



UNIVERSITÄT ZU LÜBECK  
INSTITUT FÜR  
THEORETISCHE INFORMATIK

# Algorithmen für Moving-Target-Travelling Salesman Problem

*Algorithms for Moving-Target-Travelling Salesman Problem*

## **Bachelorarbeit**

verfasst am

**Institut für Theoretische Informatik**

im Rahmen des Studiengangs

**Informatik**

der Universität zu Lübeck

vorgelegt von

**Felix Greuling**

ausgegeben und betreut von

**Prof. Dr. Maciej Liskiewicz**

Lübeck, den 29. Dezember 2019

### Eidesstattliche Erklärung

*Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.*

---

Felix Greuling

Zusammenfassung

TODO

Abstract

TODO

# Inhaltsverzeichnis

1	Einleitung	1
1.1	Beiträge dieser Arbeit	2
1.2	Verwandte Arbeiten	2
1.3	Aufbau dieser Arbeit	2
2	Grundlagen	4
2.1	Eindimensionaler Fall im MT-TSP	4
2.2	Input & Output	12
3	Zwei-orthogonale-Achsen im MT-TSP	13
3.1	Theoretische Grundlagen	13
4	Heuristiken für zwei-orthogonale-Achsen im MT-TSP	16
4.1	Problem der Modellierung bei zwei orthogonale Achsen im MT-TSP mit Zuständen als Graphen	16
4.2	Prioritätsansatz	17
4.3	Algorithmus mit dem Prioritätsansatz	18
4.4	Brute-Force für zwei orthogonale Achsen im MT-TSP	20
5	Experimente	25
6	Zusammenfassung und Ausblick	26
	Literatur	27
A	Implementierungen	28

# 1

## Einleitung

Das Travelling-Salesman-Problem (TSP) ist ein in der Informatik weit verbreitetes und seit der Formulierung im Jahre 1930 als mathematisches Problem ein langjährig erforschtes Optimierungsproblem aus der Kombinatorik. Dies hat eine besonders hohe Relevanz für Probleme aus der echten Welt, da diese auf das TSP reduziert werden können. Gesucht ist dabei die kürzeste Reihenfolge an Wegen, welche die Ziele miteinander verbinden, sodass man jedes Ziel besucht und am Ende der Tour wieder zum Startpunkt zurückkehrt. Dafür betrachten wir eine Menge an Städten. Diese sollen allesamt von einem FlixBus als Zwischenstop genutzt werden, da an diesem Reisende aus- und einsteigen wollen. Am Ende der Tour soll der FlixBus zum Auftanken wieder zu seinem Ausgangspunkt zurückgelangen. Um die Benzinkosten gering zu halten, wird also die kürzeste Abfolge an Städten gesucht. Dabei sollte man beachten, dass mehrere optimale Touren möglich sind. Für TSP gibt es gerade im heutzutage diverse Anwendungsgebiete. Wir leben in einem Zeitalter, in welchem autonome Systeme zunehmend eine große Rolle spielen. Früher oder später übernehmen autonome Fahrzeuge die Verantwortung auf unseren Straßen [7]. Der Vorreiter für diesen Markt ist die Firma Tesla. Die Fahrzeuge besitzen bereits die Software zum selbstständigen Fahren, erlaubt ist der Einsatz ganz ohne den Menschen aber noch nicht. Auch hierbei werden auf den Straßen die kürzesten Routen gesucht. Die künstliche Intelligenz kann dies unter Anderem mit der Reduzierung auf das TSP berechnen.

Darüber hinaus ist nun ein Spezialfall des TSP von großem Interesse. Dabei sind solche Ziele nicht mehr stationär, diese bewegen sich nun mit einer konstanten Geschwindigkeit. Diese spezielle Instanz wird hierbei als Moving-Target-TSP bezeichnet. Die Problematik besteht dabei, dass sich nach dem Erreichen eines Ziels die Position der anderen Ziele über die Zeit geändert haben. Somit ist eine Berechnung der optimalen Tour mindestens genauso schwer, wie TSP-Instanzen mit stationären Zielen. Das Moving-Target-TSP ist auch in ebenfalls für Probleme der realen Welt interessant. Für autonome Fahrzeuge könnten Drohnen zur Überwachung eingesetzt werden, welche ein Fahrzeug identifizieren können. Somit könnten Daten des Fahrverhaltens für einen Moment eines der zu Testfahrzeuge gesammelt werden. Als bereits integriertes System ist der Nahverkehr auf Google Maps zu erwähnen. Für eine Route wird die schnellstmögliche Route berechnet, wobei aktuelle Positionen von Bussen und Zügen betrachtet werden. Innerhalb dieser beiden Anwendungen bewegen sich die Ziele zwar nicht unbedingt mit einer konstan-

ten Geschwindigkeit, mittels eines Abwägungsmaß, z.B. einer Durchschnittsgeschwindigkeit, lassen sich diese auf Moving-Target-TSP übertragen.

Im Jahre 1998 wurde diese spezielle Instanz des TSP vorgestellt[4]. Dabei wurde dynamische Programmierung[6] in Kombination mit der Modellierung eines Graphen in topologischer Reihenfolge als Strategie für eindimensionale Moving-Target-TSP vorgeschlagen. Diese besitzt eine Zeitkomplexität von  $O(n^2)$ .

## 1.1 Beiträge dieser Arbeit

In dieser Arbeit wird ein neuer, auf dem eindimensionalen Moving-Target-TSP basierender Fall, eingeführt. Dieser enthält zusätzlich eine weitere orthogonale Achse, auf welcher sich die Ziele und die Verfolger bewegen können. Die für den eindimensionalen-Fall bereits gezeigten Eigenschaften, dass der Verfolger in einer optimalen Tour sich jederzeit mit  $v_{max} > v_i, \forall v_i \in V$  bewegt und erst seine Richtung ändern kann, sofern er das schnellste Ziel in seiner Richtung eingeholt hat, gelten auch für diese neue Modifikation. Mit dem Prioritätsalgorithmus wurde ein effizienter aber nicht optimaler Algorithmus vorgestellt, während der Brute-Force-Ansatz für optimale Ergebnisse bei kleinen Instanzen  $n < 15$  genutzt werden kann.

**TODO: Ergebnisse aus Experimente präsentieren**

## 1.2 Verwandte Arbeiten

Seit der ersten Erwähnung aus dem Jahre 1998 wurden unterschiedliche Strategien zur Lösung des MT-TSP ausprobiert und Approximationen durchgeführt. Die Approximation für generelle Fälle gilt dabei schwierig, da viele verschiedene Faktoren die Komplexität des Problems bestimmen. Die Approximations-Forschung in [3] zeigte, dass sich die Probleme nicht besser als mit einem Faktor von  $2^{\pi(\sqrt{\pi})}$  in polynomieller Zeit lösen lassen, es sei denn es gilt  $P = NP$ . Dies gilt bisher allerdings weiterhin als ungelöstes Problem in der Informatik. In jedem Fall gelten Moving-Target-TSP als NP-hart, selbst wenn sich nur zwei Ziele bewegen[3].

Über ein Jahrzehnt später, im Jahre 2013, wurde ein Genetischer Algorithmus vorgestellt [2]. Der erzielten Ergebnisse erwiesen sich dabei effektiver, als mit einer ähnlich modifizierten Greedy-Methode. Diesen Jahres wurden die evolutionären Algorithmen[10] Ameisenkolonien, Simulierte Abkühlung und Genetische Algorithmen experimentell getestet[8]. Letztere haben dabei die beste Performance bei der Suche nach akzeptablen Lösungen mit eingeschränkter Zeit und Rechenleistung erbracht.

## 1.3 Aufbau dieser Arbeit

Diese Arbeit beschreibt zunächst in Kapitel 2 die vorangegangenen Forschungen des eindimensionalen Moving-Target-TSP. Dabei wird speziell die Modellierung und der dazu

entwickelter Algorithmus erläutert. Anschließend wird in Kapitel 3 ein neuer Fall vorgestellt. Dafür wird der 1D-Fall um eine orthogonale Achse erweitert, auf der sich die Ziele sowie der Verfolger bewegen können. Für diesen neuen zwei-orthogonale-Achsen-Fall werden theoretische Eigenschaften gezeigt, die auch für den 1D-Fall gelten. So wird in Kapitel 4 ein Prioritäts- und Brute-Force-Algorithmus präsentiert. Zuletzt folgt in Kapitel 5 ein Experimentierteil für alle drei Algorithmen inklusive der Bestimmung der Güte.

# 2

## Grundlagen

Im folgenden Kapitel werden alle nötigen Grundlagen für das Moving-target TSP (MT-TSP) erläutert. Beim herkömmlichen TSP wird eine optimale Tour durch alle Ziele und Rückkehr zum Startpunkt gesucht. Diese ist die kürzeste Reihenfolge an Zielen, bei dem jedes  $z_i \in Z$  abgefangen wurde. Dabei startet und beendet der Verfolger jede Tour im Ursprung. Das MT-TSP ist eine spezielle Instanz des TSP mit der Erweiterung, dass die Ziele nicht mehr stationär sind, sondern eine konstante Bewegung haben.

