

Das Traveling Salesman Problem

Material

[Git Repo](#) | [Jupyter Notebook](#)

Ziele der heutigen Sitzung:

- Verständnis des Traveling Salesman Problems als klassisches NP-schweres Optimierungsproblem
- Kennenlernen verschiedener Algorithmen zur exakten und approximativen Lösung
- Implementierung und Analyse der Algorithmen in Python
- Vergleich der Algorithmen hinsichtlich Laufzeit und Approximationsgüte

Das **Traveling Salesman Problem (TSP)** ist ein klassisches Problem der Informatik und Optimierung. Es modelliert folgende Situation:

Ein Reisender soll mehrere Punkte besuchen. Er möchte dabei:

- Jeden Punkt genau einmal besuchen
- Am Ende zu seinem Ausgangspunkt zurückkehren
- Die zurückgelegte Gesamtstrecke minimieren

Anwendungsbeispiele

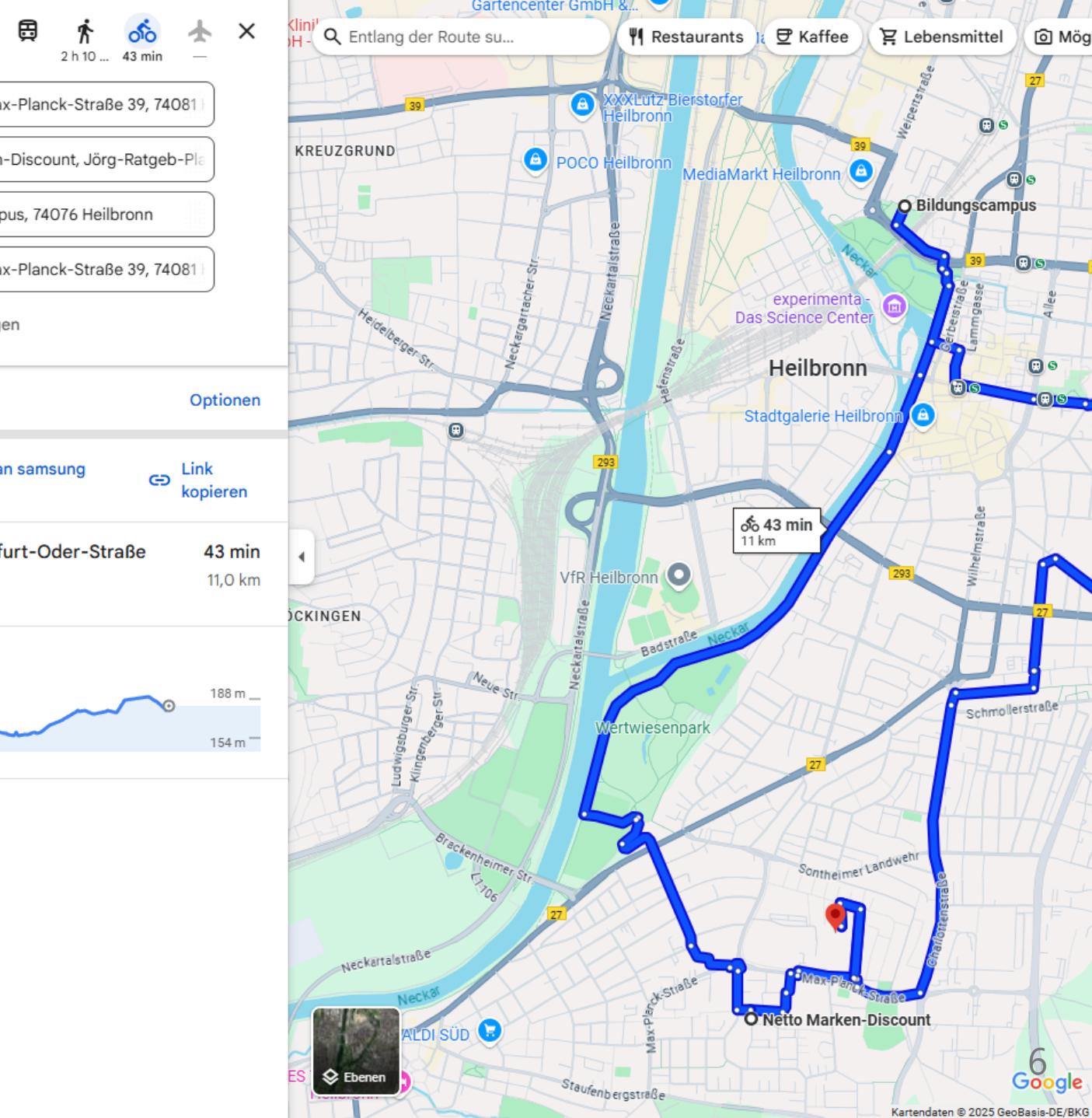
- Tourenplanung für Lieferfahrzeuge
- Optimierung von Wartungsrouten
- Leiterplattenherstellung (optimale Anordnung von Bohrungen)
- **Bestrahlungstherapie** (Bewegung des Aktuators zum Ziel)

