



UMWELT VERKEHR

Dokumentation zum Projekt der Abschlussprüfung



WINTER 2020

MOHAMMAD ALO

PRÜFLINGSNUMMER 183-05031

Inhaltsverzeichnis

Aufgabenanalyse und Beschreibung des Verfahrens.....	3
Aufgabenbeschreibung	3
Beschreibung des Verfahrens.....	3
Prinzip.....	4
Zu beachten.....	4
Algorithmus	4
Beschreibung des Algorithmus.....	4
Schnellste Verbindung.....	5
Geringste CO2-Emission	5
Klassendiagramm	6
Anmerkung	6
Klassenbeschreibung	7
Parameter.....	7
IVerbindung	7
OeffentlichenVerbindung.....	7
IndividualVerbindung	7
FlugVerbindung	7
LocationType	7
Location	7
Knoten	7
Kante	7
Graph.....	7
Flug	7
Entfernung.....	7
Datei	7
AlgoDaten.....	8
Prozessablaufpläne.....	9
GetKuerzesterPfad()	9
GetKuerzesterPfadCO2()	10
.....	10
Sequenzdiagramm.....	11
Ausblick	12
Funktionale Trennung	12
Benutzerhandbuch	13

Änderungen	13
Quellcode	14
Progam.cs	14
}	15
Parameter.cs	15
OeffentlichenVerbindung.cs.....	16
LocationType.cs	17
Location.cs	18
Knoten.cs	18
Kanten.cs	19
IVerbindung.cs.....	19
Graph.cs.....	19
FlugVerbindung.cs	22
Flug.cs	22
Entfernung.cs	23
Datei.cs	23
AlgoDaten.cs.....	27
Test	31
Beispiel1	31
Beispiel2	32
Test1	33
Test2	34
Test3	35
Test4	36
Test5	37
Test6	38
Technische Umgebung	39
Eigenständigkeitserklärung	40

Aufgabenanalyse und Beschreibung des Verfahrens

Aufgabenbeschreibung

Herr Tunburg macht sich Gedanken um das Klima der Welt. Deswegen will er den CO₂-Ausstoß reduzieren. Außerdem will die Firma viele Dienstreisen einsparen, indem Besprechungen online durchgeführt werden. Darüber hinaus sollen auch die Verkehrsmittel bewusster gewählt werden. Da Herr Tunburg keine Ahnung hat, wie viel CO₂ auf den jeweiligen Dienstreisen ausgestoßen wird und welche Auswirkungen der Wechsel des Verkehrsmittels auf die Reisezeit hat, müssen wir diese Daten mithilfe einer Software abschätzen.

Beschreibung des Verfahrens

Es soll drei Bewegungsarten geben, die unterschieden werden. Diese drei Fortbewegungsarten werden das Modell vereinfachen.

1. Individualverbindung

Die Individualverbindung kann sich mit allen anderen Punkten verbinden.

Die Individualverbindung setzt sich aus drei Teilen zusammen.

1. Zu Fuß
(max. 1km, 4km/h, 0kg/km)
2. Mit Fahrrad/Auto im Stadtverkehr
(max. 10km, 30km/h, 0,189kg/km)
3. Mit Auto auf der Autobahn
(max. 2000km, 100km/h, 0,189kg/km)

Die maximale Länge einer Individualverbindung ist also $1\text{km} + 10\text{km} + 2000\text{km} = 2011\text{km}$.

Entfernung mit Aufschlag (+20%)

Nur gleicher Kontinent

2. Verbindung mit öffentlichen Verkehrsmitteln

Die öffentlichen Verkehrsmittel sind verbunden mit den Haltstellen und Flughafen und die Verbindungen bestehen aus bis zu zwei Teilen.

- a. Bus/ Straßenbahn
(max. 25km, 30km/h, 0,055kg/km)
- b. Fernbus/ IC(E)
(unbegrenzte Reichweite, 100km/h, 0,055kg/km)

Entfernung mit Aufschlag (+10%)

Nur gleicher Kontinent

3. Flugverbindung

Die Flugverbindung besteht nur zu anderen Flughäfen.

(unbegrenzte Reichweite, 900km/h, 0,2113 kg/km).

Pro Flugverbindung wird eine Wartezeit von 3h berechnet, außer die Flugverbindung besteht im gleichen Land, wofür dann eine Wartezeit von nur 2h berechnet wird.

Pro Zwischenstopp wird die Wartezeit erneut berechnet.

Entfernung mit Aufschlag (+2%).

Die Verbindung zwischen zwei Punkten soll aus maximal fünf Komponenten bestehen, die wie folgt angeordnet sind, ohne die gleiche Verkehrsart mehrfach direkt nacheinander zu benutzen. Im Folgenden ist ein Beispiel für eine mögliche Konstellation aufgeführt.

1. Individualverbindung
2. ÖPNV
3. Flugverkehr
4. ÖPNV
5. Individualverbindung

Da die Individualverbindung und der ÖPNV zweimal genutzt werden, sind sie doppelt aufgeführt.

Prinzip

- 1) Eine Individualverbindung kann zu einem Flughafen, Zielpunkt und ÖPNV führen.
- 2) Eine Verbindung mit einem öffentlichen Verkehrsmittel ist von einem Flughafen oder Haltepunkt zu einem Flughafen, Haltepunkt oder Zielpunkt möglich.
- 3) Eine Flugverbindung.
- 4) Eine weiterführende Verbindung kann von Flughafen zu Zielpunkt oder von Haltepunkt zu Halte-/Zielpunkt sein.
- 5) Weitere Individualverbindung zum Zielpunkt.

Zu beachten

- 1) Es sind zwei Verbindungen mit öffentlichen Verkehrsmitteln (2,4) nötig, wenn eine Flugverbindung dazwischen liegt.
- 2) Es sind zwei Individualverbindungen (1,5) nötig, falls eine Flugverbindung oder eine Verbindung mit öffentlichen Verkehrsmitteln dazwischen liegt.

Algorithmus

Beschreibung des Algorithmus

Da das Problem darin besteht, den schnellsten Weg und den geringsten CO₂ Ausstoß zu berechnen, sieht es nach einem Graphen-Problem aus.

Da wir über alle Wege gehen müssen, um den kleinsten Wert zu berechnen, weil es keine negativen Zeiten/Strecken bzw. CO₂-Emissionen geben kann, bietet sich der Dijkstra-Algorithmus an, um das Problem zu lösen.

Dijkstra ist ein Greedy-Algorithmus, mit dem man den kürzesten/kostengünstigsten Weg suchen kann.

Um den kürzesten Weg zu berechnen, erstellt man zuerst den Startknoten mit dem Gewicht null ein.

Zuerst betrachtet man alle Kanten, die von dem Startknoten abgehen, und addiert die Gewichte der Kanten zu dem Gewicht des nächsten Knotens hinzu. Anschließend wird der vorherige Knoten als besucht markiert. Der Knoten mit dem geringsten Gewicht wird ausgewählt und der Vorgang wird wiederholt, solange bis alle Knoten besucht wurden.

Nachdem alle Knoten besucht wurden, geht man den Rückweg vom Zielpunkt zum Startpunkt.

