Dokumentation der Praktischen Arbeit zur Prüfung zum Mathematisch-technischen Softwareentwickler

Thema: Minimierung

3. Mai 2024

Oskar Druska

Prüfungs-Nummer: 101 20015

Programmiersprache: Python

Ausbildungsort: Jülich

Inhaltsverzeichnis

1. Aufgabenanalyse **1**
- 1.1. Aufgabenstellung . . . 1
- 1.2. Lücken und evtl. Fallstricke . . . 2

2. Verfahrensbeschreibung **3**
- 2.1. Programm- und Datenstruktur . . . 3
- 2.2. Input . . . 3
- 2.3. Algorithmus / Minimierung . . . 4
- 2.4. Ausgabe . . . 5
  - 2.4.1. Textoutput . . . 5
  - 2.4.2. Darstellung per Gnuplot . . . 5
  - 2.4.3. Gnuplot Script . . . 6

3. Programmbeschreibung **7**
- 3.1. ProblemSolver . . . 7
- 3.2. InputReader . . . 7
- 3.3. OutputWriter . . . 7
- 3.4. Stadt . . . 7
  - 3.4.1. Konstruktor . . . 8
  - 3.4.2. Getter . . . 8
  - 3.4.3. get_total_distance . . . 8
- 3.5. Positionable . . . 8
  - 3.5.1. Konstruktor . . . 8
  - 3.5.2. Vergleich . . . 8
  - 3.5.3. toString . . . 9
  - 3.5.4. get_distance . . . 9
  - 3.5.5. move . . . 9
  - 3.5.6. Getter . . . 9
- 3.6. Stadtteil . . . 9
- 3.7. RettungsStation . . . 9
  - 3.7.1. Konstruktor . . . 10
  - 3.7.2. move . . . 10
  - 3.7.3. add, remove, und clear_responsibilities . . . 10
  - 3.7.4. Getter . . . 10
  - 3.7.5. cum_dist . . . 10

4. Testdokumentation **13**
4.1. IHK-Beispiel 13
4.2. Normalfälle 13
4.3. Sonderfälle 13
4.4. Fehlerfälle 13

A. Abweichungen und Ergänzungen zum Vorentwurf **15**

B. Benutzeranleitung **17**
B.1. Usage 17
B.2. Failure 18

C. Entwicklungsumgebung **19**

D. Quellcode **21**
D.1. main.py 21
D.2. ProblemSolver 21
D.3. InputParser 24
D.4. OutputWriter 26
D.5. Stadt 29
D.6. Positionable 31
D.7. Stadtteil 34
D.8. RettungsStation 34

1. Aufgabenanalyse

1.1. Aufgabenstellung

Als Schauort der Aufgabenstellung dient eine schematische Repräsentation einer Stadt, welche durch die Errichtung von Rettungsstationen versorgt werden muss. Die Stadt wird als rechteckiges Gebiert mit fester Größe NxM dargestellt. Dieses Gebiet ist in feste, gleich große, quadratische Parzellen, auch Stadtteile genannt, aufgeteilt.

Jedem dieser Stadtteile wird eine natürlich Zahl >= 0 als Anzahl der Unfälle pro Tag zugeordnet. Jedes Quadrat hat eine Seitenlänge von 1000m. Als unveränderliche Inputdaten sind die Größe der Stadt und damit auch die Anzahl und Positionen der Quadrate gegeben. Jedes dieser Quadrate ist mit der Anzahl der Unfälle pro Tag beschriftet. Des Weiteren sind mind. 0 Rettungsstellen gegeben, welche bereits fest im Stadtgebiet stehen, sowie mindestens 1 Rettungsstation, deren neue Position gefunden werden sollen.

Die Rettungsstationen werden an den Kreuzungspunkten der Stadtteile aufgestellt. Nun sollen für die neuen Rettungsstationen Positionen und für alle Rettungsstationen Zuordnungen der Stadtteile gefunden werden, sodass sich eine gewichtete kummulative Distanz der Stadtteile zu ihren Rettungsstationen minimiert.

Umgangssprachlich: Die Rettungsstationen sollen so platziert werden, dass alle Stadtteile abhängig ihrer Unfallquote möglichst gut/schnell versorgt werden können.

Dabei gilt:

- Rettungsstationen versorgen sich nicht untereinander
- Pro Ausfahrt kann nur ein Unfall verarbeitet werden

  Bei x Unfällen muss der Stadtteil also x Mal angefahren werden
- Die Entfernung zwischen einer Rettungsstation und einem Stadtteil wird ungeachtet des Straßennetzes als Luftlinie per euklidischem Abstand berechnet:

  $d = sqrt((x_{ij} - xr_k)^2 + (y_{ij} - yr_k)^2)$

  x_{ij}, y_{ij} als Koordinaten eines Stadtteils

  $xr_k yr_k$ als Koordinaten einer Rettungsstation
- Diese Strecke wird in Abhängigkeit der Unfälle w_{ij} des ij-ten Stadtteils gewichtet: $d * w_{ij}$
- Die akkumulierte Strecke der Einsatzfahrzeuge einer Rettungsstation ist die Summe der gewichtetet Strecken zu allen zugehörgen Stadtteilen:

$d_{Rk} = \sum_{L_k} w_{ij} * d$

- Beachte: hierbei werden nur die Hinwege zu den Unfallorten in Erwägung gezogen. Diese unterscheiden sich aber nur um den Faktor 2 zu allen Wegen und spielen bei der Lösung des Problems daher keine Rolle.

- $f(R)$ ist dann die Summe aller gewichteten Wegstrecken aller Rettungsstationen. Diese soll minimiert werden.

  $f(R) = \sum_{k=1}^{n} \sum_{L_k} w_{ij} * d$

Es handelt sich hier also um ein Minimierungsproblem. Die Inputparameter dieser Funktion sind die Zuordnung der Stadtteile zu den festen Rettungsstationen sowie die Position und Zuordnungen der neuen Rettungsstationen.

1.2. Lücken und evtl. Fallstricke

- Bei der Gewichtung der Stadtteile wird nur die 1-fache Distanz (also nur der Hin- oder Rückweg) beachtet.

