Algorithmus zur Bildung transitiver Graphen

Dieser Algorithmus dient zur Gruppierung von Objekten basierend auf ihrer räumlichen Nähe. Er verwendet eine Kombination aus KD-Tree-Suche und einer transitiven Gruppierungslogik.

1 Schritte des Algorithmus

1.1 1. Paarweises Finden von Nachbarn (KD-Tree)

Ein **KD-Tree** (k-dimensionaler Baum) wird verwendet, um effizient Paare von Punkten zu finden, deren Abstand kleiner oder gleich einem gegebenen Schwellenwert t (Threshold) ist. Dies reduziert die Berechnungszeit im Vergleich zur vollständigen paarweisen Abstandsberechnung.

- **Input**: Punkte $P = \{p_1, p_2, \ldots, p_n\}$ in einem 2D-Raum mit Koordinaten (ALPHAWIN_SKY, DELTAWIN
- **Operation**:
  1. Baue einen KD-Tree auf den Punkten P.
  2. Verwende die Funktion `query_pairs(t)`, um alle Punktpaare (i, j) zu finden, die die Bedingung $d(p_i, p_j) \leq t$ erfüllen.
- **Komplexität**:
  - Aufbau des KD-Trees: $\mathcal{O}(n \log n)$.
  - Abfrage von Paaren: Effizienter als $\mathcal{O}(n^2)$.

1.2 2. Bildung von Verknüpfungen (Graphenmodell)

Die durch den KD-Tree ermittelten Nachbarpaare (i, j) werden als Kanten eines ungerichteten Graphen interpretiert:

- Jeder Punkt p_i ist ein **Knoten**.
- Eine Kante zwischen zwei Knoten (i, j) existiert, wenn $d(p_i, p_j) \leq t$.

Beispiel:

- Punkte: A, B, C, D.
- Paare: $(A, B), (B, C)$.
- Graph:
$$A \longleftrightarrow B \longleftrightarrow C, \quad D$$
- Interpretation: A, B, C bilden eine Gruppe; D ist eine separate Gruppe.

1.3 3. Transitive Gruppierung (Connected Components)

Das Ziel ist es, alle zusammenhängenden Komponenten des Graphen zu finden. Hierfür wird eine iterative Gruppierungslogik angewandt:

- **Fall 1**: Weder Punkt i noch Punkt j gehören einer existierenden Gruppe an. Eine neue Gruppe wird erstellt.
- **Fall 2**: Einer der Punkte gehört bereits zu einer Gruppe. Der andere Punkt wird dieser Gruppe hinzugefügt.

- **Fall 3**: Beide Punkte gehören unterschiedlichen Gruppen an. Die Gruppen werden zusammengeführt.

Beispiel für die Gruppierung:

- Paare: $(A, B), (B, C), (D, E)$.
- Verarbeitung:
  - Erstelle Gruppe 1: $\{A, B\}$.
  - Füge C zu Gruppe 1 hinzu: $\{A, B, C\}$.
  - Erstelle Gruppe 2: $\{D, E\}$.
- Ergebnis:

$$\text{Gruppe 1: } \{A, B, C\}, \quad \text{Gruppe 2: } \{D, E\}.$$

1.4 4. Algorithmusdetails

Initialisierung

```
tree = cKDTree(positions)  # Erstelle den KD-Tree
pairs = tree.query_pairs(threshold)  # Finde Nachbarpaare
```

Gruppenbildung

```
identical_objects = []  # Ergebnisliste
unique_id = 1  # Gruppennummer
visited = set()  # Verarbeitete Punkte

for i, j in pairs:
if i in visited or j in visited:
continue  # Überspringe bereits verarbeitete Punkte

identical_objects.append((i, unique_id))  # Füge Punkt i zur Gruppe hinzu
identical_objects.append((j, unique_id))  # Füge Punkt j zur Gruppe hinzu

visited.add(i)
visited.add(j)
unique_id += 1
```

1.5 5. Optimierungspotenziale

- **Union-Find-Algorithmus**: Effizient zur Erkennung von zusammenhängenden Komponenten. Verwendet eine disjunkte Mengenstruktur zur Gruppierung.
- **Präprozessierung des Suchraums**: Rasterbasierte Gruppierung, um den Suchraum vor der Paarbildung einzuschränken.
- **Parallelisierung**: Die paarweise Suche und Gruppenzuordnung können parallel ausgeführt werden.

2 Zusammenfassung

Der Algorithmus kombiniert:

- **KD-Tree**: Effiziente Nachbarschaftssuche.
- **Iterative Gruppierung**: Bildung von transitiven Gruppen.

Die Effizienz des Algorithmus kann durch den Einsatz von Union-Find oder Rasterbasierten Ansätzen weiter verbessert werden.