Aufgabe 1: Weniger krumme Touren

Teilnahme-ID: 64609

Bearbeiter/-in dieser Aufgabe:
Richard Ewert

31. August 2025

Inhaltsverzeichnis

1 Lösungsidee

Die grundsätzliche Idee ist, möglichst viele Lösungen zu finden und von diesen immer die kürzeste auszugeben. Als Mögliche Startstrecken werden alle Kombinationen aus zwei Strecken mit einem Innenwinkel größer 90° ausgewählt. Diese bilden die Grundlage für einen Lösungspfad.

Da eine kurze Startstrecke eine höhere Wahrscheinlichkeit hat, Teil einer kurzen Lösung zu sein, werden diese Startstrecken nach ihrer Länge aufsteigend sortiert. Beginnend mit der kürzesten Startstrecke wird die zur letzten Koordinate dichteste nächste Koordinate, welche noch nicht im Pfad enthalten ist und die Winkelbedingung erfüllt, angeflogen. Dies wird so lange fortgesetzt, bis keine weiteren Koordinaten zu erreichen sind.

Wenn alle Koordinaten im Pfad enthalten sind, wurde eine Lösung gefunden. Falls dieser Lösungspfad der bisher kürzeste ist, wird er als neue Lösung ausgegeben.

Ansonsten wird die letzte Koordinate aus dem Pfad entfernt, die nächst dichteste angehangen und die Suche wie oben beschrieben fortgesetzt.

Wenn alle oder eine vorgegebene Anzahl an Möglichkeiten für eine Startkombination ausprobiert wurden, wird der Prozess mit der nächst kürzeren Anfangsstrecke erneut gestartet. Dies wird so lange wiederholt, bis alle Anfangsstrecken abgearbeitet sind. Dieser Prozess kann mehrfach parallel ausgeführt werden, um den Prozess zu beschleunigen.

2 Umsetzung

Die Umsetztung erfolgt in der Sprache Rust. Im folgenden werden die entscheidenden Datenstrukturen und Funktionen für die Umsetzung der Lösungsidee beschrieben.

2.1 Repräsentation der Koordinaten

Eine Koordinate wird durch einen `Node` Struct, bestehend aus einer X- und Y-Koordinate, repräsentiert. Der `Node` Struct implementiert eine Funktion `distance()` zum Berechnen der Distanz zwischen sich selbst und einem anderem `Node`, sowie eine Funktion `angles()` zum Berechnen des Winkels zwischen sich selbst und zwei anderen `Node` Objekten. Die Koordinaten aus den Eingangsdaten werden in einem Vektor aus `Node` Objekten `nodes` gespeichert.

```
1     let (nodes, max_iterations, name) = read_nodes();
```

2.2 Caching der Distanzen und Winkel zwischen Nodes

Ein `Node` Objekt wird im Programm generell als Index des Datentyps Vektors repräsentiert. `nodes[o]` repräsentiert die erste Koordinate in den Eingangsdaten, und so weiter.

Da aus meinen Versuchen hervorgegangen ist, dass der Zugriff auf den Speicher schneller ist als das Berechnen der Winkel und Distanzen, wird im Vorfeld die Distanz zwischen ihnen berechnet und im Cache `distances` gespeichert.

Der Zugriff auf die im Cache gespeicherten Distanzen erfolgt über die Indices der Nodes im `distances` Vector. Beispiel: `distances[4][8]` liefert die Distanz zwischen den Nodes 4 und 8 im `nodes` Vektor.

Analog dazu werden die möglichen Ergänzungen von zwei Nodes zu einem Dreieck mit einem Innenwinkel größer 90° vorberechnet, nach Distanz zur zweiten Node aufsteigend sortiert und in `angles` gecached. Beispiel: `angles[2][5]` liefert einen Vektor mit Indices von Folgenodes der Nodes 2 und 5. Der Vektor ist nach der Distanz zum Node 5 sortiert.

Die Caches werden in der Funktion `calc_angles_distances()` aufgebaut:

```
1     let (angles, distances) = calc_angles_distances(&nodes);
```

2.3 Aufbau der Startpfade

Im Vektor `start_paths` werden alle Kombinationen aus 3 ungleichen Nodes, welche die Winkelbedingung erfüllen, als Startpfade mit `generate_start_paths()` vorberechnet. Die Startpfade werden nach Länge aufsteigend sortiert.

Der `start_paths` Vector dient als Aufgabenliste, von der sich mehrere Threads immer den obersten Startpfad nehmen, aus dem Vector entfernen und bearbeiten. Damit das Nehmen des obersten Eintrages bei paralleler Ausführung mit mehreren Threads nicht durch andere negativ beeinflusst wird, ist `start_paths` ein 'atomic reference counted mutual exclusive' Vector.

```
1     let generated_paths = generate_start_paths(&angles, &distances);
2     let start_paths: Arc<Mutex<Vec<Vec<usize>>>> =
3         Arc::new(Mutex::new(generated_paths.clone()));
```

2.4 'main()' Funktion

Die `main()` Funktion legt in Rust den Anfang des Programmes fest. Sie ruft die anderen Funktionen auf und kümmert sich um das Verteilen von Aufgaben auf unterschiedliche Threads. Sie beginnt mit dem Einlesen der vom Nutzer übergebenen Werte. Sie initialisiert die in Abschnitt 2.2 beschriebenen caches, sowie den in Abschnitt 2.3 beschriebenen `start_paths` Vektor. Schließlich werden basierend auf der Anzahl an CPU Kernen des Systems mehrere Threads erstellt.

