Aufgabe 1: Weniger krumme Touren

Teilnahme-ID: 64910

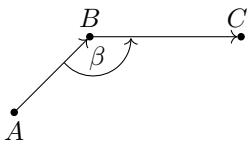
Bearbeiter dieser Aufgabe:
Jan Neuenfeld

16. April 2023

1 Lösungsidee

Die Aufgabe ist, einen Weg durch eine Punktewolke zu finden, bei der jeder Punkt genau einmal besucht wird und der Winkel zwischen zwei Wegen größer als 90° ist. Dabei ergeben sich im Endeffekt zwei Problemstellungen:

1.1 Wann ist ein Punkt erreichbar?



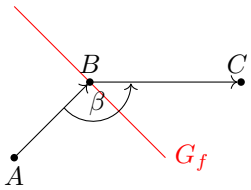
Anstatt die Größe des Winkels auszurechnen und dann zu vergleichen, ob er zwischen 90° und 270° liegt, wird bei meiner Lösung der Term der Grenzlinie errechnet und dann verglichen, ob A und C auf verschiedenen Seiten dieser Grenzgeraden liegen. Dies funktioniert, da die genaue Größe von β nicht relevant ist. Auch ist anzunehmen, dass die Berechnung sehr oft durchgeführt werden wird, und deshalb möglichst performant sein sollte.

$$f(x) = -1 \div \frac{y_B - y_A}{x_B - x_A} \cdot x$$

Hier stellt $\frac{y_B - y_A}{x_B - x_A}$ die Steigung von $\overline{AB}$ dar. Indem man -1 durch dieses Ergebnis dividiert, erhält man die Steigung der Normalen auf eine Gerade durch A und B. Dennoch ist dieser Term noch nicht vollständig, da er bisher nur eine Steigung beschreibt. Um den eigentlichen Term zu erhalten, muss man x_B von x subtrahieren, da der gesamte Graph dadurch um x_B nach rechts verschoben wird. Dann braucht man nur noch eine horizontale Verschiebung um y_B bis die Gerade durch B führt.

$$f(x) = (-1 \div \frac{y_B - y_A}{x_B - x_A}) \cdot (x - x_B) + y_B$$

Wenn nun also $f(x_C)$ kleiner ist als x_C können wir schlussfolgern, dass C oberhalb von G_f liegen muss.



Nachdem man mit dem gleichen Verfahren bestimmt hat ob A über- oder unterhalb der Geraden liegt, ist nur noch zu überprüfen ob die Punkte auf verschiedenen Seiten von G_f liegen. Wenn das der Fall ist, liegt β zwischen 90° und 270°, womit bestätigt wird, dass nach einer Bewegung von A nach B, der Punkt C mit den gegebenen Einschränkungen noch erreichbar ist.

1.2 Wie finde ich einen Weg durch die Punktewolke?

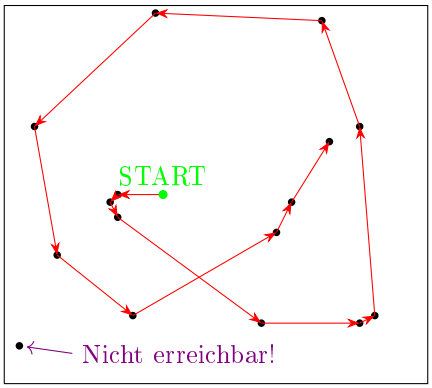
Man kann natürlich einfach jeden möglichen Weg ausprobieren, ein entsprechender Algorithmus hat aber eine Laufzeit von etwa $\mathcal{O}(n!)$. Somit könnte man die gegebenen Beispiele von bis zu 120 Punkten nicht in endlicher Zeit lösen. Deshalb ist die Frage nach der Laufzeit hier eigentlich die Entscheidendere. Dennoch bietet es sich an, den Algorithmus trotzdem zu implementieren, um ein Gefühl für die Umstände und Laufzeiten zu bekommen.

