da hier nur eine Verbindung hingeht und keine weg. Da die kürzeste mögliche Strecke gesucht wird, wird als Endpunkt der Punkt ausgewählt, dessen zweitkürzeste Verbindung die größte aller Punkte ist, da diese Verbindung im Weg nicht berücksichtigt werden muss.

Wenn nun ein Punkt zum Pfad hinzugefügt wird und der vorherige Punkt der vorher ausgewählte Endpunkt war, so wird aus den verbleibenden freien Punkten nach demselben Prinzip ein neuer Endpunkt ausgewählt. Dann wird die zweite Strecke des alten Endpunktes zum Wert hinzugefügt und die zweite Strecke des neuen Endpunktes entfernt. Der Rest der Berechnung der neuen Kosten ist dasselbe wie bei der Grundidee abgesehen davon, dass für den Startpunkt die kürzere Strecke als abgehende Strecke benutzt wird.

```rust
pub fn update_costs(&mut self, curr_pt: usize, next_pt: usize) -> f64 {
    let mut cost = 0.0;
    //Der aktuelle Endpunkt
    let curr_last = self.max_second_costs[self.curr_end_idx];
    //falls der aktuelle Punkt der Endpunkt ist
    //wird nach einem neuen Endpunkt gesucht
    if curr_pt == curr_last.0 {
        //sucht den nächsten freien Endpunkt
        let new_last = loop {
            self.curr_end_idx += 1; //geht zum nächsten Endpunkt
            //in max_second_costs wurden mögliche Endpunkte
            //sortiert nach tauglichkeit gespeichert
            let new_last = self.max_second_costs[self.curr_end_idx];
            if self.free_pts[new_last.0] { //falls der Endpunkt frei ist wird er genommen
                break new_last;
            }
        };
        let prev_last_cost = curr_last.1;
        let new_last_cost = new_last.1;
        cost += (prev_last_cost - new_last_cost) / 2.0;
    }
    //falls der vorherige punkt der Startpunkt ist
    let min_prev = if self.path.len() == 1 {
```

```
            //wird für den vorherigen Punkt die kürzere Strecke abgezogen
            self.min_costs[curr_pt].0
        } else {
            //sonst die längere
            self.min_costs[curr_pt].1
        };
        //Die kürzere Strecke zum nächsten Punkt:
        let min_next = self.min_costs[next_pt].0;
        cost -= (min_prev + min_next) / 2.0;
        cost
    }
```

Wenn auf einen Punkt wieder zurückgegangen wird, so muss auch die untere Grenze zurückgesetzt werden. Deshalb werden in jedem Schritt gespeichert, der wievielte Endpunkt gerade angenommen wird und was die aktuelle untere Grenze ist. Wenn um einen Punkt zurückgegangen wird, kann der Wert dann wieder zurückgesetzt werden.

2.4 Linienfindung

Das Finden der Linien wird in der find_lines Funktion implementiert. Wie in der Lösungsidee beschrieben werden zuerst Liniensegmente gesucht. Um ein Liniensegment zu finden werden alle Punkte in einer Schleife geprüft. Für jeden Punkt finde werden zuerst die drei dichtesten Punkte und ihren Abstand zu ihm betrachtet. Der Punkt als Mittelpunkt und die Zwei dichtesten Punkte werden unter den folgenden Voraussetzungen als Liniensegmente erkannt:

1. Der zweitdichteste Punkt ist maximal 1,5 mal so weit vom Mittelpunkt wie der dichteste Punkt entfernt
2. Der drittdichteste Punkt ist mehr als 1,5 mal weiter vom Mittelpunkt entfernt als der zweitdichteste Punkt
3. Der Winkel der drei Punkte über den Mittelpunkt ist kleiner als 90°

Die Liniensegmente werden in einer HashMap gespeichert (key ist der Mittelpunkt), um das mögliche Liniensegment für einen Punkt schnell wiederfinden zu können. Diese Liniensegmente werden hinterher zu ganzen Linien zusammengesetzt. Dafür wird mit einem zufälligen Segment begonnen. Die Enden des Segments werden nun erweitert, indem mithilfe der HashMap immer wieder die Segmente der Endpunkte gefunden werden. Der Punkt des Segments, der nicht der vorherige Endpunkt oder der Punkt vor dem Endpunkt ist, ist dann der neue Endpunkt. Das Segment wird dabei aus der HashMap entfernt. Wenn kein weiteres Segment gefunden werden kann ist die Linie an diesem Ende fertig.

```
    // Erweiterung der Linie in eine Richtung
    while let Some((a, b)) = line_segments.remove(&end) {
        //Finde den Punkt, der noch nicht Teil der Linie ist
        let new_end = if a == p_pt { b } else { a };
        if used_pts[new_end] {
            break;
        }
        line.push_back(new_end);
        used_pts[new_end] = true;
        p_pt = end;
        end = new_end;
    }
```

Nachdem die Linie in beide Richtungen erweitert wurde wird sie gespeichert, wenn sie eine bestimmte Mindestlänge hat, sodass nur tatsächliche Linien erkannt werden. Danach wird der Vorgang mit den restlichen Liniensegmenten wiederholt bis keine mehr übrig sind. Unter „Beispiele“ sind die Linien, die für einige BwInf-Beispieleingaben gefunden wurden, dargestellt.

Um die Linien beim Finden des Weges verwenden zu können, wird für jeden Punkt gespeichert, ob es einen möglichen Sprung über eine Linie gibt.

```rust
let available_skips: Vec<Option<Skip>> = get_skips(&costs, &lines);
```

Informationen über einzelne Sprünge werden in der Skip-Klasse gespeichert

```rust
pub struct Skip {
    end: usize, //Ende der Linie
    second_pt: usize, //Der Punkt nach dem Anfang der Linie
    penultimate_pt: usize, //Der Punkt vor dem Ende der Linie
    cost: f64, //Wie lange die Linie ist
}
```

Abgesehen von den Endpunkten werden die Punkte der Linien aus der dichteste-Punkte Liste (nearest_pts) entfernt, sodass nie ein Linien-Punkt als nächstes gewählt wird.
Wenn der aktuelle Punkt ein Linien-Endpunkt ist, so gibt es mehrere Möglichkeiten für den weiteren Verlauf des Pfades:

- Wenn im letzten Schritt schon gesprungen wurde, dann befindet man sich am Ende des Sprungs und es wird ganz normal nach dem nächsten Punkt gesucht.
- Wenn auf den Punkt zurückgegangen wurde oder der Winkel zum nächsten Punkt der Linie nicht passt, wird wieder zurückgegangen, da es von diesem Punkt aus sonst nur die Möglichkeit gibt über die Linie zu gehen
- Sonst wird die Linie benutzt

Dies wird in der PathFinder::find_nexts Methode implementiert. Wenn Abkürzungen benutzt werden, wird der zweite Punkt der Linie zum Berechnen des Winkels benutzt. Wenn auf den letzten Punkt gesprungen wurde wird der vorletzte Punkt der Linie benutzt.