### 2.1 Eindimensionaler Fall im MT-TSP

Der eindimensionale Fall (1D-Fall) des MT-TSP wurde das erste Mal in [4] erwähnt und stellt die Grundlage dieser Arbeit dar. Dabei sind jegliche Bewegungen der Ziele und des Verfolgers auf eine Linie beschränkt.

**Definition 2.1.** Jede Instanz  $I$  im 1D-Fall des MT-TSP enthält eine Anzahl  $n$  von Zielen  $Z = \{z_1, \dots, z_n\}$  und den Verfolger  $\kappa$ .  $I$  wird demnach beschrieben durch

$$I = (Z, \kappa).$$

Jedes Ziel  $z_i$  befindet sich zunächst an einem Startpunkt  $p_i$  und bewegt sich dann mit einer konstanten Geschwindigkeit  $v_i$  entlang einer Achse,  $p_i, v_i \in \mathbb{R}$ . Demnach kann ein Ziel als ein Tupel  $z_i = (p_i, v_i)$  dargestellt werden. Der Ursprung ist der Start und das Ziel jeder Tour und ist definiert durch einen Punkt ohne Geschwindigkeit. Die genaue Position ist dabei die Startposition des Verfolgers  $p_\kappa$ . Jegliche Positionen und Geschwindigkeiten können dabei als Vektoren  $P$  und  $V$

$$P = (p_1, \dots, p_n)$$

$$V = (v_1, \dots, v_n)$$



dargestellt werden.

Das Tupel  $(-1, 0)$  würde also als Beispiel bedeuten, dass der Verfolger an der Koordinate  $-1$  startet und ist durch die Geschwindigkeit von  $0$  stationär. Einfachheitshalber wird in dieser Arbeit der Ursprung mit  $(0, 0)$  für 1D-Fälle fest definiert. Ein Ziel mit einer negativen Koordinate befindet sich auf der linken, positive Koordinaten auf der rechten Seite des Ursprungs.

**Definition 2.2.** Der Verfolger bewegt sich mit der Maximalgeschwindigkeit

$$v_{\text{verfolger}} > |v_i|, \forall v_i \in V,$$

somit wird sichergestellt, dass der Verfolger nach einer gewissen Zeit jedes Ziel auf jeden Fall eingeholt hat.

Ohne diese Definition ist es möglich, dass eine unendlich große Tourzeit berechnet wird, da einige Ziele nicht abgefangen werden können.

**Definition 2.3.** Mit dem Zeitstempel  $t \in \mathbb{R}_0^+$  kann genau bestimmt werden, an welcher Position sich ein Ziel hinbewegt hat. Die Position eines Ziels ist also abhängig vom aktuellen Zeitstempel  $t$ . Jede Tour beginnt bei  $t = 0$ .

Es gilt

$$p_{i,t} = p_{i,0} + v_i \cdot t.$$

**Definition 2.4.** Die Zeit, die benötigt wird, um ein Ziel  $B$  von der Position von Ziel  $A$  einzuholen, ist als

$$\tau = \left\lceil \frac{\|pos_A, pos_B\|_1}{v_{\text{verfolger}} - v_B} \right\rceil$$

definiert.

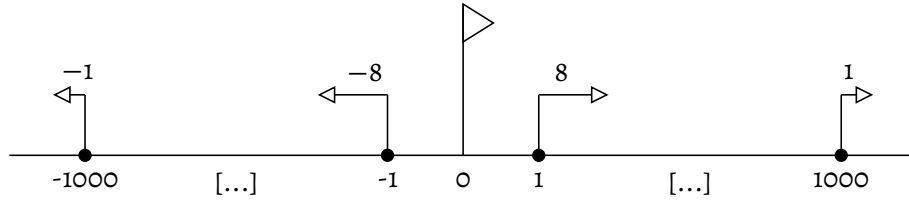
Die Berechnung beruht auf der nach der Zeit (in diesem Fall  $\tau$ ) gleichgesetzten und umgestellten physikalischen Formel <sup>1</sup>. Bemerke,  $v_A$  wird durch  $v_{\text{verfolger}}$  ersetzt, da sich der Verfolger immer mit maximaler Geschwindigkeit bewegt.

Mit diesen Voraussetzungen kann das Problem nun modelliert und gelöst werden. Dafür wird im Folgenden die Vorarbeit aus [4] detailliert beschrieben. Die Autoren modellieren das Problem als Graph-Problem, wobei der eigentliche Graph on-the-fly erstellt wird. Mit linearer Programmierung wird anschließend die schnellstmögliche Abfangzeit für jeden Zustand bestimmt. Somit kann die optimale Tourzeit und -Reihenfolge bestimmt werden. Im eindimensionalen-Fall befinden sich alle Ziele auf einer Achse und können sich nur in zwei verschiedene Richtungen bewegen. Dasselbe gilt auch für den Verfolger. Wichtig ist zunächst die Bedingung, dass der Verfolger sich immer mit seiner maximalen Geschwindigkeit  $v_{\text{verfolger}}$  bewegt. Helvig et. al. bewiesen dies mit dem Aspekt, dass eine Reisegeschwindigkeit von  $v < v_{\text{verfolger}}$  äquivalent zu einer Wartezeit an einem Punkt ist. Dies resultiert in eine längere Tourzeit, das heißt die Tour wäre nicht mehr optimal. Um das Problem der kürzesten Route zu lösen, muss sich der Verfolger an einem Ziel entscheiden, das nächste Ziel in derselben oder entgegengesetzte Richtung einzuholen. Die Kostenberechnung für eine schnellste Tour aus

<sup>1</sup> Gleichförmige Bewegung:  $s = v \cdot t + s_0$

- alle Ziele links vom Ursprung aus gesehen und danach alle rechten Ziele abfangen
- alle Ziele rechts vom Ursprung aus gesehen und danach alle linken Ziele abfangen

ist zwar simpel und einfach implementierbar, reicht aber nicht aus (siehe Abbildung 2.1). Im worst case geht  $t \rightarrow \infty$ , sobald die äußersten Ziele noch deutlich weiter entfernt vom



**Abbildung 2.1:** Offensichtlich würde der Verfolger mit der Geschwindigkeit  $v_{\text{verfolger}} = 11$  deutlich länger für eine Tour brauchen, sofern er zunächst alle Ziele auf der einen und dann auf der anderen Seite abarbeitet. Hierbei wäre es sinnvoll, zunächst die Ziele  $(-1, -10)$  und  $(1, 10)$  einzuholen.

Ursprung aus liegen. Die Autoren aus [4] definierten daraufhin Wendepunkte und bewiesen ihre Korrektheit. Nur an diesen ist es dem Verfolger möglich, die Richtung zu ändern. Wendepunkte sind dabei die schnellsten Ziele auf der rechten bzw. linken Seite des Verfolgers. Sofern der Verfolger vor einem Wendepunkt umkehrt, verlängert sich die Tour und ist damit nicht mehr optimal.

**Definition 2.5.** Ein Zustand  $A$  ist definiert durch die aktuelle Position des Ziels ( $s_k$ ), an dem sich der Verfolger zur Zeit befindet und dem schnellsten Ziel auf der gegenüberliegenden Seite des Ursprungs ( $s_f$ ). Es ist also wieder eine Tupeldarstellung

$$A = (s_k, s_f)$$

möglich, wobei  $s_k$  und  $s_f$  wiederum Tupel sind (siehe Definition 2.1).

Ein Zustand stellt eine Momentaufnahme der Tour dar. Dabei wird ein potentieller Wendepunkt repräsentiert. Um die optimale Tour zu bestimmen, muss an jedem dieser Punkte korrekt entschieden werden, ob sich der Verfolger weiter in die Richtung bewegt oder  $s_f$  auf der anderen Seite des Ursprungs verfolgt. Im Gegensatz zu den anderen Zuständen besitzen  $A_o$  und  $A_{\text{final}}$  keine Tupel. Dabei handelt es sich um den Start und Endzustand, welche bei jeder Tour gleich sind. Der Verfolger befindet bei beiden dieser Zustände im Ursprung. Mit der Funktion  $t$  wird einem Zustand die aktuell minimale Zeit zugewiesen, mit der der Zustand über andere Zustände bis dahin am schnellsten erreichbar ist (siehe Definition 2.3). Offensichtlich gilt demnach  $t[A_o] = 0$ .