Vollständige Tiefensuche

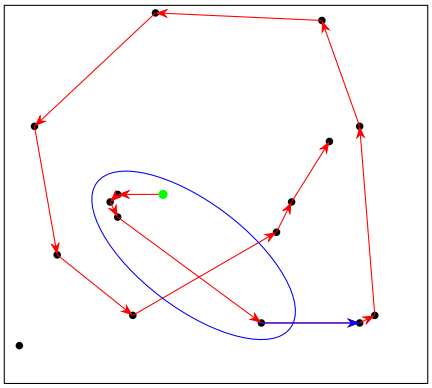
Wie schon angemerkt, ist die Laufzeit einer uneingeschränkten Tiefensuche sehr lang. Der Vorteil ist jedoch, dass man einige Informationen sammeln kann. Dieser Algorithmus legt eine wichtige Grundlage zur Bewertung aller anderen Algorithmen, da er garantiert den kürzesten Weg findet.

Caching?

Bevor ich zu den etwas besseren Algorithmen übergehe, möchte ich hier noch auf die Frage eingehen, ob eine rekursive, uneingeschränkte Tiefensuche sich mit Caching vielleicht etwas beschleunigen lässt.
Spezifisch in Hinblick auf diese Aufgabe bedeutet Caching, dass das Programm einen Weg, der schon „gegangen“ wurde, merkt und, wenn die gleiche Ausgangssituation ein weiteres Mal entsteht, auf die Daten zurückgreifen kann. So kann schnell aus dem Speicher abgerufen werden ob es sich überhaupt noch lohnt auf einem Pfad weiter zu rechnen. In unserem Fall müssen allerdings sehr strenge Kriterien erfüllt sein bevor das Programm auf seine vorhergehenden Ergebnisse zurückgreifen kann, um sicher zu gehen, dass nicht aus Versehen ein Weg als unmöglich angesehen wird, der eigentlich zur Lösung geführt hätte, oder anders herum.

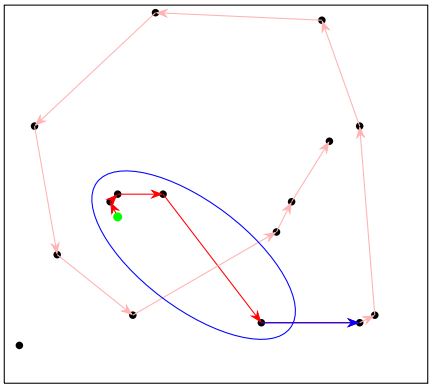


Hier würde also der Algorithmus am Ende nicht mehr weiter kommen und dieses Ergebnis in allen Variationen speichern.
Zum Beispiel:



Hier wird die Menge der sich in der blauen Ellipse befindlichen Punkte ohne Reihenfolge zusammen mit dem blau markierten Weg gespeichert. Sollte der Algorithmus zu einem anderen Zeitpunkt wieder den blauen Weg gehen, wird die Menge der von ihm besuchten Punkte mit dem Cache verglichen. Da das

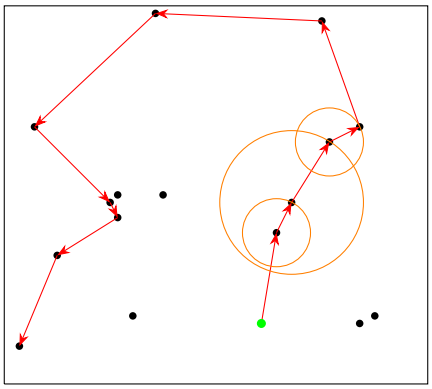
gleiche Szenario dort mit einem Ergebnis gespeichert ist, kann der Durchlauf sofort abgebrochen werden, weil schon bekannt ist, dass der Weg zu keiner Lösung führen kann.