3 Beispiele

Das Programm wird mit dem „$ cargo run –release“- Befehl ausgeführt. Man muss sich dabei im Ordner für diese Aufgabe befinden. Hier wird davon ausgegangen, dass Rust installiert ist. Die „–release“- Flag ermöglicht Optimisierungen die erlauben, dass das Programm in einer akzeptierbaren Zeit läuft. Alternativ kann auch die vorkompilierte Datei für die jeweilige Plattform verwendet werden, zum Beispiel „$ travel-windows.exe“. Mögliche Argumente können mithilfe der „–help“ Flag angezeigt werden.

```
~/bwinf/travel $ cargo run -- --help
Ein Programm das einen Weg durch mehrere Punkte finden kann ohne mehr als 90° abzubiegen

Usage: travel [OPTIONS] <PATH>

Arguments:
  <PATH>  Die Punkte-Datei

Options:
      --show-points              Ob die Punkte angezeigt werden sollen
      --show-lines               Ob die linien angezeigt werden sollen
      --dont-use-lines           Ob Linen beim finden des Pfads benutzt werden sollen
      --no-calc                  Ob der Pfad berechnet werden soll
      --stop-on-found            Ob direkt abgebrochen werden soll, wenn ein Pfad gefunden wurde
      --dont-check-angle         Ob der Winkel überprüft werden soll
      --line-min <LINE_MIN>      Die Linien-Mindestlänge [default: 5]
      --img-size <IMG_SIZE>      Die Bildgröße [default: 4000]
      --pt-size <PT_SIZE>        Punktgröße im Bild [default: 15]
      --line-width <LINE_WIDTH>  Linienbreite [default: 3]
      --img-path <IMG_PATH>      Wo das Bild gespeichert werden soll (mehrere Dateiformate möglich)
      --out-path <OUT_PATH>      Wo der Pfad gespeichert werden soll [default: out.txt]
      --max-iter <MAX_ITER>      Wie lange der Pfad berechnet werden soll [default: 10000000]
      --start-pt-n <START_PT_N>  Der wievielte Startpunkt genutzt werden soll, optional
      --search-start             Ob nach dem Startpunkt gesucht werden soll
```

```
  -h, --help                    Print help
  -V, --version                 Print version
```

Das Programm erzeugt ein Bild des Pfads sowie eine Textdatei mit den Punkten. Ich werde hier nur die Programmausgabe und Bilder einfügen, die Textdateien befinden sich Ordner für diese Aufgabe mit dem Namen „out_n.txt“, wobei n die Nummer der Beispieleingabe ist.

Bei der fünften Beispieleingabe kann mit dem ersten Startpunkt kein Weg gefunden werden, deshalb wird automatisch der nächste genommen.

```
~/bwinf/travel $ cargo run --release data/wenigerkrumm5.txt
Punkte aus Datei geladen
Suche Weg...
Linien finden...
kein Pfad gefunden, nächster Startpunkt
reset mit Startpunkt: 16
neue untere Grenze: 3506.9592304150924, Abgeschnittene Pfade: 0/67 = 0.000%
max_iter erreicht
tiefstes Level auf das zurückgegangen wurde: 21/60, Abgeschnittene Pfade: 3325907/10000001 = 33.259%
Weg gefunden, Länge: 3506.9592304150938
Überprüfe Winkel... alle Winkel in Ordnung
finden des Wegs hat 931.570084ms gedauert
Bild gespeichert
```

Wenn man die maximale Schrittanzahl erhöht (100 Mio), wird ein kürzerer Weg gefunden

```
~/bwinf/travel $ cargo run --release data/wenigerkrumm5.txt --max-iter 100000000
Punkte aus Datei geladen
Suche Weg...
Linien finden...
kein Pfad gefunden, nächster Startpunkt
reset mit Startpunkt: 16
neue untere Grenze: 3506.9592304150924, Abgeschnittene Pfade: 0/67 = 0.000%
neue untere Grenze: 3498.5132502510955, Abgeschnittene Pfade: 30945242/93078926 = 33.246%
neue untere Grenze: 3489.306019606827, Abgeschnittene Pfade: 31201451/93903507 = 33.227%
i: 100000000, Abgeschnittene Pfade: 33.376%
max_iter erreicht
tiefstes Level auf das zurückgegangen wurde: 21/60, Abgeschnittene Pfade: 33375833/100000001 = 33.376
Weg gefunden, Länge: 3489.3060196068272
Überprüfe Winkel... alle Winkel in Ordnung
finden des Wegs hat 8.201135065s gedauert
Bild gespeichert
```

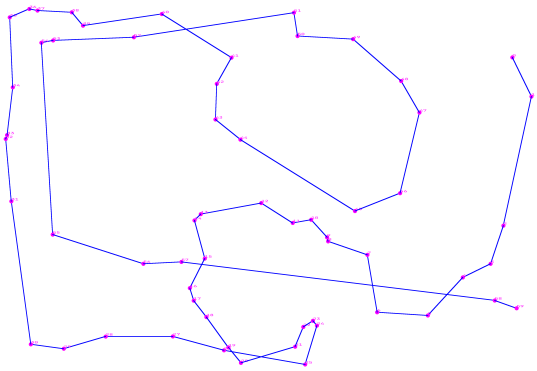

Abbildung 2: Der kürzere Weg von wenigerkrumm5.txt

3.1 Linien

Mithilfe der „–show-lines“ - Flag können Linien angezeigt werden, die in den Punkten gefunden wurden. Mithilfe der „–line-min“ - Flag kann die Linien-Mindestlänge angegeben werden (siehe „Umsetzung“). Mithilfe der „–show-points“-Flag können außerdem die Punkte angezeigt werden, die nicht Teil der Linien sind. Beispiel:

```
~/bwinf/travel $ cargo run --release data/wenigerkrumm3.txt --show-lines --show-points --no-calc
Punkte aus Datei geladen
Linien finden...
line: Line { ends: [29, 69], pts: [29, 55, 62, 77, 69] }
line: Line { ends: [93, 6], pts: [93, 98, 101, 7, 89, 80, 30, 66, 104, 90, 18, 106, 6] }
line: Line { ends: [42, 39], pts: [42, 14, 110, 1, 97, 53, 46, 58, 59, 108, 107, 22, 39] }
line: Line { ends: [9, 41], pts: [9, 65, 115, 75, 41] }
line: Line { ends: [50, 78], pts: [50, 87, 56, 24, 118, 57, 34, 0, 3, 85, 82, 114, 78] }
line: Line { ends: [99, 68], pts: [99, 119, 12, 40, 68] }
line: Line { ends: [63, 105], pts: [63, 2, 33, 37, 45, 96, 117, 49, 74, 25, 20, 109, 105] }
line: Line { ends: [28, 84], pts: [28, 112, 116, 88, 84] }
Bild gespeichert
```

Resultierendes Bild, sowie die erzeugten Bilder mit anderen Parametern:

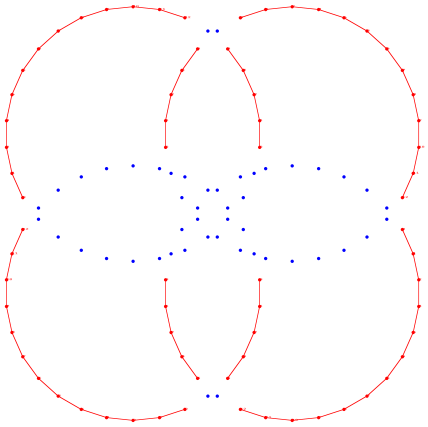

Abbildung 3: Gefundene Linien in der dritten Beispieleingabe in Rot, Punkte in Blau.

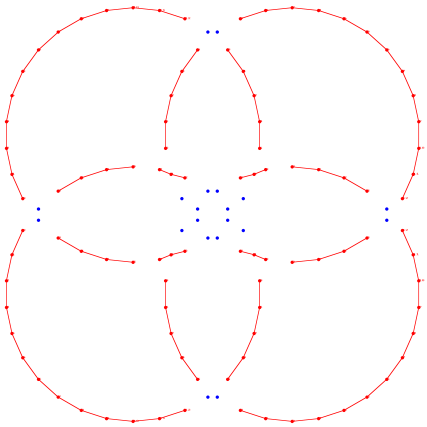

Abbildung 4: Gefundene Linien in der dritten Beispieleingabe, Mindestänge auf Null heruntergesetzt. Hier werden Linien erkannt, die möglicherweise nicht beim Finden des Pfads helfen, weshalb die Mindestlänge normalerweise fünf und nicht Null ist.