Dasselbe Prinzip wird verwendet, um die Aufgabe zu bearbeiten und die schnellsten Verbindungen mit dem geringsten CO₂-Ausstoß zu berechnen. Der Algorithmus wird zweimal durchlaufen:

Einmal, um den schnellsten Weg zu finden und das zweite Mal, um den Weg mit dem geringsten CO₂-Ausstoß zu finden.

Schnellste Verbindung

Um die schnellste Verbindung mit Hilfe des Dijkstra-Algorithmus zu suchen, muss das Problem zuerst in einen Graphen überführt werden. Ein Graph besteht aus Knoten und Kanten.

Knoten sind Orte und Kanten die Verbindungen zwischen jeweils zwei Knoten (Orten). Jede Kante enthält mehrere Informationen (Co₂, Dauer, Distanz, km/h, Start- und Zielknoten). Jeder Knoten enthält die folgenden Informationen (ID, Latitude, Longitude, Kontinent und der Name des Ortes).

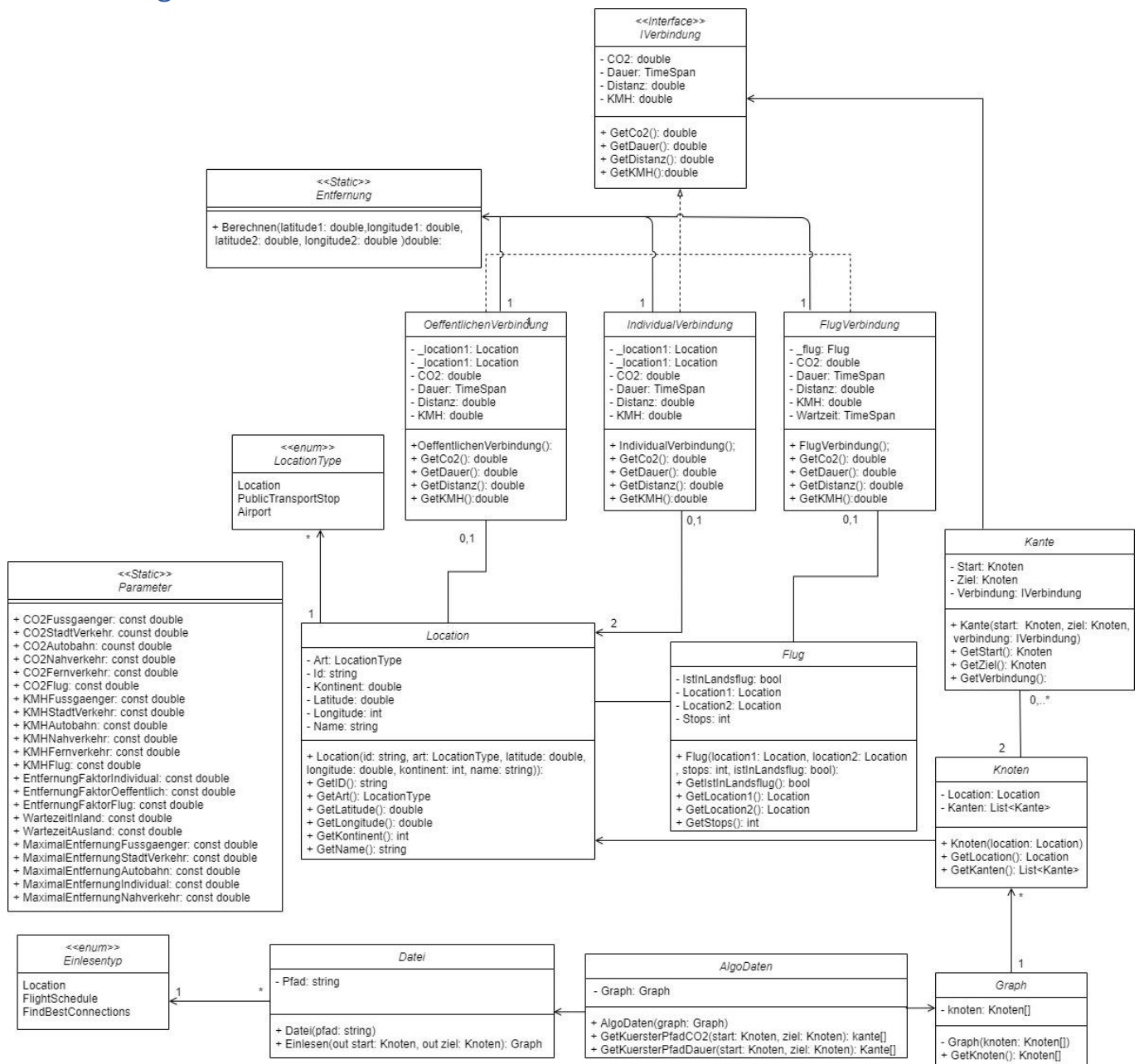
Da es in der Aufgabestellung mehrere Wege zwischen zwei Knoten gibt (Individualverbindung, öffentliche Verkehrsmittel, Flugverbindung), erstellt man eine Liste von Kanten alle möglichen Verbindungen.

Nachdem man alle Kanten hat, können Start- und Zielpunkt an den Algorithmus übergeben werden, um den schnellsten Weg oder den geringsten CO₂-Ausstoß zu finden.

Geringste CO₂-Emission

Um die Verbindung mit der geringsten CO₂-Emission zu finden, verwendet man dasselbe Prinzip wie zuvor, wobei man die Gewichtung der Kanten nicht mehr auf die Dauer, sondern auf die Emissionswerte der Verbindung setzt.

Klassendiagramm



Anmerkung

Die Parameter-Klasse wird in allen anderen Klassen verwendet. Um das Diagramm übersichtlicher zu gestalten, habe ich keine Verbindungen von dieser Klasse aus gemacht.

Klassenbeschreibung

Parameter

Die Klasse **Parameter** enthält alle Daten, die in der Aufgabenstellung gegeben sind, als Konstanten.

IVerbindung

IVerbindung ist eine Schnittstelle für alle Verbindungen (Individual-, Flug- und öffentliche Verbindung), weil alle Verbindungen ähnliche Information haben.

OeffentlichenVerbindung

Die Klasse **OeffentlichenVerbindung** enthält alle nötigen Informationen über die öffentliche Verbindung von zwei Orten.

IndividualVerbindung

Die Klasse **IndividualVerbindung** enthält alle nötigen Informationen über die Individualverbindung

FlugVerbindung

Die Klasse **FlugVerbindung** enthält alle nötigen Informationen über die Flugverbindung von zwei Orten.

LocationType

Die Enumeration **LocationType** enthält die drei Arten von Orten.

Location

Mit der Klasse **Location** werden die Informationen über einen Ort gespeichert.

Knoten

Mit der Klasse **Knoten** wird ein Ort mit all seinen Kanten (Verbindungen) gespeichert.

Kante

Eine Kante enthält einen Startknoten und einen Endknoten mit den entsprechenden Verbindungsinformationen.

Graph

Die Klasse **Graph** enthält alle gegebenen Knoten.

Flug

Jeder Flug enthält Informationen darüber, ob es sich um einen Inland- oder Auslandsflug handelt, die Anzahl der Stopps sowie Start- und Zielflughafen.