Hier wird eine ganze Menge Rechenzeit gespart. Dennoch gibt es einige Nachteile des Verfahrens. Zum einen ist sehr viel Arbeitsspeicher vonnöten, da Tausende von Ergebnissen immer wieder gespeichert werden müssen. Auch braucht das ständige Vergleichen und Speichern von Punktemengen viel Rechenzeit. Somit ergibt sich auch für diesen Algorithmus im Worst Case die Laufzeitkomplexität $\mathcal{O}(n!)$. In der Realität hängt es stark von der Konstellation der Punkte ab wie lange dieser Algorithmus braucht, so dass die Laufzeit manchmal nur ein Bruchteil des Algorithmus ohne Cache beträgt, aber manchmal auch mehr. Bei ersten Tests hat sich kein entscheidender Vorteil gezeigt, die Idee könnte allerdings mit der Implementierung von Multithreading weiterverfolgt werden.

Klare Kriterien

Kriterien, die zu jedem Zeitpunkt den besten nächsten Punkt bestimmen können, konnte ich leider nicht finden. Der nächstliegende Ansatz wäre allerdings, immer den Punkt zu wählen, dessen Distanz zum aktuellen Punkt am geringsten ist. Dies führt theoretisch zu einer Laufzeitkomplexität von $\mathcal{O}(n)$. Da aber der Startpunkt dadurch nicht bestimmt werden kann, muss der Algorithmus einmal für jeden Punkt durchgeführt werden, was zu einer fast quadratischen Laufzeit von $\mathcal{O}(n \cdot (n-1))$ führt. Um die Wahrscheinlichkeit zu erhöhen, dass der Algorithmus auch tatsächlich einen Weg findet und entlegene Punkte erreicht werden, wird auch für den zweiten Punkt des Weges noch jede Möglichkeit ausprobiert, was einen zu einer letztendlichen Laufzeit von $\mathcal{O}(n \cdot (n-1) \cdot (n-2))$ führt. Ein Beispiel, bei dem die ersten zwei Punkte schon vorgegeben sind, sieht etwa so aus:



Einige Tests haben allerdings gezeigt, dass nur bei etwa 40% von zufällig erstellen Punktewolken, die lösbar wären, eine Lösung gefunden wird. Auch ist es im Gegensatz zu den anderen Algorithmen nicht garantiert, dass der kürzeste Weg gefunden wird.

Geschwindigkeit

Bei mehr als 200 Punkten kommt leider auch dieser Algorithmus an seine Grenzen. Was also wenn man mehr Punkte in seiner Punktewolke hat, oder besonders schnell ein Ergebnis braucht? Bisher wurden immer mindestens $n \cdot (n-1)$ verschiedene Wege ausprobiert und dann der Kürzeste herausgesucht.

Um die Berechnung zu beschleunigen kann man natürlich das Programm stoppen, sobald es die erste Lösung gefunden hat, was allerdings in vergleichsweise langen Wegen resultiert. Da eine Sicherheit von 40% unzureichend ist, lohnt es sich hier den ersten und letzten Algorithmus zu verbinden. Während ursprünglich bei dem ersten Algorithmus die Punkte in zufälliger Reihenfolge durchprobiert wurden, kann man diese stattdessen nach dem Kriterium des letzten Algorithmus sortieren, also die Vorteile beider Algorithmen nutzen. In einem Worst-Case-Szenario beträgt die Laufzeit noch immer $\mathcal{O}(n!)$, der Durchschnitt liegt allerdings bei etwa $\mathcal{O}((\frac{n}{2})!)$.