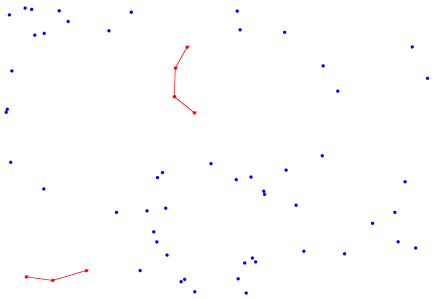

Abbildung 5: Gefundene Linien in der fünften Beispieleingabe, Mindestänge auf Null heruntergesetzt. Auch werden Linien erkannt, die möglicherweise nicht beim Finden des Pfads helfen.

Mithilfe der „–dont-use-lines“ Flag kann angegeben werden, dass Linien nicht benutzt werden sollen. Wenn man das mit der „–stop-on-found“ Flag (es wird nicht weitergesucht, wenn ein Pfad gefunden wurde) kombiniert, kann man sehen wie viel schneller der Pfad gefunden wird.

Suche ohne Linien, gekürzte Ausgabe (dritte Beispieleingabe):

```
~/bwinf/travel $ cargo run --release data/wenigerkrumm3.txt --stop-on-found --dont-use-lines
...
Pfad gefunden
        Dauer (ohne Vorberechnungen): 97.402µs
        Dauer (mit Vorberechnungen): 98.836133ms
        Länge: 2049.7847565495404
...
```

Suche mit Linien:

```
~/bwinf/travel $ cargo run --release data/wenigerkrumm3.txt --stop-on-found
...
Pfad gefunden
        Dauer (ohne Vorberechnungen): 53.801µs
        Dauer (mit Vorberechnungen): 80.006126ms
        Länge: 2028.7093959981682
...
```

Wie man hier sehen kann ist die Zeitdauer ohne Vorberechnungen etwas kürzer. Dieser Unterschied macht aber nicht viel aus, wenn man die erheblich größere Dauer mit Vorberechnungen einbezieht. Die Dauern schwanken auch erheblich. Der Vorteil der Linien zeigt sich erst, wenn man länger nach einem Weg sucht.

Tabelle 1: Suchen mit mehr Iterationen:

Iterationen:	Sofortabbruch	**10 Mio**	**100 Mio**	**1 Mrd**	**10 Mrd**
Länge (mit Linien):	2028	1947	1935	1929	1929
Länge (ohne Linien):	2049	2006	2006	1953	1947

Hier kann man sehen, dass die Linien es ermöglichen bessere Ergebnisse zu erhalten. Die Suche mit Linien kann jedoch etwas länger dauern (ca +30%). Auch kann man sehen, dass manchmal Über viele Iterationen hinweg keine bessere Lösung gefunden wird.

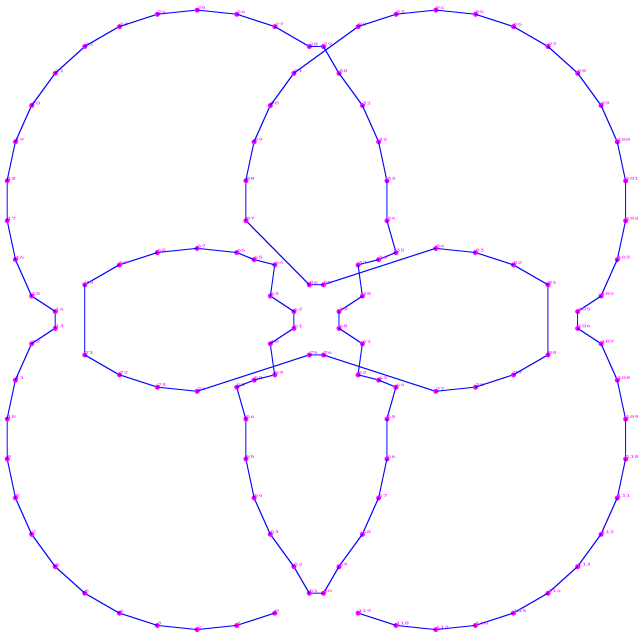

Abbildung 6: Gefundener Weg für die dritte Beispieleingabe, suche mit Linien und 10Mrd Iterationen

3.2 Startpunkt suchen / Beste Lösungen

Mithilfe der „–search-start“ Flag kann nach dem besten Startpunkt gesucht werden. Dafür wird die mehrmals mit verschiedenen Startpunkten nach dem Weg gesucht und dann die Nummer des Startpunktes mit dem kürzestem Weg zurückgegeben. Das heißt nicht, dass der gefundene Startpunkt der Startpunkt ist der tatsächlich den optimalen Weg ergeben würde, es ist aber zumindest ein Startpunkt mit dem es leichter ist einen kurzen Weg zu erreichen. Da der Startpunkt zuerst der Punkt mit dem größten zweiten Abstand ist und danach der mit dem zweitgrößten Abstand und so fort, gibt die Startpunktnummer an, der wievielte Startpunkt in dieser Folge der beste ist. Diese Startpunktnummer kann mithilfe der „–start-pt-n“ Flag wiederverwendet werden. In Tabelle 2 sind die besten Startpunkte für die Beispieleingaben sowie die Längen der Wege zu sehen. Die Punkte und Bilder der jeweiligen Wege sind im selben Ordner wie diese Dokumentation mit den Namen „out_n.txt“ und „out_n.png“ gespeichert, wobei n die Nummer des Beispiels ist. Möglicherweise könnten diese Wege noch weiter verbessert werden, indem ein bessere

Beispiel nummer	Startpunktnummer	Startpunkt	Länge mit 10 Mrd. Iter	Optimal?
1	0	20	847.43	Ja
2	0	50	2183.66	Ja
3	26	28	1891.19	Nein
4	8	21	1205.07	Ja
5	1	16	3489.31	Nein
6	64	72	4073.30	Nein
7	11	26	4840.57	Nein

Tabelle 2: Optimale Startpunktnummer der einzelnen Beispiele. Optimal gibt an, ob alle Pfade vom Startpunkt aus ausprobiert wurden. Länge auf zwei Nachkommastellen gerundet

Berechnung für die untere Grenze der Pfade benutzt wird (Branch and Bound).

