3.5.1 Abbildung einer Dreiecksmatrix auf ein Array

Wie in Kapitel 2.5 beschrieben ist ein wichtiger Schritt des ECPNN Algorithmus das Finden von Clusterkombinationen mit minimalen Δs. Um dies erreichen zu können, müssen initial alle Cluster kombiniert und die dadurch entstandenen Kombinationen gespeichert werden. Für eine Anzahl n an Clustern ist die Anzahl der möglichen Kombinationen zweier Cluster gegeben durch:

$$\binom{n}{2} = \frac{n!}{2! \cdot (n-2)!} = \frac{n \cdot (n-1)}{2}$$

Wie in Garrido, Pearlman und Finamore (1995) beschrieben lassen sich alle möglichen Cluster-Kombinationen $C_{i,j}$ in einer quadratischen Matrix darstellen.

$$n = 5$$

$$\begin{array}{c} \begin{matrix} 0 & 1 & 2 & 3 & 4 \end{matrix} \\ \begin{pmatrix} C_{0,0} & C_{1,0} & C_{2,0} & C_{3,0} & C_{4,0} \\ C_{0,1} & C_{1,1} & C_{2,1} & C_{3,1} & C_{4,1} \\ C_{0,2} & C_{1,2} & C_{2,2} & C_{3,2} & C_{4,2} \\ C_{0,3} & C_{1,3} & C_{2,3} & C_{3,3} & C_{4,3} \\ C_{0,4} & C_{1,4} & C_{2,4} & C_{3,4} & C_{4,4} \end{pmatrix} \end{array} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

Der maximale, initiale Speicheraufwand für eine Matrix mit n Clustern wäre somit $O(n) = n^2$. Der effektive Füllgrad, also das Verhältnis von tatsächlich benötigtem Speicher zu alloziertem Speicher, einer solchen Matrix für große n entspräche

$$\lim_{n\to\infty} f(n) = \lim_{n\to\infty} \frac{\frac{n\cdot(n-1)}{2}}{n^2} = \lim_{n\to\infty} \frac{n^2 - n}{2 \cdot n^2} = \lim_{n\to\infty} \frac{1 - \frac{1}{n}}{2} = 50\%$$

Um den Füllgrad auf 100% optimieren zu können müssen folgenden Eigenschaften berücksichtigt werden:

- Die benötigte quadratische Matrix ist symmetrisch.
- Die Diagonale entspricht der Kombination eines Clusters mit sich selbst und wird dementsprechend nicht benötigt.
- Da $C_{i,j} = C_{j,i}$ gilt bildet die obere Dreiecksmatrix ohne Diagonale alle benötigten Cluster-Kombinationen ab.

Gesucht wird also eine Abbildung $k(i, j, n)$, die die Indizes (i, j) einer Dreiecksmatrix mit der Dimension $n \times n$ auf ein Array eindeutig abbildet, wobei $k(i, j, n) = k(j, i, n)$ gelten muss und der Speicher des Array bestmöglich gefüllt werden soll. Für die gesuchte Funktion gilt somit:

$$K : \{0, 1, .., n\} \times \{0, 1, .., n\} \times \{n\} \to \mathbb{N}_0 : (i, j, n) \mapsto k$$

$$k \leq \frac{n \cdot (n-1)}{2}$$

Beispiel für $n = 5$:
Mögliche Kombinationen $= \frac{5 \cdot (5-1)}{2} = 10$

$$\begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ \begin{pmatrix} X & C_{1,0} & C_{2,0} & C_{3,0} & C_{4,0} \\ X & X & C_{2,1} & C_{3,1} & C_{4,1} \\ X & X & X & C_{3,2} & C_{4,2} \\ X & X & X & X & C_{4,3} \\ X & X & X & X & X \end{pmatrix} & \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{matrix} \text{Anzahl Elemente pro Zeile} \\ 5-0-1=4 \\ 5-1-1=3 \\ 5-2-1=2 \\ 5-3-1=1 \\ 5-4-1=0 \end{matrix} \end{array}$$

soll abgebildet werden auf

$$\begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ (C_{1,0} & C_{2,0} & C_{3,0} & C_{4,0} & C_{2,1} & C_{3,1} & C_{4,1} & C_{3,2} & C_{4,2} & C_{4,3}) \end{array}$$

