UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Algorithmen für bewegende Ziele im 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

Danksagungen

Inhaltsverzeichnis

1

Einleitung

2

Grundlagen

Im folgenden Kapitel werden alle nötigen Grundlagen für bewegende Ziele im Travelling-Salesman-Problem erläutert. Dabei werden zwei konkrete Fälle vorgestellt:

1. eindimensionaler Fall: Jedes Ziel kann sich nur auf einer Linie bewegen
2. zwei-orthogonale-Achsen-Fall: Erweitert den eindimensionalen Fall um eine orthogonal, auf der ersten Linie, liegenden Achse, auf der sich die Ziele bewegen können.

Die Approximation für generelle Fälle gilt dabei schwierig, da viele verschiedene Faktoren die Komplexität des Problems bestimmen. Die Approximations-Forschung in [2] zeigte, dass sich die Probleme nicht besser als mit einem Faktor von $2^{\pi(\sqrt{\pi})}$, es sei denn $P = NP$. Dies gilt bisher allerdings weiterhin als ungelöstes Problem in der Informatik.

2.1 Fallübergreifende Definitionen

TODO: kleine Einleitung

Definition 2.1. Jede Instanz enthält eine Anzahl n von Zielen $Z = \{z_1, ..., z_n\}$. 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 , $p_i, v_i \in \mathbb{R}$. Die Positionen und Geschwindigkeiten können dabei als Vektoren P und V dargestellt werden.

$$P = (p_1, ..., , p_n)^T$$
$$V = (v_1, \ ..., \ , v_n)^T$$

Demnach kann ein Ziel als ein Tupel $z_i = (p_i, v_i)$ dargestellt werden. Der Ursprung ist definiert durch einen Punkt ohne Geschwindigkeit. Das Tupel $(-1, 0)$ würde also 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 kann sich ebenfalls nur auf den Achsen bewegen. Sein Ziel ist die schnellst mögliche Tour zu finden, um alle Punkte abzufangen. Dabei bewegt sich

der Verfolger mit der Maximalgeschwindigkeit

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

Dies stellt sicher, dass der Verfolger nach einer gewissen Zeit jedes Ziel auf jeden Fall eingeholt hat. Andernfalls würde eine unendlich große Tourzeit berechnet werden. Der Ursprung ist gleichzeitig auch der Ziel der Tour. Demnach startet und endet jede Tour an diesem stationären Punkt.

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.

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

definiert. Die Berechnung beruht auf der nach der Zeit (in diesem Fall τ) gleichgesetzten und umgestellten physikalischen Formel [1]. Bemerke, v_A wird durch $v_{verfolger}$ ersetzt, da sich der Verfolger immer mit maximaler Geschwindigkeit bewegt.

Definition 2.5. Eine optimale Tour ist die kürzeste Reihenfolge an Zielen, bei dem jedes $z_i \in Z$ eingeholt wurde. Dabei startet und beendet der Verfolger jede Tour im Ursprung.

2.2 Eindimensionaler Fall

Der eindimensionale Fall wurde zuerst in [3] präsentiert. Die Autoren modellieren das Problem als Graph-Problem, wobei der eigentliche Graph on-the-fly erstellt wird. Mit linearer Programmierung wird anschließend die schnellst mögliche Abfangzeit für jeden Zustand bestimmt. Somit kann die 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 $v_{verfolger}$ bewegt. Helvig et. al. bewiesen dies mit dem Aspekt, dass ein eine Reisegeschwindigkeit von $v < v_{verfolger}$ 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

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

[1] Gleichförmige Bewegung: $s = v \cdot t + s_0$

– 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 $t \to \infty$, sobald die äußersten Ziele noch deutlich weiter entfernt vom

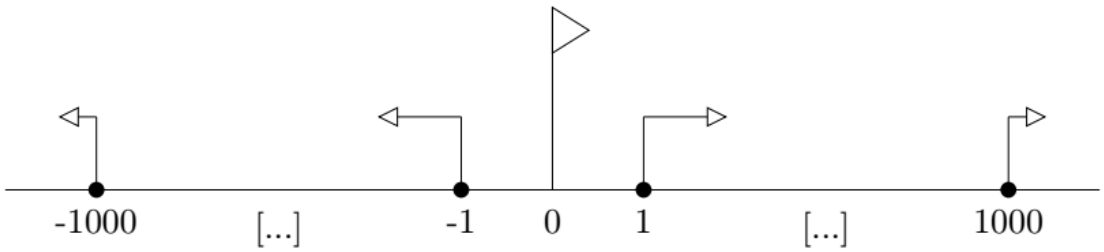


Abbildung 2.1: Offensichtlich würde der Verfolger mit der Geschwindigkeit $v_{verfolger} = 11$ deutlich länger für eine Tour brauchen, sofern er zunächst alle Ziele auf der einen und dann auf der anderen Seite abarbeitet. Hierbei wäre es sinnvoll, zunächst die Ziele $(-1, -10)$ und $(1, 10)$ einzuholen.