Jeder Thread hat eine eigene Kopie der zuvor berechneten caches, da sonst nicht sichergestellt werden kann, dass die Threads beim auslesen der Caches nicht auf einander warten müssen. Außerdem werden ihnen über mehrere Threads teilbare Referenzen zu Variablen übergeben. Dazu gehört `best_solution`, zum Speichern der Lösung, sowie der Vektor `start_paths`, um den nächsten Startpfad abnehmen zu können. Die Threads entfernen jeweils das oberste Element dieses Vektor und rufen mit diesem als `start_path` Parameter die `solve_recursive()` Funktion auf.

2.5 'solve_recursive()' Funktion

Die solve_recursive() Funktion implementiert das Finden der Lösungen für einen Startpfad rekursiv.

2.5.1 Parameter

1. 'path: &mut Vec<usize>' ist nur eine Referenz zu dem bisher erstellten Pfad, um schnell durch die Funktion gereicht werden zu können. Sie enthält alle Indices der bisher besuchten Nodes in der Reihenfolge, wie sie angeflogen wurden.
2. 'path_length: f32' enthält die Länge des Pfades in Kilometern. Die Länge ist aus dem path Vector berechenbar, es ist jedoch schneller bei einer Veränderung des Vectors die path_length Variable ebenfalls um die hinzugefügte Distanz zu verändern, als sie neu zu berechnen. Deshalb gibt es einen eigenen Parameter.
3. 'nodes: &Vec<Node>' enthält alle Node Objekte. Wird benötigt, um zwischen den Indices der Nodes und den Nodes selbst umzuwandeln.
4. 'angles: &Vec<Vec<Vec<usize>>>' wird erklärt in 2.2. Wird benötigt, um schnell Winkel auslesen zu können.
5. 'distances: &Vec<Vec<f32>>' wird ebenfalls erklärt in 2.2. Wird benötigt, um schnell Distanzen auslesen zu können.
6. 'best_solution: &mut Arc<Mutex<Vec<usize>>>' ist eine über mehrere Threads teilbare Referenz zu der aktuell besten bekannten Lösung/des besten path Vectors. Hinter der Referenz steht die selbe Datenstruktur, die auch path teilt. Wird benötigt, um eine bessere Lösung dort speichern zu können.
7. 'best_solution_length: &mut Arc<Mutex<f32>>' eine Thread teilbare Referenz zu der Länge der besten Lösung in Kilometern. Lösung und Länge sind eigene Variablen, obwohl die Länge auch aus dem Lösungspfad berechnbar wäre. Das ist nötig, da veränderbare Mutex (mutual exclusive) Referenzen vor dem Auslesen warten müssen, bis kein anderer Thread sie mehr verändert. Zwei getrennte Variablen erlauben es, Länge und Lösung unabhängig zu lesen und zu schreiben, um so Zeit die sonst zum Warten auf andere Threads benötigt werden würde zu sparen.
8. 'input_file_name: &String' Eine Referenz zum Namem der Beispieldatei, um eine Lösung unter einem ähnlichen Namen abspeichern zu können.
9. 'iterations: &mut u64' Eine Referenz zu einem Integer zum Zählen, wie häufig die solve_recursive() Funktion aufgerufen wurde.
10. 'max_iterations: &u64' Legt fest, nach wie vielen Funktionsaufrufen der Prozess abgebrochen und ein neuer Startpfad probiert werden soll.

2.5.2 Funktionsweise

Zu Beginn der solve_recursiv Funktion wird geprüft, ob die Abruchbedingungen für die Funktion erfüllt sind. Die Abbruchbedingung ist erfüllt, wenn die maximale Anzahl an Funktionsaufrufen für diesen Startpfad erreicht ist. Die Abbruchbedingung ist auch erfüllt, wenn der übergebene Pfad gleich oder länger als die beste bisher gefundene Lösung ist, da durch keine Verlängerung des Pfades eine bessere Lösung gefunden werden kann.

Wenn der Pfad so viele Einträge hat wie es Nodes in den Eingabedaten gibt, muss es sich um eine Lösung handeln, da alle Nodes im Pfad nur einmal enthalten sind und die Winkelbedingung erfüllen. Diese Lösung muss kürzer sein als die bisherige, da ansonsten die Funktion schon durch die vorherige Bedingung beendet worden wäre. Der Pfad wird als neue Lösung in best_solution gespeichert. Um sicherszustellen, dass kein anderer Thread best_solutions zwischen diesen Schritten verändert wird die Variable bis Ende des Prozesses gelocked. Im Anschluss wird die neu gefundene Lösung durch render() ausgegeben und die Funktion beendet.

Wenn weder die Abbruchbedingungen erfüllt ist, noch eine Lösung gefunden wurde, wird die Suche nach einer Lösung fortgesetzt. Dazu werden die letzten 2 Einträge im Pfad verwendet, um mithilfe des angles Caches alle möglichen Ergänzungen des Pfades nach Distanz zum letzten Element sortiert auszulesen. Aus diesen möglichen Optionen options werden alle Nodes entfernt, welche bereits im Pfad path enthalten sind.

```
    // Alle Möglichen Ergänzungen der letzten zwei Pfadeinträge
    // welche die Winkelbedingung erfüllen,
    // werden nach Distanz sortiert in "options" gespeichert
    let mut options: Vec<usize> =
        angles[path[path.len() - 2]][path[path.len() - 1]].clone();
    // Es werden nur die behalten, welche nicht im Pfad enthalten sind
    options.retain(|x| !path.contains(x));
```