Wie in der abgebildeten Matrix ersichtlich gilt für die Anzahl an Elementen in einer Zeile i einer Dreiecksmatrix ohne Diagonale:

$$g(i, n) = n - i - 1$$

Die Anzahl der Elemente in den Zeilen $\{0, 1, .., i\}$ kann berechnet werden mit

$$\begin{aligned} h(i, n) &= (n - 0 - 1) + (n - 1 - 1) + (n - 2 - 1) + ... + (n - i - 1) = \\ &= (n - 1) \cdot (i + 1) - (0 + 1 + 2 + ... + i) = \\ &= (n - 1) \cdot (i + 1) - \frac{i \cdot (i + 1)}{2} \end{aligned}$$

Der Index k ergibt sich aus der Summe der Elemente in den Zeilen vor i und den Elementen in der Zeile i vor der Spalte j:

$$k(i, j, n) = h(i - 1, n) + j - (n - g(i, n))$$
$$\text{mit } j \geq i$$

Diese Funktion $k(i, j, n)$ ermöglicht den Zugriff in O(1) auf die Kombination zweier Cluster mit den Indizes i, j. Wenn $i > j$ gilt, wird versucht auf die untere Hälfte der Dreiecksmatrix zuzugreifen. Dieses Problem kann umgangen werden indem i und j vertauscht werden wenn $i > j$ auftritt. Durch diesen Tausch wird die Forderung

$k(i, j, n) = k(j, i, n)$ erfüllt.

Die Funktion $k(i, j, n)$ erfüllt alle geforderten Eigenschaften und ermöglicht eine 100% Nutzung des allozierten Speichers, wobei der für die Berechnung des Index benötigte Aufwand gering ist.

3.5.2 Implementierung des ECPNN Algorithmus

Um eine performante Implementierung des ECPNN Algorithmus zu erhalten, wurde dieser in C++ implementiert. Die angewandten Konzepte werden hier, der Verständlichkeit wegen, in Pseudocode beschrieben. Des Weiteren werden nur die wichtigsten Punkte der Implementierung hervorgehoben. Für mathematische Definitionen wird auf andere Kapitel verwiesen.

Es wurde eine Klasse „Cluster“ wie folgt definiert:

```
class Cluster:
    unsigned int id
    double delta_s
    double num_of_clusters
    double[] center
```

Für jedes Cluster wird eine eindeutige Identifikationsnummer, das Δs des Clusters, die Anzahl der in diesem Cluster beinhalteten Datenpunkte und der Mittelpunkt des Clusters gespeichert. Die tatsächlichen Datenpunkte werden nicht gespeichert, da diese für die Berechnung der Cluster nicht benötigt werden. Die Zugehörigkeit eines Datenpunktes zu einem Cluster wird über den geringsten euklidischen Abstand zu dem Clustermittelpunkt definiert.

Als erster Schritt werden alle Cluster mit den Eingabedaten initialisiert. Da für ein nicht kombiniertes Cluster noch kein Δs definiert ist, wird dieses mit 0 initialisiert.

```
# Eingabedaten
TS = {{1,1}, {2,3}, ..}
input_dim = 2

cluster_list = []
cluster_id = 0
for input_vec in TS:
    cluster = Cluster(cluster_id=cluster_id, delta_s=0, num_of_clusters=1,
        input_dim=input_dim, center=input_vec)
    cluster_list.append(cluster)
    cluster_id += 1
```

Die Funktion „merge“ kombiniert zwei gegebene Cluster und berechnet die Werte für dieses neue Cluster. Die Funktionen „calc_merged_center“ und „calc_delta_s“