2 Umsetzung

An dieser Stelle sei anzumerken, dass jeglicher hier besprochene Algorithmus in der Programmiersprache Rust implementiert wurde. Um eine angenehme Schnittstelle zum Benutzer darzustellen, kann jeder der vier Algorithmen mit einer einzelnen Funktion aufgerufen werden, der einfach der gewünschte Modus und die Punktewolke übergeben wird.

```
1:  function RUN(input, mode)                          ▷ Stark vereinfacht
2:      res ← f32 :: MAX                               ▷ Der größtmögliche 32 Bit Float
3:      for point in points do
4:          new_res ← point.start(input, mode)
5:          if new_res < res then                      ▷ res repräsentiert hier die Länge des Weges
6:              res ← new_res
7:          end if
8:      end for
9:  return res                                         ▷ Der kürzeste gefundene Weg wird zurückgegeben
10: end function
```

Diese Funktion besteht hauptsächlich aus der Verarbeitung des Inputs und dem Sammeln von Randinformationen wie die Dauer der Berechnung. Auch wird hier schon der erste Punkt festgelegt, dessen „start“ Methode aufgerufen wird. Außerdem wird hier der Modus übergeben, da die Unterscheidung zwischen Algorithmen erst später stattfindet. Diese Methode ist sehr ähnlich zu der Funktion „run“, weshalb ich sie nicht in Pseudocode darstellen werde. Der größte Unterschied ist, dass in „start“ zwischen den Modi unterschieden wird, und abhängig von diesem entweder die Methode „check“, „check_all“, „check_cached“ oder „check_optimal“. Der Grund warum ein Punkt in einem Struct statt in einem Tupel gespeichert wird, liegt hauptsächlich in der Verbesserung der Lesbarkeit des Codes. Die Menge an Punkten wird in einem Vector, also einem dynamischen Array, gespeichert, da die Größe der Punktewolke während des Kompilierens noch nicht bekannt ist. Auch werden die Elemente eines Vectors in der Reihenfolge gespeichert, in der sie hinzugefügt werden. Dadurch wird das Nachvollziehen des verfolgten Weges sehr einfach. „check“ ist eine rekursive Funktion, das heißt sie ruft sich selber auf. Um aus diesem unendlichen Kreis auszubrechen, muss eine Abbruchsbedingung bestehen, die die Funktion vor dem Selbstaufruf beendet, wie in Zeile 3-5 und Zeile 11-13 zu sehen. In Zeile 4 wird der zurückgelegte Weg mit 0 bezeichnet, da die tatsächliche Distanz aufaddiert wird während die Rekursion verlassen wird (Zeile 17).

```
1:  function CHECK(self, points, visited, last)
2:      visited.push(self)                             ▷ Der Liste „visited“ wird „self“ hinzugefügt
3:      if visited.length() == points.length() then    ▷ Abbruchbedingung der Rekursiven Funktion
4:          return (0, visited)                        ▷ 0 stellt die Länge des Weges dar
5:      end if
6:      for point in points do
7:          if point.is_valid() and not point in visited then   ▷ is_valid(): Siehe Kapitel 1.1
8:              possible.push(point)
9:          end if
10:     end for
11:     if possible.is_empty() then                    ▷ Sackgasse
12:         return Error
13:     end if
14:     closest ← closest element in possible          ▷ Siehe Kapitel 1 „Klare Kriterien“
15:     res ← closest.check(points, visited, self)     ▷ Der rekursive Aufruf
16:     if res == Ok then
```

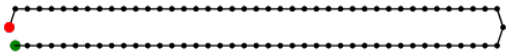
```
17:         return res + self.distance(closest)        ▷ Die Länge des Weges wird aufaddiert
18:     else                                           ▷ Siehe Z.12
19:         return Error
20:     end if
21: end function
```

Jeder der anderen Algorithmen ist sehr ähnlich zu diesem implementiert. Bei der Vollständigen Tiefensuche ist der einzige Unterschied, dass nicht der nächste Punkt gesucht wird, sondern durch die gesamte Liste an Punkten iteriert wird, und das beste Ergebnis zurückgegeben wird. Bei „Geschwindigkeit“ wird *possible* nach Entfernung sortiert, allerdings wird die rekursive Funktion auf jedes Element der Liste aufgerufen und das erste Ergebnis zurückgegeben.