In `options` sind nun die Nodes enthalten, welche die Winkelbedingung erfüllen und noch nicht im Pfad enthalten sind. Nun wird über `options` iteriert. Dabei wird die jeweilige Node zu `path` hinzugefügt, die durch diesen Node hinzukommende Pfadlänge aus dem Cache `distances` ausgelesen. Im Anschluss wird die Funktion `solve_recursive` rekursiv mit dem ergänzten Pfad `path`, der angepassten Länge des Pfades `path_length + add_length`, sowie den restlichen Parametern aufgerufen.

```
    for i in options {
        // Wird zum Pfad hinzugefügt
        path.push(i);
        // Die zusätzliche Länge wird berechnet
        let add_length = distances[path[path.len() - 2]][path[path.len() - 1]];
        // Der veränderte Pfad wird mit den anderen Parametern weitergegeben
        solve_recursive(
            path,
            path_length + add_length,
            ... // Weitere Parameter direkt weitergegeben
        );
```

Nach Abschluss des rekursiven Funktionsaufrufes, wird der hinzugefügte Node wieder entfernt und die nächste Iteration über `options` beginnt.

Auf diese Weise können alle Möglichkeiten, einen Weg zu allen Koordinaten zu finden, durchschritten werden.

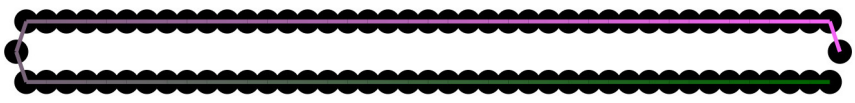
```
        // Nachdem das finden der Lösungen dieses Teilbaumes abgeschlossen ist,
        // werden die Veränderungen zum Pfad wieder rückgängig gemacht
        path.pop().unwrap();
    }
```

3 Beispiele

3.0.1 wenigerkrumm1

Das erste Beispiel löst der Algorithmus gut. Da immer die dichteste noch nicht besuchte Node als nächste gewählt wird, ist bei jedem Startpfad die erste gefunden Lösung auch die beste.

```
\$: ./bwinf-test --path pfad/zu/wenigerkrumm1.txt 1000
```

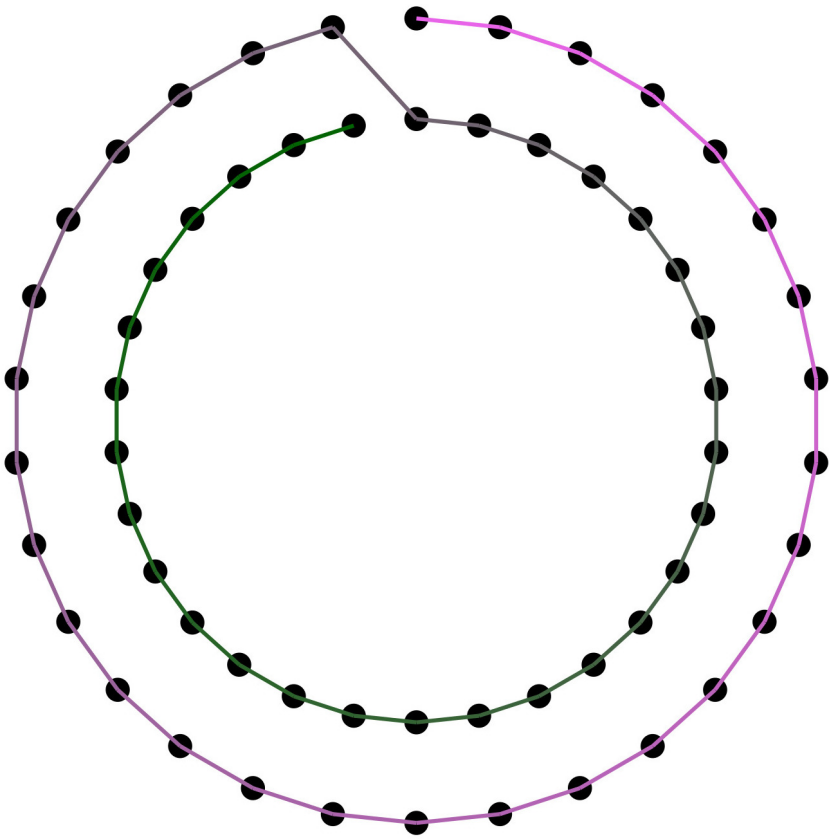


3.0.2 wenigerkrumm2

Das zweite Beispiel wird noch schneller als das erste gelöst. Der erste Startpfad liegt im inneren Punktering und der Algorithmus folgt diesem, bis alle inneren Punkte enthalten sind. Danach wird der dichteste Punkt des äußeren Kreises probiert. Von diesem kann jedoch kein anderer Punkt mehr erreicht werden,

da ansonsten die Winkelbedingung nicht erfüllt wäre. Es wird ein Punkt zurück gegangen und der nächst dichteste probiert. Von diesem kann der äußere Ring umlaufen werden. Die unten gezeigt Lösung wird gefunden.

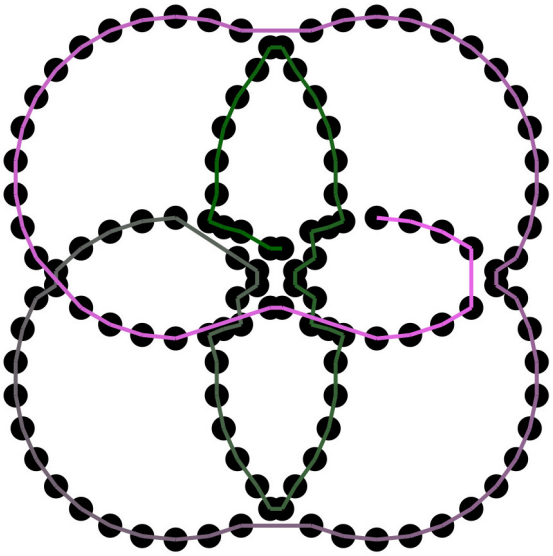
```
\$: ./bwinf-test --path pfad/zu/wenigerkrumm2.txt 1000
```



3.0.3 wenigerkrumm3

Das dritte Beispiel wird nicht so gut gelöst wie die vorherigen. Da immer die dichteste Node erreicht werden soll, folgt der Algorithmus nicht den für Menschen offensichtlich kürzeren vorgegebenen Kreisen.