  Dies kann aber vernachlässigt werden, da dies die gesamte Distanz nur um de Faktor 2 staucht und sich nicht auf die Monotonie der zu minimierenden Funktion auswirkt.

- Sollten keine neuen, bewegbaren Rettungsstationen gegeben sein, wird die Gesamtdistanz nur anhand der vorhandenen Rettungsstationen berechnet und als Status Quo zurückgegebene.

2. Verfahrensbeschreibung

2.1. Programm- und Datenstruktur

Sinning für die Repräsentation der Stadt ist ein 2D-Array der Form NxM. Dieses Array hält Objekte, welche einen Stadtteil repräsentieren; idealerweise mit Koordinaten und Unfallquote. Die Vorraussetzung, dass die Stadt in NxM gleich große Quadrate eingeteilt wird, garantiert, dass eine diskrete Darstellung der Stadt als Array genügen wird.

Über die Indizes i,j des Arrays können die Stadtteile entsprechend der Abstandsformeln angesprochen werden. Jeder Stadtteil enthält dann eine Repräsentation seines Mittelpunkts in Form von Meterkoordinate, die dann verwendet werden, um Abstände zu berechnen. Für die Instanziierung der Stadtteile gemäß der Indizes des haltenden Arrays gilt:

Stadtteil.Position auf [i, j] = $(i * 1000 + 500, j * 1000 + 500)$

Die Verschiebung von 500 ist notwendig, da sich die Mittelpunkte der Stadtteile nicht auf dem gradnetz der Stadt befinden.

Für die Rettungsstationen lässt sich das gleiche Prinzip verwenden. Diese Stehen aber im Gegensatz zu den Stadtteilen auf den Kreuzungspunkten und daher dem Gradnetz der Stadt. Die Rettungsstation auf Abb. 1 der Aufgabenstellung steht genau auf [2, 1] == (2000m, 1000m)

Hier wird oBdA davon ausgegangen, dass der Ursprung des Koordinatensystems in der unteren linken Ecke der Stadt liegt.

Die Zuordnung der Stadtteile erfolgt über eine Liste bei den Rettungsstationen, welche Referenzen auf die entsprechenden Stadtteilobjekte hält.

Bei der Implementierung muss dann darauf geachtet werden, dass keine Stadtteile mehrfach zugeordnet werden.

2.2. Input

Auf Seite 3 der Aufgabenstellung ist eine beispielhafte Eingabedatei als pure Textdatei gegeben. Diese enthält Zeilenweise Informationen zur Dimension der Stadt in Metern, die Anzahl der festen und bewegichen Rettungsstationen, die Position der festen Rettungsstationen, sowie eine zeilenweise Aufschlüsselung der Unfallquoten, schematisch in Form der Stadt angeordnet.

Aus der Dimension der Stadt kann die Größe des 2D-Arrays abgeleitet werden. Bsp: (12000, 6000) -> int[12, 6]

Äquivalt kann die Metermaßangabe der Rettungsstationen in Indizes umgewandelt werden: Bsp: (2000, 1000) -> int[2, 1]

Die schematische, zeilenhafte Aufschlüsselung der Unfallquoten wird genutzt, um die Stadtteile im Stadtteilarray zu instanziieren. Beachte: der erste Eintrag der Inputdatei entspricht dem [0, n-1]ten Eintrag des Stadtttteilarrays, insofern man den Koordinatenursprung der Stadt nach links-unten setzt.

Jede Inputdatei beginnt mit einer textuellen Beschreibung der Stadt, zeilenweise angeührt von //ünd kann vom Parsingprozess ignoriert werden.

2.3. Algorithmus / Minimierung

Das zu minimierende Problem ist die akkumulierte Abstandsfunktion $f(R)$.

$$f(R) = \sum_{k=1}^{n} \sum_{(i,j) \in L_k} w_{i,j} * \sqrt{(x_{ij} - xr_k)^2 + (y_{ij} - yr_k)^2}$$

- n: Anzahl der Rettungsstationen, fest und neu
- L_k: Liste der Koordinaten (in Metern) der zugehörigen Stadtteile der k-ten Rettungsstation
- w_{ij}: Unfallquote des Stadtteils auf [i, j]
- x_{ij}, y_{ij}: x,y Komponente des Stadtteils auf [i, j] (in Metern)
- xr_k, yr_k: x,y Komponente der k-ten Rettungsstation (in Metern)

1. Platziere die festen Rettungsstationen auf dem Stadtplan (wie per Input gegeben)
2. Platziere die neuen RS zufällig: beachte, dass nicht zwei Rettungsstationen den gleichen Standort haben können
3. Ordne alle Stadtteilen ihren nächsten Rettungsstationen zu: s. Abstandsfunktion d
4. Berechen für jede neue Rettungsstation den Pseudoschwerpunkt P als die durchscnittliche x und y Koordinate aller zugehörigen Stadtteile
5. Verschiebe die Rettungsstationen auf ihre Pseudoschwerpunkte: beachte, dass Rettungsstationen nur auf Kreuzungspunkten stehen können.
6. Wiederhole ab 3
7. Wenn sich die Positionen der Rettungsstationen nicht mehr ändern: beende und berechne f(R)

Diese finale Zuordnung ist lt. Aufgabenstellung von der zufälligen Startkofiguration abhängig und liefert nur ein lokales Minimum.

Um ggf. bessere Ergebnisse zu finden, ist es sinnvoll, den Algorithmus mehrmals mit verschiedenen Startkonfigurationen laufen zu lassen und dann das insgesamt kleinste

Ergebnis zu wählen. Evtl. lassen sich Heueristiken aus dem Machine Learning anweden, um möglichst optimale Startkonfigurationen zu finden.

Die Anzahl der Iterationen sollte ggf. Abhängig von der Größe der Stadt und Anzahl der Rettungsstationen gewählt werden, um einen möglichen Laufzeitüberhang zu vermeiden.

