UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Algorithms for Moving-Target TSP

Algorithmen für bewegende Ziele in TSP

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 28. November 2019

IM FOCUS DAS LEBEN

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

Acknowledgements

TODO

Contents

1 Introduction 1

1.1 Contributions of this Thesis 1

1.2 Related Work 1

1.3 Structure of this Thesis 1

2 The moving-target traveling salesman problem in general 2

2.1 Definitions 2

2.2 Instances of Moving-Target TSP 2

3 Moving-Target TSP in one dimension 4

3.1 Algorithm by Helvig, Robins and Zelikovsky 5

4 Conclusion 7

1

Introduction

1.1 Contributions of this Thesis

TODO

1.2 Related Work

TODO

1.3 Structure of this Thesis

TODO

2

The moving-target traveling salesman problem in general

2.1 Definitions

In MT-TSP we consider an amount of targets $T = \{t_1, ..., t_n\}$ and a set of velocities $V = \{v_1, ..., v_n\}$ so that each moving with a constant movement speed $\vec{v_i}$. A pursuer starts at the origin O (a defined position), moving with a velocity of v_p. His aim is to visit all targets once and finishes with returning to the origin.

Therefore, we can model this problem as the following graph
$G = (T, V, O, v_p)$.

2.2 Instances of Moving-Target TSP

It was proven that MT-TSP is NP-hard. Some instances can result in an unbounded error, whenever the pursuer choses a non-optimal tour. Therefore, the condition $v_p >$ \overrightarrow{v__i} must apply, to avoid these errors. This was proven by the authors in [**helvig**], since the goal is the most fast optimal tour. Thus, instead of directly calculating the tour of the pursuer, it is necessary to determine the solvability of the input. Whenever it is not possible we gain a 'No'-instance, otherwise we go ahead and calculate the tour.
This paper presents two concrete cases:

1) 1D-case: Each target's movement is limited to a single line
2) 2D-case: The movement direction of a target consists of a two-dimensional vector

Each case is investigated for the shortest and the fastest tour. Helvig , Robins and Zelikovsky presented in [**helvig**] the first heuristics in the field of MT-TSP. However, determining the approximation for general cases is hard, because there are many influences that determine the complexity of the problem. The approximation research in [**hammar**] showed that MT-TSP cannot be approximated better than by a factor of $2^{\pi(\sqrt{n})}$ by a polynomial time algorithm unless P=NP.

Later on, we will compare new heuristics and algorithms with those that recently proposed in [**moraes**]. This paper examines Genetic Algorithm, Simulated Annealing and

Ant Colony Optimization to solve MT-TSP.

One-dimensional-case

This specific case is already introduced extensive in [**helvig**]. Each movement is fixed on a single line, thus, there are two possible directions for a target and the pursuer.

Two-dimensional-case

In this case another dimension is added. We will consider different scenarios with fixed numbers of targets and corridors. Compared to the 1D-case, in which a target is fixed on a line, targets freely move within the space. Therefore, we define corridors, representing the number of vectors in which the targets move, resulting in different complexities.

This paper focusses on the 2D-case. Previous works did not construct concrete heuristics for this type of case, therefore, this paper shows new aspects in the field of MT-TSP.

Input & Output

To set up heuristics a suitable input and output are necessary. Thus, the MT-TSP can be modelled as graph problem (refering section 2), further heuristics just expect a graph G as parameter.

The Output, on the other hand, can look very different. The most obvious parameter are the length and the time needed to finish the tour. Whenever, there is no possible tour, an MT-TSP algorithm needs to return a 'No'-instance.

Furthermore, a tour should be comprehensible. Therefore, the targets are displayed in the order in that the pursuer executes the tour. For each target the initial coordinates and the time and the position whenever the pursuer intercepts the target are monitored. To fully understand and analyse the tour a visualization is a good application, but not necessary.

3

Moving-Target TSP in one dimension

This specific instance was introduced in [**helvig**]. Each target and the pursuer are restricted to a single line, so there are only two possible directions of movement. The first naive approach is to calculate the cost of the tour to intercept all the targets on the right side of the origin and then on the left and vice versa. Choose the tour with lower costs. This will probably result in an unbounded error with simple counter examples (TODO: create a graphic with a counter example), where the pursuer probably takes and eternity to intercept a target.

Therefore, the pursuer is only allowed to change his direction, whenever he intercepts the fastest target either on the left or right side of the origin. These targets are named as turning points. Turning points that are not the fastest target can not reduce the tour time. This time not used to catch up with the fastest target is equivalent to waiting at one point. As already mentioned (TODO), waiting at one point results in not optimal tours.

With this knowledge about turning points, the term *state* is introduced. A state is a snapshot of a tour. It represents a potential turning point at which the pursuer then attempts to reach either the next fastest target on the same or the other side of the origin. It required two targets to define a state. First, the target, where the pursuer is currently located (s_k), and second, the fastest target (s_f) on the other side of the origin. Thus, a state can be described as the a tuple (s_k, s_f). As special cases, there are the states A_0 and A_{final}. Neither A_0 or A_{final} have such a tuple (s_k, s_f), these states just define the start and end of each tour. For each state A_i, the shortest time to reach the state can be calculated with the time function t. It applies $t(A_0) = 0$.