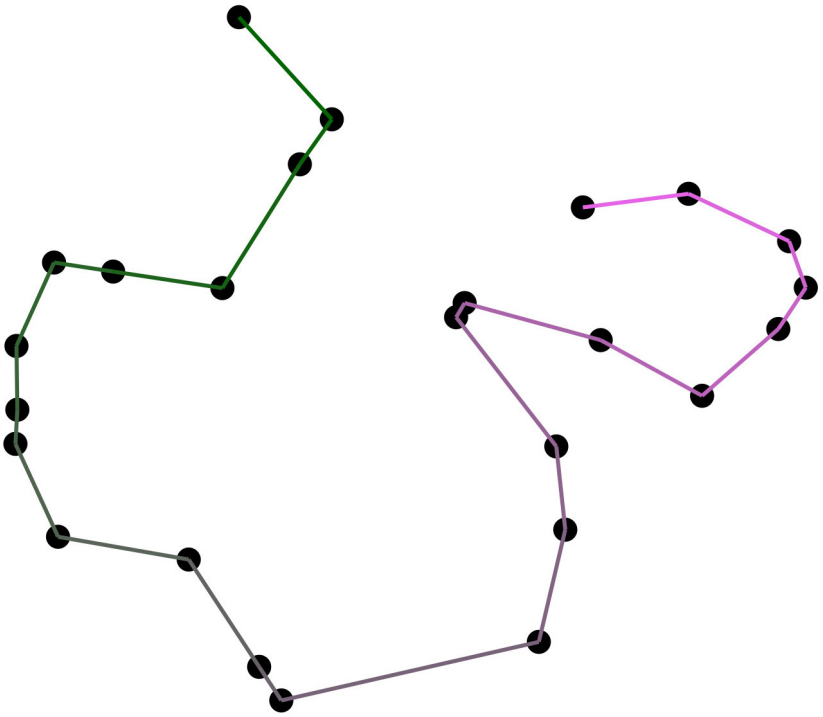
```
\$: ./bwinf-test --path pfad/zu/wenigerkrumm3.txt 1000
```



3.0.4 wenigerkrumm4

Beim vierten Beispiel ist es schwieriger Nachzuvollziehen, wie die Lösung erreicht wurde. Da es keinerlei Überschneidungen im Pfad gibt und fast alle Nodes mit ihren dichtesten Nachbarn verbunden sind, handelt es sich aber offensichtlich um eine sehr gute Lösung. Hier wird auch klar wie nützlich das Wechseln der Startposition ist. Bei diesem Beispiel begint der Pfad nähmlich an einer Stelle, welche keine leicht zu identifizierbaren Merkmale hat.

```
\$: ./bwinf-test --path pfad/zu/wenigerkrumm4.txt 1000
```