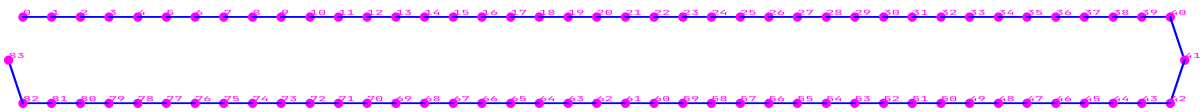

Abbildung 7: Der beste Weg für wenigerkrumm1.txt

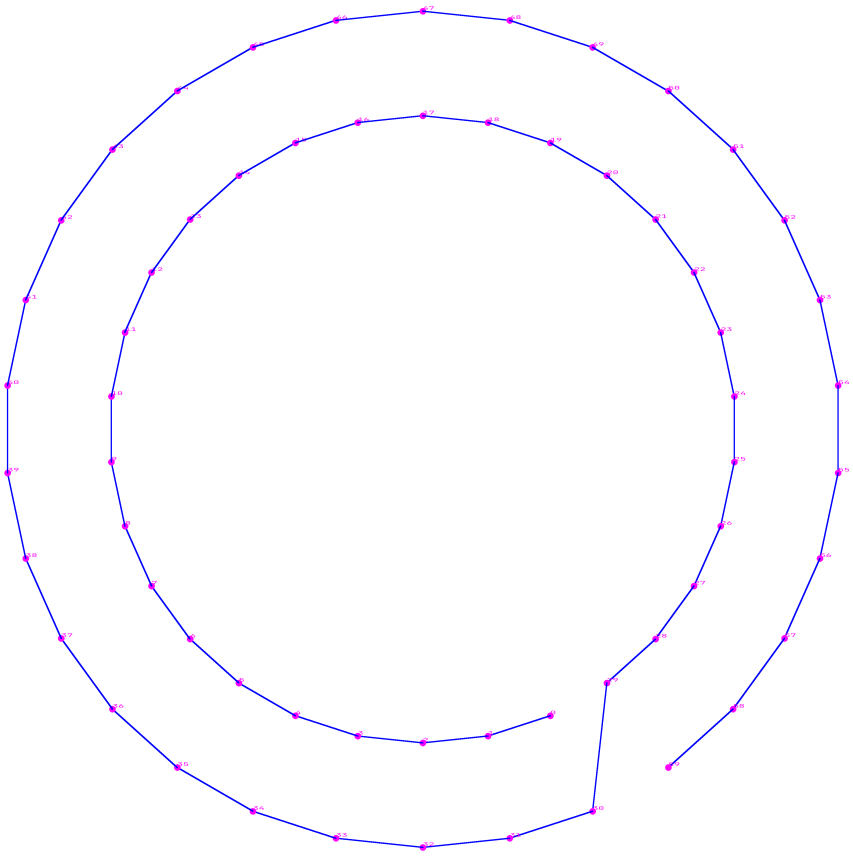

Abbildung 8: Der beste Weg für wenigerkrumm2.txt

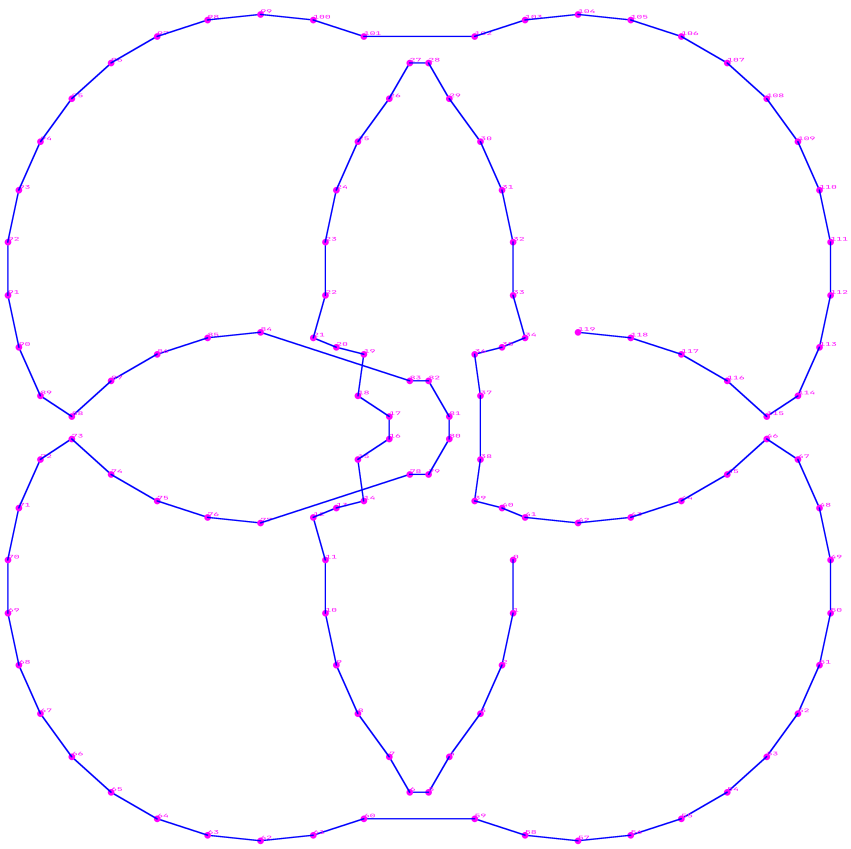

Abbildung 9: Der beste gefundene Weg für wenigerkrumm3.txt

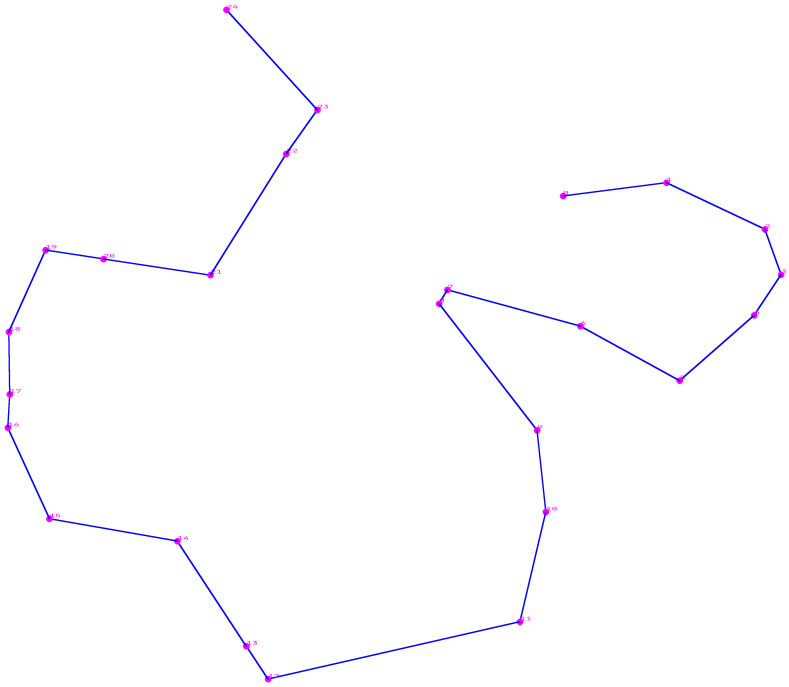

Abbildung 10: Der beste Weg für wenigerkrumm4.txt

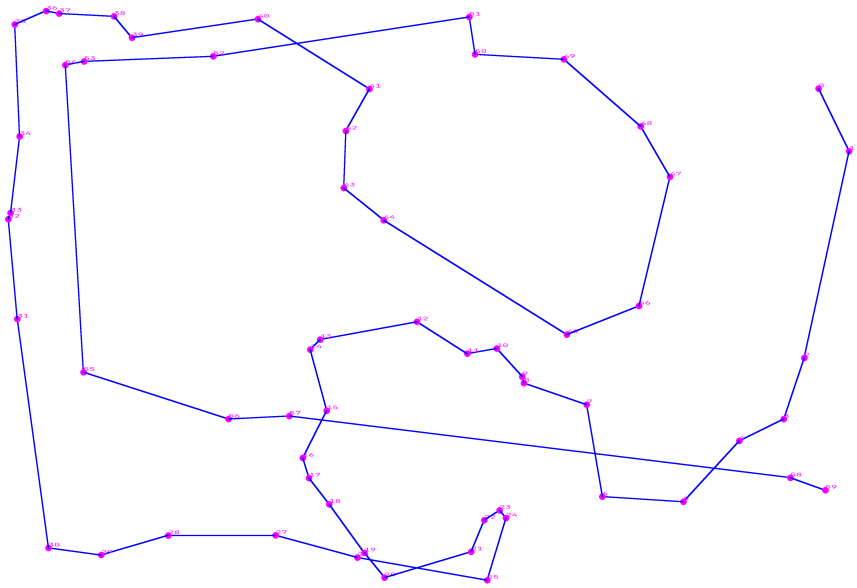

Abbildung 11: Der beste gefundene Weg für wenigerkrumm5.txt

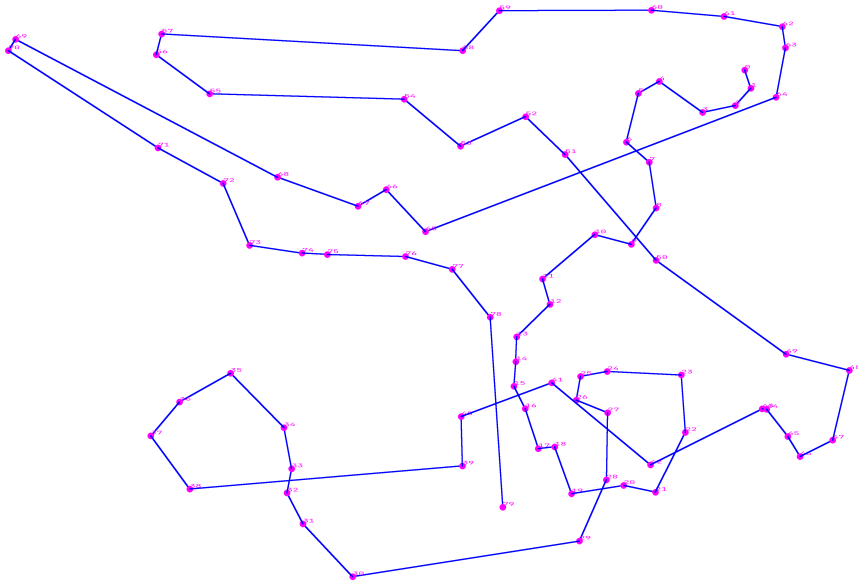

Abbildung 12: Der beste gefundene Weg für wenigerkrumm6.txt

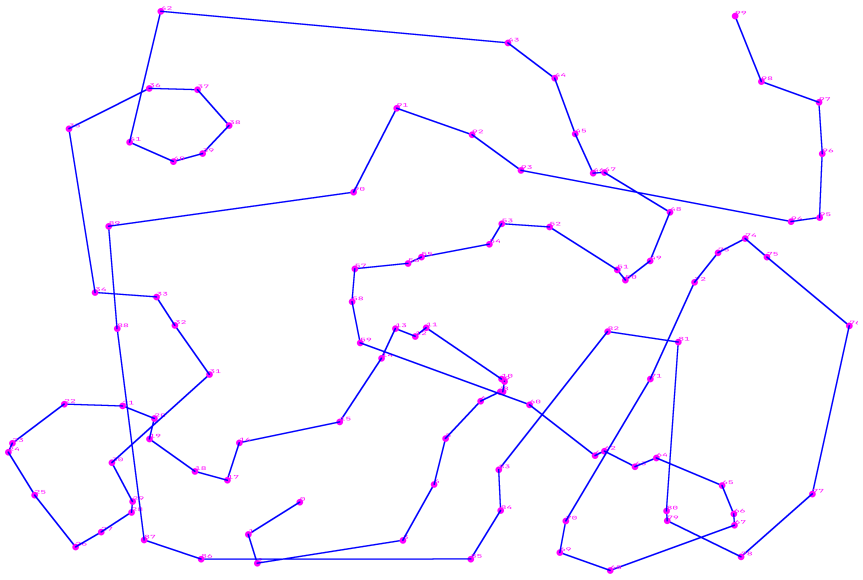

Abbildung 13: Der beste gefundene Weg für wenigerkrumm7.txt

4 Theorethische Analyse

4.1 Laufzeit

Da es möglich ist, dass es von einem bestimmten Startpunkt nur einen möglichen Pfad gibt der den Anforderungen entspricht kann es sein, dass alle Pfade ausprobiert werden müssen. Die Worst-Case Laufzeit ist also $O(n!)$. Um die erwartete Laufzeit des Algorithmus analysieren zu können, muss man zwischen verschiedenen Fällen unterscheiden.

- Abbruch nach dem ersten Ergebnis: Wenn die Suche nach dem ersten gefundenen Pfad abgebrochen wird, überwiegt generell die Zeit für die Vorberechnungen. Die größte dieser Vorberechnungen ist die der Winkel für jeweils drei Punkte (AngleOkList). Diese hat, da sie eine verschachtelte For-Schleife mit einer Tiefe von drei benutzt eine Laufzeit von $O(n^3)$. Deshalb hat das Programm in diesem Fall eine Laufzeitkomplexität von $O(n^3)$.