Ursprung aus liegen.

Die Autoren aus [3] definierten anschließend 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. rechten Seite des Verfolgers. Sofern der Verfolger vor einem Wendepunkt umkehrt, verlängert sich die Tour und ist damit nicht mehr optimal.

Definition 2.6. 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_0 und A_{final} keine Tupel. Dabei handelt es sich um den Start und Endzustand, welche bei jeder Tour gleich sind. Der Verfolger befindet bei beiden dieser

Zuständen 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 $t[A_0] = 0$.

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 (siehe Abbildung TODO: Bsp mit den Zielen aus der Abbildung 2.1). Somit befinden sich die Kombinationen bzw. Zustände aus den schnellsten Zielen am Listenanfang von *States*.

Definition 2.7. Ein Zustandsübergang von Zustand A in den Zustand B wird mit

$$\tau = A \rightarrow B$$

definiert. Der Übergang τ gibt dabei die Zeit τ (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 τ_{left} und τ_{right} bezeichnet. Sofern jedes Ziel auf einer Seite eingeholt wurde, wird der Übergang τ_{final} in A_{final} gewählt. Für die Berechnung von τ_{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_0 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 τ_{Left} und τ_{right} führt jeder Zustand in einen anderen Zustand mit einem höheren Index. Dabei erhält der Startzustand A_0 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. Anhand der Abbildung 2.3 (siehe Abbildung TODO: Bsp mit der Abbildung 2.2) ist diese Modellierung anhand des bisherigen Beispiels aus Abbildung 2.1 und 2.2 gut nachvollziehbar.

Definition 2.8. Ein Graph sei durch

$$G = (V, E)$$

definiert. Hierbei werden die Knoten V durch die Zustände $A_i \in States$ und E durch die jeweiligen Zustandsübergänge τ 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_0) 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_0 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.

2.3 Algorithmus von Helvig, Robins and Zelikovsky

Mit diesen Voraussetzungen haben die Autoren von [3] 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 τ_{final} in A_{final}. Berechne die Zeit, um die verbleibenden Ziele auf der anderen Seite einzuholen und zusätzlich die Retour zum Ursprung.
- Berechne ansonsten τ_{Left} und τ_{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 (auch 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.

2.4 Zwei-orthogonale-Achsen-Fall

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 Ziele 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, während der Ursprung, im Gegensatz zum 1D-Fall (fest definiert auf die Koordinate 0), auf einer der Achsen liegen kann.

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 a nötig. Die Position p_i wird dabei mit

$$p_i = (\mathbb{R}, a) \mid a \in \{true, false\}$$

neu definiert. Dabei gibt a 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.

2.5 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

Theoretische Grundlagen für den zwei-orthogonale-Achsen-Fall

In diesem Kapitel soll mit der Vorarbeit aus [3] gezeigt werden, dass einige Eigenschaften des 1D-Falls genauso auch im zwei-orthogonale-Achsen-Fall (2OA-Fall) gelten.

Hilfssatz 3.1. *In jeder optimalen Tour beim 2OA-Fall beim bewegende Ziele in TSP bewegt sich der Verfolger immer mit maximaler Geschwindigkeit.*

Beweis. Fallunterscheidung:

1. Das nächste Ziel des Verfolgers liegt auf der selben Achse:
   Mit dem Beweis für 1D-Fälle in [3] gilt dies für diesen Fall.
2. Das nächste Ziel des Verfolgers bewegt sich auf der anderen Achse:
   Indirekter Beweis: Nehmen an, der Verfolger bewegt sich mit $v_{verfolger} < v_{max}$. Dies ist äquivalent dazu, dass der Verfolger an seiner aktuellen Position eine Zeit τ wartet und sich dann mit v_{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 Schnittpunkt 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 τ. 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 Schnittpunkt wartet, sondern von diesem aus p direkt erreicht. Bis zum Zeitpunkt t_2 wird nun wieder für die Dauer von τ 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 Schnittpunkt 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 $v_{verfolger} = v_{max}$.