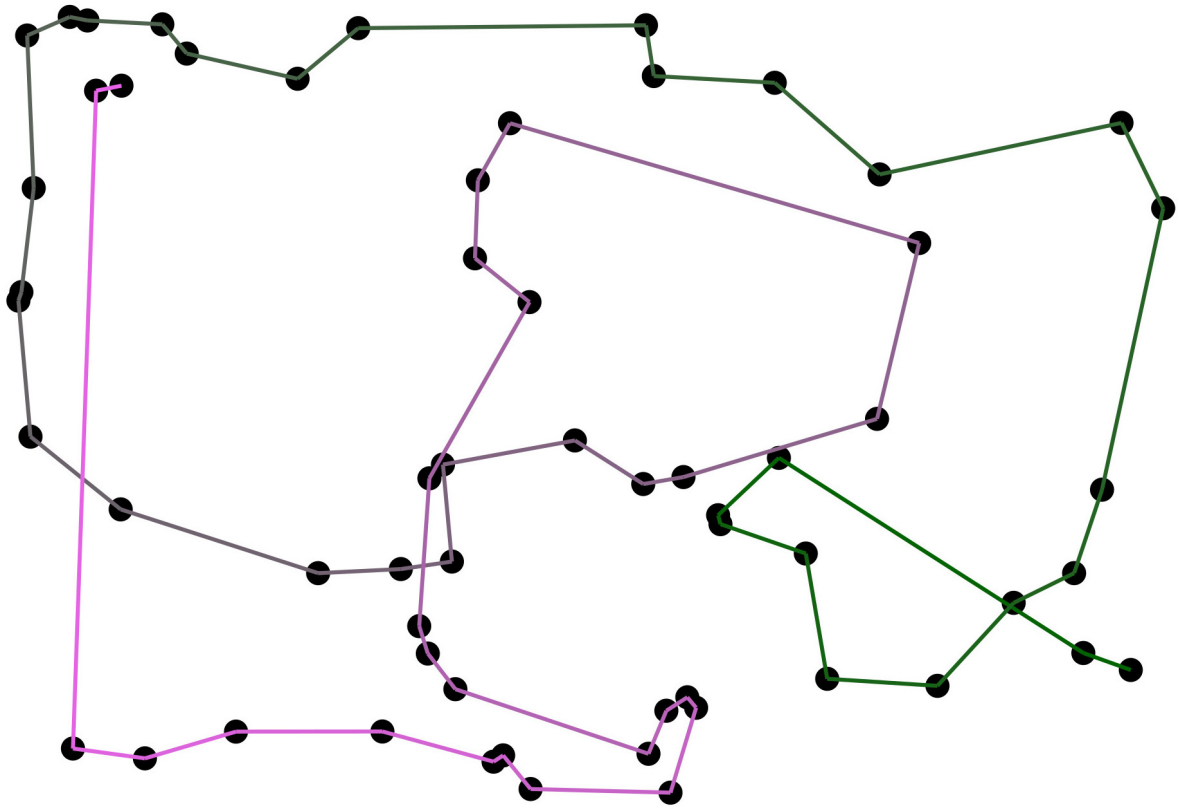


3.0.5 wenigerkrumm 5 bis 7

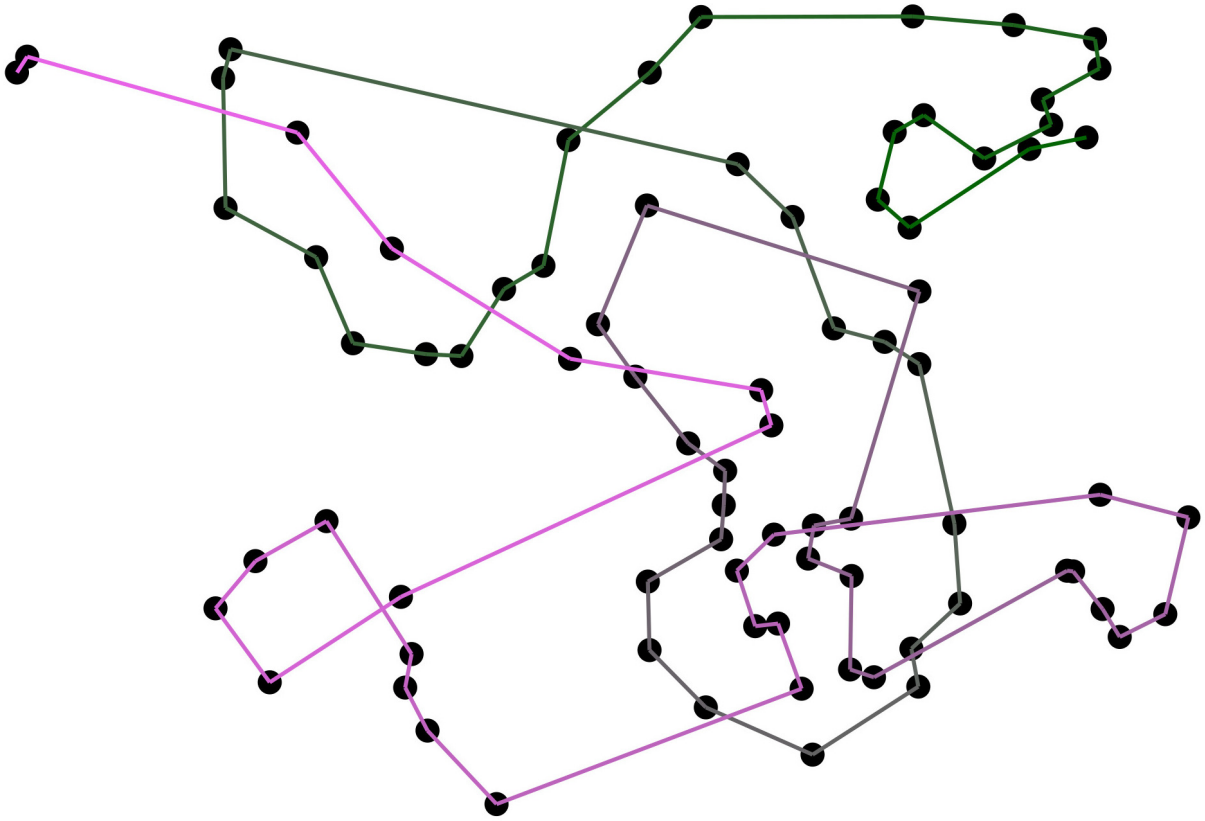
Die Beispiele 5 bis 7 sind zusammengefasst, da sie alle ähnlich sowohl in ihrem Aufbau als auch ihrer Lösung sind. Es fällt auf, dass sich der Algorithmus mit größeren Netzen schwerer tut. Der Pfad nimmt

manchmal große Umwege, um Nodes zu erreichen, welche eigentlich schon früher erreichbar gewesen wären. Es wird deutlich, dass bei diesem Ansatz die Variation in den anfänglichen Teilen des Pfades fehlt. Wenn besonders zu Begin eine Route gewählt wird, die spätere Möglichkeiten verbaut, kann dies nicht mehr korrigiert werden. Eine Möglichkeit diesen Aspekt zu verbessern ist, nachdem eine Lösung gefunden wurde, Abschnitte und einzelne Nodes miteinander zu vertauschen, um so eine schnellere Route zu finden.