- Erreichen von der maximalen Schrittanzahl: Wenn die maximale Schrittanzahl nicht ausreicht, um den Weg zu finden so wird die Suche abgebrochen. Deshalb steigt in diesem Fall die Laufzeit linear mit der maximalen Schrittanzahl ($O(n)$).
- Optimaler Weg: Die Zeit die benötigt wird um einen Optimalen Weg zu finden ist im voraus schwer einzuschätzen, da sie je nach Punkten und Startpunkt sehr stark schwanken kann, selbst wenn die Punkte Anzahl gleich ist.

4.2 Platz

Am Anfang der Berechnung werden viele Werte vorbestimmt und gespeichert. Das kann bei großer Punktanzahl erheblichen Platz verbrauchen. Die Vorberechnung die am meisten Platz verbraucht ist wie bei der Laufzeit die Berechnung welche Winkel in Ordnung sind. Diese Werte werden in einer Liste der Länge n^3 gespeichert, also ist die Platzkomplexität $O(n^3)$

5 Quellcode

Der Quellcode ist in mehrere Dateien gegliedert. „main.rs“ enthält Code zum Auslesen der Dateien sowie dem Verarbeiten der Programmargumente, ist also nicht weiter für die Lösung der Aufgabenstellung interessant und hier deshalb nicht mit eingezogen. Die anderen Dateien enthalten folgenden Code:

- angle_list.rs enthält die AngleOkList Klasse und Code um festzustellen, ob ein Winkel in Ordnung ist.
- cost_list.rs enthält die CostList Klasse.
- line.rs enthält Code zum finden von Linien und zum Erstellen von Abkürzungen
- path_finder.rs enthält die PathFinder Klasse.

Da der Quellcode ziemlich lang ist habe ich einige Funktionen entfernt. Der gesamte Quellcode kann im Ordner für diese Aufgabe unter „src/“ gefunden werden.