2.4. Ausgabe

Als Ausgabe sind zwei verschiedene Ausgabeformate gefordert:

2.4.1. Textoutput

Die Standardausgabe beginnt wie die Input Datei mit einer textuellen beschreibung der Stadt. Wie auch beim Input beginnen diese mit //üm einem Parser zu signalisieren, dass diese ignoriert werden können. Daraufhin folgen 2 Zeilen, welche die Anzahl der festen und neuen REttungsstationen angeben. Nun werden die Rettungsstationen und ihre zugehörigen Stadtteile aufgeschlüsselt. Pro Rettungstation wird genannt:

- neue oder alte Rettungsstation
- Position in Meterkoordinaten
- alle zugewiesenen Stadtteile in Meterkoordinaten
- zudem die gewichtete Distanz zu diesen Stadtteilen

Sobald alle Rettungsstationen wie oben aufgeführt wurden, wird zuletzt die als minimal bestimmte Gesamtdistanz $f(R)$ ausgegeben.

2.4.2. Darstellung per Gnuplot

Gnuplot ist ein open-source Plottingtool, welches ursprünlich entwickelt wurde, um mathematische Funktionen zu plotten. Es kann in diesem Kontext genutzt werden, um eine schematische Darstellung der Stadt mit ihren Stadtteilen und Rettungstationen zu erzeugen.

Es ermöglicht das Plotten per Skriptsprache. Es soll nun eine CSV Datei erstellt werden, auf dessen Basis Gnuplot einen Plot der Stadt erstellen kann. Die Stadt wird als Koordinatensystem gezeigt, Stadtteile und Rettugnsstationen werden als Kreise dargstellt. Jede Zeile der CSV Datei repräsentiert einen Stadtteil und sieht wie folgt aus:

<X>, <Y>, <sqrt(unfallquote)>, <unfallquote>, <farbe_rot>, <farbe_gruen>, <farbe_blau>

X und Y sind die Meterkoordinaten eines Stadtteils oder eine Rettungsstation. Die Unfallquote sollen innerhalb des Kreises angezeigt werden. Die Wurzel der Unfallquote wird als Radius des Kreises verwendet und wird bei einer Unfallquote von 0 auf 0.7 gesetzt um dennoch einen Kreis zu zeigen.

Die Farbkomponenten dienen, um den Kreis einzufärben. Alle Stadtteile, die einer Rettungsstation angehören, sollen die gleiche Farbe tragen. Rettungsstationen werden immer in Einheitsrot eingefärbt.

Die Größe der Rettungsstationen wird allgemein auf 2 gesetzt, die Zahl im zugehörgen Kreis stellt die ID der Station dar.

2.4.3. Gnuplot Script

Die CSV wird von Gnuplot innerhalb eines Plottingskripts verarbeitet. Dieses Skript ist gegeben. Auf eine genaue Erläuterung wird an dieser Stelle verzichtet. Das Skript muss allerdings an jeden Input angepasst werden, um die entsprechende Outputdatei finden zu Können. Desweiteren müssen die Dimensionen der Koordinatenachsen angegeben werden. Diese sind auch je nach Input unterschiedlich. Eine Vorlage für das gnuplot wird in den Outputprozess des Programms gehardcoded.

Bemerke: Aufgrund des gewählten Algorithmus ist das Ergebnis nicht eindeutig. Textoutput und Gnuplots können also von den Beispielergebnissen abweichen.

3. Programmbeschreibung

Das Kapitel Aufgabenanalyse1 enthält bereits eine textuelle Verfahrensbeschreibung.

Dieses Kapitel bietet eine genauerer Aufschlüsselung der verwendeten Datentypen sowie UML Diagramme.

3.1. ProblemSolver

Die Hauptklasse ProblemSolver kontrolliert den Input-, Rechen- und Outputprozess. Sie hält die Objekte Stadt, InputReader, und OutputWriter.

Auf dem Objekt Stadt führt sie den in 2.3 beschriebenen Algorithmus aus.

Die interne Methode _init_problem() initialisiert die zufälligen Startpositionen der neuen Rettungsstationen

3.2. InputReader

Die Inputklasse InputReader erhält einen Dateipfad und liest aus diesem die nötigen Informationen für ein Stadtobjekt aus. Die Methode create_stadt() gibt eine populiertes Stadtobjekt mitsamt Stadtteilen und Rettungsstationen zurück

3.3. OutputWriter

Die Outputklasse OutputWriter erhält ein Stadt Objekt wahlweise als Text- oder CSV-Datei geschrieben wird. Im Falle eines CSV Outputs wird auch ein passendes Gnuplot Skript erstellt und geschrieben.

Der Name des Stadtobjekts dient als Grundlagen für die Namen der Outputdateien.

3.4. Stadt

Das Stadt Objekt bildet die zentrale Datenstruktur zur Lösung des vorliegenden Minimierungsproblems. Es hält ein 2D-Python-Array an Stadtteilen, eine Liste aller Rettungsstationen, Informationen über Länger und Breite der Stadt, sowie einen Namen. Stadtteile und Rettungsstationen sind Instanzen der gleichnamigen Klassen, Länger und Breite werden als Python-Integer gespeichert; der Name ist ein Python-String.

3.4.1. Konstruktor

Der Stadt-Konstruktor erhält einen Python-String als name, ein 2D-Integer-Python-Array welches die Unfallquoten der einzelnen Stadtteile und eine 1D-Array an Rettungsstationen. Aus der Aufgabenstellung heraus ist gegebene, dass das Stadtgebiet in NxM 1qkm große Quadrate eingeteilt wird. Aus den Dimensionen der Unfallquotenarrays lassen sich daher die Maße der Stadt ableiten. Für jeden Eintrag der Unfallquoten wird ein Stadtteil mit einer Koordinatenangabe und der Unfallquote erstellt und in das Stadtteilarray populiert.

3.4.2. Getter

Die Klasse Stadt bietet für alle Attribute eine Getter-Methode an. Insbesondere die Maßangaben dürfen nicht extrinsisch verändert werden, da dies sonst zu inkonsistenzen bzgl. des Stadtteilearrays führt.

3.4.3. get_total_distance

Dies ist die Implementierung der Funtion $f(R)$ aus der Aufgabenstellung. Für jede Rettungsstation im Stadtobjekt wird die gewichtetet Distanz cum_dist() zu allen zugehörigen Stadtteilen berechnet. Dies wird aufsummiert und bildet dann die Gesamtdistanz.