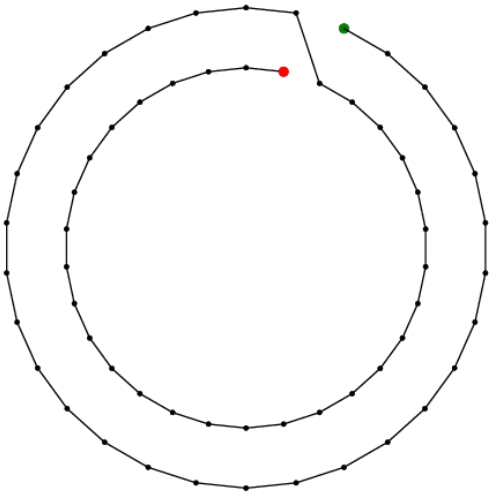
```
1: function CHECK_OPTIMAL(self, points, visited, last)
2:     do_stuff()                                     ▷ Siehe Zeile 2-13 oben
3:     possible.sort()          ▷ Die möglichen Punkte werden nach Distanz zu self sortiert
4:     for i in possible do
5:         res ← i.check_optimal(points, visited, self)
6:         if res == Ok then
7:             return res + self.distance(closest)
8:         end if
9:     end for
10: end function
```

3 Beispiele

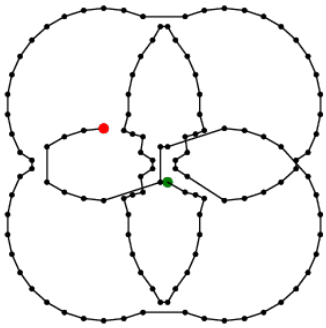
Im Weiteren werden die berechneten Wege für die gegebenen Beispiele dargestellt. Dabei stellt der grüne Punkt jeweils den Start- und der rote Punkt den Endpunkt dar. Außerdem ist unter dem Bild die berechnete Distanz und die Berechnungszeit auf einem AMD Ryzen 5 Prozessor angegeben. Der benutzte Algorithmus ist der in Kapitel 1 „Klare Kriterien“ beschriebene. Während bei den Beispielen 1 bis 4 offensichtlich ein ziemlich optimaler Weg gefunden wurde, ist bei den Beispiele 5, 6 und 7 ein wesentlicher Nachteil des Algorithmus zu erkennen. Die Abstände zwischen den Punkten werden gegen Ende des Weges deutlich länger, was darauf hinweist, dass der Weg nicht optimal ist.



