**Lemma 1** (The No-Waiting Lemma). In any optimal Moving-Target TSP tour, the pursuer

must always move at **maximum speed**.

* Beweis mit Kontradiktion: Sich mit weniger als maximaler Geschwindigkeit ist äquivalent dazu, dass der Verfolger an bestimmten Positionen auf ein Ziel wartet. Anders herum könnte er auch direkt das Ziel einholen und dann an dem Ziel warten, d.h. wir verschieben die Wartezeit nach hinten. Das können wir mit weiteren Zielen fortführen und verschieben alle Wartezeiten ans Ende der Tour. Das ist unnötig, weshalb wir eine bessere Tour ohne diese Wartezeit bestimmen können. -> Kontradiktion

Deshalb betrachten wir nur Touren mit maximaler Verfolgergeschwindigkeit

* 1. MT-TSP in **one dimension**
* Alle Ziele und der Verfolger sind für den eindimensionalen Fall auf einer Linie fixiert. Wir werden nun einen exakt O(n^2)-Algorithmus, basierend auf dynamic programming, vorgestellt bekommen.
* Naiver Ansatz: Kosten der Tour, um alle Ziele auf der rechten Seite des Ursprungs einzuholen und anschließend auf der linken, berechnen. Dasselbe berechnen wir dann für den Start auf der linken Seite. Nehmen dann die Tour mit den geringeren Kosten. Erhalten damit möglicherweise sehr lange Touren, die nicht optimal sind, bzw. in einem unbounded error landen (es könnte ewig lange dauern, bis ein Ziel eingeholt werden kann)
  + Bsp: 4 Ziele, 2 links, 2 rechts. 2 sind dabei jeweils nah um Urspung und bewegen sich extremst schnell. Die anderen 2 sind sehr weit entfernt, aber bewegen sich kaum. Würden wir die beiden schnellen Ziele nacheinander einholen und anschließend die anderen verbleiben, würden wir relativ schnell die Tour durchführen. Mit dem naiven Ansatz würden wir uns für eine Seite entscheiden, wobei das schnelle Ziel sehr schnell weit weg ist. Um das Ziel einzuholen, würde der Verfolger eine sehr lange Zeit und Strecke benötigen.
* Dafür benötigen wir Lemma 2

**Lemma 2**. In an optimal tour for one-dimensionalMoving-Target TSP, the pursuer cannot

change direction until it intercepts the fastest target ahead of it (**Turning-points**-Lemma).

* Kernaussage des Beweises:
  + Wendepunkte, die nicht am schnellsten Ziel stattfinden, können nicht die Tourzeit verringern. Zeit, die nicht genutzt wird, um das schnellste Ziel einzuholen ist äquivalent zu Wartezeiten an einem Punkt. Mit dem vorherigen Beweis aus Lemma 1 ist die Tour damit nicht optimal.

**Definition state**:

* Eine Tour wird über eine Abfolge von Zuständen definiert. Ein Zustand ist eine Momentaufnahme einer Tour. Sie stellt einen potentiellen Wendepunkt dar, an dem der Verfolger als anschließend entweder das nächstschnellste Ziel auf derselben oder auf der anderen Seite des Ursprungs versucht zu erreichen. Um einen Zustand zu definieren benötigt man zwei Ziele. Erstens das Ziel, indem der Verfolger sich gerade befindet (s\_k) und zweitens das schnellste auf der anderen Seite des Ursprungs liegende Ziel (s\_f). Ein Zustand wird also als Tupel (s\_k, s\_f) beschrieben. Als Spezialfälle gibt es noch die Zustände A\_0 und A\_final. Dabei handelt es sich um den Start- und Endzustand der Tour. Weder A\_0 noch A\_final besitzen ein solches Tupel (s\_k, s\_f), hier wird nur die Tour begonnen, bzw beendet. Für jeden Zustand A\_i kann mit der Funktion t die kürzeste Zeit bis zum Erreichen des Zustandes berechnet werden. Es gilt t(A\_0) = 0
* Um nun alle Zustände zu bestimmen, ist eine Einteilung der Ziele in die Listen Left und Right notwendig. Jedes Ziel, welches links vom Ursprung aus liegt, wird in die Liste Left eingefügt. Analog dazu die Liste Right. Nun werden die Ziele nach der Geschwindigkeit wegführend vom Ursprung in absteigender Reihenfolge sortiert. Ziele, welche Näher am Ursprung liegen und zusätzlich langsamer sind als andere, werden in den jeweiligen Listen entfernt. Somit erhalten wir alle potentiellen Wendepunkte. Um nun die Liste aller Zustände zu bestimmen, werden für s\_k und s\_f alle Kombinationen aus den Listen Left und Right und umgekehrt eingefügt. Die Liste wird nun in aufsteigender Reihenfolge der Summe der Indizes der Ziele aus den Listen Left und Right sortiert. Somit stehen vorne die Kombinationen aus den schnellsten und hinten asus den langsamsten Zielen. [Insert graphic!]