3.5. Positionable

Die Pythonklasse Positionable wird als Interface gehandhabt und bietet eine Schnittstelle für alle Objekte, die eine definierte Position auf dem Stadtplan haben. Da Python an sich keine Interfaces bietet, wird diese wie eine normale Klasse gehandhabt. Sie hält die X und Y Komponente einer 2-dimensionalen Koordiante sowie einen Boolean, der besagt, ob das Positionable mit Anderen kollidieren kann. Sie führt Protokoll über alle Positionables mit Kollision, um ebensolche ggf. detektieren zu können.

3.5.1. Konstruktor

Der Konstruktor des Positionables erhält 2 Integer, eine X und Y Komponente, sowie einen Boolean, der besagt, ob das Positionable kollidieren kann oder nicht. Standardmäßig ist dieser auf False gesetzt.

3.5.2. Vergleich

Zwei Positionables sind gleich, wenn diese die gleiche X und Y Komponente haben.

3.5.3. toString

Die Methode __str__() ermöglicht implizietes String-Casting. Ein Positionable wird textuell simpel als "(x,y)"dargestellt.

3.5.4. get_distance

Die Methode get_distance erhält ein weiteres Positionable und gibt den euklidischen Abstand zwischen diesen beiden zurück (auch L2-Norm genannt).

3.5.5. move

Die methode move() bewegt self auf die Position des gegebenen Positionables. Es können 2 Exceptions geworfen werden:

1. NoMovementException - Abgeleitet von Exception: wird geworfen, wenn self und Positionable gleich sind

2. CollisionException - Abgeleitet von Exception: wird geworfen, wenn die Verschiebung eine Kollision auslösen würde

3.5.6. Getter

Die Klasse Positionable bietet Getter-Methoden, um die X und Y Komponenten, sowie die Kollisionseigenschaft eines Positionable abfragen zu können.

Die Koordinaten eines Positionables dürfen nicht einfach so verändert werden, um Kolliosionen etc. überprüfen zu können.

3.6. Stadtteil

Die Stadtteil Klasse ist eine Erweiterung der Positionable Klasse und repräsentiert einen Stadtteil. Es speichert die Unfallquote des Stadtteils und überschreibt die Methode move(), sodass Stadtteile nicht bewegt werden können. Sie hält einen Getter für die Unfallquote.

3.7. RettungsStation

Die Klasse RettungsStation ist eine Erweiterung der Positionable Klasse und repräsentiert eine Rettungsstation. Sie hält eine Liste an schutzbefohlenen Stadtteil Objekten sowie einen Boolean, welcher besagt, ob die Rettungsstation alt und daher fest, oder neu und beweglich ist. Zudem enthält sie eine Klassenvariable, welche die Anzahl der erstellten Rettungsstationen

3.7.1. Konstruktor

Der Konstruktor nimmt die Beweglichkeit als Boolean und sowohl eine X und Y Komponente zur Initialisierung des zugrundeliegenden Positionables. Des Weiteren wird der Rettugnsstation mittels des statischen Counters eine sequentiell inkrementierte ID zugeordnet. Diese wird lediglich genutzt, um den Rettungsstationen im resultierende Gnuplot Nummern zuordnen zu können.

3.7.2. move

Die Methode move() der Superklasse Positionable wird überschrieben, um vorher überprüfen zu können, ob eine REttungsstation überhaupt beweglich ist. Solte diese es sein, dann wird das Zielpositionable am Gradnetz der Stadt ausgerichtet. Rettungsstationen können laut Aufgabenstellung nur auf den Kreuzungspunkten der Stadt stehen, also Punkte, dessen X und Y Komponente Vielfache von 1000 sind.

3.7.3. add, remove, und clear_responsibilities

Der Liste der schutzbefohlenen Stadtteile sollen von außen Stadtteile angefügt und entfernt werden können. Außerdem muss diese vor der erneuten berechnung der Pseudoschwerpunkt geleert werden, um Mehrfachzuordnungen zu vermeiden.

3.7.4. Getter

Die Klasse RettungsStation bietet getter für all ihre Attribute an. So können z. B. weder die Beweglichkeit noch die ID einer Rettungsstation nachträglich geändert werden.

3.7.5. cum_dist

Die Methode cum_dist gibt die gewichtete Gesamtsumme der Distanzen zu allen schutzbefohlenen Stadtteilen zurück. Die Summe dieser Größe über alle Rettungsstationen hinweg, bildet die zu minimierende Größe der gesamten Problemstellung. Diese Methode wird vom Stadtobjekt genutzt, um get_total_distance zu berechnen.