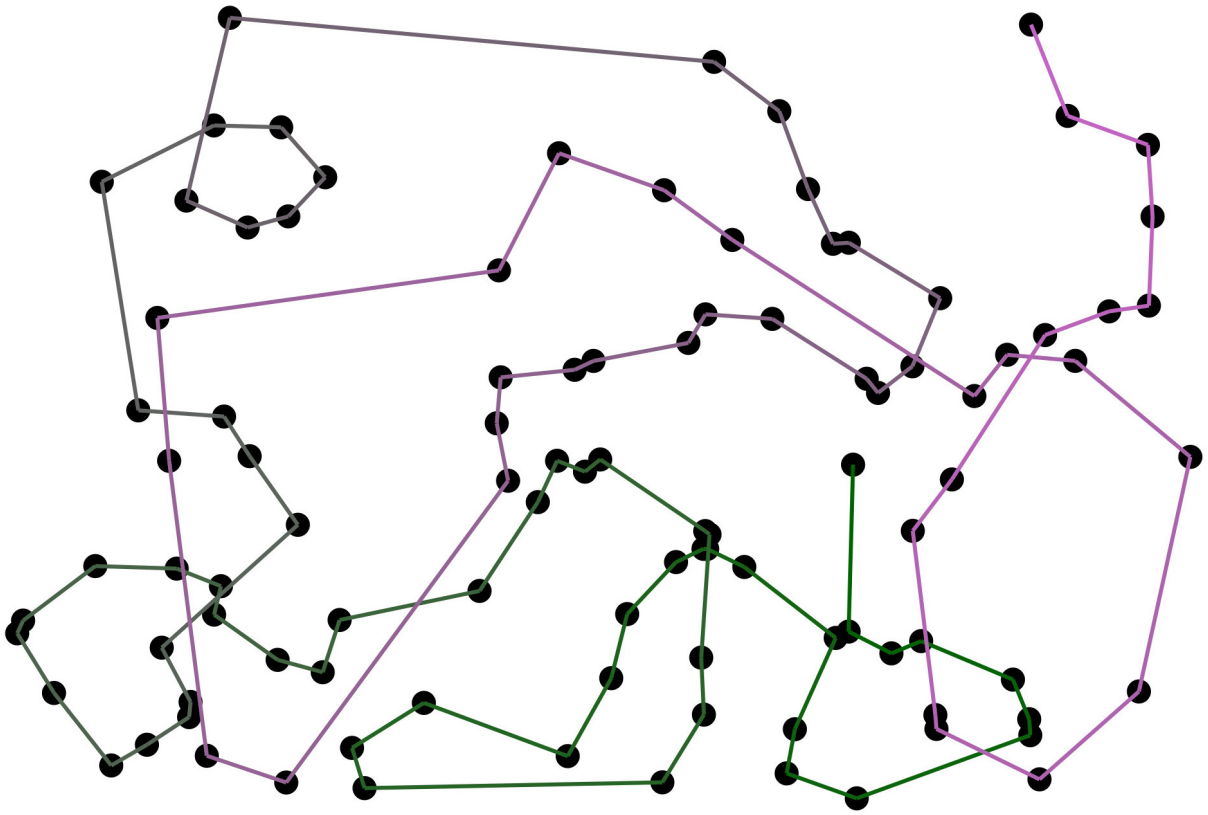
1. `\$: ./bwinf-test --path pfad/zu/wenigerkrumm5.txt 1000`