5.1 angle_list.rs

```rust
// Die Richtung von einem Punkt zu einem Anderen
#[derive(Debug)]
pub struct Dir(pub f64, pub f64);
impl Dir {
    // Gibt den Winkel zwischen zwei Richtungen zurück
    pub fn angle_to(&self, other: &Dir) -> f64 {
        let dot = self.0 * other.0 + self.1 * other.1;
        f64::acos(dot / (self.len() * other.len())) * 180.0 / PI
    }
    // Gibt die Länge der Richtung zurück
    fn len(&self) -> f64 {
        f64::sqrt(self.0 * self.0 + self.1 * self.1)
    }
}
// Gibt zurück ob der Winkel in Grad zwischen 0° und 90° oder 270° und 360° ist
pub fn angle_ok(a: f64) -> bool {
    //schneidet ab der 3. Nachkommastelle ab
    //um bei 90° und 270° Fehler zu vermeiden
    let a = (a * 1000.0).round() / 1000.0;
    a <= 90.0 || a >= 270.0
}
#[derive(Clone)]
pub struct AngleOkList {
    /// data ist ein 1D Array der Größe n_points * n_points * n_points
    /// hier wird gespeichert ob der Winkel zwischen den Punkten i1, i2, i3
    /// ok ist
    data: Vec<bool>,
    n_points: usize,
}
impl AngleOkList {
    // Speichert, ob drei Punkte einen Winkel haben der in Ordnung ist
    pub fn new(points: &Vec<Point>, args: &Args) -> Self {
        let n_points = points.len();
        let mut data = vec![false; n_points * n_points * n_points];
        for (i3, p3) in points.iter().enumerate() {
            for (i2, p2) in points.iter().enumerate() {
                for (i1, p1) in points.iter().enumerate() {
                    //19, 73, 7
                    let dir1 = p1.dir_to(p2);
                    let dir2 = p2.dir_to(p3);
                    let idx = Self::get_idx(n_points, (i1, i2, i3));
```

```
                    let angle = dir1.angle_to(&dir2);
                    data[idx] = angle_ok(angle) || args.dont_check_angle;
                }
            }
        }
        Self { n_points, data }
    }
    // Gibt den Index im 1D Array zurück
    fn get_idx(n_points: usize, idx: (usize, usize, usize)) -> usize {
        n_points * n_points * idx.2 + n_points * idx.1 + idx.0
    }
    // Gibt zurück ob der Winkel zwischen den Punkten i1, i2, i3 ok ist
    pub fn is_ok(&self, i1: usize, i2: usize, i3: usize) -> bool {
        self.data[Self::get_idx(self.n_points, (i1, i2, i3))]
    }
}
```

5.2 cost_list.rs

```
#[derive(Clone, Debug)]
pub struct CostList {
    /// data ist ein 1D Array der Größe n_points * n_points
    /// hier werden die Abstände zwischen den Punkten gespeichert
    /// Wird wie ein 2D Array verwendet Spaltennummer ist ein Punkt
    /// und Zeilennummer der Anderere
    data: Vec<Option<f64>>,
    pub size: usize,
}

impl CostList {
    // Speichert die Abstände zwischen den Punkten
    pub fn new(points: &Vec<Point>) -> Self {
        let size = points.len();
        let mut data = vec![None; size * size];
        let mut i = 0;
        for y in 0..size {
            for x in 0..size {
                if x != y {
                    let dist = points[x].dist_to(&points[y]);
                    data[i] = Some(dist);
                }
                i += 1;
            }
        }
        Self { data, size }
    }
    // Gibt den Index im 1D Array zurück
    fn get_idx(&self, x: usize, y: usize) -> usize {
        self.size * y + x
    }
    pub fn get(&self, x: usize, y: usize) -> &Option<f64> {
        &self.data[self.get_idx(x, y)]
    }
}
```

5.3 line.rs

```
#[derive(Debug, Clone)]
pub struct Line {
```

```rust
    pub ends: [usize; 2], // Start und Endpunkt
    pub seconds: [usize; 2], // Zweiter und vorletzter Punkt
    pub pts: Vec<usize>, // Alle Punkte der Linie
}
impl Line {
    // Erstellt eine neue Linie
    fn new(pts: Vec<usize>) -> Self {
        let ends = [pts[0], pts[pts.len() - 1]];
        let seconds = [pts[1], pts[pts.len() - 2]];
        Self { ends, pts, seconds }
    }
}
/// Findet alle Linien in den Punkten
pub fn find_lines(costs: &CostList, angle_list: &AngleOkList, line_min: usize) -> Vec<Line> {
    println!("Linien finden...");
    let n_pts = costs.size;
    // Erstellen einer HashMap, um schnell auf mögliche Liniensegmente zugreifen zu können
    let mut line_segments = HashMap::new();

    // Finden von Liniensegmenten durch Iterieren über alle Punkte
    for start in 0..n_pts {
        // Bestimmen der drei nächsten Punkte und ihrer Abstände
        let mut nexts = (0..n_pts)
            .filter_map(|b| Some((b, (*costs.get(start, b))?)))
            .collect::<Vec<_>>();
        nexts.sort_by(|a, b| a.1.total_cmp(&b.1));
        let (pt_0, dist_0) = nexts[0];
        let (pt_1, dist_1) = nexts[1];
        let (_, dist_2) = nexts[2];

        // Überprüfen, ob die nächsten drei Punkte ein Liniensegment bilden
        let security = 1.5;
        if dist_1 < dist_0 * security
            && dist_2 > dist_1 * security
            && angle_list.is_ok(pt_0, start, pt_1)
        {
            // Speichern des möglichen Liniensegments in der HashMap
            line_segments.insert(start, (pt_0, pt_1));
        }
    }

    // Zusammenfügen von Liniensegmenten zu Linien
    let mut lines = vec![];
    let mut used_pts = vec![false; n_pts];
    while !line_segments.is_empty() {
        // Erstellen einer neuen Linie
        let mut line = LinkedList::new();

        // Auswahl eines zufälligen Liniensegments als Startpunkt
        let pt = *line_segments.keys().next().unwrap();
        let (mut start, mut end) = line_segments.remove(&pt).unwrap();
        line.push_back(start);
        line.push_back(pt);
        line.push_back(end);
        if used_pts[pt] || used_pts[start] || used_pts[end] {
            continue;
        }
```

```
        used_pts[pt] = true;
        used_pts[start] = true;
        used_pts[end] = true;
        let mut p_pt = pt;

        // Erweiterung der Linie in eine Richtung
        while let Some((a, b)) = line_segments.remove(&end) {
            //Finde den Punkt, der noch nicht Teil der Linie ist
            let new_end = if a == p_pt { b } else { a };
            if used_pts[new_end] {
                break;
            }
            line.push_back(new_end);
            used_pts[new_end] = true;
            p_pt = end;
            end = new_end;
        }
        p_pt = pt;

        // Erweiterung der Linie in die andere Richtung
        while let Some((a, b)) = line_segments.remove(&start) {
            //Finde den Punkt, der noch nicht Teil der Linie ist
            let new_start = if a == p_pt { b } else { a };
            if used_pts[new_start] {
                break;
            }
            line.push_front(new_start);
            used_pts[new_start] = true;
            p_pt = start;
            start = new_start;
        }

        // Überprüfen, ob die Linie ausreichend lang ist und in eine Linie umgewandelt werden kann
        let line = line.into_iter().collect::<Vec<_>>();
        if line.len() >= line_min {
            lines.push(Line::new(line));
        }
    }
    lines
}
/// Speichert Sprünge
pub struct Skip {
    pub end: usize,            //Ende der Linie
    pub second_pt: usize,      //Der Punkt nach dem Anfang der Linie
    pub penultimate_pt: usize, //Der Punkt vor dem Ende der Linie
    pub cost: f64,             //Wie lange die Linie ist
}

/// Erstellt eine Liste von Sprüngen aus einer Liste von Linien
pub fn get_skips(costs: &CostList, lines: &Vec<Line>) -> Vec<Option<Skip>> {
    //Gibt an, ob der Punkt an einem bestimmten Index Endpunkt einer Linie ist
    let mut skips = vec![None; costs.size];
    for line in lines {
        // Ein Sprung für beide Enden der Linie erstellen
        for (skip_start, skip_end) in [(0, 1), (1, 0)].iter() {
            let second_pt = line.seconds[*skip_start];
            let penultimate_pt = line.seconds[*skip_end];
            let skip = Skip::new(
```

```
                line.ends[*skip_end],
                second_pt,
                penultimate_pt,
                line.get_cost(costs),
            );
            // Füge den Sprung in die Liste ein
            skips[line.ends[*skip_start]] = Some(skip);
        }
    }
    skips
}
```