In dieser Vorlesung untersuchen wir drei **algorithmische Ansätze**:

1. **Brute-Force-Algorithmus:** Probiert alle möglichen Routen aus (optimal, aber sehr langsam)
2. **Nearest-Neighbor-Algorithmus:** Eine einfache Greedy-Strategie, die immer zum nächstgelegenen Punkt geht
3. **Christofides-Algorithmus:** Ein fortgeschrittenes Approximationsverfahren mit garantierter Qualität

Analyse der Algorithmen

- Die theoretischen Grundlagen
- Die praktische Implementierung
- Die Laufzeitkomplexität
- Die Qualität der Lösung



Mathematische Formulierung

Das Traveling Salesman Problem lässt sich mathematisch wie folgt formalisieren:

Sei $G = (V, E)$ ein vollständiger Graph mit:

- $V = \{1, 2, \dots, n\}$ ist die Menge der Knoten (Punkte)
- $E = \{(i, j) | i, j \in V, i \neq j\}$ ist die Menge der Kanten
- $c : E \rightarrow \mathbb{R}^+$ ist eine Kostenfunktion, die jeder Kante $(i, j) \in E$ ein Gewicht c_{ij} zuordnet

Ziel und Problemformulierung

Ziel: Finde einen Hamiltonkreis mit minimalen Gesamtkosten.

Mathematische Formulierung: Gesucht ist eine Permutation π von $\{1, 2, \dots, n\}$, die die folgende Kostenfunktion minimiert:

$$\sum_{i=1}^{n-1} c_{\pi(i), \pi(i+1)} + c_{\pi(n), \pi(1)}$$

Komplexitätsanalyse

Komplexitätsklasse: Das TSP ist NP-schwer. Dies bedeutet, dass kein effizienter (polynomieller) Algorithmus bekannt ist, der optimale Lösungen für beliebig große Instanzen garantiert.

Konsequenz für die Praxis:

- Bei n Punkten gibt es $(n - 1)!/2$ verschiedene mögliche Routen
- Bei 10 Punkten sind dies bereits über 180.000 Routen
- Bei 15 Punkten über 43 Milliarden

Brute Force Algorithmus

- Probiert **systematisch** alle möglichen Routen aus
- Garantiert die optimale Lösung
- Laufzeit: $O(n!)$ → exponentiell
- Praktisch nur für kleine Instanzen (<15 Punkte) anwendbar
- **Speicherkomplexität:** $O(n)$ nur eine Permutation im Speicher

Grundprinzip

1. Generiere systematisch alle möglichen Rundwege (Permutationen der Punkte)
2. Berechne für jeden Rundweg die Gesamtlänge
3. Wähle den kürzesten Rundweg

Mathematische Analyse

Anzahl möglicher Rundwege: Bei n Punkten gibt es $(n)!$ verschiedene

Korrektheit: Der Algorithmus findet garantiert das globale Optimum, da er alle möglichen Lösungen prüft.