Entfernung

Die Klasse **Entfernung** ist eine statische Klasse, in der die Entfernung mithilfe des Kosinussatz berechnet wird.

Datei

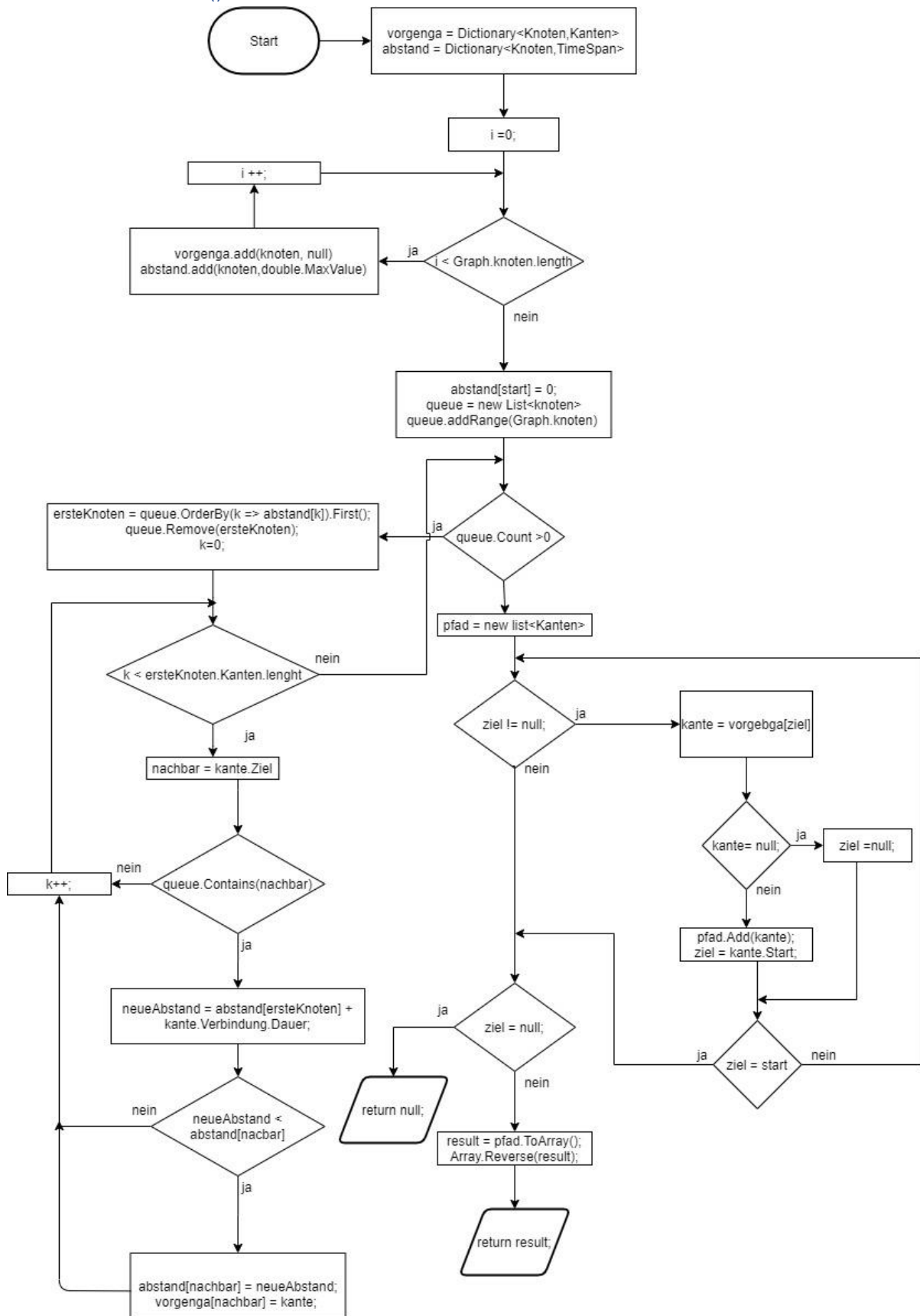
Die Klasse **Datei** ist dafür zuständig, die Daten aus einer Textdatei einzulesen und daraus einen Graphen zu bilden.

AlgoDaten

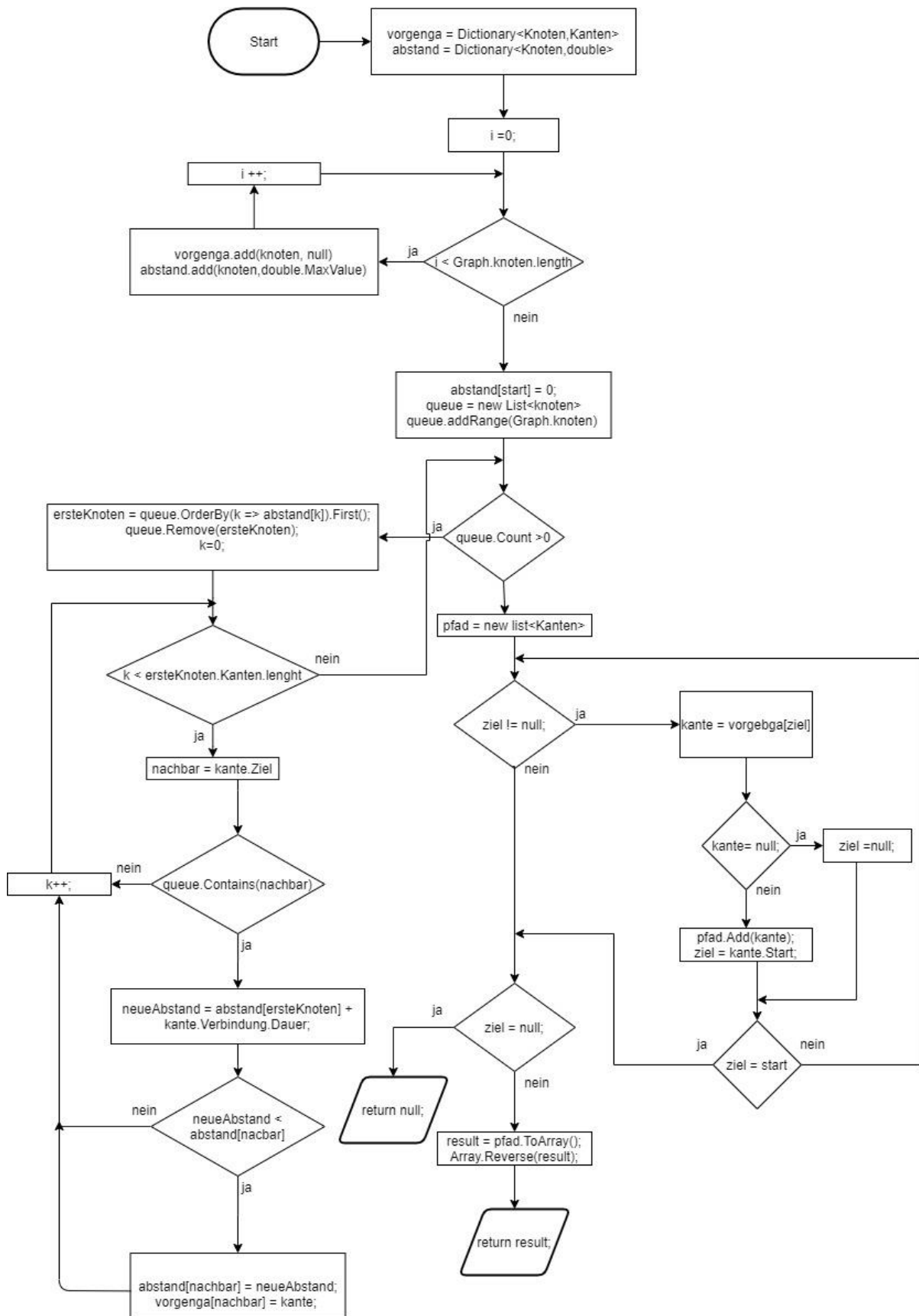
In dieser Klasse wird der aufgestellt Graph gehalten und kann mittels Dijkstra-Algorithmus den schnellsten Weg und den mit dem geringsten CO2-Ausstoß bestimmen.