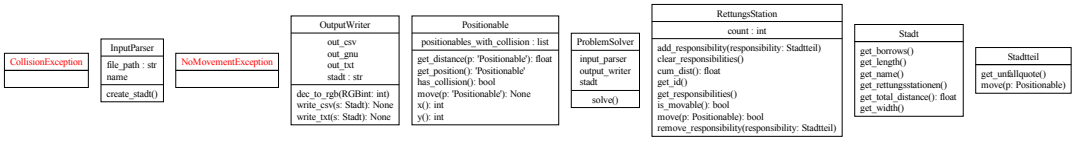

Abbildung 3.1.: Klassendiagramme

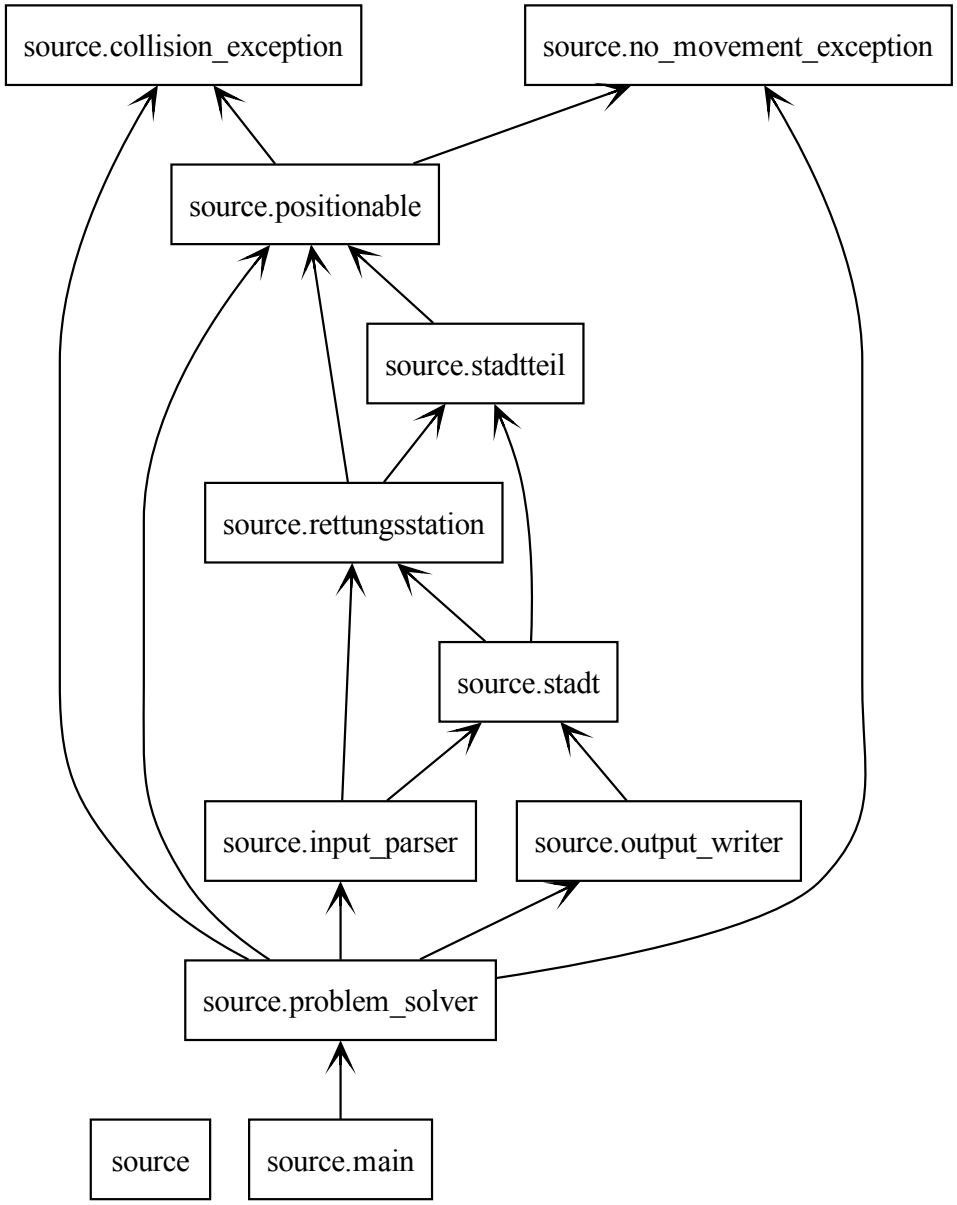

Abbildung 3.2.: Klassenstruktur

4. Testdokumentation

4.1. IHK-Beispiel

4.2. Normalfälle

4.3. Sonderfälle

4.4. Fehlerfälle

A. Abweichungen und Ergänzungen zum Vorentwurf

Ich bin in folgenden Punkte von meinem Entwurf des ersten Tages abgewichen:

- Anstatt die Stadt nur als Integer-Array zu speichern, habe ich mich für einen objektorientierteren Ansatz entschieden.

  Das Stadtarray hält nun Objekte des Typs Stadtteil, welche neben der Unfallquote auch die Position in Metern tracken. So kann auf den Standort eines Stadtteils einfach per borrow.get_Positionable() zugegriffen werden. Bei der Berechnung der euklidischen Distanz sind dann keiene Indizekonversionen abhängig vom Typ des POsitionables nötig.

- Die Einführung der Superklasse Positionable erlaubt die gemeinsame Handhabung von Stadtteilen und Rettungsstationen als Entitäten, welche einen Standort im Stadtgebiet haben.

  Beide lassen sich dann durch eine methode move() bewegen. Stadtteil.move() verhindert ein Umsetzten des Stadtteils, während RettungsStation.move() den neuen Standort einer Station am Gradnetz der Stadt ausrichtet.

  Das Bewegen einer Rettungsstation und die DIstanzberechnung zweier Positionables wird nun über move() und get_distance() gekapselt.

- Bei der Berechnung der Pseudoschwerpunkte werden nun nur Stadtteile mit einer Unfallquote größer als 0 berücksichtigt. Dies war zwar auch so von der Aufgabenstellung gefordert, hatte ich aber am ersten Tag außer Acht gelassen.

B. Benutzeranleitung

B.1. Usage

Stellen Sie sicher, dass Sie sich im Root-Directory des Projekte befinden. Ihr

```
ls -l
```

Output sollte in etwa so aussehen:

```
$ ls -l
total 7784
-rw-r--r--   1 odruska  staff      486 May  3 09:14 README.md
-rwxr-xr-x   1 odruska  staff       81 May  3 09:14 TEST_GROPRO_2024.sh
drwx------@ 32 odruska  staff     1024 May  3 07:26 docu
drwxr-xr-x@  5 odruska  staff      160 May  2 14:42 input
-rw-r--r--@  1 odruska  staff  3971378 May  1 21:01 matse_test_gropro_20
drwxr-xr-x@  5 odruska  staff      160 May  3 08:33 output
drwxr-xr-x@ 14 odruska  staff      448 May  3 09:11 source
```

Stellen Sie sicher, dass gnuplot und Python3.x installiert sind. Die automatisierte Ausführung des Programms wird von einer ZShell gehandhabt. Folgendes Kommando sollte daher nicht scheitern:

```
which gnuplot;
which python3;
which zsh;
```

Den Lösungsprozess der Beispiel sowie der Testfälle übernimmt ein Shellscript. Es ruft auch gnuplot auf, um die Ergebnisse schemenhaft darzustellen:

```
./TEST_GROPRO_2024.sh
```

Dieses Projekt nutzt Python Virtual Environments. Dieses befindet sich unter

```
./venv_test_gropro_2024
```

und sollte vom ShellScript automatisch aktiviert und deaktiviert werden.