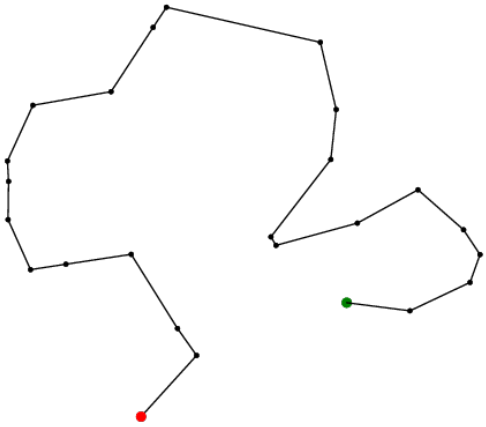
Beispiel 1: $847,4342\,km$, $219\,ms$



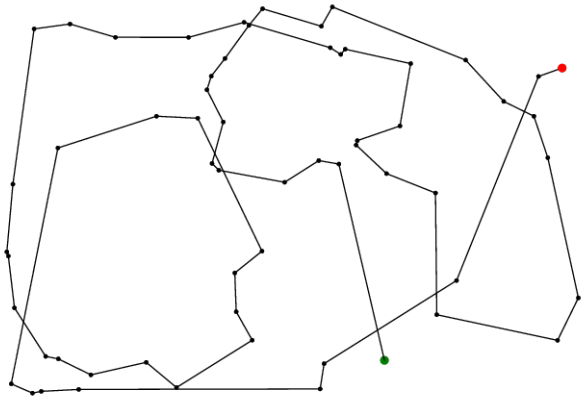
Beispiel 2: $2183,662\,km,\ 36\,ms$



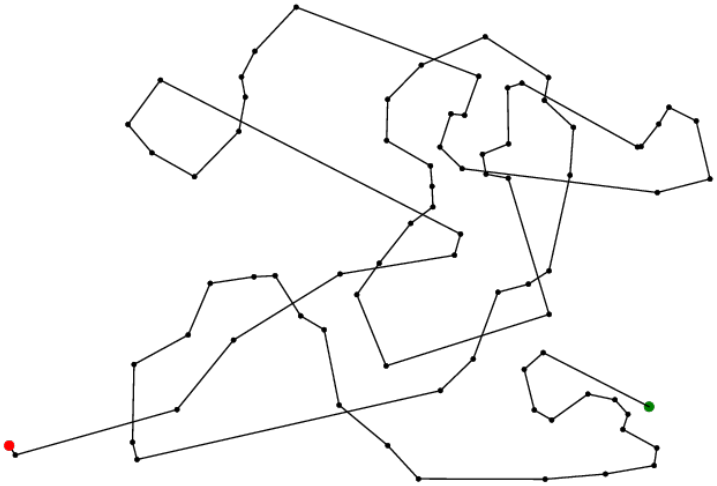
Beispiel 3: $1944,605\,km,\ 1041\,ms$



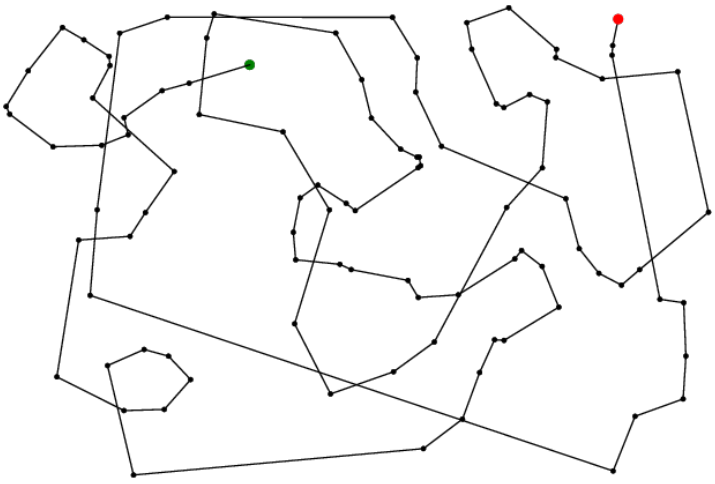
Beispiel 4: $1205,0686\,km,\ 2\,ms$



Beispiel 5: $3854,2773\,km$, $35\,ms$



Beispiel 6: $4052,955\,km$, $81\,ms$



Beispiel 7: $5035,684\,km$, $253\,ms$

4 Quellcode

run.rs

```
pub fn run(input: In, mode: Mode) -> (f32, Vec<(f32, f32)>, Duration) {
    let time = Instant::now();
    // setting this to false results in more efficient Caching,
    // but turns off multithreading
    let multithreaded_cache = false;
    // its possible to pass a Vec<(f32, f32)> and a path
    let points: Vec<Dot> = match input {
        In::AVec(the_points) => the_points.into_iter()
            .map(|(x,y)| Dot { x, y })
            .collect(),
        In::AFile(ref filename) => {
            let file = File::open(filename)
                .expect(format!("can not open file {filename}").as_str());

            BufReader::new(file).lines()
                .filter_map(|l| l.ok())
                .map(parse_line)
                .collect()
        },
    };

    if mode == Mode::Cached && !multithreaded_cache {
        let res = start_cached(&points);
        return (res.0, Vec::from_iter(res.1.iter().map(|a| a.get_infos())), time.elapsed())
    }

    let mut results = Vec::new();
    for i in points.clone() {
        let p = points.clone();
        results.push(thread::spawn(move || i.start(&p, mode)))
    }

    let mut best: (f32, Vec<Dot>) = (f32::MAX, Vec::new());

    for i in results {
        let res = i.join().unwrap().unwrap();
        if res.0 < best.0 {
            best = res;
            // delete this if you want shorter paths, let it be for faster results
            if mode == Mode::Optimal {
                return (
                    best.0,
                    Vec::from_iter(best.1.iter().map(|a| a.get_infos())), time.elapsed()
                )
            }
        }
    }

    if best.1.is_empty() {
        return (0., Vec::new(), time.elapsed());
    }
    return (best.0, Vec::from_iter(best.1.iter().map(|a| a.get_infos())), time.elapsed())
}
```