Prozessablaufpläne

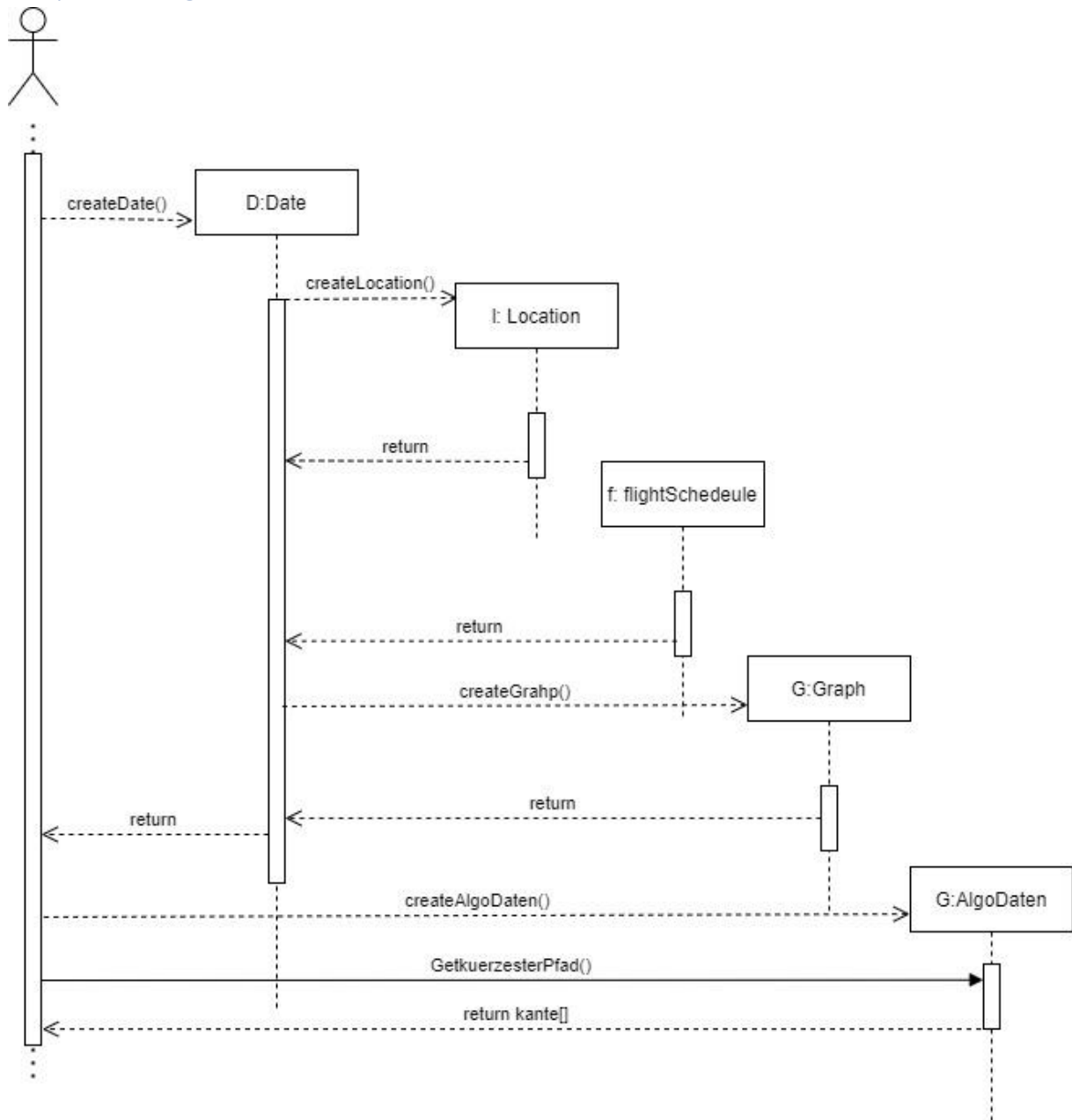
GetKuerzesterPfad()



GetKuerzesterPfadCO2()



Sequenzdiagramm



Ausblick

Das Programm liefert bisher nur eine, nämlich die beste Verbindung für den schnellsten Weg und den geringsten CO₂-Ausstoß. Um mehr als den besten Pfad zu finden, müsste sich das Programm mehr als einen Vorgänger pro Knoten merken.

Die Methoden `GetKuerzesterPfadCO2` und `GetKuerzesterPfadDauer` in der Klasse `AlgoDaten` könnten zu einer Methode zusammengefasst werden, da die Methodendefinition nahezu identisch ist.

Des Weiteren könnten die Kantengewichtungen soweit verallgemeinert werden, dass auch beispielweise Fahrkosten betrachtet werden können.

Funktionale Trennung

Um die funktionale Trennung in meinem Programm zu bewahren, bin ich nach dem Modell-View-Controller Schema vorgegangen. Die Anwendungslogik ist also getrennt von der View und der Controller kümmert sich um die Kommunikation zwischen beiden Schichten. Dies ermöglicht es flexibel zu bleiben und vereinfacht zukünftige Veränderungen an Komponenten.

Benutzerhandbuch

In folgenden Schritten kann man das Programm benutzen:

1. Programmordner anklicken
2. Eine Textdatei mit den Input-Daten in den Ordner legen
3. Anschließend „Abschlussprüfung_Umwelt_verkehr.exe“ starten
4. Die Lösungen werden als Textdatei in dem Ausgabeordner gespeichert
5. Jede Ausgabe-Datei ist genauso benannt wie die dazugehörige Eingabedatei.

Änderungen

In meiner ursprünglichen Planung habe ich mich auf den Dijkstra-Algorithmus im Allgemeinen bezogen. Während der Lösungsentwicklung habe ich meine Planung jedoch verfeinert und den Algorithmus an die Problemstellung angepasst. Das Programm führt die Methoden [GetKuerzesterPfadDauer](#) und [GetKuerzesterPfadCo2](#) nacheinander aus.

Da ich mich am Montag hauptsächlich auf den Algorithmus und die Datenstrukturen konzentriert habe, sind zu meiner ursprünglichen Planung noch folgende Klassen hinzugekommen. Dazu zählen Kante, Knoten, Parameter und weitere.