GnuPlot öffnet ein Fenster pro Ausgabe. Diese können evtl. exakt übereinander liegen. Lassen Sie sich davon nicht irritieren. (Mir ist das nämlich passiert ;-))

B.2. Failure

Sollte das ShellScript nicht das tun, was es tun soll, können Sie die Beispiele und den Gnuplot Output einzeln erstellen.

```
python3.12 ./source/main.py <InputFile>
```

löst das Problem, welches in InputFile definiert wurde. Es kreiert auch das GnuPlot Skript und die benötigte CSV Datei im Order output

```
gnuplot -p <.gnuFile>
```

erstellt dann den Plot. Wichtig ist, dass die benötigte CSV Datei im Ordner output liegt.

C. Entwicklungsumgebung

Programmiersprache	:	Python
Compiler	:	Python3.12
Rechner	:	2021 MacBook Pro, Apple M1 Pro
Betriebssystem	:	MacOS 14.3

D. Quellcode

D.1. main.py

```
from problem_solver import ProblemSolver
import argparse


def main() -> None:
    """
    main
    :return: None
    """
    parser = argparse.ArgumentParser(description=
                                     "find optimal city configuration")
    parser.add_argument('-f', '--file', type=str, required=True)
    args = parser.parse_args()

    solver = ProblemSolver('Matsehausen.txt')
    solver.solve()
    print("problem solved")


if __name__ == '__main__':
    main()
```

D.2. ProblemSolver

```
from input_parser import InputParser
from output_writer import OutputWriter
from positionable import Positionable
from no_movement_exception import NoMovementException
from collision_exception import CollisionException
import random


class ProblemSolver:
    def __init__(self, input_file: str):
```

```
        self.input_parser = InputParser(input_file)
        self.stadt = self.input_parser.create_stadt()
        self.output_writer = OutputWriter(self.stadt)

    def _init_problem(self, seed=None):
        """
        Initialize the problem by choosing random
        starting Positionables for movable rescue stations
        :return:
        """

        print(f"initializing problem")

        if seed:
            random.seed(seed)

        # find random starting points for movable _rettungsstationen
        rs = [r for r in self.stadt.get_rettungsstationen() if r.is_mo
        for r in rs:
            while True:
                x = random.randint(0, self.stadt.get_length())*1000
                y = random.randint(0, self.stadt.get_width())*1000
                try:
                    # if moving passes -> place next Rettungsstation
                    r.move(Positionable(x, y))
                    break
                except ValueError as e:
                    print(e)
                    pass

    def solve(self):
        """
        Solve Stadt problem using the following algorithm:
        1. Initialize the problem by choosing random starting position
        2. Compute distances from each borrow to each resuce station
        3. assign each rescue station all their nearest borrows
        4. Compute pseudo center-of-mass for each rescue station weigh
        5. Move each movable RS to the crossing nearest to that pseudo
        6. loop through 2.
        7. once no RS moves anymore, compute collective distances and
        :return:
        """
        self._init_problem()
```

```
        positions_changed = True

        while positions_changed:
            print("Keep moving rescue stations")
            positions_changed = False

            # Alle Zuständigkeiten aller Rettungsstationen löschen
            print("clear reponsibilities")
            for r in self.stadt.get_rettungsstationen():
                r.clear_responsibilities()

            # Stadtteile ihren nächsten Rettungsstationen zuordnen
            print("assigning borrows to rescue stations")
            for s in self.stadt.get_borrows():
                dist_r = []
                for r in self.stadt.get_rettungsstationen():
                    dist_r.append((s.get_distance(r), r))
                min(dist_r, key=lambda p: p[0])[1].add_responsibility(s)

            # Schwerpunkte berechnen und bewegliche Rettungsstationen ve
            print("compute pseudo center-of-mass for each rescue station
            for r in [r for r in self.stadt.get_rettungsstationen() if r
                borrows = r.get_responsibilities()  # alle Stadtteile ei
                # die Stadtteile aus borrows, welche mind, 1 Unfall pro
                # -> Stadtteile ohne Unfälle bei der Berechnung vernachl
                borrows_not_zero = [b for b in borrows if b.get_unfallqu

                if not borrows_not_zero: # Borrows empty
                    # Sollte eine RS keine nicht-null-unfälle Zugehörigk
                    continue

                pseudo_x = sum([b.x() * b.get_unfallquote() for b in bor
                pseudo_y = sum([b.y() * b.get_unfallquote() for b in bor

                # make sure, RS dont get positioned outside of stadt bou
                if pseudo_x > self.stadt.get_length()*1000:
                    pseudo_x = self.stadt.get_length()*1000
                if pseudo_x < 0:
                    pseudo_x = 0

                if pseudo_y > self.stadt.get_width()*1000:
                    pseudo_y = self.stadt.get_width()*1000
                if pseudo_y < 0:
                    pseudo_y = 0
```

```
                pseudo = Positionable(round(pseudo_x), round(pseudo_y)

                try:
                    r.move(pseudo)
                    positions_changed = True
                except CollisionException as c:
                    print(c)
                    continue
                except NoMovementException as n:
                    print(n)
                    continue

        # Positionen nicht geändert
        print("Rescue stations havent changed positions anymore. Finis
        # lokales Minimum der aktuellen startkonfiguration gefunden
        self.output_writer.write_txt()
        self.output_writer.write_csv()
```