Wie bereits erwähnt, gibt es in den meisten Fällen Ziele, welche keine potentiellen Wendepunkte darstellen und somit nicht zur optimalen Tour beitragen. Um die Laufzeit und Speicherkomplexität zu reduzieren, können diese zunächst eliminiert werden. Es handelt sich dabei um Ziele, die sowieso eingeholt werden. Zunächst wird jedes Ziel in die Liste *Left* oder *Right* eingefügt, abhängig davon, ob sich das Ziel bei *timestamp* = 0 auf der linken oder rechten Seite des Ursprungs befindet. Anschließend werden *Left* und *Right* in absteigender Reihenfolge nach den Geschwindigkeiten sortiert. Ziele, welche sich nun

näher am Ursprung befinden und zugleich langsamer sind als ein anderes aus der jeweiligen Liste, werden eliminiert. Damit beinhalten *Left* und *Right* ausschließlich potentielle Wendepunkte.

Um nun alle Zustände zu bestimmen, wird jede Kombination aus den Listen *Left* und *Right* und umgekehrt für  $s_k$  und  $s_f$  eingesetzt und in die Zustandsliste *States* eingefügt. Anschließend wird *States* in absteigender Reihenfolge nach der Summe der Indizes der Ziele aus den Listen *Left* und *Right* sortiert. Somit befinden sich die Kombinationen bzw. Zustände aus den schnellsten Zielen am Listenanfang von *States*. Ein Zustandsübergang von Zustand A in den Zustand B wird mit

$$\tau = A \rightarrow B$$

beschrieben. Der Übergang  $\tau$  gibt dabei die Zeit  $\tau$  (siehe Definition 2.4), um von dem aktuellen Zustand A in den nächsten Zustand B zu gelangen). Ausgehend von einem Zustand, gibt es bis zu zwei Zustandsübergänge. Dabei handelt es sich von dem nächst-schnellsten auf der linken oder rechten Seite des Ursprungs. Die Übergänge werden dann als  $\tau_{left}$  und  $\tau_{right}$  bezeichnet. Sofern jedes Ziel auf einer Seite eingeholt wurde, wird der Übergang  $\tau_{final}$  in  $A_{final}$  gewählt. Für die Berechnung von  $\tau_{final}$  wird die Zeit vom aktuellen Zustand bis zum Abfangen der restlichen Ziele auf der anderen Seite des Ursprungs und zusätzlich die Rückkehr zum Ursprung berechnet. Offensichtlich existiert eingehend in  $A_o$  und ausgehend von  $A_{final}$ .

Wie bereits erwähnt, ist die Zustandsliste *States* nach der Summe der Indizes aus *Left* und *Right* in absteigender Reihenfolge sortiert. Mit zwei Möglichkeiten von  $\tau_{left}$  und  $\tau_{right}$  führt jeder Zustand in einen anderen Zustand mit einem höheren Index. Dabei erhält der Startzustand  $A_o$  die Summe  $-1$ , wodurch die Zustandsübergänge in die Zustände mit dem Summenwert von 0 führen. Die Zustände mit der höchsten Summe führen in  $A_{final}$  (Dies gilt sowieso mit der vorherigen Regelung). Mit diesen Bedingungen konnte nun aus *States* ein Graph  $G$  erzeugt werden. Im Graph

$$G = (V, E)$$

werden die Knoten  $V$  durch die Zustände  $A_i \in States$  und  $E$  durch die jeweiligen Zustandsübergänge  $\tau$  repräsentiert. Mit der Bedingung, dass Übergänge nur in Zustände mit höheren Summenwerten führen, ist  $G$  gerichtet und azyklisch. Man gelangt also nach spätestens  $n$  Zuständen (exklusive  $A_o$ ) in  $A_{final}$ .

Mit der Modellierung des Problems als Graphen und den Eigenschaften, dass dieser azyklisch und in topologischer Reihenfolge sortiert ist, kann das Problem mit einem einfachen *Kürzeste-Wege*-Algorithmus gelöst werden. Hierbei kann eine simple Heuristik, zum Beispiel [1], verwendet werden, um den kürzesten Weg von  $A_o$  nach  $A_{final}$  zu bestimmen. Mit der Bestimmung des kürzesten Pfades muss am Ende noch bestimmt werden, welche Ziele zwischen den Zuständen eingeholt wurden. Damit werden die anfangs eliminierten Zustände wieder der Tour hinzugefügt und bestenfalls (richtige Implementierung) ist somit die optimale Tour bestimmt.

### Algorithmus von Helvig, Robins und Zelikovsky

Mit diesen Voraussetzungen haben die Autoren von [4] einen exakten  $O(n^2)$ -Algorithmus für eindimensionale Fälle entwickelt, welcher auf dynamischer Programmierung basiert. Dieser bestimmt dabei die optimale Tour für die Eingabeinstanz.

Nach dem Schema aus dem vorherigen Abschnitts werden wieder die Listen *Left*, *Right* und *States* generiert. Anschließend wird durch jeden der  $n^2$  Zustände iteriert. Für das einfachere Verstehen des Algorithmus wurde der Graph  $G$  zwar beschrieben, aber nicht generiert. Somit wird der Speicherplatz reduziert und damit die Effizienz des Algorithmus verbessert. Diese *On-the-fly*-Methode, um den Graph  $G$  zu generieren, ist durch die topologische Sortierung möglich. Damit ist für jeden Zustand sichergestellt, dass dieser mit minimaler Zeit erreicht wurde. Zudem ist nicht für jeden Zustand eine Berechnung der Übergänge in andere Zustände nötig. Einige werden ausgelassen oder führen direkt in  $A_{final}$ . Dies lässt sich auf das Vorgehen des Algorithmus zurückführen. Für einen Zustand  $A_i$  wird dabei eines der folgenden Schritte ausgeführt:

- Wenn in  $A_i$  keine eingehenden Übergänge besitzt, führe mit dem nächsten Zustand in der Liste fort. Dies tritt genau dann auf, wenn  $t[i] = \infty$ .
- Falls der Verfolger jedes Ziel auf einer Seite des Ursprungs eingeholt hat, erzeuge einen Übergang  $\tau_{final}$  in  $A_{final}$ . Berechne die Zeit, um die verbleibenden Ziele auf der anderen Seite einzuholen und zusätzlich die Retour zum Ursprung.
- Berechne ansonsten  $\tau_{Left}$  und  $\tau_{Right}$ , welche den Verfolger entweder zum schnellsten Ziel auf der rechten oder linken Seite schickt. Falls die Zeit addiert mit dem aktuellen Zeitstempel  $t[i]$  kleiner ist, als bisher von einem anderen Zustand, aktualisiere  $t[A_{Left}]$  bzw.  $t[A_{Right}]$  mit diesem Wert.

Schließlich werden alle Ziele, einschließlich der zuvor eliminierten Ziele, zwischen den Wendepunkten berechnet und in der richtigen Reihenfolge zusammengefügt. Somit wird eine optimale Tour durch die Kombination aus topologischer Reihenfolge und linearer Programmierung garantiert. Der dazu entwickelte Pseudocode ist in Algorithmus 1 abgebildet[4].

**Algorithmus 1** Exact Algorithm for One-Dimensional Moving-Target TSP[4]**Input:** The initial positions and velocities of  $n$  targets, and the maximum pursuer speed**Output:** A time-optimal tour intercepting all targets, and returning back to the origin**Preprocessing**

Partition the list of targets into the targets on the left side, the right side of the origin

Sort the targets on the left into list *Left* in order of nonincreasing speedsSort the targets on the right into list *Right* in order of nonincreasing speedsDelete targets from *Left* which are closer to the origin than faster targets in this listDelete targets from *Right* which are closer to the origin than faster targets in this list**if** *Left* or *Right* is empty **then**

Calculate the time required to intercept all remaining targets; and

Go to the postprocessing step

**end if****Main Algorithm**Let  $A_o$  be the start stateLet  $A_{final}$  be the final state*STATE* is the sorted list of states in order of nondecreasing sum of the indices  
    of each state's targets in lists *Left* and *Right*Place  $A_o$  first in the list *STATE*Place  $A_{final}$  last in the list *STATE* $t(A) \leftarrow \infty$  for any state  $A \neq A_o$  $t(A_o) \leftarrow 0$ *current*  $\leftarrow 0$ **while** *current*  $\leq$  the number of states in *STATE* **do**     $A = STATE[current]$     **if** there are no transitions into  $A$  **then**        Increment *current* and jump back to the beginning of the while loop    **end if**    **if** for state  $A$ , all remaining targets are on one side of the origin **then**         $t(\tau_{final}) \leftarrow$  time required to intercept the remaining targets and  
        return to the origin    **else**        Calculate the two transitions  $\tau_{left}$  and  $\tau_{right}$  from state  $A$  using lists *Left* and *Right*        **if**  $t(A) + t(\tau_{left}) < t(A_{left})$  **then**             $t(A_{left}) \leftarrow t(A) + t(\tau_{left})$         **end if**        **if**  $t(A) + t(\tau_{right}) < t(A_{right})$  **then**             $t(A_{right}) \leftarrow t(A) + t(\tau_{right})$         **end if**    **end if**    Increment *current***end while***OUTPUT*  $\leftarrow$  the reverse list of states from  $A_{final}$  back to  $A_o$ **Postprocessing****for** pair of consecutive states in *OUTPUT* **do**

Calculate which targets are intercepted between the state pair

Sort the intercepted targets by the interception order

**end for**

Output the concatenated sorted lists of targets

### Beispiel für die Funktionsweise des Algorithmus

Oftmals ist es schwierig, dynamische Programmierung nachzuvollziehen. Dafür werden im Folgenden anhand eines Beispiels die einzelnen Schritte und insbesondere die Iterationen durch jeden Zustand erläutert.