Quellcode

Progam.cs

```
using System.IO;
using System;

namespace Abschlussprüfung_Umwelt_Verkehr
{
    class Program
    {
        static void Main(string[] args)
        {
            string currentPath = Directory.GetCurrentDirectory();
            currentPath = Environment.CurrentDirectory;
            string testFolderPath = currentPath + @"\Daten\";
            string testAusgabePath = currentPath + @"\Ausgabe\";
            Directory.CreateDirectory(testAusgabePath);

            string[] filePaths = Directory.GetFiles(testFolderPath, "*.txt",
                SearchOption.AllDirectories);

            for (int i = 0; i < filePaths.Length; i++)
            {
                try
                {
                    string testflie = filePaths[i];
                    Datei datei = new Datei(testflie);

                    Graph graph = datei.Einlesen(out Knoten start, out Knoten ziel);
                    AlgoDaten algodaten = new AlgoDaten(graph);

                    var dauerPfad = algodaten.GetKuersterPfadDauer(start, ziel);
                    var dauer = new TimeSpan(0);

                    var co2Pfad = algodaten.GetKuerzesterPfadCO2(start, ziel);
                    double co2 = 0;

                    string fileName = testAusgabePath +
Path.GetFileName(filePaths[i]);
                    using (StreamWriter sw = File.CreateText(fileName))
                    {
                        sw.WriteLine(dauerPfad[0].Start.Location.Name + "-->" +
dauerPfad[dauerPfad.Length - 1].Ziel.Location.Name);
                        sw.WriteLine("-----");
                        sw.WriteLine("Strecke Luftline: " +
Entfernung.Berechnen(start.Location.Latitude, start.Location.Longitude,
                        ziel.Location.Latitude, ziel.Location.Longitude));
                        sw.WriteLine("");
                        sw.WriteLine("");
                        sw.WriteLine("Schnellsete Verbindungen");
                        foreach (Kante kante in dauerPfad)
                        {
                            dauer += kante.Verbindung.Dauer;
                            co2 += kante.Verbindung.CO2;

                            sw.WriteLine(kante.Start.Location.Name + "-- " +
kante.Verbindung.GetType().Name + "-->" + kante.Ziel.Location.Name);
                        }
                        sw.WriteLine("Dauer: " + dauer + ", CO2-Emission: " + co2);
                    }
                }
            }
        }
    }
}
```



```

        public const double MaximalEntfernungNahverkehr = 25;
    }
}

```

OeffentlichenVerbindung.cs

```

using System;

namespace Abschlussprüfung_Umwelt_Verkehr
{
    class OeffentlichenVerbindung : IVerbindung
    {
        private Location _location1;
        private Location _location2;

        public double Distanz => Parameter.EntfernungFaktorOeffentlich
* Entfernung.Berechne(_location1.Latitude, _location1.Longitude, _locat
ion2.Latitude, _location2.Longitude);

        public double KMH
        {
            get
            {
                double kmh = 0;

                double distanz = Distanz;
                double d = distanz;
                if (distanz > 0) // Nah Verkher
                {
                    double distanzGehen = Math.Min(Parameter.MaximalEnt
fernungNahverkehr, distanz); // maximal 25km gehen
                    kmh += Parameter.KMHNahverkehr * distanzGehen;

                    distanz -= distanzGehen;
                }
                // Fern Verkehr
                kmh += Parameter.KMHFernverkehr * distanz;
                return kmh / d;
            }
        }

        public TimeSpan Dauer
        {
            get
            {
                double dauer = 0;

                double distanz = Distanz;
                if (distanz > 0) // Nah Verkher
                {
                    double distanzGehen = Math.Min(Parameter.MaximalEnt
fernungNahverkehr, distanz); // maximal 25km gehen
                    dauer += distanzGehen / Parameter.KMHNahverkehr;
                }
            }
        }
    }
}

```

```

        distanz -= distanzGehen;
    }
    // Fern Verkehr
    dauer += distanz / Parameter.KMHFernverkehr;
    return TimeSpan.FromHours(dauer);
}

public double CO2
{
    get
    {
        double co2 = 0;

        double distanz = Distanz;
        if (distanz > 0) // Nah Verkehr
        {
            double distanzGehen = Math.Min(Parameter.MaximalEntfernungNahverkehr, distanz); // maximal 25km gehen
            co2 += Parameter.CO2Nahverkehr * distanzGehen;

            distanz -= distanzGehen;
        }
        // Fern Verkehr
        co2 += Parameter.CO2Fernverkehr * distanz;
        return co2;
    }
}

public OeffentlichenVerbindung(Location location1, Location location2)
{
    _location1 = location1;
    _location2 = location2;
}
}

```

LocationType.cs

```

namespace Abschlussprüfung_Umwelt_Verkehr
{
    internal enum LocationType
    {
        Location,
        PublicTransportStop,
        Airport
    }
}

```

Location.cs

```
namespace Abschlussprüfung_Umwelt_Verkehr
{
    internal class Location
    {
        public string Id { get; }
        public LocationType Art { get; }
        public double Latitude { get; }
        public double Longitude { get; }
        public int Kontinent { get; }
        public string Name { get; }

        public Location(string id, LocationType art, double latitude, double longitude, int kontinent, string name)
        {
            Id = id;
            Art = art;
            Latitude = latitude;
            Longitude = longitude;
            Kontinent = kontinent;
            Name = name;
        }
    }
}
```

Knoten.cs

```
using System.Collections.Generic;

namespace Abschlussprüfung_Umwelt_Verkehr
{
    internal class Knoten
    {
        public Location Location { get; }
        public List<Kante> Kanten { get; } = new List<Kante>();

        public Knoten(Location location)
        {
            Location = location;
        }
    }
}
```

Kanten.cs

```
namespace Abschlussprüfung_Umwelt_Verkehr
{
    internal class Kante
    {
        public Knoten Start { get; }
        public Knoten Ziel { get; }

        public IVerbindung Verbindung { get; }

        public Kante(Knoten start, Knoten ziel, IVerbindung verbindung)
        {
            Start = start;
            Ziel = ziel;
            Verbindung = verbindung;
        }
    }
}
```

IVerbindung.cs

```
using System;

namespace Abschlussprüfung_Umwelt_Verkehr
{
    interface IVerbindung
    {
        double Distanz { get; }
        double KMH { get; }
        TimeSpan Dauer { get; }
        double CO2 { get; }
    }
}
```

