Aufgabe 1: Weniger krumme Touren

Teilnahme-ID: 66048

Bearbeiter/-in dieser Aufgabe:
Karolina Schnitter

28. March 2023

Inhaltsverzeichnis

Lösungsidee

Für die Aufgabe muss als Grundlage die Distanz zwischen zwei Punkten berechnet werden sowie der Winkel zwischen drei Punkten. Für die Distanz kann der Satz des Pythagoras verwendet werden, für den Winkel der Cosinussatz.

Ein passender möglichst kurzer Weg kann mit folgenden 2 Lösungsschritten gefunden werden

1. einen möglichst kurzen Weg mit heuristischen Algorithmen finden

2. den gefundenen Weg durch Veränderung einzelner Abschnitte verbessern

Zu 1.

Ich habe drei verschiedene Varianten des Nearest Neighboor Algorithmus verwendet. Bei dem Ersten wird von einem Startknoten aus, der Reihe nach in einer Richtung nach nahen Nachbarknoten gesucht und darauf geachtet, dass die Abbiegewinkel erlaubt sind. Der Zweite sucht in beide Richtungen von einem Startknoten aus. Der Dritte verbindet immer die Knoten oder Weg-Abschnitte miteinander, die die kürzeste Distanz zueinander haben. Alle 3 Varianten lassen sich mit Backtracking umsetzen. Backtracking gehört zu den Brute-force Algorithmen, das heißt, dass die Zeitkomplexität, um eine Lösung zu finden, O(n!) sein kann. Das kann z.B. auftreten, wenn von einem gewählten Startknoten kein Weg existiert, bei dem die Bedingung mit den Abbiegewinkeln erfüllt ist. Um das zu umgehen habe ich festgelegt, dass nach einer bestimmten Anzahl von backtracking steps ein neuer Startknoten gewählt wird. Diese Anzahl ist unabhängig von n = Anzahl

der Knoten. Damit sinkt die Zeitkomplexität auf $O(n^2)$. Um die Warscheinlichkeit einen guten Weg zu finden zu erhöhen werden die erste und zweite Variante für jeden Startknoten verwendet. Alle gefundenen Wege können bei 2. optimiert werden

Zu 2.

Ich verbessere den Weg indem ich 1 bis n verbundene Punkte nehme und sie an allen anderen möglichen Positionen im Weg wieder einfüge und dabei überprüfe, ob die Gesamtdistanz kleiner wird und noch alle Winkel zulässig sind. Diese Verbesserung wird für alle gefundenen Wege angewendet. Damit ergibt sich eine Zeitkomplexität von $O(n^5)$ (Anzahl gefundene Wege * Anzahl Aufruf der Verbesserung * ausgesuchte Wege * einfügen an bestimmter Position * tatsächliches Einfügen, Gesamtdistanz und Winkel berechnen; jeder dieser Faktoren entpricht n).

Zur Performance der verschiedenen Varianten:

Die erste und zweite Variante geben ungefähr gleich viele Wege zurück. Die dritte Variante gibt nicht immer einen Weg zurück, was warscheinlich daran liegt, dass, wenn viele einzelne Weg-Abschnitte existieren, es unwarscheinlicher ist, dass sie sich zusammenfügen können unter erlaubten Abbiege Winkeln. Diese Wege sind auch meistens länger, da um die Abbiege-Winkel-Regel einzuhalten auch weiter entfernte Weg-Abschnitte kombiniert werden.

Anmerkung zur Zeitkomplexität:

Wenn der Backtracking Algorithmus verwendet wird ohne das die Wege nachher optimiert werden, ist die Zeitkomplexität trotzdem $O(n^3)$, da ich, bevor der Backtracking Algorithmus vewendet wird, für alle möglichen Tripel von Punkten die Winkel ausrechne und diese dann im Backtracking Algorithmus verwende. Man könnte aber auch darauf verzichten und hätte eine $O(n^2)$ Zeitkomplexität.