necessaries.rs

```
impl Dot {
    pub fn start<'a>(&'a self, points: &'a Vec<Dot>, mode: Mode) -> Result<(f32, Vec<Dot>), i32> {
        let mut best: (f32, Vec<Dot>) = (f32::MAX, Vec::new());
        let visited: Vec<Dot> = vec![*self];
        let mut buff: Vec<(Vec<Dot>, (Dot, Dot))> = Vec::new();

        for point in points {
            if point == self {continue}
            if let Ok(res) = match mode {
```

```
                Mode::Normal => point.check(&points, visited.clone(), *self),
                Mode::Expensive => point.check_all(&points, visited.clone(), *self),
                Mode::Cached => point.check_cached(&points, visited.clone(), *self, &mut buff),
                Mode::Optimal => point.check_optimal(&points, visited.clone(), *self)
            } {
                if res.0 + self.distance(point) < best.0 {
                    best = (res.0 + self.distance(point), res.1);
                    if mode == Mode::Optimal {break;}
                }
            };
        }
        Ok(best)
    }

    // responsible for most of the runtime
    fn check<'a>
    (
        &'a self,
        points: &'a Vec<Dot>,
        mut visited: Vec<Dot>,
        last: Dot
    ) -> Result<(f32, Vec<Dot>), i32> {

        visited.push(*self);

        // a result is found if theres no points left to visit
        if visited.len() == points.len() {return Ok((0., visited))}

        let mut possible = Vec::new();

        for i in points {
            if !visited.contains(&i) && is_valid(last, *self, *i){
                possible.push((i.distance(self), i));
            }
        }

        if possible.is_empty() {return Err(1)}

        possible.sort_by(|a, b| a.0.total_cmp(&b.0));

        let perfect = possible.get(0).unwrap().1;

        if let Ok(res) = perfect.check(points, visited, *self) {
            return Ok((res.0 + self.distance(perfect), res.1));
        } else {
            return Err(1)
        }
    }

    fn check_all<'a>
    (
        &'a self,
        points: &'a Vec<Dot>,
        mut visited: Vec<Dot>,
        last: Dot
    ) -> Result<(f32, Vec<Dot>), i32> {

        visited.push(*self);

        // a result is found if theres no points left to visit
        if visited.len() == points.len() {return Ok((0., visited))}

        let mut perfect = &Dot { x: 0., y: 0. };
        let mut perfect_path = (f32::MAX, Vec::new());

        for point in points {
            if !visited.contains(&point) && is_valid(last, *self, *point) {
                // "if let" unstable in connection with other expressions, nested if's necessary
                if let Ok(res) = point.check_all(points, visited.clone(), *self) {
                    if res.0 < perfect_path.0 {
                        perfect_path = res;
                        perfect = point;
                    }
```

```
                }
            }
        }

        if perfect_path.1.is_empty() {
            return Err(1)
        }
        return Ok((perfect_path.0 + self.distance(perfect), perfect_path.1));
    }

    fn check_optimal<'a>
    (
        &'a self,
        points: &'a Vec<Dot>,
        mut visited: Vec<Dot>,
        last: Dot
    ) -> Result<(f32, Vec<Dot>), i32> {

        visited.push(*self);

        // a result is found if theres no points left to visit
        if visited.len() == points.len() {return Ok((0., visited))}

        let mut possible = Vec::new();

        for i in points {
            if !visited.contains(&i) && is_valid(last, *self, *i){
                possible.push((i.distance(self), i));