**Beispiel 2.6.** Für den Input sei folgendes gegeben:

- Ziele  $Z = \{(-1000, -1), (-500, -1), (-1, 8), (1, 8), (500, 1), (1000, 1)\}$
- Verfolger  $\kappa = (0, 10)$

Demnach startet und beendet der Verfolger die Tour an der Koordinate 0. Bemerke, dass einfachheitshalber trotz voranschreitender Zeit die Koordinaten der Ziele gleich bleiben. Die Ziele werden nun in *Left* und *Right* eingeteilt:

- *Left* =  $\{(-1000, -1), (-500, -1), (-1, 8)\}$
- *Right* =  $\{(1, 8), (500, 1), (1000, 1)\}$

Nach der Sortierung in absteigender Reihenfolge nach den Geschwindigkeiten und der Eliminierung von Zielen, welche sowieso eingeholt werden, sehen die Listen wie folgt aus:

- *Left* =  $\{(-1, 8), (-1000, -1)\}$
- *Right* =  $\{(1, 8), (1000, 1)\}$

Dies ist äquivalent zu der Startkonfiguration aus Abbildung 2.1. Als nächstes wird die *States* in absteigender Reihenfolge der Indizes aus den Listen *Left* und *Right* erstellt:

$$\begin{aligned}
 &A_0 \\
 &A_1 = \{(-1, -8), (1, 8)\} \\
 &A_2 = \{(1, 8), (-1, -8)\} \\
 &A_3 = \{(-1, -8), (1000, 2)\} \\
 &A_4 = \{(1000, 2), (-1, -8)\} \\
 &A_5 = \{(-1000, -1), (1, 8)\} \\
 &A_6 = \{(1, 8), (-1000, -1)\} \\
 &A_7 = \{(-1000, -1), (1000, 2)\} \\
 &A_8 = \{(1000, 2), (-1000, -1)\} \\
 &A_{\text{final}}
 \end{aligned}$$

Nun beginnt die dynamische Programmierung. Dafür wird durch jeden Zustand aus *States* in chronologischer Reihenfolge iteriert. Dabei ergeben sich für jede Iteration folgende

benötigte Zeiten bis zum Erreichen eines jeden Zustands:

Iteration 0:  $t = [0.0, 0.5, 0.5, \infty, \infty, \infty, \infty, \infty, \infty, \infty]$

Iteration 1:  $t = [0.0, 0.5, 0.5, \infty, \infty, 111.11, 5.5, \infty, \infty, \infty]$

Iteration 2:  $t = [0.0, 0.5, 0.5, 5.5, 125.0, 111.11, 5.5, \infty, \infty, \infty]$

Iteration 3:  $t = [0.0, 0.5, 0.5, 5.5, 125.0, 111.11, 5.5, 112.23, 137.5, \infty]$

Iteration 4:  $t = [0.0, 0.5, 0.5, 5.5, 125.0, 111.11, 5.5, 112.23, 137.5, 2251.0]$

Iteration 5:  $t = [0.0, 0.5, 0.5, 5.5, 125.0, 111.11, 5.5, 112.23, 137.5, 2001.0]$

Iteration 6:  $t = [0.0, 0.5, 0.5, 5.5, 125.0, 111.11, 5.5, 112.23, 126.25, 2001.0]$

Iteration 7:  $t = [0.0, 0.5, 0.5, 5.5, 125.0, 111.11, 5.5, 112.23, 126.25, 585.17]$

Iteration 8:  $t = [0.0, 0.5, 0.5, 5.5, 125.0, 111.11, 5.5, 112.23, 126.25, 529.61]$

Somit benötigt die optimale Tour durch alle Ziele eine Reisezeit von  $t[9] = 529, 61$ . Beachte, dass keine Iteration 9 nötig ist, da mit  $A_8$  das letzte mal  $\tau_{final}$  berechnet wird. Nun wurde im Algorithmus nicht explizit beschrieben, in welcher Reihenfolge der Zustände die optimale Tour erfolgt. Dafür kann einfach eine Liste *parent* verwendet werden, um bei einer schnelleren Zeit  $t(A) + t(\tau_{left/right}) < t(A_{left/right})$  den Elternzustand abzuspeichern. Diese Liste sieht nach den Iteration wie folgt aus:

$parents = [-1, 0, 0, 2, 2, 1, 1, 3, 6, 8]$

Beim Backtracking ergibt sich nun die Zustandsreihenfolge

$A_0$

$A_1 = \{(-1, -8), (1, 8)\}$

$A_2 = \{(1, 8), (-1, -8)\}$

$A_4 = \{(1000, 2), (-1000, -1)\}$

$A_{final}$ .

Damit erhalten wir 4 Wendepunkte<sup>2</sup>, da vor  $A_{final}$  alle verbleibenden Ziele auf der anderen Seite eingeholt werden. Damit wäre theoretisch beim Ziel  $(-1000, -1)$  der letzte Wendepunkt der Tour. Zum Schluss wird nun überprüft, zu welchem Zeitpunkt die Ziele aus  $Z$  zwischen den einzelnen Wendepunkten eingeholt werden.

$(0, 0)$ , Abfangzeit : 0.0

$(-1, -8)$ , Abfangzeit : 0.5

$(1, 8)$ , Abfangzeit : 5.5

$(500, 1)$ , Abfangzeit : 56.67

$(1000, 2)$ , Abfangzeit : 126.25

$(-500, -1)$ , Abfangzeit : 335.0

$(-1000, -1)$ , Abfangzeit : 390.55

$(0, 0)$ , Abfangzeit : 529.61

Dies ist nun die optimale Tour, in der alle Ziele aus  $Z$  abgefangen wurden.

<sup>2</sup> Exklusive  $A_0$  und  $A_{final}$ , dies ist nur der Start und das Ziel der Tour.



## 2.2 Input & Output

Um Heuristiken aufzustellen und zu bewerten ist ein sinnvoller und einheitlicher Input und Output notwendig. Für den Input wird eine Menge  $T$  von Zielen sowie die initiale Position und Geschwindigkeit des Verfolgers erwartet. Dies reicht aus, um eine Tour zu bestimmen.

Beim Output kommt es darauf an, wie detailliert die Tour beschrieben werden soll. Als offensichtliche Parameter werden die Tourlänge und Tourzeit zurückgegeben. Damit ist allerdings die Tour schlecht nachvollziehbar. Demnach werden die Ziele in der vom Verfolger eingeholten Reihenfolge zurückgegeben. Dabei verfügt jedes Ziel über die Position und Zeit in der der Verfolger es eingeholt hat. Um die Tour komplett nachvollziehen, ist eine graphische Anwendung sinnvoll, aber nicht notwendig.

Die Algorithmen dieser Arbeit werden dabei einfach nur die Ziele in der eingeholten Reihenfolge zurückgeben. Je nach Implementierung kann dann einem Ziel dabei die eingeholte Zeit zugeordnet werden, womit man dann anschließend alle restlichen Informationen berechnen kann.

Sobald allerdings eine ungültige Eingabe, z.B. wenn eine Zielgeschwindigkeit größer als die Verfolgergeschwindigkeit ist, wird eine „Nein“-Instanz zurückgegeben. Dies wird allerdings in den Algorithmen vorausgesetzt und nicht extra behandelt.



# 3

## Zwei-orthogonale-Achsen im MT-TSP

Als neue Modifikation des 1D-Falls wird nun der Achse eine weitere hinzugefügt. Alle Bewegungen und Positionierungen der Ziele und des Verfolgers sind ebenfalls auf die Achsen beschränkt. Die Achsen liegen dabei orthogonal zueinander. Den Zielen ist es dabei nicht erlaubt, an dem Schnittpunkt die Achse zu wechseln. Ein Ziel befindet sich also entweder auf der waagerechten oder senkrechten Achse. Der Schnittpunkt ist dabei festgesetzt auf die Koordinate 0. Dies gilt einfachheitshalber ebenfalls für den Ursprung. Im Folgenden wird der Ursprung als Begriff für diesen Punkt verwendet.