Umsetzung

Ich habe den Hauptteil der Aufgabe in Rust und einzelne Teile, wie das grafische Darstellen eines Weges in Python implementiert. Die Einzelteile werden in einem Bash script zusammengeführt (das Programm kann nur ausgeführt werden, wenn run.sh, program, und program2 executable permissions bekommen, ich habe das Ganze auf 2 verschiedenen Geräten mit Linux ausprobiert)

Ich habe am Anfang des Programmes ein 3D Vector mit booleans für die Winkel, und einen 2D Vector mit f64 für die Distanzen zwischen den Punkten angelegt. Diese beiden Vectoren werden der solve function übergeben (siehe Quellcode Abschnitt). Der Parameter „rand_mode“ gibt an, ob die Reihenfolge der Startknoten zufällig sein soll, „fast_mode“, ob die Funktion nachdem sie einen Weg gefunden hat, verlassen werden soll. Für den Algorthmus an sich wird ein visited HashSet verwendet, das speichert, welche Knoten schon besucht wurden und ein Stack, der die Reihenfolge der Knoten speichert. Die beiden äußeren Schleifen iterieren über alle Paare von Startknoten, der innere Loop is der eigentliche Backtracking Algorithmus. Damit ein Knoten als nächster Knoten ausgewäht werden kann muss er verschiedene Kriterien erfüllen (siehe if else ladder), z.B muss der

Winkel zu den vorherigen Knoten erlaubt sein, er darf nicht im visited HashSet sein oder muss in der Reihenfolge nach dem Knoten kommen wenn gerade ein backtracking step ausgeführt wurde. Wenn kein passender Knoten gefunden wird wird ein Backtrackinf step ausgeführt, der letzte Knoten vom Stack und aus visited gelöscht.

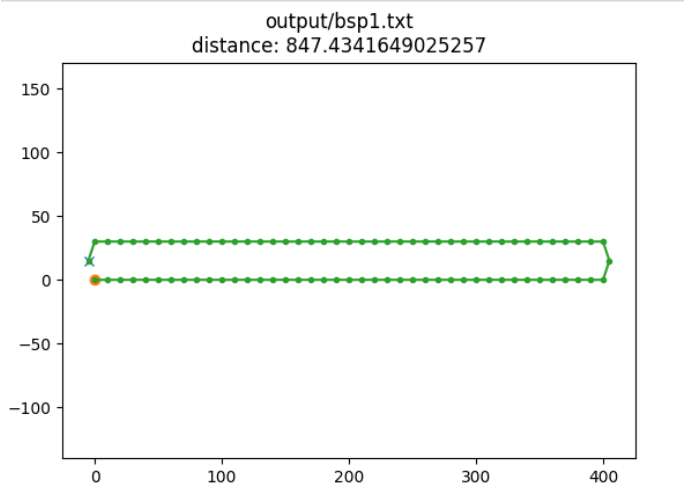
Die funktion opt1_move_n_points führt den Hauptteil der Optimierung durch.

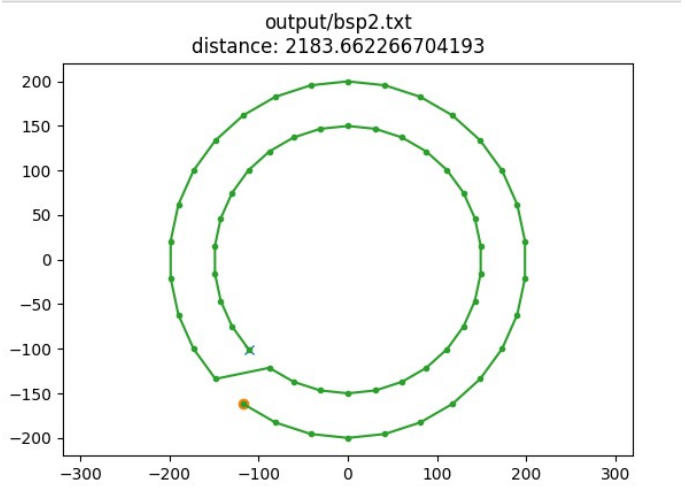
Beispiele

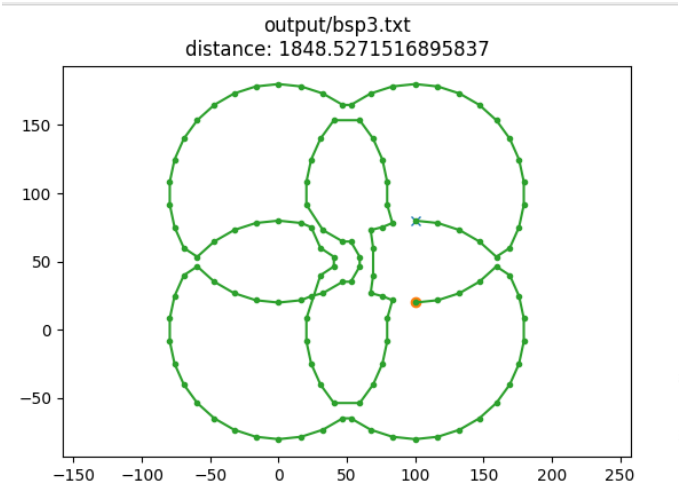
Ich hab für die Visualisierung der Ausgabe den Graph mit der Python library matplotlib gezeichnet:

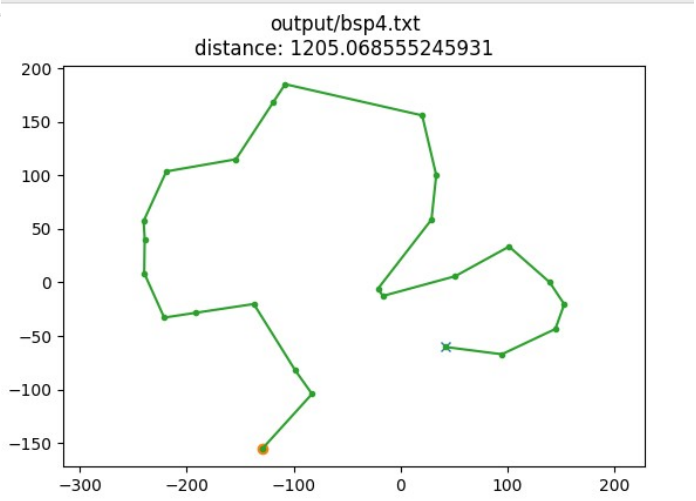
Wege mit Optimierung:

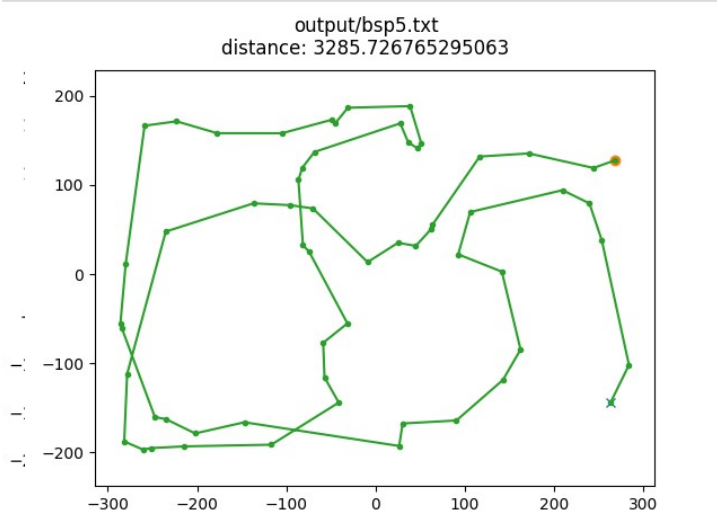
```
$ ./run.sh
program 1 or 2: 1
"all" fuer alle Beispiele oder name eines files im testcase folder: all
return first valid path and do not compare paths from other startpoints Y / N: N
choose startpoints at random Y / N: Y
skip optimization Y / N: N
fast mode: false
rand mode: true
skip_opt: false
Zeit für den Test: 7.301035453s
Zeit für den Test: 1.516477366s
Zeit für den Test: 39.329444977s
Zeit für den Test: 47.901814ms
Zeit für den Test: 759.418352ms
Zeit für den Test: 2.874049609s
Zeit für den Test: 5.624392695s
Zeit für alle Tests: 57.453198624s
```