**Definition transition**:

* In einem Zustand A gibt es zwei Möglichkeiten zur Wahl: Entweder wird das schnellste Ziel auf der linken oder auf rechten Seite des Verfolgers (auch vom Ursprung aus gesehen!) als nächstes verfolgt. Diese Punkte wiederum sind potentielle Wendepunkte, d.h. es wird ein Übergang in den nächsten Zustand B erzeugt. Die Notation ist hierbei tau : A -> B. Mit der Wahl der linken und rechten Seite werden zwei Übergänge erzeugt, *tau\_left* und *tau\_right*. Somit hat jeder Zustand bis zu zwei Übergänge. Wenn mit einem Zustand alle auf der linken Seite befindlichen Wendepunkte abgegrast sind, gibt es somit nur einen Übergang *tau\_right*. A\_final hat wiederum keine Übergänge.
* Der Zeitfunktion kann mit t(*tau\_left*) oder t(tau\_right) auch Übergänge übergeben. Damit wird die Zeit berechnet, um von dem betrachteten s\_k im Zustand A an Zeitpunkt t(A) das schnellste Ziel auf der linken oder rechten Seite abzufangen.

**Definition graph G=(V,E)**

* Wir können nun unser Problem mit den Zuständen und Übergängen in ein Graphproblem überführen. Dabei stellen die Zustände die Knoten V und die Übergänge die Kanten zwischen den Knoten dar. Der Graph G kann also mit G=(V,E) generiert werden.
* Nun muss der genaue Aufbau spezifiziert werden. Zunächst wird der Startzustand A\_0 eingefügt. Wie bereits erwähnt, wird die Zustandsliste in aufsteigender Reihenfolge der Summe der Indizes aus Left und Right generiert. Nun werden die Zustände mit dem niedrigsten Summenwert neben A\_0 plaziert. Als nächstes wird dasselbe mit dem nächsthöheren Summenwert gemacht, wobei dessen Zustände neben den Zuständen des darunterliegenden Summenwertes platziert werden. Das wird bis zum Ende der Liste fortgeführt, wobei am Ende der Endzustand A\_final angehängt wird. Übergänge können ausschließlich nur in Zustände des höherliegenden Summenwertes führen. Damit ist der Graph azyklisch, da es keine Kanten zu vorherigen oder den gleichen Knoten gibt. Sofern auf einer Seite alle Ziele abgefangen wurde, kann direkt eine Kante in A\_final gezogen werden. Das Gewicht der Kante wird dann von der Zeit bestimmt, alle restlichen Ziele auf der entsprechend anderen Seite abzufangen. Dadurch gibt es einige Knoten, zu denen keine Kante gezogen wird.
* Das neue Graphproblem kann somit als shortest path problem gelöst werden. Dabei wird der kürzeste Weg von A\_0 bis A\_final gesucht. Da der Graph azyklisch ist, kann ein einfaches Verfahren gewählt werden. Hierfür haben sich die Autoren aus [x] für dynamische Programmierung entschieden. Dabei wird durch jeden einzelnen Zustand in topologischer Reihenfolge iteriert und mit *tau\_left* und *tau\_right* überprüft, ob der nachfolgende Zustand schneller erreicht werden kann, als ggf. zuvor mit einer anderen Reihenfolge an Zuständen. Durch die Generierung der States wird die Laufzeit allerdings quadriert, wodurch der Algorithmus eine Laufzeit von exakt O(n^2) benötigt.