□

4

Heuristiken für den zwei-orthogonale-Achsen-Fall

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 des zwei-orthogonale-Achsen-Falls mit Zuständen als Graphen

Im 1D-Fall konnte über über die topologische Reihenfolge der Wendepunkte bzw. Zustände ein Graph erzeugt werden. Somit wurde mit der Bestimmung des kürzesten Pfads das Problem die optimale Tour bestimmt. Es wäre also zunächst sinnvoll, diesen Ansatz für den 2OA-Fall zu übernehmen, da nur eine Achse hinzugekommen ist. Hierfür werden wieder die einzelnen Ziele in die Listen *Left, Right, Top, Bottom* aufgeteilt. Bei dem Algorithmus von [3] 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 Schnittpunkt bewegen. Das liegt daran, dass Ziele nun den Schnittpunkt 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. Zunächst [...] TODO: Finish section

TODO: Abbildung

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.

Definition 4.1. Mit den Faktoren ϕ wird die Priorität eines Ziels berechnet und werden definiert durch

$$\kappa = \{\varphi_1, ..., \varphi_5\}^T \mid k_i \in \mathbb{R}.$$

Definition 4.2. Die Gewichte ω beeinflussen die Faktoren ϕ, um den jeweiligen Anteil an der Priorität zu erhöhen bzw. zu verringern und werden definiert durch

$$\omega = \{w_1, ..., w_5\}^T \mid w_i \in \mathbb{R}.$$

Definition 4.3. Der Geschwindigkeitsfaktor ϕ_1 erhöht Priorität eines schnellen Ziels z_i und ist definiert durch

$$\phi_1(z_i, w_1) = \frac{|v_i|}{v_{verfolger}} \cdot w_1.$$

Definition 4.4. Der Positionsfaktor ϕ_2 erhöht Priorität bei Zielen, die sich vom Ursprung wegbewegen, andernfalls verringert sich die Priorität. Dies wird mit den Faktor $a \in \{-1, 1\}$ verrechnet. ϕ_2 wird definiert durch

$$\phi_2(z_i, w_2) = \frac{|p_i|}{v_{verfolger}} \cdot a \cdot w_2.$$

Definition 4.5. TODO: Sicherstellen, dass diese Def. notwendig ist, ich vermute nein
Der Schnittpunktsfaktor ϕ_3 verringert die Priorität bei Zielen in der Nähe des Schnittpunktes und ist definiert durch

$$\phi_3(z_i, w_3) = \left|\frac{\|o, p_i\|_1}{o - v_i}\right| \cdot w_3 \qquad \text{(siehe Definition 2.4).}$$

Diese Definition scheint im Konflikt mit Definition 4.4 zu stehen. Allerdings fördert die Kombination aus den beiden Definitionen weit entfernte Ziele und solche, von denen man schnell die anderen Achsen erreichen kann.

Definition 4.6. TODO: Sicherstellen, dass diese Def. notwendig ist, ich vermute nein
Der Ursprungsfaktor ϕ_4 erhöht die Priorität bei Zielen Nahe des Ursprungs geringfügig und ist definiert durch

$$\phi_4(z_i, w_4) = \left|\frac{\|p_{ursprung}, p_i\|_1}{o - v_i}\right| \cdot w_4 \quad \text{(siehe Definition 2.4).}$$

Definition 4.7. Der Distanzfaktor ϕ_5 verringert die Priorität bei großen Abständen zur aktuellen Position und ist definiert durch

$$\phi_5(z_i, w_5) = \left|\frac{\|p_{verfolger}, p_i\|_1}{v_{verfolger} - v_i}\right| \cdot w_5 \quad \text{(siehe Definition 2.4).}$$

Definition 4.8. Die Priorität $\alpha \in \mathbb{R}$ eines Ziels z_i bestimmt die Wichtigkeit eines Ziels, als nächstes eingeholt zu werden

$$\alpha(z_i, \omega) = \varphi_1(z_i, w_1) + \varphi_2(z_i, w_2) - \varphi_3(z_i, w_3) - \varphi_4(z_i, w_4) - \varphi_5(z_i, w_5)$$

weqhgurfwef

Definition 4.9. Die Prioriätätswarteschlange Q ist eine sortierte Liste an verbleibenden Zielen in absteigender Reihenfolge nach α und ist definiert durch

$$Q = T \subseteq Z.$$

Dabei wird das erste Element von Q als nächstes Ziel eingeholt.

4.3 Algorithmus mit dem Prioritätsansatz

In diesem Abschnitt wird ein konkreter 20A-Fall-Algorithmus vorgestellt. Während der 1D-Algorithmus aus [3] 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 Schnittpunkt 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 α neu berechnet und in Q absteigend nach α 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 1.

