* Willkommen
* Bei BA mit Moving-Target-TSP beschäftigt
* Kurzer Überblick

**Einleitung**

**Folie 1**

* Das TSP ist ein Optimierungsproblem aus der Kombinatorik
* Die Aufgabe besteht darin, eine Reihenfolge für den Besuch mehrerer Ziele so zu wählen, dass die gesamte Reisestrecke minimal ist
* Dabei muss jedes Ziel min. 1x besucht werden, jede Tour endet wieder im Startpunkt
* Das TSP ist NP-vollständig, sodass das generelle Problem nicht in polynomieller Zeit gelöst werden kann

**Folie 2**

* Dies hat zum Beispiel für Städtereisen eine hohe Relevanz
* In diesem Fall ist die 1. Rundreise nicht optimal, während die 2. die schnellst-mögliche ist
* Dies ist allerdings nicht in allen Fällen so offensichtlich, wofür es verschiedene Lösungsstrategien gibt

**Folie 3**

* Das Moving Target TSP wurde das erste Mal im Jahre 1998 von Helvig et al. erwähnt
* Dies ist eine Spezialinstanz des herkömmlichen TSP in dem Sinne, dass die Ziele nun nicht mehr unbedingt stationär sind, sondern eine konstante Geschwindigkeit besitzen
* Die generelle Problematik bleibt allerdings dieselbe, das heißt es wird wieder die schnellst-mögliche Tour durch alle Ziele und Rückkehr zum Startpunkt gesucht

**Folie 4**

* Als Anwendungsbeispiel betrachten wir ein Versorgungsschiff, dessen Aufgabe ist, alle Flugzeugträger zu versorgen.
* Man könnte nun behaupten, diese bewegen diese sich nicht immer mit einer konstanten Geschwindigkeit.
* Beim herkömmlichen TSP fließen allerdings auch weitere Faktoren mit ein, z.B. Stau auf der Autobahn, schlechtes Wetter etc., sodass man bei beiden Berechnungen jeweils zusätzliche Faktoren mitberechnet werden müssten

**Grundlagen**

**Folie 1**

* Die Autoren Helvig et. al. haben das Problem wie folgt definiert: -ablesen-

**Folie 2**

* Das heißt wir haben eine Anzahl von Zielen Z
* Diese haben jeweils eine Startposition und eine Geschwindigkeit
* Die Position ändert sich also nach einer Zeiteinheit um die Geschwindigkeit
* Zudem haben wir einen Verfolger, welcher ebenfalls eine Geschwindigkeit besitzt und an einer Position startet
* Die Geschwindigkeit ist dabei echt größer, als die des schnellsten Ziels, sodass der Verfolger in jedem Fall alle Ziele nach einer gewissen Zeit abfängt
* Die Startposition wird im Folgenden als Ursprung definiert

**Folie 3**

* Gegeben haben wir nun 4 Ziele mit jeweils 2 Zielen auf jeder Seite des Ursprungs
* Die erste Strategie wäre zunächst alle Ziele einer Seite abzufangen und dann die auf der anderen Seite, um dann diese Tourzeit mit der umgekehrten Reihenfolge zu vergleichen
* Diese Strategie funktioniert leider nicht so einfach, da eines der Ziele (1,8) oder (-1,-8) sich sehr schnell weit weg vom Ursprung entfernt und der Verfolger danach lange brauchen würde, dieses Ziel einzuholen
* Stattdessen haben sich die Autoren des eben erwähnten Papers eine andere Strategie überlegt. Dafür werden nur noch alle potenziellen Wendepunkte betrachtet. In dem Fall würde ein Ziel, welches sich zwischen 1 und 1000 befindet und eine Geschwindigkeit zwischen 1 und 8 besitzt, sowieso eingeholt werden und somit für die Tourberechnung zunächst weggelassen werden
* Für die Berechnung der Tour werden nun die Ziele in die Listen Left und Right eingesetzt, je nachdem, auf welcher Seite sich diese vom Ursprung befinden. Die Listen sind jeweils in absteigender Reihenfolge nach der Geschwindigkeit sortiert, sodass immer die schnellsten Ziele vorne stehen. Dies ist für den nächsten Schritt wichtig
* Es werden nun Zustände generiert. Zustände sind Momentaufnahmen einer Tour. Diese bestehen aus sk – aktuelles Ziel, welches der Verfolger gerade abgefangen hat und sf- das schnellste Ziel auf der anderen Seite des Ursprungs
* Nun werden alle möglichen Kombinationen aus den Listen Left und Right gebildet und für sk und sf eingesetzt. Die Indizes der beiden Listen werden dabei zusammenaddiert, wodurch wir jeweils einen Summenwert erhalten. Bei A0 und A9 handelt es sich um den Start- und Endzustand, diese sind quasi leer, aber der Verfolger befindet sich dann quasi jeweils im Ursprung. Lasst uns nun merken, dass wir 2x SW 0 haben, 4x 1 und 2x 2. Zudem befinden sich nun die schnellsten Ziele in den Zuständen mit dem kleinsten SW. Die Zustände ordnen wir im folgenden Schritt in aufsteigender Reihenfolge nach den Summenwerten an