In order to determine all states, it is necessary to divide the targets into the lists *Left* and *Right*. Each target, which is located on the left of the origin, is inserted *Left*. Analogous for the list *Right*. Now the targets are sorted by the speed leading away from the origin in descending order. Targets that are closer to the origin and additionally slower than others are removed in the respective lists. Thereby, just the potential turning points are remaining. In order to determine the list of all states, named *States*, all combinations of the lists *Left* and *Right* and vice versa are inserted for s_k and s_f. The list is now sorted in ascending order of the sum of the indices of the targets from the lists *Left* and *Right*. Therefore, the combinations of the fastest targets are in front of *States*.

In a state A, there are two options: Either the fastest target on the left or on the right side of the pursuer (also from the perspective of the origin!) is intercepted next. These targets in turn are potential turning points, i.e. a *transition* into the next state B is generated. The notion of a transition between the states A and B is $A \rightarrow B$. A transition applies to the respective time $t[A]$. Therefore, the position of the target s_k of each A and B must be updated with $s_k^{'} = s_k + v_{s_k} \cdot t[A]$. With simple physics equations the time to reach B can be calculated with $t = \dfrac{s_{k_B}^{'} - s_{k_A}^{'}}{v_{pursuer} - v_B}$. Thus, the pursuer always travels with maximum speed, this equation uses $v_{pursuer}$ instead of v_A. This time represents the weight of the transition. Summarized the transition $A \rightarrow B$ depends on the timestamp of A and the transition weight varies over time.

With the options to move to the left or right turning point, there are two transitions outgoing the current state: τ_{left} and τ_{right}. Thus, each state has up to two transitions into other states. As soon as each turning point is intercepted on the left side of the origin, there is only the transition τ_{right} into the final state, which is then named as τ_{final}. The weight of the edge is then determined by time needed to intercept all remaining targets on the other side. The final state has no transitions into other states. With $t[\tau_{left}]$ or $t[\tau_{right}]$ the time is calculated to intercept the fastest target on the left or right side from the considered s_k in state A at time $t(A)$.

Now the problem with states and transitions between them can be transformed into a graph problem. The states represent the vertices V and the transitions are the edges E between the vertices. Thus, the Graph G can be modelled as $G = (V, E)$. Next the exact structure of G needs to be specified. First, the start state A_0 is inserted. As previously mentioned, *States* is generated in ascending order of the sum of the indices in the lists *Left* and *Right*. Now the states with the lowest sum value are placed next to A_0. Next, the same thing is done with the next higher sum value with its states placed next to the states of the underlying sum value. Proceed with this process for all other states until the end of the list and append A_{final} at the end. Transitions can only lead to states of higher sum values. Therefore, the graph is acyclic, as there are no edges into the current or previous vertices. Caused by some transitions immediately into A_{final}, there are probably vertices to which no edge is drawn.

The new graph problem can thus be solved as a shortest path problem by finding the shortest way from A_0 to A_{final}. Since the graph is acyclic, a simple procedure can be chosen.

3.1 Algorithm by Helvig, Robins and Zelikovsky

With this requirements the authors of [**helvig**] showed an exact $O(n^2)$-time algorithm for 1D-cases, which is based on dynamic programming. The goal is to receive the fastest, optimal tour, intercepting each target.

First the algorithm generates the lists *Left*, *Right* and *States* as described above. Next, iterate through the state list in topological order. To make it easier to understand, modelling as a graph makes sense. The algorithm solves this rather *on-the-fly* than generating each transition between the states. This method is possible through the topological order which is gained by the special sorting and eliminations of *Left* and *Right* and further the sorting of *States*. Furthermore, by far not not all transitions are considered. Some states get skipped or directly lead into A_{final}, caused by performing one of the following steps, while iterating through each state A_i:

– If A_i has no transitions into it, proceed with the next state. This case can be recognized whenever the time function array at index i is still initialized with $t[i] = \infty$.
– If the pursuer has intercepted each target on one side, create a transition from A_i into the final state A_{final}. Note, to calculate the transition weight, determine the maximum time from A_i to each extant target on the other side of the origin.
– Otherwise calculate the transitions τ_{left} and τ_{right}, which correspond to sending the pursuer after either the fastest target on the left or the fastest target on the right. If $t[A_i]$ added with the time needed to perform the transition is faster than possibly before with other states, update $t[A_i]$ with this result.

Therefore, the time used to reach a state B_i with the shortest sequence of states is defined by $t(A_i) = min\{t(A) + t(\tau) | \tau : A \rightarrow B\}$. Then traverse backwards from A_{final} to A_0, which denotes the reversed list of turning points. Last calculate which targets are intercepted between the state pair, so we can describe an optimal tour of targets. Iterating the states squares the runtime, which requires the algorithm to run exactly in $O(n^2)$.

4

Conclusion

TODO