Algorithmus 1 Prioritäts-Algorithmus für zwei-orthogonale Achsen beim bewegende Ziele in TSP

Input: Ziele Z, Ursprung $z_{ursprung}$, Verfolgergeschwindigkeit v_{max}
Output: Ziele Z in der Tour-Reihenfolge, inklusive Retour zum Ursprung

Sei t das Zeit-Array, welches für jedes $z_i \in Z$ die Abfangzeit angibt
Sei *current* das Ziel, welches der Verfolger soeben eingeholt hat
Sei *OUTPUT* die Liste an Zielen in der Abfangreihenfolge
current $\leftarrow$ *ursprung*
OUTPUT.*add*(*current*)
Q $\leftarrow$ Prioritätswarteschlange mit Zielen, welche diese in absteigender Reihenfolge nach ihrer Priorität sortiert.
for $z_i \in Z$ **do**
    $t[z_i] \leftarrow \infty$
    Q.*add*(z_i)
    Berechne $\alpha(z_i)$
end for

while Q beinhaltet verbleibende Ziele **do**
    if jedes verbleibende Ziel liegt auf einer der vier Seiten des Schnittpunktes **then**
        for $z_i \in Q$ **do**
            $t[z_i] \leftarrow$ Zeit von der aktuellen Position bis zum Abfangen von z_i
        end for
        Berechne Rückkehr zum Ursprung von dem Ziel, welches als letztes aus Q erreicht wird
        OUTPUT.*addAll*(Q)
        break
    end if

    Berechne $\alpha(z_i)$, $\forall z_i \in Q$ und update Q
    prev $\leftarrow$ *current*
    current $\leftarrow$ Q.*poll*()
    $t[current] \leftarrow time[prev] + \pi[prev \rightarrow current]$
    OUTPUT.*add*(*current*)
    Update die Position von jedem $z_i \in Q$
    eingeholteZiele $\leftarrow$ Ziele zwischen *prev* und *current*
    Q.*removeAll*(*eingeholteZiele*)
    OUTPUT.*addAll*(*eingeholteZiele*)
end while

Sortiere *eingeholteZiele* in aufsteigender Reihenfolge nach $t[z_i]$
return *OUTPUT*

4.4 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 durchiterieren der Warteschlange werden die soeben genannten Laufzeiten um den Faktor n erhöht. Jegliche Sortierungen befinden sich außerhalb der Warteschlange und haben damit eine Zeitkomplexität von $O(n \log(n))$.

Somit ergibt sich beim Prioritätsalgorithmus eine Gesamtzeitkomplexität von $O(n^2)$.

4.5 Korrektheit Prioritätsansatz

TODO

4.6 Brute-Force im 2OA-Fall

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 den Algorithmus abschätzen zu können.

Mit einem Brute-Force werden alle Möglichkeiten durchprobiert, um eine Lösung zu bestimmen, in unserem 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 beispielweise 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 τ_{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 τ_{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 τ_{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 Beipiel von Abbildung 4.1 2 Berechnungen eingespart (siehe Abbildung 4.2). Bei jeder Permutation wird am Anfang und am Ende der Ursprung eingerechnet. Formal zusammengefasst wird der Brute-Force-Algorithmus in Algorithmus 2.

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

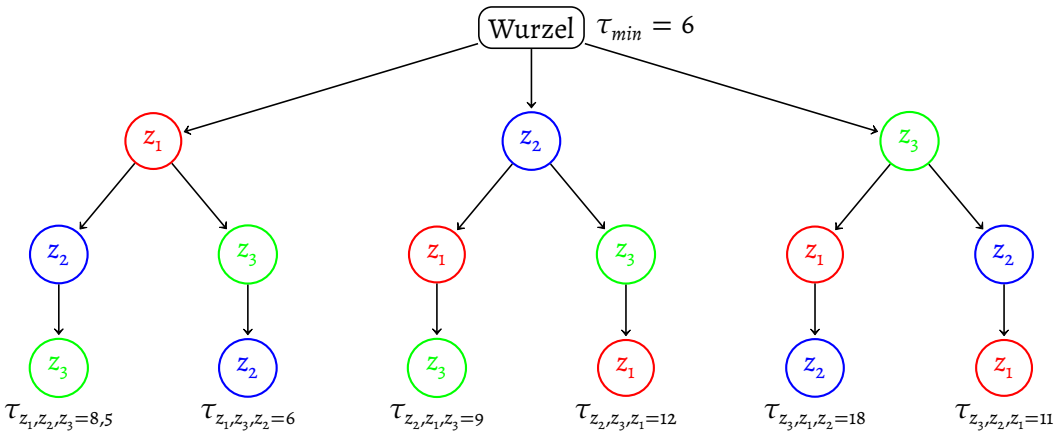
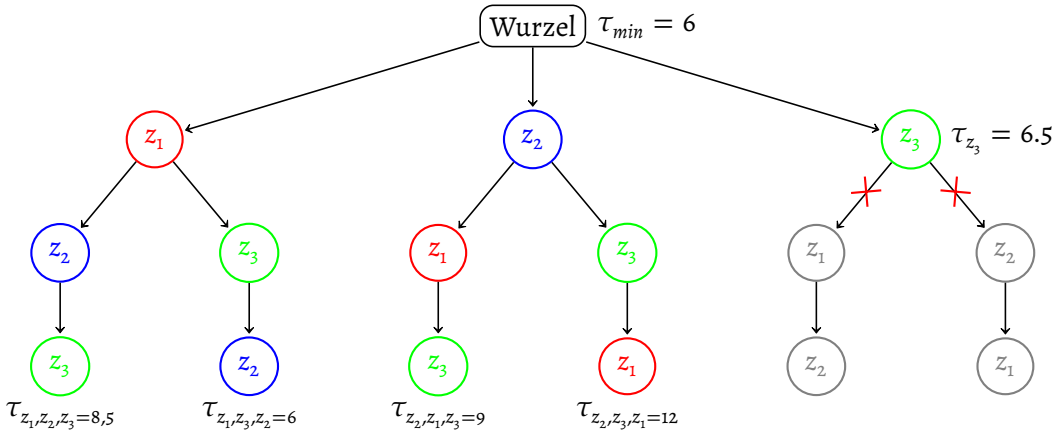