Zeitkomplexität: $O(n!)$

- Bei $n = 10$ Punkten: ca. 3 628 800 Permutationen
- Bei $n = 15$ Punkten: ca. 1 307 674 368 000 Permutationen
- Bei $n = 20$ Punkten: ca. 10^{18} Permutationen | 2 432 902 008 176 640 000

Brute-Force-Algorithmus mit festem Startpunkt

In der Python Implementierung des Brute-Force-Algorithmuses werden alle möglichen Permutationen sämtlicher Punkte betrachtet. Bei n Punkten gibt es $n!$ solcher Permutationen.

In vielen praktischen Anwendungen ist jedoch der Startpunkt vorgegeben. In diesem Fall reduziert sich die Anzahl der zu betrachtenden Permutationen auf $(n - 1)!$, da nur noch die Reihenfolge der anderen Punkte variiert wird.

Vorteile eines festen Startpunkts:

1. **Reduzierte Berechnungskomplexität:** Statt $n!$ müssen nur $(n - 1)!$ Permutationen überprüft werden
2. **Realistische Modellierung:** In vielen praktischen Anwendungsfällen ist der Startpunkt tatsächlich fest

Mathematische Auswirkung:

Bei 10 Punkten mit variablem Startpunkt: $10! = 3.628.800$ Permutationen

Bei 10 Punkten mit festem Startpunkt: $9! = 362.880$ Permutationen

Die Berechnung wird also um den Faktor 10 schneller, was bei größeren Problemen einen **erheblichen Unterschied** macht.

Wahl des optimalen Startpunkts:

- Wenn der Startpunkt variabel ist und wir den besten Startpunkt wählen könnten, müssten wir dennoch alle $n!$ Permutationen betrachten

Die exakte Definition des Problems kann erhebliche Auswirkungen auf die Berechnungskomplexität und die gefundenen Lösungen haben.

Sucht die kürzeste
Strecke durch
Myrtana





Start bei 1, 2 existiert nicht
[1,3,4,8,6,5,13,7,9,10,11,12]

Vorgehen:

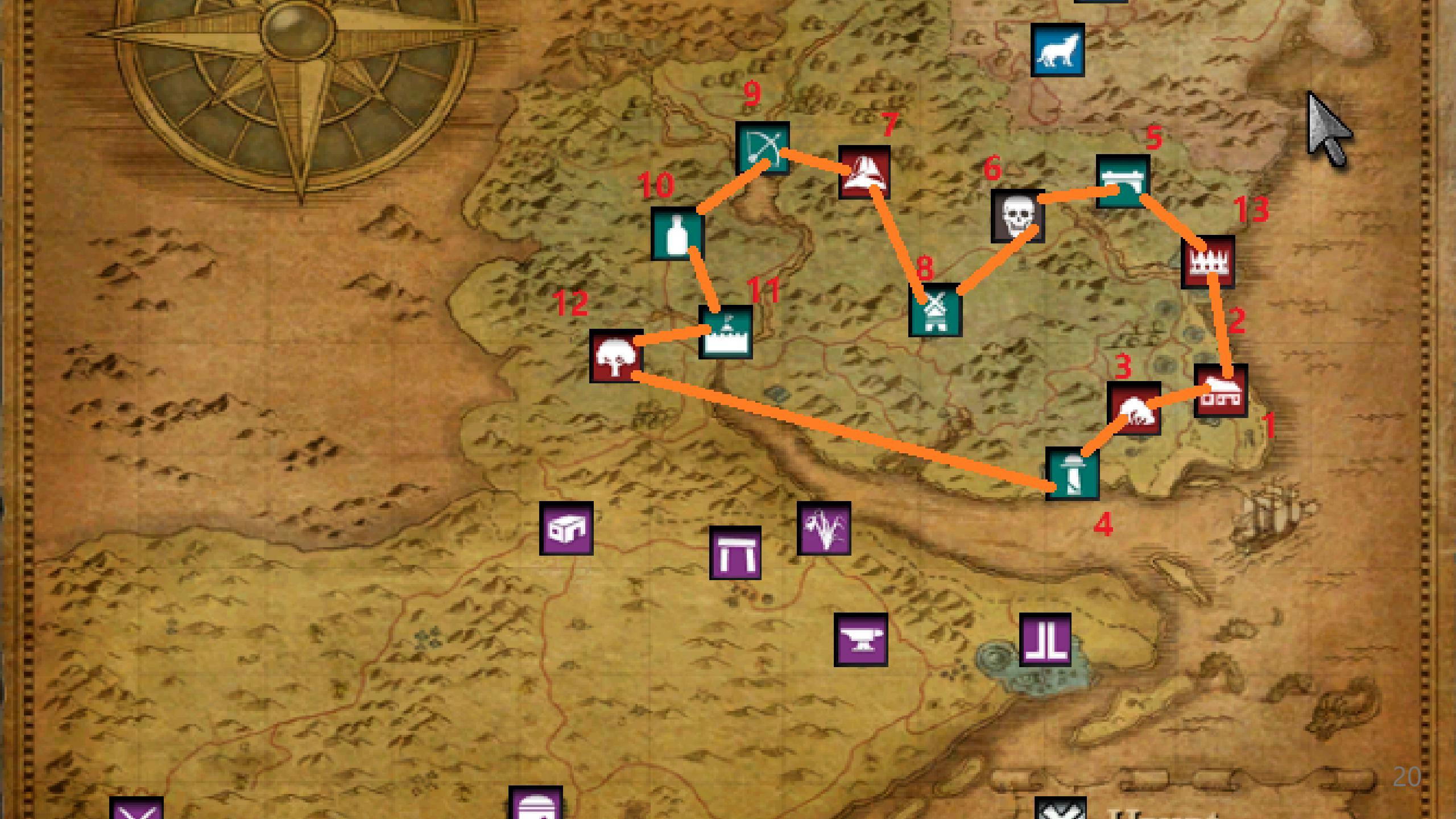
$$p_{next} = \arg \min_{p \in \text{unvisited}}$$

$$\sqrt{(p_x - \text{current}_x)^2 + (p_y - \text{current}_y)^2}$$





Wie sieht die Route aus wenn P2 existiert?



Nearest-Neighbor-Algorithmus

Der Nearest-Neighbor-Algorithmus (Nächster-Nachbar-Algorithmus) ist eine einfache Strategie, die auf gesundem Menschenverstand basiert:

1. Wähle einen Startknoten
 2. Finde den nächstgelegenen Punkt, der noch nicht besucht wurde
 3. Gehe zu diesem Punkt
 4. Wiederhole Schritte 2-3, **bis alle Punkte besucht wurden**
 5. Kehre zurück zum Ausgangspunkt, um die Tour abzuschließen
- Laufzeit: $O(n^2)$
 - **Greedy-Strategie:** meist gute, aber selten optimale Lösungen

Vorteile:

- Sehr einfach zu verstehen und zu implementieren
- Deutlich schneller als Brute-Force (funktioniert auch für viele Punkte)
- Liefert recht gute Lösungen (**kein garantiertes minimum**)

Nachteile:

- Kann in bestimmten Fällen sehr schlechte Lösungen liefern
- Die Qualität der Lösung hängt stark vom gewählten Startpunkt ab

Mathematische Grundlage

Der Nearest-Neighbor-Algorithmus ist eine **Greedy-Heuristik** mit folgender Vorgehensweise:

1. Starte bei einem beliebigen Knoten v_0 als aktuellem Knoten.
2. Finde den nächsten unbesuchten Knoten v_i zum aktuellen Knoten, d.h. minimiere $d(v_{current}, v_i)$ über alle unbesuchten Knoten v_i .
3. Füge v_i zur Tour hinzu und setze v_i als aktuellen Knoten.
4. Wiederhole Schritte 2 und 3, bis alle Knoten besucht sind.
5. Kehre zum Startknoten v_0 zurück, um den Kreis zu schließen.