D.3. InputParser

```
from stadt import Stadt
from rettungsstation import RettungsStation


class InputParser:
    def __init__(self, file_path: str):
        self.file_path = file_path
        self.name = file_path.split("/")[-1].split(".")[0]
        # self._parse_input()

    def _parse_input(self):
        """
        parse given file path and read out the values
        neded to create Stadt object

        DOESNT WORK
        :return:
        """
        _unfallquote = []
        _r_old = []
        _r_new = []
```

```
        _city_shape = ()
        _oldnew_keyword = {"neu": True, "alt": False}

        # return tuple containing count and wether old or new
        read_rescue_station = lambda l: int(l.split(" ")[2])
        read_city_shape = lambda l: ( int(l.split(" ")[1].split(",")[0])
                                      int(l.split(" ")[1].split(",")[1])
                                      )
        read_position = lambda l: (int(l.split(",")[0]), int(l.split(","


        print('parsing input file')
        with open(self.file_path, "r") as f:
            lines = f.readlines()
            # Throw out all comment lines
            lines = [l for l in lines if not l.startswith("//")]

            for line in lines:
                try:
                    _city_shape = read_city_shape(line)
                    continue
                except Exception as e:
                    try:
                        r_count = read_rescue_station(line)
                        for i in range(r_count):
                            _r_old.append(read_position(line+i))
                    except Exception as e:
                        pass



    def create_stadt(self):
        """
        create Stadt object based on input file
        :return:
        """
        print(f"creating stadt")

        # currently hardcoded bc parsing doesnt work yet
        u = [[3, 2, 1, 0, 0, 2, 1, 2, 3, 0, 0, 4],
             [3, 1, 0, 2, 1, 0, 0, 0, 1, 2, 1, 0],
             [0, 0, 0, 2, 0, 0, 1, 2, 0, 1, 0, 0],
             [0, 0, 0, 1, 3, 1, 0, 0, 0, 2, 3, 1],
             [0, 1, 2, 1, 2, 0, 0, 2, 2, 1, 2, 1],
```

```
            [2, 0, 1, 1, 2, 0, 0, 0, 0, 4, 3, 2]]

        name = self.name

        r1 = RettungsStation(immovable=True, x=2000, y=1000)
        r2 = RettungsStation(immovable=False)
        r3 = RettungsStation(immovable=False)

        r = [r1, r2, r3]

        return Stadt(name, u, r)
```

D.4. OutputWriter

```
from stadt import Stadt
import csv
import math
import numpy as np


class OutputWriter:
    _gnu_template = \
'''
# {0}
set title "{0}"
rgb (r,g,b) = int(r)*65536 + int(g)*256 + int(b)
set xrange [0:{1}]
set yrange [0:{2}]
set size ratio -1
unset key
set datafile separator whitespace
scale=200
textColor = 'black'
set multiplot layout 1,1
plot "./output/{0}.csv" using 1:2:($3*scale):(rgb($5,$6,$7)) \\
     with circles lc rgb variable fs transparent solid 0.5, \\
     "./output/{0}.csv" using 1:2:(sprintf("%d", $4)) with labels noti
     textcolor rgb textColor
'''

    _movable_keyword = {True: 'neu', False: 'alt'}
```

```
    def __init__(self, s: Stadt):
        """
        Creates OutputWriter object to write stadt config
        into a txt or csv file
        :param file_out:
        """
        self.stadt = s
        self.out_txt = ("output/" + self.stadt.get_name() + ".txt")
        self.out_csv = ("output/" + self.stadt.get_name() + ".csv")
        self.out_gnu = ("output/" + self.stadt.get_name() + ".gnu")

    @staticmethod
    def dec_to_rgb(RGBint: int):
        """
        Converts an integer to RGB values
        :param RGBint:
        :return: (red, green, blue)
        """
        blue = RGBint & 255
        green = (RGBint >> 8) & 255
        red = (RGBint >> 16) & 255
        return red, green, blue

    def write_txt(self) -> None:
        """
        Writes stadt config to txt file
        :param s: Stadt
        :return: None
        """
        print("write txt output")

        s = self.stadt

        out = ""
        out += "//**********************************\n"
        out += f"// Stadtplan {s.get_name()}\n"
        out += f"// {len([r for r in s.get_rettungsstationen() if not r.
               + f"und {len([r for r in s.get_rettungsstationen() if r.i
               + f"neue Stationen\n"
        out += "//**********************************\n"
        out += f"Rettungsstellen alt: {len([r for r in s.get_rettungssta
        out += f"Rettungsstellen neu: {len([r for r in s.get_rettungssta

        out += "\n"
```

```
        for i, r in enumerate(s.get_rettungsstationen()):
            out += f"Rettungsstelle: {i} - {OutputWriter._movable_keyw
            out += str(r.get_position())
            out += "\nZugeordnete Stadtteile:\n"

            for count, borrow in enumerate(r.get_responsibilities()):
                out += str(borrow.get_position())
                out += " "
                count += 1
                if count % 5 == 0:
                    out += "\n"
            out += "\n"
            out += f"Gewichtete Stadtteile: {r.cum_dist()}\n"
            out += "\n"

        out += "\n"
        out += f"Gesamtstrecke: {s.get_total_distance()}"

        with open(self.out_txt, "w", newline="") as f:
            f.write(out)

    def _write_gnu_script(self, s):
        """
        Write gnuplot script fitted to current problem.
        Makes use of class variable _gnu_template
        :param s:
        :return:
        """
        print("write gnuplot script")
        # modify and write gnuplot script for specific problem
        with open(self.out_gnu, 'w', newline='') as gnu_script:
            gnu_script.write(OutputWriter._gnu_template.format(
                s.get_name(),
                s.get_length() * 1000,
                s.get_width() * 1000
            ))
        gnu_script.close()

    def write_csv(self) -> None:
        """
        Writes stadt config to csv file
        :return: None
        """
```

```
        print("write CSV output")
        s = self.stadt

        self._write_gnu_script(s)

        # create and write CSV file to be read by Gnuplot
        with (open(self.out_csv, 'w', newline='') as csv_file):
            writer = csv.writer(csv_file, delimiter=" ")

            dec_colors = np.linspace(0x0, 0xFFFFFF,
                                     num=len(s.get_rettungsstationen()), d

            for r, dec_color in zip(s.get_rettungsstationen(), dec_color
                rgb = self.dec_to_rgb(dec_color)
                for borrow in r.get_responsibilities():
                    root_u = math.sqrt(borrow.get_unfallquote()
                                       )if borrow.get_unfallquote() > 0

                    writer.writerow(
                        [borrow.get_position().x(),
                         borrow.get_position().y(),
                         root_u,
                         borrow.get_unfallquote(),
                         rgb[0],  # red
                         rgb[1],  # green
                         rgb[2]]  # blue
                    )
            for r in s.get_rettungsstationen():
                writer.writerow(
                    [r.get_position().x(),
                     r.get_position().y(),
                     2,
                     r.get_id(),
                     255,  # red
                     0,  # green
                     0]  # blue
                )
```