Abbildung 4.2: Mit der Brute-Force-Optimierung wird direkt hinter z_3 abgeschnitten, da an diesem Knoten bereits der Zeitpunkt τ_{min} überschritten wird.



Algorithmus 2 Brute-Force-Algorithmus für zwei-orthogonale Achsen beim bewegende Ziele in TSP

Input: Ziele Z, Ursprung $z_{ursprung}$, Verfolgergeschwindigkeit v_{max}
Output: Ziele Z in der Tour-Reihenfolge, inklusive Retour zum Ursprung

$B \rightarrow$ Permutationsbaum, bei n Zielen muss es n Kinderknoten geben.
$\tau_{min} \leftarrow \infty$
Sei *aktuelleZiele* das Array mit den aktuellen Indizes von Z, initialisiert mit einsen
Sei k die aktuelle Tiefe im Baum
Sei t das Zeit-Array welches für die aktuelle Ziel-Reihenfolge die Abfangzeit angibt
$k \leftarrow 1$
while $aktuelleZiele[0] \leq n$ und $aktuelleZiele[n] \leq n$ **do**
    $current \leftarrow Z[aktuelleZiele[k]]$
    $prev \leftarrow Z[aktuelleZiele[k-1]]$
    $t[current] \leftarrow t[prev] + \pi[prev \rightarrow current]$
    if $t[current] \geq \tau_{min}$ oder mindestens ein Ziel der verbliebenen Indizes wird zwischen
    current und *prev* abgefangen **then**
        if $aktuelleZiele[k] = n$ **then**
            Dekrementiere k
            Inkrementiere $aktuelleZiele[k]$
        else
            Inkrementiere $aktuelleZiele[k]$
        end if
    else
        if $k < n$ **then**
            Inkrementiere k
        else
            $t[current] \leftarrow t[current] + \pi[current \rightarrow ursprung]$
            if $t[current] < \tau_{min}$ **then**
                $\tau_{min} \leftarrow t[current]$
                Dekrementiere k
                Inkrementiere $aktuelleZiele[k]$
            end if
        end if
    end if
end while

4.7 Laufzeitkomplexität Brute-Force-Ansatz

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 τ_{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 der 2. Bedingung definitiv Teile des Baumes abgeschnitten, da immer Ziele zwischen den einzelnen Zielen eingeholt werden[2]. An sich 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[3].

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

4.8 Korrektheit Brute-Force-Ansatz

Mit dem Brute-Force-Algorithmus wird jede mögliche Permutation an Zielen ausprobiert. Sofern eine Permutation schneller als die aktuell schnellste, wird diese übernommen und τ_{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.

[2] Dies gilt für $n > 2$
[3] Später dazu mehr im Kapitel Experimente

5

Experimente

TODO

6

Zusammenfassung und Ausblick

TODO

Literatur

[1] Brandstädt, A. Kürzeste Wege. In: *Graphen und Algorithmen*. Springer, 1994, S. 106–123.

[2] 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.

[3] Helvig, C. S., Robins, G. und Zelikovsky, A. The moving-target traveling salesman problem. In: *Journal of Algorithms* 49(1):153–174, 2003.