Mit der neuen Achse könnte für die Positionsbestimmung eine zweidimensionale Koordinate verwendet werden. Allerdings wäre einer dieser Koordinaten immer gleich 0, da jegliche Bewegungen der Ziele und des Verfolgers auf die Achsen beschränkt sind. Somit ist nur eine einfache Ergänzung der Definition 2.1 um einen booleschen Wert  $b$  nötig. Die Position  $p_i$  wird dabei mit

$$p_i = (a, b) \text{ mit } a \in \mathbb{R}, b \in \{true, false\}$$

neu definiert. Dabei gibt  $b$  die Achse an, *true* steht für die waagerechte, *false* für die senkrechte Achse. Die Ziele werden nun neben den Listen *Left* und *Right* auch in die *Top* und *Bottom* einsortiert. Dabei deckt *Top* den positiven und *Bottom* den negativen Koordinatenbereich ab. Mit dieser Ergänzung gelten weiterhin alle anderen der vorherigen Definitionen.

### 3.1 Theoretische Grundlagen

In diesem Abschnitt soll mit der Vorarbeit aus [4] gezeigt werden, dass einige Eigenschaften des 1D-Falls genauso auch im zwei-orthogonalen-Achsen-Fall des MT-TSP gelten.

**Lemma 1.** In jeder optimalen Tour bei zwei orthogonalen Achsen im MT-TSP bewegt sich der Verfolger immer mit maximaler Geschwindigkeit.

**Beweis.** Der Beweis basiert darauf, dass in jedem Fall eine Reduzierung auf den Beweis von [4] vorgenommen wird. Dafür nehmen wir eine Fallunterscheidung vor:

1. Das nächste Ziel des Verfolgers liegt auf der selben Achse:  
Mit dem Beweis für 1D-Fälle in [4] gilt dies unmittelbar auch für diesen Fall.

2. Das nächste Ziel des Verfolgers bewegt sich auf der anderen Achse:

Indirekter Beweis:

Nehmen an, der Verfolger bewegt sich mit  $v_{\text{verfolger}} < v_{\text{max}}$ . Dies ist äquivalent dazu, dass der Verfolger an seiner aktuellen Position eine Zeit  $\tau$  wartet und sich dann mit  $v_{\text{max}}$  weiterbewegt, um dann das nächste Ziel  $s$  einzuholen. Dabei befindet sich  $s$  auf der anderen Achse. Nach der Wartezeit erreicht der Verfolger an Zeitpunkt  $t_1$  den Ursprung und holt das Ziel  $s$  an der Position  $p$  zum Zeitpunkt  $t_2$  ein.

Nehmen nun an, dass der Verfolger sich direkt zum Mittelpunkt bewegt. Bis zum Eintreffen des Zeitpunktes  $t_1$  wartet der Verfolger nun wieder die Zeit  $\tau$ . Das Ziel  $s$  wird nun zum selben Zeitpunkt  $t_2$  bei  $p$  erreicht, wie im vorherigen Szenario.

Dies wird nun fortgeführt, indem der Verfolger nicht im Ursprung wartet, sondern von diesem aus  $p$  direkt erreicht. Bis zum Zeitpunkt  $t_2$  wird nun wieder für die Dauer von  $\tau$  gewartet. Außerdem kann der Verfolger schon zu einem Zeitpunkt  $t_1 \leq t_s \leq t_2$  abfangen, sofern sich  $s$  vom Verfolger wegbewegt.

Wird dies nur für alle restlichen Ziele der Tour fortgeführt, resultiert dies letztendlich in Wartezeit am Ende der Tour, was offensichtlich nicht optimal ist. Dieser Fall ist demnach nur eine Erweiterung des 1D-Fall-Beweises um den Ursprung zwischen Zielen, die auf unterschiedlichen Achsen liegen.

In jedem der Fälle wird eine Wartezeit erzeugt, welche an das Ende der Tour verschoben werden kann. Somit ist die Tour offensichtlich nicht mehr optimal. Der Verfolger bewegt sich also zu jeder Zeit mit  $v_{\text{verfolger}} = v_{\text{max}}$ .  $\square$

**Lemma 2.** In jeder optimalen Tour bei zwei orthogonale Achsen im MT-TSP gelten für den Verfolger folgende Eigenschaften:

- Bewegt sich der Verfolger wegführend vom Ursprung, kann dieser erst seine Richtung ändern, sofern er das schnellste Ziel in seiner Richtung abgefangen hat.
- Bewegt sich der Verfolger in Richtung des Ursprungs, kann dieser solange nicht seine Richtung ändern, bis er das den Ursprungs erreicht hat.

*Beweis.* Für den Beweis des Lemmas müssen beide Eigenschaften bewiesen werden.

Fallunterscheidung:

**TODO: Beweis überarbeiten!**

1. Der Verfolger bewegt sich vom Ursprung weg in Richtung des schnellsten Ziels  $s$  auf der selben Achse:

Mit dem Beweis für 1D-Fälle in [4] gilt dies unmittelbar auch für diesen Fall. Dies gilt für jede Seite jeder Achse, auf der sich der Verfolger vom Ursprung wegbewegt.

2. Der Verfolger hat soeben das schnellste Ziel auf einer Achse abgefangen und bewegt sich nun zum Ursprung:

Nach [4] kann sich nun der Verfolger nicht umdrehen und ein anderes Ziel auf der Achse einholen. Damit hätte er eine Zeit lang nicht das schnellste Ziel einer Richtung eingeholt und somit nach [4] äquivalent zum Warten in einem Punkt ist. Dies resultiert nach Lemma 1 in eine nicht optimale Tour.

Sei der Verfolger nun am Ursprung angekommen. Dieser hat nun drei Richtungsmöglichkeiten. Dabei spielt es allerdings keine Rolle für welche der drei Richtungen

sich der Verfolger entscheidet, denn ab dem Ursprung gilt für jede der Richtungen wieder der 1. Fall.

Es reicht also wieder aus, den 1D-Fall-Beweis für Wendepunkte aus [4] als Grundlage für den zwei-orthogonale-Achsen-Fall-Beweis zu verwenden.  $\square$

# 4

## Heuristiken für zwei-orthogonale-Achsen im MT-TSP

In diesem Kapitel werden konkrete Heuristiken für den zwei-orthogonale-Achsen-Fall präsentiert. Zunächst wird der Prioritätsansatz erläutert. Anschließend wird eine Verbesserung dieses Ansatzes betrachtet. Zuletzt wird die Brute-Force Methode vorgeschlagen. In jedem der Fälle werden die Laufzeit und Güte analysiert und die Korrektheit gezeigt.

### 4.1 Problem der Modellierung bei zwei orthogonale Achsen im MT-TSP mit Zuständen als Graphen

Im 1D-Fall konnte über die topologische Reihenfolge der Wendepunkte bzw. Zustände ein Graph erzeugt werden. Somit wurde mit der Bestimmung des kürzesten Pfades das Problem die optimale Tour bestimmt.

Es wäre also zunächst sinnvoll, diesen Ansatz für den zwei-orthogonale-Achsen-Fall zu übernehmen, da nur eine Achse hinzugekommen ist. Hierfür werden wieder die einzelnen Ziele in die Listen *Left*, *Right*, *Top* und *Bottom* aufgeteilt. Bei dem Algorithmus von [4] wurden nun die Ziele eliminiert, welche sowieso eingeholt werden und dementsprechend keine Wendepunkte darstellen. Mit der zusätzlichen Achse gilt dies nur noch für Ziele, welche sich wegführend vom Ursprung bewegen. Das liegt daran, dass Ziele nun den Ursprung überqueren können, während der Verfolger sich auf der anderen Achse befindet. Somit werden solche Ziele nicht automatisch eingeholt.

Der nächste Schritt ist nun die Generierung der Zustandsliste *States*, wobei eine Reihe von Problemen auftreten. Die Ziele befinden sich dabei im Laufe der Zeit nicht mehr unbedingt auf der Seite des Verfolgers, welche die jeweilige Liste aus *Left*, *Right*, *Top* und *Bottom* vorgibt. Sei der Verfolger zum Zeitpunkt  $t$  auf der oberen Seite der senkrechten Achse. Das Ziel  $z$  befindet sich in der Liste *Left* und befindet zum Zeitpunkt  $t$  auf der rechten Seite der waagerechten Achse. Damit befindet sich  $z$  nun auch auf der rechten Seite des Verfolgers und ist nicht mehr repräsentativ für die Liste *Left*.

Mit dieser Überlegung muss nun jedes solcher Ziele, die den Ursprung überqueren, betrachtet werden. Dieses hat potentiell eine hohe Geschwindigkeit und muss somit ggf.

möglichst schnell abgefangen werden. Problematisch ist dabei, dass ein Ziel vielleicht eine hohe Geschwindigkeit besitzt, aber noch sich sehr weit entfernt vom Ursprung befindet. (siehe Abbildung **TODO**). Es müsste also für jegliche Ziele, die einen Ursprung überqueren extra Zustände geben, da diese als die nächsten Wendepunkte in Betracht gezogen werden könnten. Demnach gäbe es von einem Zustand weitaus mehr Übergänge, als nur  $\tau_{left}$ ,  $\tau_{right}$ ,  $\tau_{top}$  und  $\tau_{bottom}$ .