Graph.cs

```
using System;

namespace Abschlussprüfung_Umwelt_Verkehr
{
    class IndividualVerbindung : IVerbindung
    {
        private readonly Location _location1;
        private readonly Location _location2;
        public double Distanz => Parameter.EntfernungFaktorIndividual *
Entfernung.Berechne(_location1.Latitude, _location1.Longitude, _location2.Latitude, _location2.Longitude);

        public double KMH
        {
            get
            {
                return Distanz / Duration;
            }
        }
    }
}
```

```

    {
        double kmh = 0;

        double distanz = Distanz;
        double d = distanz;
        if (distanz > 0) // gehen
        {
            double distanzGehen = Math.Min(Parameter.MaximalEntfernungFussgaenger, distanz); // maximal 1km gehen
            kmh += Parameter.KMHFussgaenger * distanzGehen;

            distanz -= distanzGehen;
        }

        if (distanz > 0) // Stadtverkehr
        {
            double distanzStadt = Math.Min(Parameter.MaximalEntfernungStadtVerkehr, distanz); // maximal 10km in stadt
            kmh += Parameter.KMHStadtVerkehr * distanzStadt;

            distanz -= distanzStadt;
        }

        // rest mit Autobahn
        kmh += Parameter.KMHAutobahn * distanz;
        return kmh / d;
    }
}

public TimeSpan Dauer
{
    get
    {
        double dauer = 0;
        double distanz = Distanz;
        // gehen = 0co2
        if (distanz > 0)
        {
            double distanzGehen = Math.Min(Parameter.MaximalEntfernungFussgaenger, distanz); // maximal 1km gehen
            dauer += distanzGehen / Parameter.KMHFussgaenger;
            distanz -= distanzGehen;
        }

        if (distanz > 0) // Stadtverkehr
        {
            double distanzStadt = Math.Min(Parameter.MaximalEntfernungStadtVerkehr, distanz); // maximal 10km in stadt
            dauer += distanzStadt / Parameter.KMHStadtVerkehr;

            distanz -= distanzStadt;
        }

        // rest mit Autobahn
        dauer += distanz / Parameter.KMHAutobahn;
    }
}

```

```

        return TimeSpan.FromHours(dauer);
    }
}
public double CO2
{
    get
    {
        double co2 = 0;
        double distanz = Distanz;
        // gehen = 0co2
        if (distanz > 0)
        {
            double distanzGehen = Math.Min(Parameter.MaximalEntfernungFussgaenger, distanz); // maximal 1km gehen
            co2 += Parameter.CO2Fussgaenger * distanzGehen;
            distanz -= distanzGehen;

            if (distanz > 0) // Stadtverkehr
            {
                double distanzStadt = Math.Min(Parameter.MaximalEntfernungStadtVerkehr, distanz); // maximal 10km in stadt
                co2 += Parameter.CO2StadtVerkehr * distanzStadt;

                distanz -= distanzStadt;
            }
            // rest mit Autobahn
            co2 += Parameter.CO2Autobahn * distanz;

            return co2;
        }
    }
}
public IndividualVerbindung(Location location1, Location location2)
{
    _location1 = location1;
    _location2 = location2;
}
}

```

FlugVerbindung.cs

```
using System;

namespace Abschlussprüfung_Umwelt_Verkehr
{
    class FlugVerbindung : IVerbindung
    {
        private readonly Flug _flug;

        public double Distanz => Parameter.EntfernungFaktorFlug * Entfernung.Berechne(_flug.Location1.Latitude, _flug.Location1.Longitude, _flug.Location2.Latitude, _flug.Location2.Longitude);

        public double KMH => Parameter.KMHFlug;

        public TimeSpan Dauer => TimeSpan.FromHours(Distanz / KMH + Wartezeit * (1 + _flug.Stops));
        private double Wartezeit => _flug.IstInlandsflug ? Parameter.WartezeitInland : Parameter.WartezeitAusland;

        public double CO2 => Parameter.CO2Flug * Distanz;

        public FlugVerbindung(Flug flug)
        {
            _flug = flug;
        }
    }
}
```

Flug.cs

```
namespace Abschlussprüfung_Umwelt_Verkehr
{
    internal class Flug
    {
        public Location Location1 { get; }
        public Location Location2 { get; }
        public int Stops { get; }
        public bool IstInlandsflug { get; }

        public Flug(Location location1, Location location2, int stops, bool istInlandsflug)
        {
            Location1 = location1;
            Location2 = location2;
            Stops = stops;
            IstInlandsflug = istInlandsflug;
        }
    }
}
```

Entfernung.cs

```
using static System.Math;

namespace Abschlussprüfung_Umwelt_Verkehr
{
    static class Entfernung
    {
        /// <summary>
        /// Die Berechnung Methode berechnet Luftlinienstrecke mit Hilfe
        /// der Seitenkosinussatz funktion
        /// </summary>
        /// <param name="latitude1">Gib das erste latitude ein</param>
        /// <param name="longitude1">Gib das erste longitude ein</param>
        >
        /// <param name="latitude2">Gib die zweite latitude ein</param>
        /// <param name="longitude2">Gib die zweite longitude ein</para
        m>
        /// <returns> Gibt die Entfernung zurück</returns>
        public static double Berechne(double latitude1, double longitude1, double latitude2, double longitude2)
        {
            latitude1 = latitude1 / 180 * PI;
            longitude1 = longitude1 / 180 * PI;

            latitude2 = latitude2 / 180 * PI;
            longitude2 = longitude2 / 180 * PI;

            double erdeRadius = 6378.388;
            return erdeRadius * Acos(Sin(latitude1) * Sin(latitude2) +
            Cos(latitude1) * Cos(latitude2) * Cos(longitude2 - longitude1));
        }
    }
}
```

Datei.cs

```
using System.Linq;
using System.IO;
using System.Collections.Generic;

namespace Abschlussprüfung_Umwelt_Verkehr
{
    internal class Datei
    {
        private enum EinleseTyp
        {
            Locations, FlightSchedule, FindBestConnections
        }
        public string Pfad { get; }

        public Datei(string pfad)
```



```

    {
        Pfad = pfad;
    }
    /// <summary>
    /// Hier wird die Daten eingelesen um eine Graph zu bauen
    /// </summary>
    /// <param name="start">gib das start knoten ein (Ort)</param>
    /// <param name="ziel">gib das ziel knoten ein (Ort)</param>
    /// <returns>gibt eine Graph aus mit Start und ziel knoten ein<
/returns>
    public Graph Einlesen(out Knoten start, out Knoten ziel)
    {
        start = null;
        ziel = null;
        Location startLocation = null;
        Location ziellocation = null;

        List<Location> locations = new List<Location>();
        List<Flug> fluge = new List<Flug>();
        string[] text = File.ReadAllLines(Pfad);
        EinleseTyp einleseType = EinleseTyp.Locations;

        foreach (var line in text)
        {
            if (line.StartsWith("#") || line.Length == 0)
            {
                continue;
            }
            if (line == "Locations:")
            {
                einleseType = EinleseTyp.Locations;
                continue;
                // Locations lesen
            }
            else if (line == "FlightSchedule:")
            {
                einleseType = EinleseTyp.FlightSchedule;
                continue;

                // FlightSchedule: Lesen
            }
            else if (line == "FindBestConnections:")
            {
                einleseType = EinleseTyp.FindBestConnections;
                continue;
                // FindBestConnections: lesen
            }

            if (einleseType == EinleseTyp.Locations)
            {
                string[] lineSplit = line.Split(';');
                string id = lineSplit[0];
                string kindOfLocation = lineSplit[1].Trim();
            }
        }
    }