5.4 path_finder.rs

```
//Findet die dichtesten Punkte für jeden Punkt
fn find_nearest_pts(costs: &CostList, lines: &Vec<Line>) -> Vec<Vec<(usize, f64)>> {
    //Punkt die auf einer Linie liegen
    let line_pts: HashSet<usize> = lines
        .iter()
        .flat_map(|l| -> Vec<usize> { (l.pts[1..l.pts.len() - 1]).to_vec() })
        .collect();
    let n_pts = costs.size;
    let result = (0..n_pts)
        .map(|start| {
            let mut nexts = (0..n_pts)
                .filter_map(|b| {
                    // ignoriert, wenn der Punkt auf einer Linie liegt
                    if line_pts.contains(&b) {
                        return None;
                    }
                    Some((b, (*costs.get(start, b))?))
                })
                .collect::<Vec<_>>();
            // sortiert nach Abstand
            nexts.sort_by(|a, b| a.1.total_cmp(&b.1));
            nexts
        })
        .collect::<Vec<_>>();
    result
}
//Ist für die Berechnung des Pfades zuständig
//Da diese Klasse für viele Sachen zuständig ist,
//ist die Anzahl der Member-Variablen relativ groß.
//Ich habe jedoch festgestellt, dass das trennen der Klasse
//in mehrere Verschiedene Einheiten zu mehr Komplexität führt,
//da verschiedene Werte oft an vielen Stellen benötigt werden,
//weshalb die Klasse jetzt etwas mehr Klassenvariablen als ideal hat.
pub struct PathFinder<'a> {
    points: &'a Vec<Point>,  //alle Punkte
    args: &'a Args,          //Parameter
    n_pts: usize,            //Anzahl der Punkte (ohne Linienpunkte)
    angle_list: AngleOkList, //Welche Punkte mit welchem Punkt verbunden werden können

    cost_list: CostList,                //Abstand zwischen allen Punkten
    min_costs: Vec<(f64, f64)>,         //Mindestabstände für jeden Punkt
    nearest_pts: Vec<Vec<(usize, f64)>>, //nächste Punkte für jeden Punkt
    //Die Punkte sortiert nach zweitbestem Abstand (siehe Branch and Bound Umsetzung)
    max_second_costs: Vec<(usize, f64)>,
```

```
    start_pt: usize, //Startpunkt

    lines: Vec<Line>,                    //gefundene Linien
    available_skips: Vec<Option<Skip>>, //von welchen Punkten aus gesprungen werden kann

    free_pts: Vec<bool>, //welche Punkte noch frei sind

    path: Vec<usize>, //aktueller Pfad
    //speichert für jeden Punkt im Pfad,
    //der wievielt nähste Punkt als nächstes drankommt
    idxs: Vec<usize>,
    prev_costs: Vec<f64>,          //Die vorherigen Kosten
    prev_end_idxs: Vec<usize>,     //Die vorherigen Endpunkte (für jeden Punkt im Pfad)
    prev_skips: Vec<Option<Skip>>, //Ob die vorherigen Punkte übersprungen wurden

    cost: f64,           //aktuelle untere Grenze
    curr_end_idx: usize, //der wievielte Endpunkt der aktuelle ist
    //wie oft die Funktion run() aufgerufen wurde,
    //wird benutzt um einen neuen Startpunkt zu bestimmen
    n_runs: usize,

    went_back: bool, //ob schon zurückgegangen wurde
}
impl<'a> PathFinder<'a> {
    /// erzeugt ein neues PathFinder Objekt, berechnet dabei viele Werte vorab
    pub fn new(points: &'a Vec<Point>, args: &'a Args) -> Self {
        let costs = CostList::new(points);
        let angle_list = AngleOkList::new(&points, &args);
        /*
        ...
        gekürzt
        */

        let path = vec![start_pt];
        let idxs = vec![0];
        let prev_costs = vec![cost];
        let prev_skips = vec![None];
        let prev_end_idxs = vec![0];
        let n_runs = 0;
        /*
        ...
        gekürzt
        */
    }
    /// berechnet neue mindestkosten
    pub fn update_costs(&mut self, curr_pt: usize, next_pt: usize) -> f64 {
        let mut cost = 0.0;
        //Der aktuelle Endpunkt
        let curr_last = self.max_second_costs[self.curr_end_idx];
        //falls der aktuelle Punkt der Endpunkt ist
        //wird nach einem neuen Endpunkt gesucht
        if curr_pt == curr_last.0 {
            //sucht den nächsten freien Endpunkt
            let new_last = loop {
                self.curr_end_idx += 1; //geht zum nächsten Endpunkt
                                        //in max_second_costs wurden mögliche Endpunkte
                                        //sortiert nach tauglichkeit gespeichert
                let new_last = self.max_second_costs[self.curr_end_idx];
```

```
                if self.free_pts[new_last.0] && new_last.0 != self.start_pt {
                    //falls der Endpunkt frei ist wird er genommen
                    break new_last;
                }
            };
            let prev_last_cost = curr_last.1;
            let new_last_cost = new_last.1;
            cost += (prev_last_cost - new_last_cost) / 2.0;
        }
        //falls der vorherige punkt der Startpunkt ist
        let min_prev = if self.path.len() == 1 {
            //wird für den vorherigen Punkt die kürzere Strecke abgezogen
            self.min_costs[curr_pt].0
        } else {
            //sonst die längere
            self.min_costs[curr_pt].1
        };
        //Die kürzere Strecke zum nächsten Punkt:
        let min_next = self.min_costs[next_pt].0;
        cost -= (min_prev + min_next) / 2.0;
        cost
    }
    /// findet den nächst-nähsten punkt, der passt
    fn find_next_nearest(&mut self, curr_pt: usize) -> Option<(usize, f64)> {
        let level = self.path.len() - 1; //gibt an, wie weit man sich im Pfad befindet
        //der vorherige punkt, ist None, wenn der aktuelle punkt der erste ist
        let p_pt = self.get_prev_pt();
        //der wivielt-nähste punkt als nächstes genommen wird
        let curr_idx = self.idxs.get_mut(level).unwrap();
        //die liste der nähsten punkte
        let next_pts = &self.nearest_pts[curr_pt];
        //In rust gibt es kein do-while. Um trotzdem zuerst einmal durchzulaufen und dann zu prüfen,
        //ob abgebrochen werden soll, wird ein loop mit break benutzt
        loop {
            // falls alle nächsten punkte ausgeschlossen wurden
            // wird wieder zurückgegangen, da kein nächster Punkt gefunden wurde
            if *curr_idx >= next_pts.len() {
                return None;
            }
            let (next_pt, cost) = next_pts[*curr_idx];
            //falls der näheste punkt schon benutzt wurde oder der winkel nicht passt
            if !self.free_pts[next_pt]
                //überprüft den winkel nur, wenn es einen vorherigen punkt gib
                //das heißt, dass der aktuelle punkt nicht der startpunkt ist
                || (p_pt.is_some() && !self.angle_list.is_ok(p_pt.unwrap(), curr_pt, next_pt))
            {
                *curr_idx += 1; //wird der nächstbeste punkt genommen
            } else {
                //falls der nächste punkt passt, wird er zurückgegeben
                break Some((next_pt, cost));
            }
        }
    }
    /// findet den nächsten punkt, entweder den nächsten nähsten oder einen Sprung
    pub fn find_nexts(&mut self, curr_pt: usize) -> Option<(usize, f64, Option<Skip>)> {
        //gibt an, wie weit man sich im Pfad befindet
        let level = self.path.len() - 1;
        //der vorherige punkt, ist None wenn der aktuelle punkt der erste ist
```

```
        let p_pt = self.get_prev_pt();
        //der wivielt-nähste punkt als nächstes genommen wird,
        //wird benutzt, um zu prüfen, ob auf diesen punkt schon zurückgesprungen wurde
        let curr_idx = *self.idxs.get(level).unwrap();

        let prev_skip = &self.prev_skips[level];
        let skip = &self.available_skips[curr_pt];