Mathematisch ausgedrückt: In jedem Schritt i wählen wir den Knoten v_i , sodass:

$$v_i = \arg \min_{v \in V_{unvisited}} d(v_{current}, v)$$

Die Zeitkomplexität beträgt $O(n^2)$, was deutlich effizienter als die Brute-Force-Methode ist $O(n!)$.

Einfach ausgedrückt: Bei 100 Punkten müssen wir nur etwa **10.000 Vergleiche** durchführen, statt 10^{158} wie bei Brute-Force. Das ist ein enormer Unterschied!

Allerdings garantiert dieser Algorithmus **keine optimale Lösung** und kann im schlimmsten Fall eine Tour liefern, die $\Theta(\log n)$ mal länger als die optimale Tour ist.

Der Christofides-Algorithmus

- Entwickelt 1976 vom griechischen Mathematiker(Zypern,1942-2019) **Nicos Christofides**
- Revolutionärer Ansatz durch Kombination von:
 - Minimalen Spannbäumen
 - Perfekten Matchings

- **Approximationsgarantie:** Höchstens 1,5-mal länger als die optimale Lösung
- Übertrifft ältere Heuristiken wie den Double-Tree-Algorithmus (2-Approximation)
- Setzt bis heute den Maßstab für Approximationsgüte im metrischen TSP

Christofides Algorithmus

1. Finde einen minimalen Spannbaum des Graphen
 2. Bestimme die Knoten mit ungeradem Grad
 3. Berechne ein minimales Matching auf diesen Knoten
 4. Kombiniere Spannbaum und Matching zu einem Eulerschen Graphen
 5. Konstruiere einen Eulerkreis und leite daraus einen Hamiltonkreis ab
- Laufzeit: $O(n^3)$

Bestimme den Minimalen Spannbaum

Ein minimaler Spannbaum (MST) T eines Graphen $G = (V, E)$ ist ein Baum, der alle Knoten in V verbindet und die geringste Gesamtkantenlänge aufweist. Formal minimiert T die Summe:

-
-

$$\sum_{(u,v) \in T} c(u, v)$$



Identifizierung der Knoten mit ungeradem Grad

Der Grad eines Knotens v in einem Graphen G ist die Anzahl der inzidenten Kanten an v , bezeichnet als $\deg(v)$.

In einem Baum mit n Knoten gibt es genau $n - 1$ Kanten. Nach dem Handschlaglemma der Graphentheorie ist die Summe aller Knotengrade gleich dem Doppelten der Kantenanzahl:

$$\sum_{v \in V} \deg(v) = 2|E| = 2(n - 1)$$

Daraus folgt, dass die Anzahl der Knoten mit ungeradem Grad in jedem Graphen gerade sein muss. In einem MST gibt es oft mehrere Knoten mit ungeradem Grad, und wir müssen diese identifizieren, um im nächsten Schritt ein perfektes Matching für sie zu finden.

Für jeden Knoten v im MST berechnen wir:

$$\deg(v) = |\{u \in V : (u, v) \in T\}|$$

und sammeln diejenigen Knoten v , für die $\deg(v) \bmod 2 = 1$ gilt.