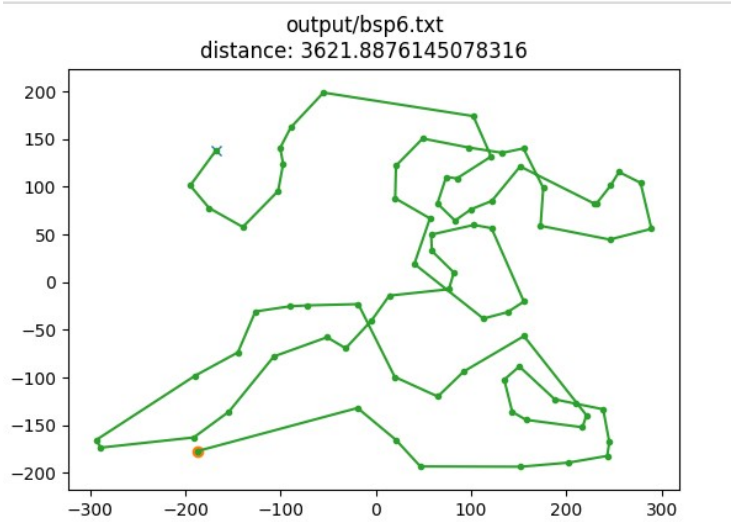


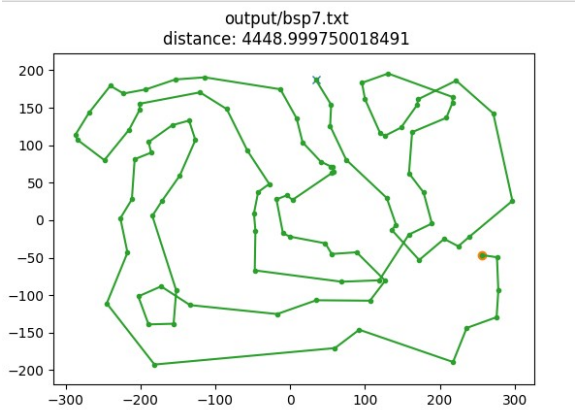



output/bsp3.txt
distance: 1848.5271516895837


output/bsp4.txt
distance: 1205.068555245931

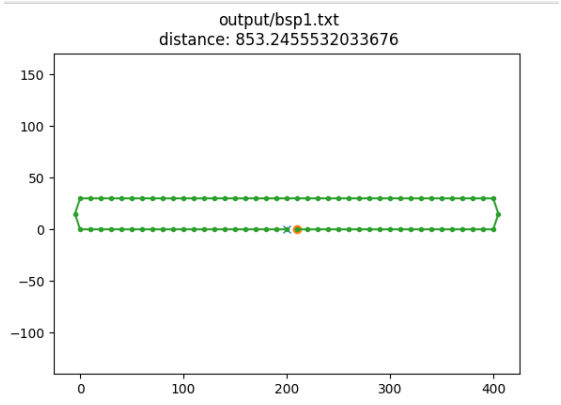

output/bsp5.txt
distance: 3285.726765295063

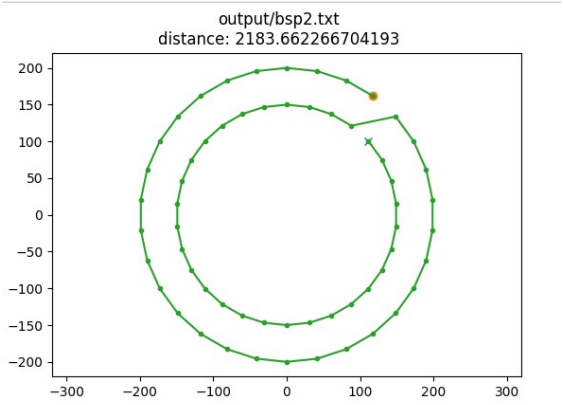

output/bsp6.txt
distance: 3621.8876145078316

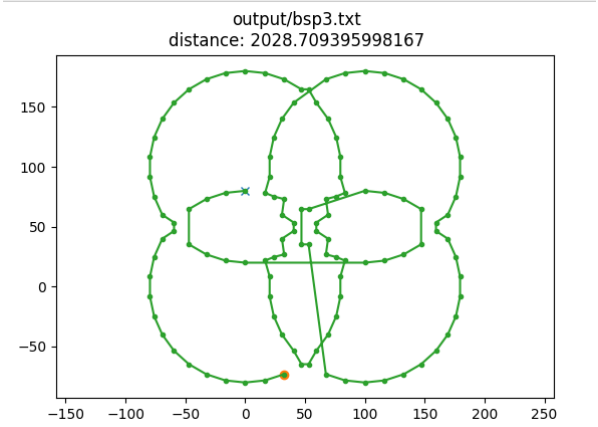

output/bsp7.txt
distance: 4448.999750018491

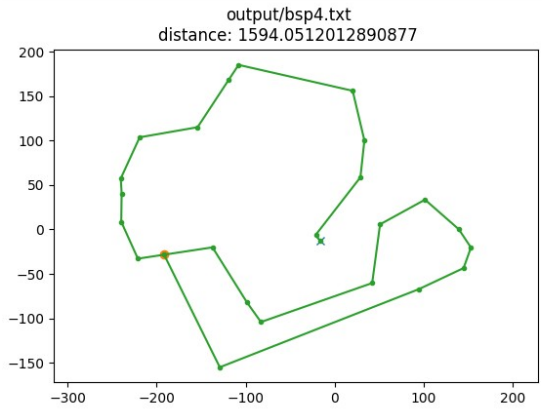
Wege ohne Optimierung

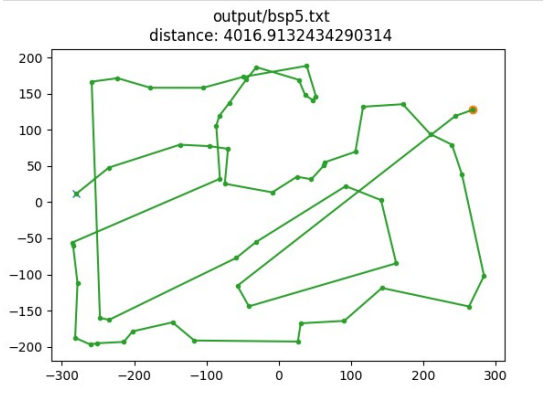
```
$ ./run.sh
program 1 or 2: 1
"all" fuer alle Beispiele oder name eines files im testcase folder: all
return first valid path and do not compare paths from other startpoints Y / N: Y
choose startpoints at random Y / N: N
skip optimization Y / N: Y
fast mode: true
rand mode: false
skip_opt: true
Zeit für den Test: 8.058174ms
Zeit für den Test: 1.886849ms
Zeit für den Test: 24.07342ms
Zeit für den Test: 3.47195ms
Zeit für den Test: 15.407479ms
Zeit für den Test: 20.00269ms
Zeit für den Test: 302.379474ms
Zeit für alle Tests: 375.627018ms
```