Die nächste Überlegung wäre nun, die Listen nach jeder Iteration zu aktualisieren. Das Problem ist hierbei, dass damit die Modellierung des Graphen so nicht mehr funktioniert. Dabei entsteht mitten im Graphen ein komplett neuer Graph.

Letztendlich würde ein optimaler Algorithmus eine sehr große Laufzeit benötigen und gerade bei vielen Randbedingungen ist eine korrekte Implementierung schwer. Daher wird im nachfolgenden Abschnitt ein anderer Ansatz, bei dem die Ziele nach einer Priorität abgefangen werden, mit der Laufzeitkomplexität von  $O(n^2)$  vorgeschlagen.

## 4.2 Prioritätsansatz

Mit dem Prioritätsansatz wird für ein Ziel  $z_i$  die Dringlichkeit bestimmt, dieses als nächstes einzuholen. Hierfür fließen unterschiedliche Faktoren für die Berechnung der Priorität mit ein. Die Priorität wird nach jedem Abfangen eines Ziels neu berechnet. Mit den Faktoren  $\varphi_1$ ,  $\varphi_2$  und  $\varphi_3$ , wobei  $\varphi_i \in \mathbb{R}$ , wird die Priorität eines Ziels berechnet. In die jeweilige Berechnung fließen die Gewichte

$$\omega = (w_1, w_2, w_3) \mid w_i \in \mathbb{R}.$$

mit ein, um den jeweiligen Anteil an der Priorität zu erhöhen bzw. zu verringern.

Der Geschwindigkeitsfaktor  $\varphi_1$  erhöht Priorität eines schnellen Ziels  $z_i$  und wird mit

$$\varphi_1(z_i, w_1) = \frac{|v_i|}{v_{verfolger}} \cdot w_1. \quad (4.1)$$

berechnet. Der Positionsfaktor  $\varphi_2$  erhöht Priorität bei Zielen, die sich vom Ursprung bewegen, andernfalls verringert sich die Priorität. Dies wird mit dem Faktor  $a \in \{-1, 1\}$  verrechnet.

$$\varphi_2(z_i, w_2) = \frac{|p_i|}{v_{verfolger}} \cdot a \cdot w_2. \quad (4.2)$$

Der Distanzfaktor  $\varphi_3$  verringert die Priorität bei großen Abständen zur aktuellen Position und wird auf Grundlage der Definition 2.4 berechnet mit

$$\varphi_3(z_i, w_3) = \left| \frac{\|p_{verfolger} - p_i\|_1}{v_{verfolger} - v_i} \right| \cdot w_3. \quad (4.3)$$

**Definition 4.1.** Die Priorität  $\alpha_i \in \mathbb{R}$  eines Ziels  $z_i$  bestimmt die Wichtigkeit eines Ziels, als nächstes eingeholt zu werden und wird mit

$$\alpha_i(z_i, \omega) = \varphi_1(z_i, w_1) + \varphi_2(z_i, w_2) - \varphi_3(z_i, w_3)$$

definiert.

Die einzelnen Ziele werden nun in eine Prioritätswarteschlange

$$Q = T \subseteq Z.$$

eingesetzt. Diese wird in absteigender Reihenfolge nach  $\alpha_i$  sortiert. In jeder Iteration wird die Priorität neu berechnet, sodass  $Q$  ebenfalls neu sortiert wird.

### 4.3 Algorithmus mit dem Prioritätsansatz

In diesem Abschnitt wird ein konkreter 2oA-Fall-Algorithmus vorgestellt. Während der 1D-Algorithmus aus [4] einmalig durch jeden Zustand iteriert, wird in diesem Algorithmus immer für das erste Element *current* aus der Prioritätswarteschlange  $Q$  eine Abfolge von Operationen durchgeführt.

Zunächst wird überprüft, ob sich inklusive *current* alle verbleibenden Ziele auf einer Seite des Ursprungs befinden. Dabei muss gelten, dass die Ziele sich zu dem Zeitpunkt auf einer Seite befinden müssen, zu dem der Verfolger den Ursprung erreicht hat. Das Abfangen der verbleibenden Ziele und die Rückkehr zum Ursprung ist dabei einfach zu berechnen. Dies stellt die Abbruchbedingung des Algorithmus dar und ist spätestens mit dem letzten Ziel erfüllt. Ist die Bedingung nicht erfüllt, wird für jedes verbleibende Ziel die Priorität  $\alpha$  neu berechnet und in  $Q$  absteigend nach  $\alpha$  neu sortiert. Anschließend wird die Zeit  $\pi[prev \rightarrow current]$  vom vorherigen Ziel *prev* bis *current* berechnet und auf den aktuellen Zeitstempel addiert. Anschließend wird mit der ermittelten Zeit die Position jedes Ziels in  $Q$  gemäß Definition 2.3 aktualisiert. Nun werden neben *current* alle Ziele, welche zwischen *prev* und *current* abgefangen wurden, aus  $Q$  entfernt und ein Output-Array *OUTPUT* eingefügt. Dies wird dann solange bis zum Abbruchkriterium fortgeführt. Am Ende wird dann *OUTPUT* in der Reihenfolge des Abfangens sortiert und zurückgegeben. Der Algorithmus ist formal beschrieben in Algorithmus 2.

**Algorithmus 2** Prioritäts-Algorithmus für zwei-orthogonale Achsen beim MT-TSP

---

**Input:** Targets  $Z$ , pursuer**Output:** Targets  $Z$  in order of nondecreasing intercepting timeLet  $t$  be the time-array, which represents the intercepting time for each targetLet  $current$  be the target, that was just intercepted by the pursuerLet  $OUTPUT$  be the target-array in order of nondecreasing intercepting time $OUTPUT.add(current)$  $current \leftarrow origin$ , which is determined by the start position of the pursuer $Q \leftarrow$  priorityqueue with targets, which are sorted in order of nonincreasing priorities**for**  $z_i \in Z$  **do** $t[z_i] \leftarrow \infty$  $Q.add(z_i)$ Calculate  $\alpha(z_i)$ **end for****while**  $Q$  is not empty **do****if** each remmaining target is located on one of the four sides of the intersection  
**then****for**  $z_i \in Q$  **do** $t[z_i] \leftarrow$  Time to intercept  $z_i$ **end for**

Calculate return to the origin of the target, that will be reached last

 $OUTPUT.addAll(Q)$ **break****end if**Calculate  $\alpha(z_i)$ ,  $\forall z_i \in Q$  und update  $Q$  $prev \leftarrow current$  $current \leftarrow$  retrieve the first target of  $mathcal{Q}$  $t[current] \leftarrow time[prev] + \pi[prev \rightarrow current]$  $OUTPUT.add(current)$ Update the position of each  $z_i \in Q$  $interceptedTargets \leftarrow$  intercepted targets between  $prev$  und  $current$  $Q.removeAll(interceptedTargets)$  $OUTPUT.addAll(interceptedTargets)$ **end while**Assign the respective  $t[z_i]$  to each targetSort  $interceptedTargets$  in order of nondecreasing intercepting time**return**  $OUTPUT$ 

---

### Laufzeitkomplexität Prioritätsansatz

Der Prioritätsansatz geht mit der Prioritätswarteschlange linear jedes Ziel maximal einmal durch. Wird ein Ziel zwischen dem vorherigem und dem aktuell betrachteten eingeholt, wird dieses direkt aus der Warteschlange entfernt. Für die Überprüfung auf abgefangene Ziele ergibt sich eine Zeitkomplexität von  $O(n)$ , da diese für alle verbleibenden Ziele aus der Warteschlange durchgeführt wird. Die selbe Komplexität ergibt sich für das Updaten der Prioritäten. Die Berechnung der Zeit zum Einholen eines Ziels erfolgt in  $O(1)$  (siehe Definition 2.4). Die Überprüfung auf die Abbruchbedingung beläuft sich auf eine Laufzeit von  $O(n)$ , da dies für alle verbleibenden Ziele überprüft wird, ob sich diese auf der selben Seite befinden. Ist dem so, wird für jedes Ziel die finale Zeit und anschließend die Rückkehr zum Ursprung in konstanter Zeit berechnet. In Kombination mit dem Iterieren durch die Warteschlange werden die soeben genannten Laufzeiten um den Faktor  $n$  erhöht. Jegliche Sortierungen befinden sich außerhalb der Warteschlange und haben mit einem gängigen Sortiervorgang [5] eine Zeitkomplexität von  $O(n \log(n))$ . Somit ergibt sich beim Prioritätsalgorithmus eine Gesamtzeitkomplexität von  $O(n^2)$ .