Minimales perfektes Matching

- Ein **perfektes Matching** ist eine Menge von Kanten, bei der jeder Knoten genau einmal vorkommt
- Für die Menge der Knoten mit ungeradem Grad suchen wir ein **minimales perfektes Matching**
- Dies minimiert die Summe der Kantengewichte:

$$M = \arg \min_{M'} \sum_{(u,v) \in M'} c(u, v)$$



Kombination zu einem Multigraphen

Der nächste Schritt vereint den minimalen Spannbaum (MST) und das minimale perfekte Matching:

- Ein **Multigraph** $G_M = (V, E_M)$ entsteht durch:

$$E_M = E_{MST} \cup E_{Matching}$$

- Dies erlaubt mehrere Kanten zwischen denselben Knotenpaaren
- **Entscheidende Eigenschaft:** Alle Knoten haben nun **geraden Grad**

Mathematische Grundlage

- Knoten mit ungeradem Grad im MST erhalten exakt eine zusätzliche Kante durch das Matching
- Damit gilt für jeden Knoten $v \in V$:

$$\deg_{G_M}(v) \equiv 0 \pmod{2}$$

Nach dem **Satz von Euler** enthält ein zusammenhängender Graph genau dann einen Eulerkreis, wenn alle Knoten geraden Grad haben. Diese Bedingung ist nun erfüllt.

Der Eulerkreis und seine Berechnung

Definition und Bedeutung

Ein **Eulerkreis** ist ein geschlossener Rundweg, der jede **Kante** des Graphen genau einmal durchläuft. Die Existenz eines Eulerkreises ist durch die Konstruktion unseres Multigraphen garantiert, da alle Knoten geraden Grad haben.

Der Hierholzer-Algorithmus

Der **Hierholzer-Algorithmus** berechnet einen Eulerkreis effizient:

1. Beginne bei einem beliebigen Knoten und folge einem Pfad, wobei jede benutzte Kante gelöscht wird
2. Fahre fort, bis ein geschlossener Rundweg entsteht (Kreislauf)
3. Solange der Kreislauf noch nicht alle Kanten enthält:
 - Wähle einen Knoten des Kreislaufs, der noch ausgehende Kanten hat
 - Erzeuge von diesem Knoten aus einen neuen Kreislauf
 - Verschmelze diesen neuen Kreislauf mit dem bestehenden