```

```

double latitzde = double.Parse(lineSplit[2].Replace
('.', ','));
double longitude = double.Parse(lineSplit[3].Replac
e('.', ','));

int continent = int.Parse(lineSplit[4]);
string name = lineSplit[5].Trim();

LocationType art = LocationType.Location;
if (kindOfLocation == "PublicTransportStop")
{
    art = LocationType.PublicTransportStop;
}
else if (kindOfLocation == "Airport")
{
    art = LocationType.Airport;
}

else if (kindOfLocation == "Location")
{
    art = LocationType.Location;
}

Location location = new Location(id, art, latitzde,
longitude, continent, name);
locations.Add(location);

}
else if (einleseType == EinleseTyp.FlightSchedule)
{
    string[] lineSplit = line.Split(';');
    string id1 = lineSplit[0].Trim();
    string id2 = lineSplit[1].Trim();
    int stops = 0;
    bool domesticFlifht = false;

    if (lineSplit.Length > 2 && !string.IsNullOrEmptySp
ace(lineSplit[2]))
    {
        stops = int.Parse(lineSplit[2]);
    }
    if (lineSplit.Length > 3 && !string.IsNullOrEmptySp
ace(lineSplit[3]))
    {
        domesticFlifht = bool.Parse(lineSplit[3]);
    }

    Location location1 = locations.First(l => l.Id == i
d1);
    Location location2 = locations.First(l => l.Id == i
d2);

```

```

        Flug flug = new Flug(location1, location2, stops, domesticFlight);
        fluge.Add(flug);
    }
    else if (einleseType == EinleseTyp.FindBestConnections)
    {
        string[] lineSplit = line.Split(';');
        string id1 = lineSplit[0].Trim();
        string id2 = lineSplit[1].Trim();
        startLocation = locations.First(l => l.Id == id1);
        zielLocation = locations.First(l => l.Id == id2);
    }
}
Dictionary<Location, Knoten> knoten = new Dictionary<Location, Knoten>();
foreach (var loction in locations)
{
    knoten.Add(loction, new Knoten(loction));
}
start = knoten[startLocation];
ziel = knoten[zielLocation];
foreach (var flug in fluge)
{
    Knoten knoten1 = knoten[flug.Location1];
    Knoten knoten2 = knoten[flug.Location2];
    FlugVerbindung flugVerbindung = new FlugVerbindung(flug);

    Kante k1 = new Kante(knoten1, knoten2, flugVerbindung);
    Kante k2 = new Kante(knoten2, knoten1, flugVerbindung);
    knoten1.Kanten.Add(k1);
    knoten2.Kanten.Add(k2);
}
foreach (var knoten1 in knoten.Values)
{
    foreach (var knoten2 in knoten.Values)
    {
        if (knoten1 != knoten2)
        {
            IndividualVerbindung individualVerbindung = new IndividualVerbindung(knoten1.Location, knoten2.Location);
            if (individualVerbindung.Distanz <= Parameter.MaximalEntfernungIndividual
                && knoten1.Location.Kontinent == knoten2.Location.Kontinent)
            {
                Kante k1 = new Kante(knoten1, knoten2, individualVerbindung);
                knoten1.Kanten.Add(k1);
            }
            OeffentlichenVerbindung oeffentlichenVerbindung = new OeffentlichenVerbindung(knoten1.Location, knoten2.Location);

```

```

        if (knoten1.Location.Art != LocationType.Locati
on && knoten2.Location.Art != LocationType.Location
        && knoten1.Location.Kontinent == knoten2.Lo
cation.Kontinent)
        {
            Kante k1 = new Kante(knoten1, knoten2, oeff
entlichenVerbindung);
            knoten1.Kanten.Add(k1);
        }
    }
}
return new Graph(knoten.Values.ToArray());
}
}
}

```

AlgoDaten.cs

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace Abschlussprüfung_Umwelt_Verkehr
{
    internal class AlgoDaten
    {
        public Graph Graph { get; }

        public AlgoDaten(Graph graph)
        {
            Graph = graph;
        }

        /// <summary>
        /// Die folgende Methode wird die Dijkstra verwendet um die ger
ingste CO2 berechnen
        /// </summary>
        /// <param name="start"> Gib das Start Knoten ein (Ort)</param>
        /// <param name="ziel"> Gib das Ziel Knoten ein (Ort)</param>
        /// <returns>gibt eine array von kanten aus (Pfad)</returns>
        public Kante[] GetKuersterPfadCO2(Knoten start, Knoten ziel)
        {
            Dictionary<Knoten, Kante> vorgenga = new Dictionary<Knoten,
Kante>();
            Dictionary<Knoten, double> abstand = new Dictionary<Knoten,
double>();

            foreach (var knoten in Graph.Knoten)
            {
                vorgenga.Add(knoten, null);
                abstand.Add(knoten, double.MaxValue);
            }
        }
    }
}

```

```

        abstand[start] = 0;
        List<Knoten> queue = new List<Knoten>();
        queue.AddRange(Graph.Knoten);

        while (queue.Count > 0)
        {
            Knoten ersteKnoten = queue.OrderBy(k => abstand[k]).First();
            queue.Remove(ersteKnoten);

            foreach (var kante in ersteKnoten.Kanten)
            {
                Knoten nachbar = kante.Ziel;
                if (queue.Contains(nachbar))
                {
                    double neueAbstand = abstand[ersteKnoten] + kante.Verbindung.CO2;
                    if (neueAbstand < abstand[nachbar])
                    {
                        abstand[nachbar] = neueAbstand;
                        vorgenga[nachbar] = kante;
                    }
                }
            }
        }
        List<Kante> pfad = new List<Kante>();
        while (ziel != null)
        {
            Kante kante = vorgenga[ziel];
            if (kante == null)
            {
                ziel = null;
            }
            else
            {
                pfad.Add(kante);
                ziel = kante.Start;
            }

            if (ziel == start)
            {
                break;
            }
        }
        if (ziel == null)
        {
            // es gibt kein weg von Start zum Ziel
            return null;
        }
        else
        {
            Kante[] result = pfad.ToArray();
            Array.Reverse(result);
        }
    }
}

```

```

        return result;
    }
}
/// <summary>
/// Die folgende Methode wird die Dijkstra verwendet um die schnellste
weg(Dauer) berechnen
/// </summary>
/// <param name="start"> Gib das Start Knoten ein (Ort)</param>
/// <param name="ziel"> Gib das Ziel Knoten ein (Ort)</param>
/// <returns>gibt eine array von kanten aus (Pfad)</returns>
public Kante[] GetKuersterPfadDauer(Knoten start, Knoten ziel)
{
    Dictionary<Knoten, Kante> vorgenga = new Dictionary<Knoten,
Kante>();
    Dictionary<Knoten, TimeSpan> abstand = new Dictionary<Knoten, TimeSpan>();

    foreach (var knoten in Graph.Knoten)
    {
        vorgenga.Add(knoten, null);
        abstand.Add(knoten, TimeSpan.MaxValue);
    }
    abstand[start] = new TimeSpan(0);
    List<Knoten> queue = new List<Knoten>();
    queue.AddRange(Graph.Knoten);

    while (queue.Count > 0)
    {
        Knoten ersteKnoten = queue.OrderBy(k => abstand[k]).First();
        queue.Remove(ersteKnoten);

        foreach (var kante in ersteKnoten.Kanten)
        {
            Knoten nachbar = kante.Ziel;
            if (queue.Contains(nachbar))
            {
                TimeSpan neueAbstand = abstand[ersteKnoten] + k
ante.Verbindung.Dauer;
                if (neueAbstand < abstand[nachbar])
                {
                    abstand[nachbar] = neueAbstand;
                    vorgenga[nachbar] = kante;
                }
            }
        }
    }
    List<Kante> pfad = new List<Kante>();
    while (ziel != null)
    {
        Kante kante = vorgenga[ziel];
        if (kante == null)
        {

```