        //falls der vorherige Schritt ein Sprung war, ist der aktuelle Punkt das Ende der Linie.
        //Wieder zu springen würde also bedeuten wieder an den Anfang der Linie zu springen,
        //wo man gerade hergekommen ist.
        if skip.is_some() && !prev_skip.is_some() {
            let skip = skip.clone().unwrap();
            if curr_idx == 0 // falls nicht auf den Punkt zurückgesprungen wurde
                // wenn der vorherige punkt nicht None ist wird der winkel geprüft
                // hier wird als dritter Punkt der zweite Punkt der Linie genommen
                && (p_pt.is_none() || self.angle_list.is_ok(p_pt.unwrap(), curr_pt, skip.second_pt))
            {
                Some((skip.end, skip.cost, Some(skip)))
            } else {
                //Man befindet sich am Anfang der Linie und kann nicht springen,
                //also wird zurückgegangen
                None
            }
        } else {
            //Falls man sich nicht am Anfang einer Linie befindet,
            //wird der nächste Punkt normal gesucht
            let (next_pt, cost) = self.find_next_nearest(curr_pt)?;
            Some((next_pt, cost, None))
        }
    }
    /// findet mögliche Wege
    pub fn find(&mut self) -> Option<Vec<Point>> {
        let start = Instant::now();
        let mut best_start_n = None;
        let mut best_start = None;
        let mut best_cost = f64::INFINITY;
        loop {
            if let Some(result) = self.run() {
                if self.args.search_start {
                    let cost = self.cost_list.calc_len(&result);
                    if cost < best_cost {
                        best_cost = cost;
                        best_start_n = Some(self.n_runs);
                        best_start = Some(self.start_pt);
                        println!(
                            "\nNeuer bester Startpunkt, Nummer: {}, Punkt: {}\n",
                            self.n_runs, self.start_pt
                        );
                    } else {
                        println!("\nStartpunkt Nummer {} ist nicht besser als der bisher beste Startp
                    }
                    if self.n_runs >= self.n_pts - 1 {
                        if let Some(best_start_n) = best_start_n {
                            println!(
                                "Bester Startpunkt ist Startpunkt Nummer {}: {}, Cost: {}",
                                best_start_n,
                                best_start.unwrap(),
                                best_cost
```

```
                        );
                    } else {
                        println!("Kein Startpunkt gefunden");
                    }
                } else {
                    self.next_start();
                    continue;
                }
            }
            println!("Pfad gefunden");
            println!("\tDauer (ohne Vorberechnungen): {:?}", start.elapsed());
            //Indexe werden in Punkte umgewandelt
            let points = idxs_to_pts(result, self.points);
            return Some(points);
        }
        if self.n_runs >= self.n_pts - 1 {
            if self.args.search_start {
                if let Some(best_start_n) = best_start_n {
                    println!(
                        "Bester Startpunkt ist Startpunkt Nummer {}: {}, Cost: {}",
                        best_start_n,
                        best_start.unwrap(),
                        best_cost
                    );
                } else {
                    println!("Kein Startpunkt gefunden");
                }
            }
            println!("kein Pfad gefunden");
            return None;
        }
        println!("kein Pfad gefunden, nächster Startpunkt");
        self.next_start();
    }
}
fn run(&mut self) -> Option<Vec<usize>> {
    //kürzester Pfad
    let mut min_path: Option<Vec<usize>> = None;
    let mut min_cost = f64::INFINITY;
    //Anzahl der Pfade, die abgeschnitten wurden
    let mut n_cut: u128 = 0;
    //tiefstes Level, auf das zurückgesprungen wurde
    let mut min_level = usize::MAX;

    let mut i: u128 = 0; //Anzahl der durchgeführten Schritte
    'main_loop: loop {
        i += 1;
        if i > self.args.max_iter && min_path.is_some() {
            println!("max_iter erreicht");
            println!(
                "tiefstes Level auf das zurückgegangen wurde: {}/{}, Abgeschnittene Pfade: {}/{}
                min_level,
                self.n_pts,
                n_cut,
                i,
                n_cut as f64 / i as f64 * 100.0
            );
            return min_path;
```

```
}
if i % 100_000_000 == 0 {
    println!(
        "i: {}, Abgeschnittene Pfade: {:.3}%",
        i,
        n_cut as f64 / i as f64 * 100.0
    );
}
//wenn nach 1.000.000 Schritten kein Pfad gefunden wurde, wird abgebrochen
// Dadurch wird zum nächsten Startpunkt weitergegenagen
if i > 1000_000 && min_path.is_none() {
    return None;
}
if self.path.is_empty() {
    if min_path.is_some() {
        println!("Alle möglichen Pfade durchprobiert");
        return min_path;
    }
    return None;
}
// gibt an, wie weit man sich im Pfad befindet
let level = self.path.len() - 1;

if self.went_back && level < min_level {
    min_level = level;
}
// der aktuelle Punkt
let curr_pt = self.path[level];

// der nächste Punkt wird gesucht
let (next_pt, move_cost, did_skip) = if let Some(result) = self.find_nexts(curr_pt) {
    result
} else {
    // wenn kein nächster Punkt gefunden wurde, wird zurückgesprungen
    self.backtrack(curr_pt);
    continue 'main_loop;
};
self.prev_costs.push(self.cost);
self.prev_end_idxs.push(self.curr_end_idx);
// Die untere Grenze wird aktualisiert
self.cost += move_cost + self.update_costs(curr_pt, next_pt);

// wenn der Pfad zu lang ist, wird er abgeschnitten
if self.cost > min_cost {
    n_cut += 1;
    self.prev_costs.pop();
    self.prev_end_idxs.pop();
    self.backtrack(curr_pt);
    continue 'main_loop;
}

//Vorbereiten zum nächsten Punkt weiterzugehen:

// erhöht den index des aktuellen punkts um 1,
// damit beim nächsten mal der nächste punkt genommen wird
let curr_idx = self.idxs.get_mut(level).unwrap();
*curr_idx += 1;
//fügt 0 als index für den nächsten punkt hinzu
```

```
            self.idxs.push(0);

            //Speichert den nächsten punkt in dem Pfad
            self.path.push(next_pt);
            //markiert den nächsten punkt als belegt
            self.free_pts[next_pt] = false;

            self.prev_skips.push(did_skip);

            // Alle Punkte wurden besucht
            if self.path.len() == self.n_pts {
                // füge die Linien wieder ein
                let full_path = insert_lines(&self.path, self.lines.clone());
                if self.cost < min_cost {
                    let n_high_per = n_cut as f64 / i as f64;
                    println!(
                        "neue untere Grenze: {}, Abgeschnittene Pfade: {}/{} = {:.3}%",
                        self.cost,
                        n_cut,
                        i,
                        n_high_per * 100.0,
                    );
                    min_cost = self.cost;
                    min_path = Some(full_path.clone());
                    if self.args.stop_on_found {
                        println!("--stop_on_found benutzt, beende Suche");
                        return min_path;
                    }
                }
                // gehe zurück um weiterzusuchen
                self.backtrack(next_pt);
                continue 'main_loop;
            }
        }
    }
}
```