Theoretische Analyse

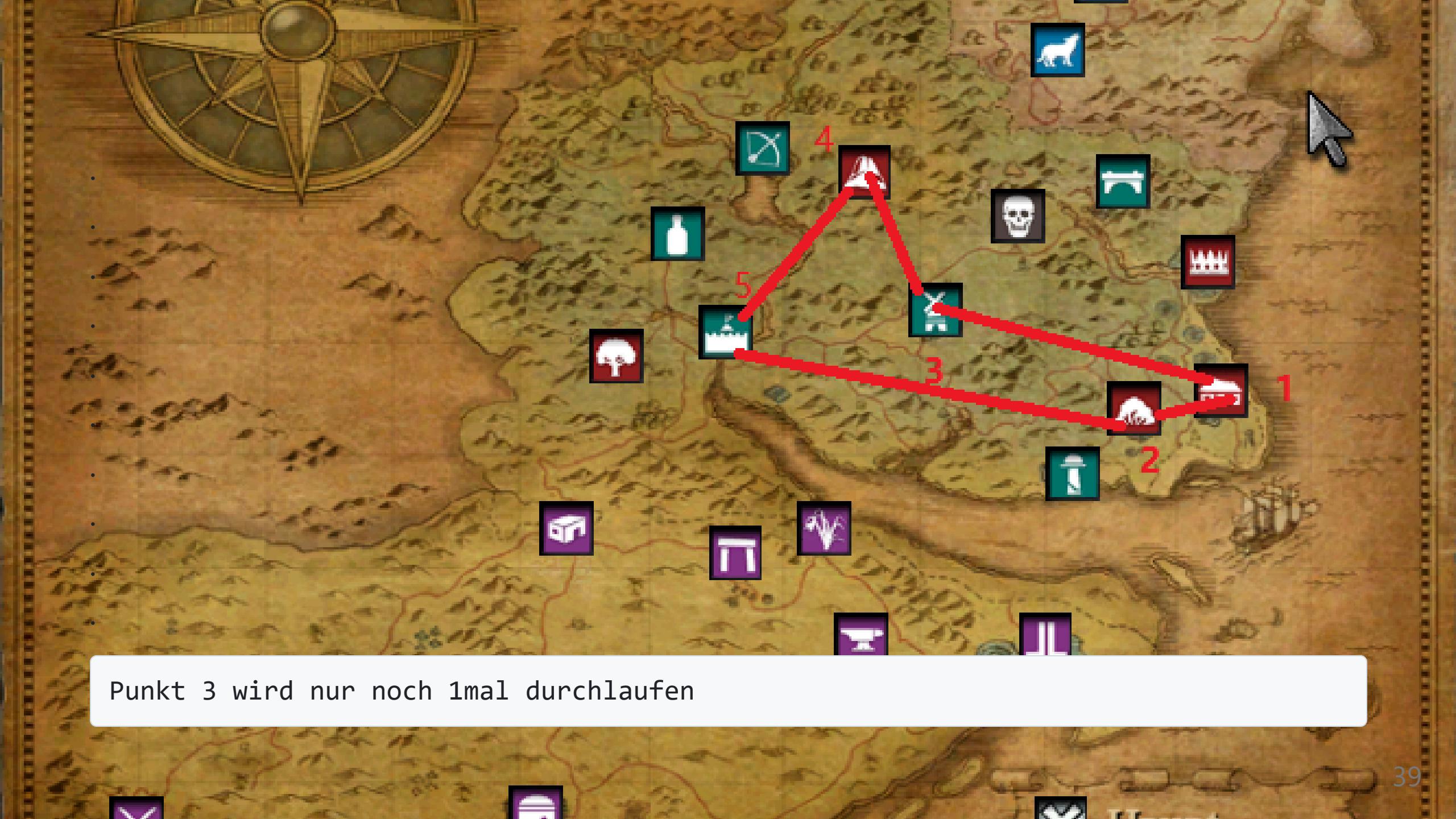
- **Korrektheit:** Der Algorithmus findet garantiert einen Eulerkreis, wenn alle Knoten geraden Grad haben
- **Zeitkomplexität:** $O(|E|)$, wobei $|E|$ die Anzahl der Kanten ist
- **Speicherkomplexität:** $O(|E|)$ zur Speicherung des Graphen und des Eulerkreises

Umwandlung zum Hamiltonkreis

Um vom Eulerkreis zum Hamiltonkreis (TSP-Lösung) zu gelangen:

- Durchlaufe den Eulerkreis und füge jeden Knoten zur TSP-Tour hinzu
- Überspringe Knoten, die bereits besucht wurden
- Diese "Kurzschlüsse" sind dank Dreiecksungleichung nie länger als der ursprüngliche Pfad





Punkt 3 wird nur noch 1mal durchlaufen

Vergleich der Algorithmen

Algorithmus	Laufzeit	Qualität	Anwendbarkeit
Brute Force	$O(n!)$	Optimal	Sehr kleine Instanzen
Nearest-Neighbor	$O(n^2)$	Keine Garantie	Mittelgroße Instanzen
Christofides	$O(n^3)$	Max. $1,5 \times$ optimal	Größere Instanzen

Zusammenfassung

- Das TSP ist ein fundamentales Problem der kombinatorischen Optimierung
- Es ist NP-schwer, was bedeutet, dass kein effizienter exakter Algorithmus bekannt ist
- Für praktische Anwendungen werden meist Approximationsalgorithmen eingesetzt
- Die Wahl des Algorithmus hängt von der Problemgröße und den Anforderungen ab:
 - Kleine Instanzen → Brute Force
 - Einfache Implementation → Nearest-Neighbor
 - Approximationsgarantie → Christofides

Vielen Dank!

Weiterführende Ressourcen

- Weitere Informationen und Implementierungsbeispiele: deadlinedriven.dev
- Machine/Deep Learning Themen : <https://deadlinedriven.dev/TechnicalContent/>
- Auch coole Seite zum Thema : [Inspiration](#)