D.5. Stadt

```
from stadtteil import Stadtteil
```

```
from rettungsstation import RettungsStation


class Stadt:
    def __init__(self, name: str, unfallquoten: list[list[int]], rettu
        """
        Create Stadt object holding rescue stations, borrows and their
        Is assigned a name which is used to create the output files
        :param name:
        :param unfallquoten: used to create borrows
        :param rettungsstationen: List of RettungsStation objects
        """
        self._name = name
        self._rettungsstationen = rettungsstationen

        self.__length = len(unfallquoten[0])
        self.__width = len(unfallquoten)

        # TODO: fix assigning coordinates so that
        #  bottom left input is 0,0 in coordinate system
        self._borrows = [Stadtteil(x=i * 1000 + 500, y=k * 1000 + 500,
                         for i in range(self.__length)
                         for k in range(self.__width)]
        pass

    def get_name(self):
        """
        :return: Stadt name
        """
        return self._name

    def get_width(self):
        """
        :return: Stadt width
        """
        return self.__width

    def get_length(self):
        """
        :return: Stadt length
        """
        return self.__length

    def get_borrows(self):
```

```
        """
        :return: Stadt borrows as 2d-array
        """
        return self._borrows

    def get_rettungsstationen(self):
        """
        :return: Stadt rettungsstationen as list
        """
        return self._rettungsstationen

    def get_total_distance(self) -> float:
        """
        Compute the total distance based on the weighed distances
        of each rescue station
        :return:
        """
        print("compute total distance")
        sum_ = 0.0
        for r in self._rettungsstationen:
            sum_ += r.cum_dist()
        return sum_
```

D.6. Positionable

```
import math
from no_movement_exception import NoMovementException
from collision_exception import CollisionException


class Positionable:
    """
    Coordinate object to track positions.
    Can only have integer coordinates.
    Also tracks all Positionables with collision.
    """
    positionables_with_collision = []

    def __init__(self, x: int, y: int, collision=False):
        """
        create Positionable object to track coordinates of entitites
        :param x: x-coordiante
```

```
        :param y: y-coordiante
        :param collision: wether or not two Positionable objects colli
        """

        self._x = int(x)
        self._y = int(y)
        self._collision = collision

        if self._collision:
            self.positionables_with_collision.append(self)

    def __eq__(self, other: 'Positionable') -> bool:
        """
        True, if all coordinates of Positionable are equal element-wis
        :param other: Positionable object to compare
        :return: bool
        """
        return self._x == other.x() and self._y == other.y()

    def __str__(self) -> str:
        """
        String representation of Positionable object.
        Shows current coordinates.
        :return: str
        """
        return f"({self._x},{self._y})"

    def get_distance(self, p: 'Positionable') -> float:
        """
        Calculates distance between two Positionable object
        :param p: Positionable object
        :return: float
        """
        return math.sqrt(
            (self._x - p.x()) ** 2
            + (self._y - p.y()) ** 2
        )

    def get_position(self) -> 'Positionable':
        """
        returns self
        :return: self: Positionable
        """
        return self
```

```
    def has_collision(self) -> bool:
        """
        returns True, if Positionable has collision
        :return:
        """
        return self._collision

    def x(self) -> int:
        """
        return x-coordinate of Positionable object
        :return: int
        """
        return self._x

    def y(self) -> int:
        """
        return y-coordinate of Positionable object
        :return: int
        """
        return self._y

    def move(self, p: 'Positionable') -> None:
        """
        Move self to the given Positionable p.
        :throws: NoMovementException if self == p
        :throws: CollisionException if p is occupied by Positionable wit
        :param: p - target Positionable
        :return: None
        """
        print("Move " + str(self) + " to " + str(p))

        if self == p:
            raise NoMovementException("Self and p are on the same spot")

        if self.has_collision():
            # new position matches any of the positionables with collisi
            # ergo p is a position with collision
            if any([p == collisionable for collisionable in self.positio
                raise CollisionException('COLLISION occured - cannot mov

        self._x = p.x()
        self._y = p.y()
```

D.7. Stadtteil

```
from positionable import Positionable


class Stadtteil(Positionable):
    def __init__(self, x: int, y: int, unfallquote: int):
        Positionable.__init__(self, x, y, collision=False)
        self._unfallquote = unfallquote

    # Override move methode, because _borrows shall not be movable
    def move(self, p: Positionable):
        print("Stadtteil immovable")
        pass

    def get_unfallquote(self):
        return self._unfallquote
```

D.8. RettungsStation

```
from positionable import Positionable
from stadtteil import Stadtteil


class RettungsStation(Positionable):
    count = 0

    def __init__(self, immovable: bool, x=-1, y=-1):
        print("create RettungsStation")

        Positionable.__init__(self, x, y, collision=True)

        # Private, bc solidity does not change
        self.__immovable = immovable

        # Private, bc IDs should not be changed by user just-like-that
        RettungsStation.count += 1
        self.__id = RettungsStation.count

        self._responsibilities = []

    def move(self, p: Positionable) -> bool:
```

```
        print("trying to move RS")
        if self.is_movable():
            # snap target coordinates to nearest crossing
            # round to thousand
            nearest_crossing = Positionable(x=1000 * round(p.x()/1000),
                                            y=1000 * round(p.y()/1000))

            print(f"nearest_crossing is {nearest_crossing}")

            Positionable.move(self, nearest_crossing)
        else:
            print("Rettungsstation nicht bewegbar -- noop")
            return False

    def add_responsibility(self, responsibility: Stadtteil):
        self._responsibilities.append(responsibility)

    def remove_responsibility(self, responsibility: Stadtteil):
        self._responsibilities.remove(responsibility)

    def get_responsibilities(self):
        return self._responsibilities

    def clear_responsibilities(self):
        self._responsibilities.clear()

    def get_id(self):
        return self.__id

    def is_movable(self) -> bool:
        return not self.__immovable

    def cum_dist(self) -> float:
        """
        Compute the cumulative distance between self and all assigned
        responsibilities, weighed by number of Unfaelle per responsibili
        :return:
        cumulative distance
        """
        print("compute cumulative distance")

        sum_ = 0.0
        for responsibility in self._responsibilities:
            sum_ += (self.get_distance(responsibility.get_position())
```

```
                * responsibility.get_unfallquote())
    return sum_
```