### Korrektheit Prioritätsansatz

Bei der Korrektheit des Prioritätsansatzes ist die Bedingung wichtig, dass der Algorithmus nicht unbedingt ein optimales Ergebnis liefert. Dieser betrachtet nach jedem Abfangen eines Ziels jedes verbliebene Ziel und updatet deren neue Priorität. Die Gewichte für die Prioritätsberechnung können dabei willkürlich gewählt werden, allerdings empfiehlt sich ein geeignetes Set zu wählen, um eine schnelle Tour zu ermöglichen. In jeder Iteration wird dabei das Ziel mit der höchsten Priorität ausgewählt und abgefangen. Sofern ein Ziel bereits zwischen zwei anderen eingeholt wurde, wird dieses aus der Warteschlange entfernt. In der Abbruchbedingung befinden sich alle Ziele auf der selben Seite, dieser Teil ist einfach zu berechnen und garantiert die schnellstmögliche Abfolge am Ende der Tour. In jedem Fall wird eine Tour zurückgegeben, dessen Reihenfolge von den gewählten Gewichten abhängig ist.

## 4.4 Brute-Force für zwei orthogonale Achsen im MT-TSP

Der Prioritätsansatz liefert zwar eine effiziente und schnelle Lösung, garantiert aber keine optimalen Ergebnisse. Um die genaue Güte zu bestimmen, ist ein optimaler Algorithmus nötig. Zwar ist dieser für große Eingaben nicht zu gebrauchen, dennoch empfiehlt sich in diesem Fall die Brute-Force-Methode, um die Güte des Prioritäts-Algorithmus abschätzen zu können.

Mit einem Brute-Force werden alle Möglichkeiten durchprobiert, um eine Lösung zu bestimmen, in diesem Fall die optimale Tour. Dafür wird für die Zielliste des Inputs alle Permutationen erstellt. Um alle Permutationen zu bestimmen, wird ein Permutationsbaum generiert. Dieser beginnt bei der Wurzel, wobei für diese auch der Ursprung direkt eingesetzt werden kann. Anschließend wird jedes Ziel als Kind des Wurzelknotens eingesetzt. Die Kinder wiederum bekommen weitere  $k - 1$  Ziele als Kinderknoten, wobei  $k$  die Tiefe des Baumes darstellt. Damit wird sichergestellt, dass keine Ziele in einer Tour



doppelt vorkommen und damit, statt  $10^{10}$ ,  $n!$  Permutationen generiert werden. Mit dieser Variante wird bei  $n$  Zielen insgesamt eine Anzahl von  $V_n = n + n \cdot (V_{n-1})$  Knoten in den Baum eingefügt. Dies kann mittels Rekursion einfach berechnet werden. Für 10 Ziele wären dies also 9.864.100 Einträge.

Anschließend kann für jede der Permutationen die Tour bestimmt werden. Nachdem durch alle der Permutationen iteriert wurde, ist die Permutation mit der kürzesten Tourzeit die optimale Tour. Dabei zu beachten ist, dass es mehrere optimale Touren geben kann, die Tourzeit hingegen bleibt jeweils gleich. Gerade bei wenigen Zielen ist die Anwendung zu empfehlen, da beispielsweise der Input  $Z = \{z_1, z_2, z_3\}$  nur 6 Kombinationen durchrechnen muss (siehe Abbildung 4.1). Wie bereits erwähnt ist die Brute-Force bei Eingaben mit vielen Zielen nicht zu empfehlen. Bereits mit  $n = 10$  Zielen gibt es  $10! = 3.628.800$  Permutationen, die alle ausprobiert werden müssen. Um dem Abhilfe zu verschaffen, kann der Permutationsbaum an vielen Stellen beschnitten werden, sodass große Teile gar nicht erst berechnet werden müssen. Betrachten dafür zwei Szenarien an einer Permutation bis zum Index  $k$ .

1. Sei  $\tau_{min}$  die aktuell schnellste Tourzeit einer Permutation. Der Baum wird nun unterhalb von  $k$  beschnitten, sofern das Ziel  $z_k$  bereits eine größere Tourzeit benötigt, als  $\tau_{min}$ .
2. Überprüfe, ob zwischen dem vorherigen  $z_{k-1}$  und dem jetzigen Ziel  $z_k$  eines der verbleibenden Ziele abgefangen wird. Sofern ein Ziel  $z_i$ ,  $i > k$  dabei angefangen wird, ist dies äquivalent dazu, dass eine andere Permutation existiert, in der  $z_i$  vor  $z_k$  eingeholt wird. Diese wird ggf. später noch berechnet und somit wird der Baum ebenfalls bei  $k$  beschnitten.

Somit werden Permutationen mit der selben Reihenfolge an Zielen bis  $k$  ausgelassen, da diese keine optimale Route erzeugen werden. Zusätzlich kann der Algorithmus in den meisten Fällen mit einer Sortierung nach der Geschwindigkeit in absteigender Reihenfolge verbessert. Oftmals wird damit schnell ein gutes  $\tau_{min}$  gefunden und damit werden umso mehr Pfade abgeschnitten. Darüber hinaus könnte ebenfalls ein Prioritätsmaß genutzt werden, ist aber nicht notwendig. Mit diesen Bedingungen wird im Beispiel von Abbildung 4.1 2 Berechnungen eingespart (siehe Abbildung 4.2). Bei jeder Permutation wird am Anfang und am Ende der Ursprung eingerechnet.

Zuletzt kann die Laufzeitkomplexität stark verbessert werden, indem der Permutationsbaum *on-the-fly* generiert wird. Dafür wird für jeden Knoten bzw. jedes Ziel direkt alle Berechnungen durchgeführt und äquivalent zu den Beschneidungen, bricht der Algorithmus einfach bei dem Eintritt eines der oben genannten Szenarien ab. Formal zusammengefasst wird der Brute-Force-Algorithmus in Algorithmus 3.

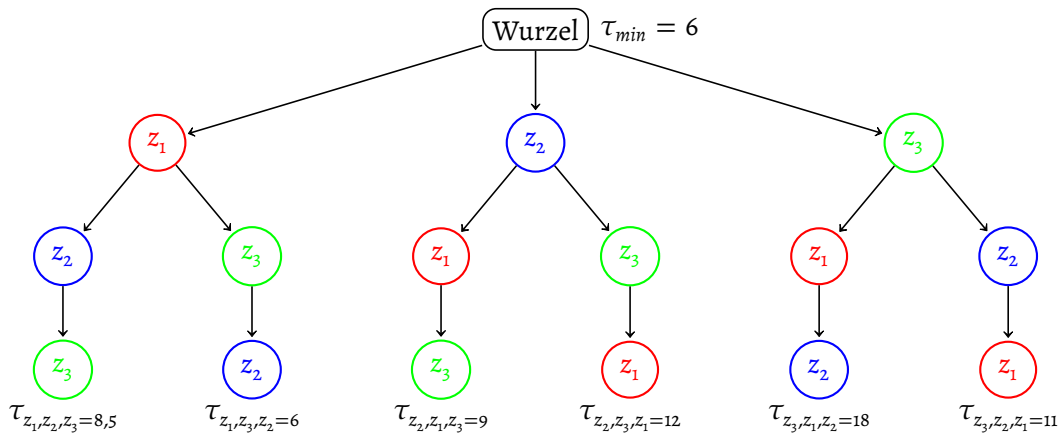


Abbildung 4.1: Die Berechnung der jeweiligen (Teil-)Tourzeit  $\tau_i$ ,  $1 \leq i \leq n!$  ist in diesem Fall bei nur  $n! = 3! = 6$  Permutationen noch problemlos.

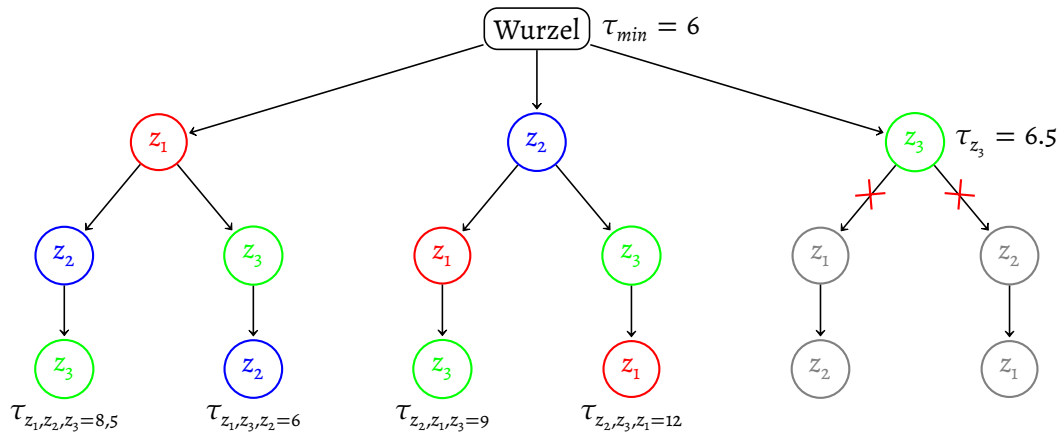


Abbildung 4.2: Mit der Brute-Force-Optimierung wird direkt hinter  $z_3$  abgeschnitten, da an diesem Knoten bereits der Zeitpunkt  $\tau_{min}$  überschritten wird.