2. `\$: ./bwinf-test --path pfad/zu/wenigerkrumm6.txt 1000`



3. `\$: ./bwinf-test --path pfad/zu/wenigerkrumm7.txt 1000`



4 Quellcode

```rust
// Für den eigentlchen Algorithmus irrelevanter Code,
// zum schreiben und lesen von Beispielen und Lösungen
mod input_output_mod;

// Enthält den Node Struct und dessen Funktionen
mod node_mod;

// Funktionen zum lesen und schreiben
use input_output_mod::{read_nodes, render};
// Logging crate
use log::{debug, info};
use node_mod::Node;
use std::cmp::Ordering;
use std::f32::MAX;
use std::num::NonZeroUsize;
use std::path::PathBuf;
use std::sync::{Arc, Mutex};
// use std::time::Instant;
use std::thread::available_parallelism;
use std::thread::{self, JoinHandle};
use indicatif::{ProgressBar, ProgressStyle};

// Berechnet die Länge Eines Pfades bestehend aus Indexen von Nodes
fn path_len(path: &Vec<usize>, distances: &[Vec<f32>]) -> f32 {
    let mut distance: f32 = 0f32;
    // "rest" enthält jetzt den Pfad bis auf das letzte Element
    let (_last, rest) = path.split_last().unwrap();
    for (i, _node_index) in rest.iter().enumerate() {
        // Für jede Node im path wird die Distanz zur nächsten berechnet
        distance += distances[path[i]][path[i + 1]];
    }
    distance
}

fn calc_angles_distances(nodes: &Vec<Node>) ->
        (Vec<Vec<Vec<usize>>>, Vec<Vec<f32>>) {
    // 2d Vector, um alle Distanzen zwischen 2 Nodes zu speichern
    let mut distances: Vec<Vec<f32>> = vec![];
    // 3d Vector, um alle Ergänzungen für 2 Nodes zu speichern
    let mut angles: Vec<Vec<Vec<usize>>> = vec![];
    // Debug Variable, um Menge von einträgen zu zählen
    let mut cache_entries = 0;
    // Es wird zum ersten Mal über jede Node iteriert
    for (start_node_index, start_node) in nodes.iter().enumerate() {
        // Beide Vectoren werden "2d gemacht"
        distances.push(vec![]);
        angles.push(vec![]);
        for (main_node_index, main_node) in nodes.iter().enumerate() {
            distances[start_node_index].push(start_node.distance(main_node));
            angles[start_node_index].push(vec![]);
            for (end_node_index, end_node) in nodes.iter().enumerate() {
                let angle = main_node.angle(start_node, end_node);
                debug!(
                    "Angle between {:?}, {:?}, {:?} : {:?}",
                    start_node_index,
                    main_node_index,
```

```
                    end_node_index,
                    angle);
                if 90f32 <= angle {
                    angles[start_node_index][main_node_index].push(end_node_index);
                    cache_entries += 1;
                }
            }
            // Die erstellte Liste wird nach Distanz zur mittleren Node sortiert
            angles[start_node_index][main_node_index].sort_by(|a, b| {
                let node_a = &nodes[*a];
                let node_b = &nodes[*b];
                // Distanz wird ausgerechnet
                let node_a_distance = main_node.distance(node_a);
                let node_b_distance = main_node.distance(node_b);

                // Distanz wird mit Wert zum vergleichen gleichgesetzt
                let val_a = node_a_distance;
                let val_b = node_b_distance;
                if val_a < val_b {
                    Ordering::Less
                } else if val_a == val_b {
                    Ordering::Equal
                } else {
                    Ordering::Greater
                }
            });
        }
    }
    info!("Cache entries count: {}", cache_entries);
    debug!("Cached entries: {:?}", angles);
    debug!("Cached distances: {:?}", distances);
    (angles, distances)
}

fn sort_paths(tasks: &mut Vec<Vec<usize>>, distances: &Vec<Vec<f32>>) {
    tasks.sort_by(|a, b| {
        let val_a = path_len(&a, distances);
        let val_b = path_len(&b, distances);
        if val_a > val_b {
            Ordering::Less
        } else if val_a == val_b {
            Ordering::Equal
        } else {
            Ordering::Greater
        }
    });
}

// Gibt alle Kombinationen aus 3 unterschiedlichen Nodes nach länge sortiert zurück
fn generate_start_paths(
    angles: &Vec<Vec<Vec<usize>>>,
    distances: &Vec<Vec<f32>>,
) -> Vec<Vec<usize>> {
    let mut paths: Vec<Vec<usize>> = vec![];
    for (first_node_index, second_node_indices) in
            angles.iter().enumerate() {
        for (second_node_index, third_node_indices) in
                second_node_indices.iter().enumerate() {
```

```
            for valid_third_node_index in
                    third_node_indices.iter() {
                if first_node_index != second_node_index
                    && second_node_index != *valid_third_node_index
                    && first_node_index != *valid_third_node_index
                {
                    let path = vec![
                        first_node_index,
                        second_node_index,
                        *valid_third_node_index
                    ];
                    paths.push(path);
                }
            }
        }
    }
    sort_paths(&mut paths, distances);
    debug!("Start paths: {:?}", paths);
    info!("Generated {} start paths", paths.len());
    paths
}

fn indices_to_nodes(
        nodes: Vec<Node>,
        indices_path: &Vec<usize>) -> Vec<Node> {
    let mut node_path: Vec<Node> = vec![];
    for i in indices_path {
        node_path.push(nodes[*i]);
    }
    node_path
}

fn solve_recursive(
    path: &mut Vec<usize>,
    path_length: f32,
    nodes: &Vec<Node>,
    angles: &Vec<Vec<Vec<usize>>>,
    distances: &Vec<Vec<f32>>,
    best_solution: &mut Arc<Mutex<Vec<usize>>>,
    best_solution_length: &mut Arc<Mutex<f32>>,
    input_file_name: &String,
    iterations: &mut u64,
    max_iterations: &u64,
) {
    // Jeder Aufruf der Funktion erhöht den iterationszähler um 1
    *iterations += 1;
    // ========== ANFANG DER ABBRUCHBEDINGUNGEN ============================
    // Die Länge der besten bekannten Lösung ist über alle Threads geteilt.
    // Deshalb wird die Variable zuerst "gelocked",
    // um andere Threads am Verändern zu hindern.
    let mut sol_len = best_solution_length.lock().unwrap();
    // Abgebrochen wird, wenn die maximale Menge an Iterationen
    // erreicht oder die Länge des eigenen Pfades größer
    // als die der kürzesten bekannten Lösung ist.
    if *iterations > *max_iterations || path_length >= *sol_len {return};
    // Wenn es so viele Einträge im Pfad, wie Nodes gibt, wurde eine Lösung
    // gefunden, wird sie als neue gespeichert und die Funktion abgebrochen.
    if path.len() == nodes.len() {
```

```rust
        // Auch die beste Lösung wird "gelocked"
        let mut best_solution_lock =
                best_solution.lock().unwrap_or_else(|e|panic!("{}", e));
        // Der aktuelle Pfad wird in die Stelle der besten Lösung geklont
        path.clone_into(&mut best_solution_lock);
        // Die Länge der besten Lösung wird aktuallisiert
        *sol_len = path_length;
        // Die Lösung wird als txt und svg gespeichert