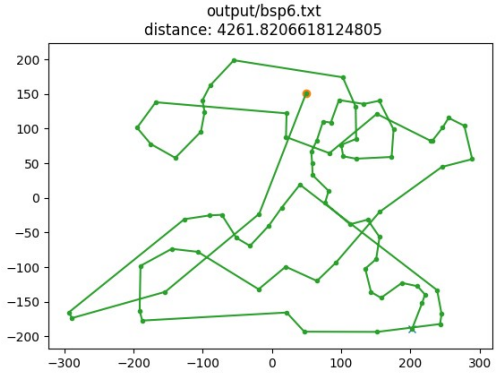


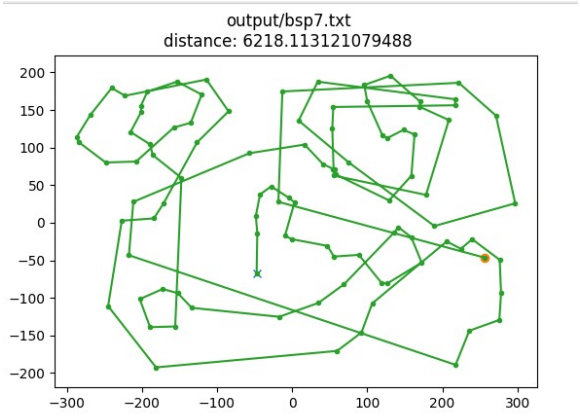






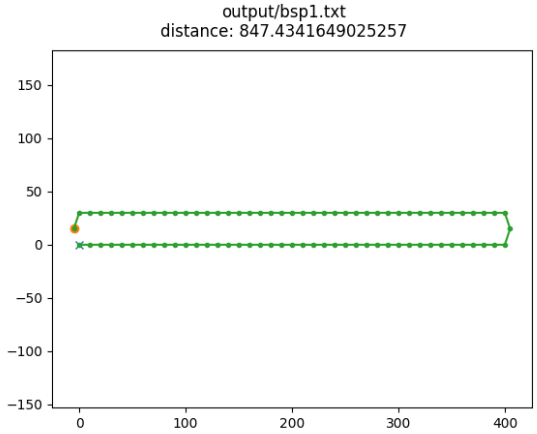


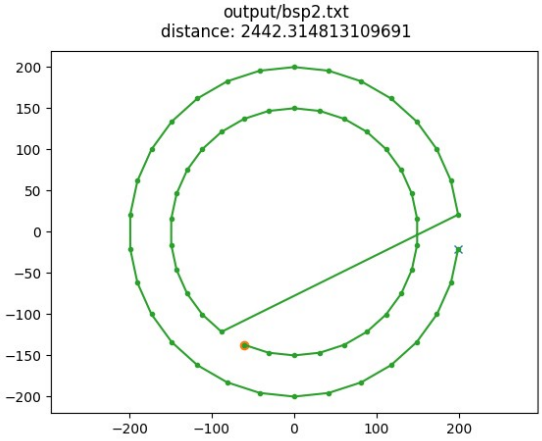


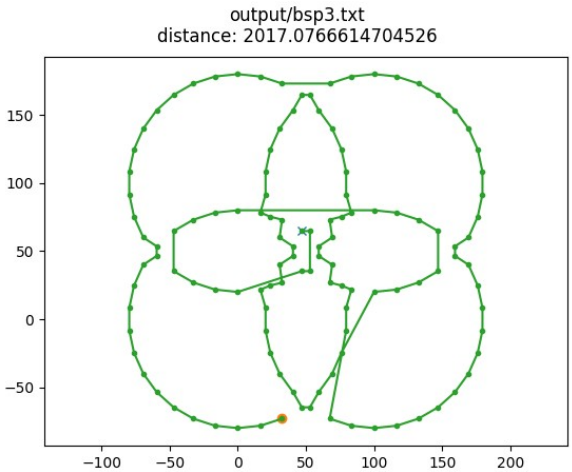


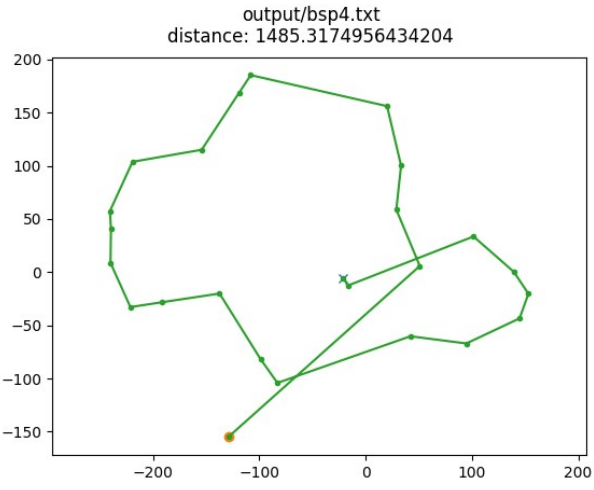
Wege mit Program 2

`Program 2 findet nur für die Tests 1 – 4 eine Lösung, in normaler Zeit`









Quellcode

```
fn solve_greedy0(
    n: i32,
    ad: &Vec<Vec<f64>>,
    aa: &Vec<Vec<Vec<bool>>>,
    rand_mode: bool,
    fast_mode: bool,
) -> Vec<(f64, Vec<usize>)> {
    let mut valid_paths: Vec<(f64, Vec<usize>)> = Vec::new(); // container for
all valid paths
    let n: usize = n as usize;
    let mut stack: Vec<usize>; // current path
    let mut visited: HashSet<usize> = HashSet::with_capacity(n); // visited
points
    let mut start_points: Vec<usize> = (0..n).collect(); // all start points
    if rand_mode {
        start_points.shuffle(&mut rand::thread_rng());
    } // shuffle if rand mode is on
    let mut action_count: i32 = 0; // counter for cut action count
    // loop over all possible start points
    'pathfinder: for sp1 in start_points {
        // initialize start point with start point
        stack = Vec::with_capacity(n);
        stack.push(sp1);
        visited.insert(sp1);
        // order all second point by distance to the first point
        let ordered_distances: Vec<usize> = get_ordered_distances(&ad[sp1]);
        for sp2 in ordered_distances {
            if sp1 == sp2 {
                continue;
            }
            // add second starting point to visited and stack
            stack.push(sp2);
            visited.insert(sp2);
            let mut backwards: bool = false; // tracks if the last set point
should be removed
            let mut last_on_place: usize = sp1; // holds the last set point
after it was removed
            // Backtracking loop
            loop {
                if backwards {
                    if stack.len() == 2 {
                        // there exists no path for the two starting points
                        break;
                    } else {
                        // set last_on_place
                        last_on_place = stack.pop().unwrap();
                        visited.remove(&last_on_place);
                        backwards = false;
                    }
                } else {
                    // search for next point
                    // set last distance to 0 if there was no point at this
position before,
                    // else to the distance between the last point and the
previous in the stack
```

```
                    let last_distance: f64 = {
                        if last_on_place == sp1 {
                            0.0
                        } else {
                            ad[*(stack.last().unwrap())][last_on_place]
                        }
                    };
                    let mut best: usize = n;
                    let last_num: usize = stack[stack.len() - 1]; // previous
points on stack
                    let last_last_num: usize = stack[stack.len() - 2];
                    for next_point in 0..n {
                        if visited.contains(&next_point) {
                            // skip when point is already visited
                            continue;
                        } else if !aa[last_last_num][last_num][next_point] {
                            // skip if the angle is too small
                            continue;
                        } else if last_distance > ad[last_num][next_point] {
                            // skip cause distance of last choosen before
backwards is longer
                            continue;
                        } else if last_distance == ad[last_num][next_point]
                            && last_on_place >= next_point
                        {
                            // skip because this next_point was already checked
                            continue;
                        } else if best != n && ad[best][last_num] <
ad[next_point][last_num] {
                            // skip cause this is worse than the best distance
                            continue;
                        } else if best != n