**Folie 4**

* Es sind nun die Zustände in aufsteigender Reihenfolge nach den SW übereinander angeordnet. SW 0: 2x, SW 1: 4x und SW 2: 2x. Den Start und Endzustand fügen wir vorne bzw. hinten ein
* Die Zustandsübergänge können einfach berechnet werden. Dafür wird die Zeit vom aktuellen sk des derzeitigen Zustands bis zum sk des darauffolgenden Zustands gemessen.
* Durch z.B. dynamische Programmierung kann nun die optimale Tour bestimmt werden, da mit dieser Modellierung ein Graph generiert wurde. Mit den jeweiligen sk’s wird dann die Tourreihenfolge bestimmt.
* O(n^2), opt. Ergebnisse

**Zwei-orthogonale-Achsen im MT-TSP**

**Folie 1**

* Ich habe nun im Rahmen meiner BA eine neue Modifikation des MT-TSP untersucht. Dafür wird die Achse des 1D-Falls um eine orthogonale erweitert.
* Jegliche Bewegungen des Verfolgers sind dabei auf die Achsen beschränkt, wobei der Verfolger am Schnittpunkt die Achse wechseln darf
* Der Ursprung ist in der Arbeit einfachheitshalber auf den Schnittpunkt der Achsen gesetzt

**Folie 2**

* In der Arbeit von Helvig et al. wurden Lemmata für optimale Touren aufgestellt. Diese habe ich für den 2oA-Fall untersucht und gezeigt, dass diese gelten.
* Es gilt, dass der Verfolger jederzeit mit maximaler Geschwindigkeit reist, da sonst Wartezeit erzeugt wird und diese einfach an das Ende der Tour verschoben werden kann. Damit wäre die Tour nicht mehr optimal.
* 2. Lemma ablesen
* Bei diesem habe ich in der BA den Fall nicht berücksichtigt, in dem der Verfolger sich im Ursprung befindet. Dieser gilt allerdings auch.

**Folie 3**

* Der erste Gedanke war nun, wieder die Modellierung als Graphen zu nutzen, um die optimale Tour zu bestimmen
* Dabei traten allerdings eine Reihe von Problemen auf, die ich nicht lösen konnte
* Zunächst würde man die Ziele gemäß des vorherigen Algorithmus in die Listen Left Right Top und Bottom aufteilen. Während der Verfolger nun das schnellste Ziel auf dem oberen Abschnitt der vertikalen Achse verfolgt, kann das Ziel (-10,5) den Ursprung überqueren und wäre damit auf der rechten Seite des Verfolgers. Diesbezüglich müsste das Ziel in Right eingefügt werden, wodurch ein komplett neuer Graph entstehen würde.
* Man könnte jetzt sagen, dass einfach zu einem bestimmten Zeitpunkt geschaut werden, welches das schnellste Ziel auf der jeweiligen Seite ist. Dies stellt allerdings ein großes Problem dar, da man nun mit einem weiteren Ziel nicht mehr abwägen kann, ob man ein sehr weit entferntes Ziel einholen sollte, oder das, welches gerade den Ursprung überquert hat, aber schneller ist, als das andere.