**Algorithmus 3** Brute-Force-Algorithmus für zwei-orthogonale Achsen beim MT-TSP**Input:** Targets  $Z$ , pursuer**Output:** Targets  $Z$  Targets  $Z$  in order of nondecreasing intercepting timeSort  $Z$  in order of nonincreasing absolute values of the respective velocitiesLet  $currentTargets$  be the current target order (partial permutation)Let  $t$  be the time-array, which represents the intercepting time for each  
target  $z_i \in currentTargets$  $\tau_{min} \leftarrow \infty$  $current \leftarrow z_0$  $prev \leftarrow origin$ , which is determined by the start position of the pursuer**while** there are possible permutations remaining **do**     $current \leftarrow$  the target just intercepted     $prev \leftarrow$  the target previously intercepted     $t[current] \leftarrow t[prev] + \pi[prev \rightarrow current]$     **if**  $t[current] \geq \tau_{min}$  or at least one target of  $Z \setminus currentTargets$  is located  
    between  $current$  and  $prev$  **then**

Step back and follow the next possible permutation-path

**else**        **if**  $currentTargets.length \neq Z.length$  **then**            Choose the next target  $z_i, z_i \notin currentTargets$         **else**             $t[current] \leftarrow t[current] + \pi[current \rightarrow ursprung]$             **if**  $t[current] < \tau_{min}$  **then**                 $\tau_{min} \leftarrow t[current]$ 

Step back and follow the next possible permutation-path

**end if**        **end if**    **end if****end while**

## Laufzeitkomplexität Brute-Force-Ansatz

**TODO: Überarbeiten, diese ist nun viel besser** Zunächst wird die benötigte Laufzeit zur Erstellung des Baums betrachtet. Dafür werden  $V_n = n + n \cdot (V_{n-1})$  Knoten eingefügt. Die Generierung des Baums kann demnach mit  $O(n!)$  abgeschätzt werden.

Trotz der zuvor vorgeschlagenen Verbesserungen des Brute-Force-Algorithmus muss der worst-case betrachtet werden. Dabei wird der Baum an keiner Stelle wegen der Tourzeit-Bedingung beschnitten, da  $\tau_{min}$  nach jeder Permutation kleiner wird. Der Fall ist zwar sehr unwahrscheinlich, aber möglich. Dies würde in eine Laufzeitkomplexität von  $O(n!)$  resultieren. Allerdings werden mit dem zweiten Szenario definitiv Teile des Baumes abgeschnitten, da immer Ziele zwischen den einzelnen Zielen eingeholt werden<sup>3</sup>. An sich

---

<sup>3</sup> Dies gilt für  $n > 2$

benötigt die Berechnung eine Laufzeit von  $O(n^2)$ , schneidet dafür aber große Teile des Baums weg. Es hat sich gezeigt, dass bei den eigentlichen  $n!$  Permutationen meistens weniger als 5% der Permutationen letztendlich ausgerechnet werden<sup>4</sup>.

Nach verschiedenen worse-case-Analysen hat sich schlussendlich gezeigt, dass die Erstellung des Permutationsbaums mit Abstand die größte Laufzeit einbüßt. Somit hat der Brute-Force-Algorithmus eine Zeitkomplexität von  $O(n!)$ .

#### Korrektheit Brute-Force-Ansatz

Mit dem Brute-Force-Algorithmus wird jede mögliche Permutation an Zielen ausprobiert. Bei der Berechnung einer Permutation werden einfach alle Ziele nacheinander abgefangen und der Zeitstempel mit der Zeit erhöht, die zum Abfangen des jeweiligen Ziels benötigt wird. Sofern eine Permutation schneller als die aktuell schnellste, wird diese übernommen und  $\tau_{min}$  mit der benötigten Tourzeit überschrieben. Mit den Verbesserungen wird der Permutationsbaum beschnitten und es werden Berechnungen eingespart. Somit wird garantiert, dass die optimale Tour zurückgegeben wird.

---

<sup>4</sup> Später dazu mehr im Kapitel Experimente

# 5

## Experimente

### TODO

Meine Vorschläge für dieses Kapitel:

- 1D für  $n = 10, 100, 1000, 10000$  testen, dabei eine Laufzeitkurve erstellen und analysieren
- Brute-Force-Algorithmus auf Effektivität testen mit zufällig generierten Inputs für  $n = 10, 12, 14$
- 1.000.000 mal den Prioritäts-Algorithmus gegen den BF-Algorithmus laufen lassen. Mit den Tourzeiten die Güte des 2OA-Algorithmus bestimmen.
- 2OA-Algorithmus in allen Kombinationen der folgenden Parametern testen:

$$n = 5, 25, 50, 75, 100$$

$$(v_{\text{verfolger}}, v_i) = (200, 20), (100, 40), (59, 60)$$

$$\text{Startpositionen} = [10.000]$$

Die zweite Notation bedeutet dabei, dass die Verfolgergeschwindigkeit für 200, 100, 59 und die Zielgeschwindigkeit (fest gesetzt für alle Ziele) für 20, 40, 60 getestet wird (Parameterwahl nach [9] orientiert).

# 6

## Zusammenfassung und Ausblick

### TODO: Zusammenfassung der Ergebnisse, v.a. aus Experimente

Zwar wurden die Probleme bei der Modellierung als Graphen aufgezeigt, ausgeschlossen sei diese allerdings nicht. Mit dem Ansatz über das Lemma 2 sollte eine optimale Strategie definitiv möglich sein und eine geringere Laufzeit aufweist, als der Brute-Force-Algorithmus mit  $O(n!)$ . Der Brute-Force-Ansatz kann zudem um einiges verbessert werden, indem der Permutationsbaum bereits bei der Generierung die einzelnen Ziele berechnet. Somit werden viele Permutationen gar nicht erst erstellt. Zwar wird damit eine deutlich bessere Laufzeit erzielt, bei worst-case-Fällen mit Eingaben von  $n > 14$  ist Brute-Force allerdings weiterhin untauglich.

Mit dem Prioritätsansatz wurde zum Einen ein effizienter und mit dem Brute-Force-Ansatz zum Anderen ein optimaler Algorithmus für wenig Ziele vorgestellt. Als nächsten Schritt wäre ein einigermaßen effizienter, aber optimaler Algorithmus vom großen Interesse. Dies wäre ein notwendiger Schritt, um theoretische Arbeit für  $k$ -Achsen in MT-TSP zu leisten. Darüber hinaus ist das nächste Ziel in dieser Thematik die Betrachtung von zweidimensionalen MT-TSP und das Aufstellen von Heuristiken. Diesen Jahres wurde für generelle Fälle eine Arbeit[8] veröffentlicht, in der mit evolutionären Strategien versucht wird, auf eine optimale Tour in vielen Iterationen hinzuarbeiten. Dabei ist allerdings die optimale Tour von vielen verschiedenen Parametern abhängig und garantieren auch keine optimale Lösung.

## Literatur

- [1] Brandstädt, A. Kürzeste Wege. In: *Graphen und Algorithmen*. Springer, 1994, S. 106–123.
- [2] Choubey, N. S. Moving target travelling salesman problem using genetic algorithm. In: *International Journal of Computer Applications* 70(2), 2013.
- [3] Hammar, M. und Nilsson, B. J. Approximation results for kinetic variants of TSP. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1999, S. 392–401.
- [4] Helvig, C. S., Robins, G. und Zelikovsky, A. The moving-target traveling salesman problem. In: *Journal of Algorithms* 49(1):153–174, 2003.
- [5] Kaaser, D. S. Algorithmen und Datenstrukturen. In: 2014.
- [6] Künzi, H. P. und Nievergelt, E. *Einführungskursus in die dynamische Programmierung*. Bd. 6. Springer-Verlag, 2013.
- [7] Minx, E. und Dietrich, R. *Autonomes Fahren*. Piper ebooks, 2015.
- [8] Moraes, R. S. de und Freitas, E. P. de Experimental Analysis of Heuristic Solutions for the Moving Target Traveling Salesman Problem Applied to a Moving Targets Monitoring System. In: *Expert Systems with Applications* 136:392–409, 2019.
- [9] Stieber, A., Fügenschuh, A., Epp, M., Knapp, M. und Rothe, H. The multiple traveling salesmen problem with moving targets. In: *Optimization Letters* 9(8):1569–1583, 2015.
- [10] Weicker, K. *Evolutionäre Algorithmen*. Springer-Verlag, 2015, S. 24–25.

# A

## Implementierungen

TODO