        render(
            nodes,
            &indices_to_nodes(nodes.clone(), &best_solution_lock),
            path_length, input_file_name.clone()
        );
        return;
    }
    // Die Länge der besten Lösung wird nicht mehr benötigt
    // und fallen gelasssen, um anderen Threads den Zugriff zu gewähren
    drop(sol_len);
    // ========== ENDE DER ABBRUCHBEDINGUNGEN ============================

    // Alle Möglichen Ergänzungen der letzten zwei Pfadeinträge
    // welche die Winkelbedingung erfüllen,
    // werden nach Distanz sortiert in "options" gespeichert
    let mut options: Vec<usize> =
        angles[path[path.len() - 2]][path[path.len() - 1]].clone();
    // Es werden nur die behalten, welche nicht im Pfad enthalten sind
    options.retain(|x| !path.contains(x));

    // Jede dieser Nodes
    for i in options {
        // Wird zum Pfad hinzugefügt
        path.push(i);
        // Die zusätzliche Länge wird berechnet
        let add_length = distances[path[path.len() - 2]][path[path.len() - 1]];
        // Der veränderte Pfad wird mit den anderen Parametern weitergegeben
        solve_recursive(
            path,
            path_length + add_length,
            nodes,
            angles,
            distances,
            best_solution,
            best_solution_length,
            input_file_name,
            iterations,
            max_iterations,
        );
        // Nachdem das finden der Lösungen dieses Teilbaumes abgeschlossen ist,
        // werden die Veränderungen zum Pfad wieder rückgängig gemacht
        path.pop().unwrap();
    }
}

fn main() {
    env_logger::init();

    assert!(PathBuf::from("./outputs/txt/").is_dir(),
    "Txt Ordner nicht gefunden. Bitte sicherstellen, dass der Pfad ./outputs/txt/ valide ist");
```

```
    assert!(PathBuf::from("./outputs/svg/").is_dir(),
    "Svg Ordner nicht gefunden. Bitte ebenfalls sicherstellen, dass der Pfad ./outputs/svg/ valide

    // Ließt alle Nodes und die Suchlänge ein
    let (nodes, max_iterations, name) = read_nodes();

    // Winkel und Distanzen werden zum schnellen auslesen berechnet
    let (angles, distances) = calc_angles_distances(&nodes);

    // Alle Anfangspfade werden bestimmt
    let generated_paths = generate_start_paths(&angles, &distances);

    // Diese Variablen sind über alle Threads geteilt:
    // Vector mit Startpfaden die noch probiert werden müssen
    let start_paths: Arc<Mutex<Vec<Vec<usize>>>> =
        Arc::new(Mutex::new(generated_paths.clone()));
    // Die aktuell beste bekannte Lösung
    let best_solution: Arc<Mutex<Vec<usize>>> = Arc::new(Mutex::new(vec![]));
    // Die Länge der aktuell besten bekannten Lösung
    let best_solution_length: Arc<Mutex<f32>> = Arc::new(Mutex::new(MAX));
    let done_threads: Arc<Mutex<u32>> = Arc::new(Mutex::new(0));

    // Hier werden die handles für die Threads eingetragen werden
    let mut handles: Vec<JoinHandle<()>> = vec![];
    // Bestimmt, wie viele CPU Kerne zur Verfügung stehen
    let total_threads: usize = available_parallelism()
        .unwrap_or(NonZeroUsize::new(1).unwrap()).into();

    // Erstellen der Fortschrittsleiste
    let bar: ProgressBar = ProgressBar::new(generated_paths.len() as u64)
        .with_style(ProgressStyle::with_template(
        "[{elapsed_precise}] {bar:40.cyan/blue} {pos:>7}/{len:7} {msg} (eta: {eta})"
        ).unwrap()
    );
    bar.set_message("Geprüfte Startpfade");
let bar = Arc::new(Mutex::new(bar));

    info!("Starting up {:?} threads", total_threads);
    for _i in 0..(total_threads - 1) {
        // Jeder Thread benötigt Zugriff auf die obigen Variablen.
        // Deshalb werden diese vom Hauptthread in neue mit "l_"
        // notierte Variablen geklont.
        // Jede der geteilten Variablen benötigt eine Referenz
        let mut l_best_solution = Arc::clone(&best_solution);
        let mut l_best_solution_length = Arc::clone(&best_solution_length);
        let l_start_paths = Arc::clone(&start_paths);
        let l_done_threads = Arc::clone(&done_threads);
        let l_bar = Arc::clone(&bar);

        // Konstante Werte werden geklont
        let l_nodes = nodes.clone();
        let l_angles = angles.clone();
        let l_distances = distances.clone();
        let l_name = name.clone();
        let l_max_iterations = max_iterations.clone();
        let l_total_tasks = generated_paths.len();

        // Der neue Thread wird gestartet.
```

```
        // Die obigen "l_" Variablen ziehen in den Thread um,
        // wenn sie Referenziert werden.
        let handle = thread::spawn(move || {
            info!("Started thread {:?}", thread::current().id());
            loop {
                let mut todo = l_start_paths.lock().unwrap();
                // Falls noch ein ungeprüfter Startpfad existiert
                if let Some(mut l_start_path) = todo.pop() {
                    let thread_number = todo.len();
                    drop(todo);
                    let l_path_length = path_len(&l_start_path, &l_distances);
                    let mut l_iterations = 0;
                    // Finde alle Lösungen
                    solve_recursive(
                        &mut l_start_path,
                        l_path_length, &l_nodes,
                        &l_angles, &l_distances,
                        &mut l_best_solution,
                        &mut l_best_solution_length,
                        &l_name,
                        &mut l_iterations,
                        &l_max_iterations
                    );
                    // Debug und ausgabe (irrelevant)
                    let mut done = l_done_threads.lock().unwrap();
                    let bar = l_bar.lock().unwrap();
                    bar.inc(1);
                    drop(bar);
                    *done += 1;
                    debug!(
                        "Thread {:?} finished path with priority {:?}  {:?}/{:?}",
                        thread::current().id(),
                        thread_number,
                        done,
                        l_total_tasks
                    );
                    drop(done);
                } else {
                    info!("Thread {:?} returned", thread::current().id());
                    // Beende die Schleife und damit den Thread,
                    // wenn alle Startpfade probiert worden sind
                    break;
                }
            }
        });
        // Thread handle wird gespeichert
        handles.push(handle);
    }
    // Iteriert über alle Threadhandles und wartet bis sie fertig sind
    while let Some(handle) = handles.pop() {
        handle.join().unwrap();
    }
    bar.lock().unwrap().finish();

    let final_solution = best_solution.lock().unwrap();
    // Stellt die gefundene Lösung dar
    if !final_solution.is_empty() {
        render(
```

```
            &nodes,
            &indices_to_nodes(nodes.clone(), &final_solution),
            path_len(&final_solution, &distances),
            name
        );
    }
}
```