                            && ad[best][last_num] == ad[next_point][last_num]
                            && next_point > best
                        {
                            // if distances are the same the point with lower
index should be choosen
                            continue;
                        } else {
                            // better next point was found
                            best = next_point;
                        }
                    }
                    // apply next point
                    if best == n {
                        // no new point so backward
                        backwards = true;
                        if CUT_ACTION_COUNT {
                            // check if max action count was reached
                            action_count += 1;
                            if action_count == MAX_ACTION_COUNT {
                                // restart with a new starting point
                                visited.clear();
                                action_count = 0;
                                continue 'pathfinder;
                            }
                        }
```

```
                    } else {
                        last_on_place = sp1; // reset last_on_place
                        stack.push(best); // append best to path and visited
                        visited.insert(best);
                    }
                    // check if solution was found
                    if stack.len() == n {
                        // add found path to valid paths
                        let tup: (f64, Vec<usize>) =
(calculate_path_length(&stack, &ad), stack);
                        valid_paths.push(tup);
                        if fast_mode {
                            // If fast mode is on return after first found valid
path
                            return valid_paths;
                        }
                        continue 'pathfinder;
                    }
                }
            }
            // remove second point
            visited.remove(&sp2);
            stack.pop();
        }
        // remove first point
        visited.remove(&sp1);
    }
    // return all found valid paths
    valid_paths
}

fn opt1_move_n_points(
    total_distance: &mut f64,
    stack: &mut Vec<usize>,
    ad: &Vec<Vec<f64>>,
    aa: &Vec<Vec<Vec<bool>>>,
    n: usize,
) {
    let last_distance: f64 = *total_distance; // keep track of the last total
distance
    // iterate over all possible points to remove from the stack
    'outer: for choosen_point in 0..(stack.len() + 1 - n) {
        // remove a slice of n points
        let mut slice: Vec<usize> = stack
            .splice(choosen_point..(choosen_point + n), std::iter::empty())
            .collect::<Vec<_>>();
        // iterate over all possible locations to insert the removed points
        for new_location in 0..(stack.len() + 1) {
            // try inserting the slice in the new location, and its reverse
            for _ in 0..2 {
                // insert the removed points into the new position
                stack.splice(new_location..new_location, slice);
                // check if all angles are valid
                if proove_all_angles(&stack, aa) {
                    let distance = calculate_path_length(&stack, ad);
                    // if the new path is shorter, update the total distance and
the stack
```

```
                    if distance < *total_distance {
                        *total_distance = distance;
                        //*stack = new_stack;
                        break 'outer;
                    }
                }
                // remove the inserted points from the stack and reverse them
                slice = stack
                    .splice(new_location..(new_location + n),
std::iter::empty())
                    .collect::<Vec<_>>();
                slice.reverse();
            }
        }
        // insert the removed points back to their original position
        stack.splice(choosen_point..choosen_point, slice);
    }
    // if the total distance was updated, repeat the optimization
    if last_distance != *total_distance {
        opt1_move_n_points(total_distance, stack, ad, aa, n);
    }
}
```