**Folie 4**

* Dafür werden nun allen Zielen Prioritäten zugewiesen. Dafür sind 3 Gewichte notwendig, die die 3 Prioritätsfaktoren beeinflussen, die wiederum die Priorität beeinflussen.
* Der Geschwindigkeitsfaktor ist besonders wichtig für die Berechnung. Je höher die Geschwindigkeit eines Ziels, desto höher die Priorität
* Der Positionsfaktor erhöht die Priorität bei Zielen, die sich vom Ursprung wegbewegen, andernfalls verringert sich die Priorität
* Distanzfaktor verringert die Priorität bei großen Abständen zur aktuellen Position

**Folie 5**

* Die Ziele werden im Algorithmus nun in eine Prioritätswarteschlange Q eingesetzt
* In jeder Iteration des Algorithmus wird die Priorität jedes Ziels in Q neu berechnet
* Dann wird das Ziel mit der höchsten Priorität genommen und als nächstes abgefangen
* In der Zeit bewegen sich alle anderen Ziele ebenfalls weiter, weshalb die Position aktualisiert werden muss
* Ziele, welche beim Abfangen des Ziels abgefangen wurden, können ebenfalls aus Q herausgenommen werden
* Dieser Algorithmus garantiert keine optimalen Ergebnisse, da Greedy versucht wird, das nächst-beste Ziel zu wählen
* Trotzdem ist dieser mit einer quadratischen Laufzeit sehr effizient

**Folie 6**

* Als optimale Alternative habe ich einen Brute-Force-Ansatz vorgestellt. Dabei werden alle möglichen Permutationen aus allen Zielen betrachtet, sodass am Ende die optimale Tour berechnet wurde. Die Knotenstellen dabei die jeweiligen Ziele dar. Problematisch ist hierbei aber offensichtlich die Komplexität mit O(n!), welche ab n=10 schon viele Ressourcen benötigt
* Deshalb habe ich zusätzliche Modifikationen entwickelt, sodass große Teile des Suchbaums weggeschnitten werden.
* Dafür wird geprüft, ob zwischen zwei Knoten ein Ziel eingeholt wurde. Falls dem so ist, tritt dieses Ziel im Laufe der Permutation ein weiteres Mal auf, wodurch die Tour nicht optimal sein kann.
* Zudem wird bei einer Berechnung eines Endknotens ein tau\_min gesetzt, welches eine Abbruchbedingung darstellt, sobald an einem Knoten diese Tourzeit bereits überschritten wurde.
* In beiden Fällen wird dann ab diesem Knoten der Baum abgeschnitten und muss nicht berechnet werden

**Ergebnisse**

* Beim Testen des Algorithmus von Helvig et al. bin ich zunächst von einem quadratischen Anstieg der Laufzeit bei steigender Anzahl an Zielen ausgegangen. Das in der Grafik lineare Wachstum kann man mit damit erklären, dass mit der steigenden Anzahl an Zielen die Wahrscheinlichkeit an Eliminierungen ansteigt und die Berechnungen in dem Haupt-Berechnungs-Schritt äquivalent zu kleineren Eingabegrößen sind
* Beim Brute-Force-Algorithmus ist aufgefallen, dass bei steigender Anzahl an Zielen der Anteil an betrachteten Knoten gegen 0 konvergiert. Trotz des kleinen Faktors besitzt der BF-Algorithmus weiterhin eine Laufzeit von O(n!), weshalb ab einer gewissen Eingabegröße der Algorithmus zu viele Ressourcen frisst und damit nicht mehr geeignet ist
* Der Prioritäts-Algorithmus liefert überraschend viele optimale Ergebnisse, besonders bei Geschwindigkeitsdifferenzen von vi\_max und v\_verfolger.

**Zusammenfassung und Ausblick**

* 1D-Algorithmus garantiert optimale Touren in polynomieller Zeit ist auch bei großen Instanzen effizient
* Der Prioritätsalgorithmus ist zwar effizient mit O(n^2), garantiert aber keine optimalen Touren. Zudem hängen die Touren stark von den gewählten Gewichten ab
* Der Brute-Force-Algorithmus ist für kleine Eingabegrößen besonders bei großen Beschneidungen im Suchbaum effizient. Vor allem, wenn schnell ein kleines tau\_min gefunden wird.
* Es ist also weiterhin für die neue Modifikation des MT-TSP eine Heuristik mit polynomieller Laufzeit von großem Interesse. Als Basis dafür könnten die vorgestellten Lemmata eine entscheidende Rolle spielen
* Damit könnte das nächste Ziel mit k-Achsen im MT-TSP gelöst werden. Diese Modifikation wäre eine wichtige Grundlage für 2D-Fälle
* Bisher sind allerdings Evolutionäre Algorithmen die besten Strategien, diese hängen allerdings von sehr vielen Parametern ab, wodurch keineswegs optimale Touren garantiert werden