            }
        }

        if possible.is_empty() {return Err(1)}

        possible.sort_by(|a, b| a.0.total_cmp(&b.0));

        for i in 0..possible.len() {
            if possible.len() > i {
                if let Ok(a) = possible[i].1.check_optimal(points, visited.clone(), *self) {
                    return Ok((a.0 + self.distance(possible[i].1), a.1));
                }
            }
        }

        return Err(1)
    }

    fn check_cached<'a>(
        &'a self,
        points: &'a Vec<Dot>,
        mut visited: Vec<Dot>,
        last: Dot,
        buff: &mut Vec<(Vec<Dot>, (Dot, Dot))>
    ) -> Result<(f32, Vec<Dot>), i32> {

        visited.push(*self);

        let mut a = visited.clone();
        a.sort_by(|a, b| (a.x, a.y).partial_cmp(&(b.x, b.y)).unwrap());
        if buff.contains(&(a.clone(), (*self, last))) {
            return Err(1)
        }

        // a result is found if theres no points left to visit
        if visited.len() == points.len() {return Ok((0., visited))}

        let mut perfect = &Dot { x: 0., y: 0. };
        let mut perfect_path = (f32::MAX, Vec::new());

        for point in points {
            if !visited.contains(&point) && is_valid(last, *self, *point) {
                // "if let" unstable in connection with other expressions, nested if's necessary
                if let Ok(res) = point.check_cached(points, visited.clone(), *self, buff) {
                    if res.0 < perfect_path.0 {
```

```
                        perfect_path = res;
                        perfect = point;
                    }
                }
            }
        }

        if perfect_path.1.is_empty() {
            if visited.len() < 5 {
                buff.push((a, (*self, last)));
            }
            return Err(1)
        }
        return Ok((perfect_path.0 + self.distance(perfect), perfect_path.1));
    }

}

pub fn start_cached(points: &Vec<Dot>) -> (f32, Vec<Dot>) {
    let mut buff: Vec<(Vec<Dot>, (Dot, Dot))> = Vec::new();

    let mut results = Vec::new();
    for last in points.clone() {
        let visited: Vec<Dot> = vec![last];
        let mut local_results = Vec::new();
        let mut best: (f32, Vec<Dot>) = (f32::MAX, Vec::new());

        for point in points.clone() {
            if point == last {break}
            if let Ok(res) = point.check_cached(&points, visited.clone(), last, &mut buff) {
                local_results.push((res.clone().0 + last.distance(&point), res.1));
            }
        }
        for res in local_results {
            if res.0 < best.0 {
                best = res;
            }
        }
        results.push(best);
    }

    let mut best: (f32, Vec<Dot>) = (f32::MAX, Vec::new());

    for res in results {
        if res.0 < best.0 {
            best = res;
        }
    }
    best
}

pub fn is_valid(from: Dot, current: Dot, to_check: Dot) -> bool {
    if current.x == from.x {
        if to_check.y == current.y {return true;}
        return (from.y - current.y).is_sign_positive() == (to_check.y - current.y).is_sign_negative();
    }

    if current.y == from.y {
        if to_check.x == current.x {return true;}
        return (from.x - current.x).is_sign_positive() == (to_check.x - current.x).is_sign_negative();
    }

    let gradient_normal = -1. / ((current.y - from.y) / (current.x - from.x));
    let calc = |x: f32| gradient_normal * (x - current.x) + current.y;

    if calc(from.x) > from.y {
        return calc(to_check.x) <= to_check.y;
    } else {
        return calc(to_check.x) >= to_check.y;
    }
}
```