```

        ziel = null;
    }
    else
    {
        pfad.Add(kante);
        ziel = kante.Start;
    }

    if (ziel == start)
    {
        break;
    }
}
if (ziel == null)
{
    // es gibt kein weg von Start zum Ziel
    return null;
}
else
{
    Kante[] result = pfad.ToArray();
    Array.Reverse(result);
    return result;
}
}
}

```

Test

Beispiel1

Input

```
Beispiel1.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe

Locations:
#id;          kind of location;      latit; longitude;    continent;    name;
csu_zen;      Location;                      48.176971;    11.5895754;   1;            CSU-Zentrale, Muenchen
mum_hbf;      PublicTransportStop;            48.140235;    11.559417;   1;            Muenchen Hbf
mum_flug;     Airport;                      48.35333;     11.770723;   1;            Muenchen Flughafen
reichstag;    Location;                      52.518191;    13.3751725;   1;            Reichstagsgebaeude, Berlin
ber_flug;     Airport;                      52.553625;    13.2901544;   1;            Berlin Flughafen Tegel
ber_hbf;      PublicTransportStop;            52.524195;    13.3693013;   1;            Berlin Hbf

FlightSchedule:
#id1;         id2;          (stops;    domestic Flifht)
mum_flug;     ber_flug;      0;         True

FindBestConnections:
csu_zen;      reichstag
```

Output

```
Beispiel1.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe
CSU-Zentrale, Muenchen-->Reichstagsgebaeude, Berlin
-----
Strecke Luftline: 499,608820675903

Schnellste Verbindungen
CSU-Zentrale, Muenchen-- IndividualVerbindung-- > Muenchen Flughafen
Muenchen Flughafen-- FlugVerbindung-- > Berlin Flughafen Tegel
Berlin Flughafen Tegel-- OeffentlichenVerbindung-- > Berlin Hbf
Berlin Hbf-- IndividualVerbindung-- > Reichstagsgebaeude, Berlin
Dauer: 03:45:58.1520000, CO2-Emission: 108,995786561707

Verbindungen mit der gringsten CO2-Emission:
CSU-Zentrale, Muenchen-- IndividualVerbindung-- > Muenchen Hbf
Muenchen Hbf-- OeffentlichenVerbindung-- > Berlin Hbf
Berlin Hbf-- IndividualVerbindung-- > Reichstagsgebaeude, Berlin
Dauer: 06:46:15.8670000, CO2-Emission: 31,400314664451
```

Erwartung:

Die Ausgabe gibt die richtige Lösung aus.

Beispiel2

Input

```
Beispiel2.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe

Locations:
#id;          kind of location;          latit;         longitude;      continent;     name;
ka_kit;       Location;                  49.0116423;    8.4078883;    1;            KIT Karlsruhe
ka_hbf;       PublicTransportStop;         48.9939436;    8.3987803;    1;            Karlsruhe Hbf
fra_flug;     Airport;                     50.0529421;    8.5688149;    1;            Frankfurt Flughafen
s_flug;       Airport;                     48.6910584;    9.1903943;    1;            Stuttgart Flughafen
sanfra_airpot; Airport;                     37.616435;     -122.388382;   2;            San Francisco Airpot
google_hp;    PublicTransportStop;                 37.4219999;    -122.0856089;  2;            Google Headquarter

FlightSchedule:
#id1;         id2;          (stops;        domestic Flight)
fra_flug;     sanfra_airpot
s_flug;       sanfra_airpot;  1

FindBestConnections:
ka_kit;       google_hp
```

Output

```
Beispiel2.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe

KIT Karlsruhe-->Google Headquarter
-----
Strecke Luftline: 9249,02871401207

Schnellste Verbindungen
KIT Karlsruhe-- IndividualVerbindung-- > Frankfurt Flughafen
Frankfurt Flughafen-- FlugVerbindung-- > San Francisco Airpot
San Francisco Airpot-- IndividualVerbindung-- > Google Headquarter
Dauer: 16:08:12.8990000, CO2-Emission: 2007,70417596625

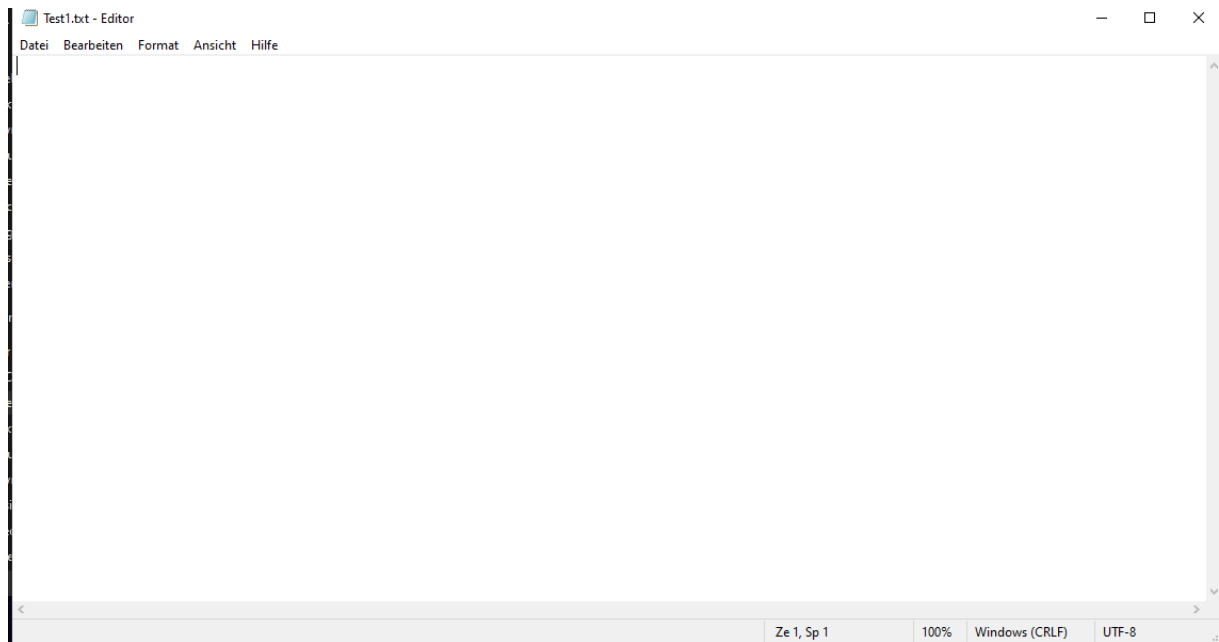
Verbindungen mit der gringsten CO2-Emission:
KIT Karlsruhe-- IndividualVerbindung-- > Karlsruhe Hbf
Karlsruhe Hbf-- OeffentlichenVerbindung-- > Frankfurt Flughafen
Frankfurt Flughafen-- FlugVerbindung-- > San Francisco Airpot
San Francisco Airpot-- OeffentlichenVerbindung-- > Google Headquarter
Dauer: 16:31:41.3880000, CO2-Emission: 1983,39263939682
```

Erwartung:

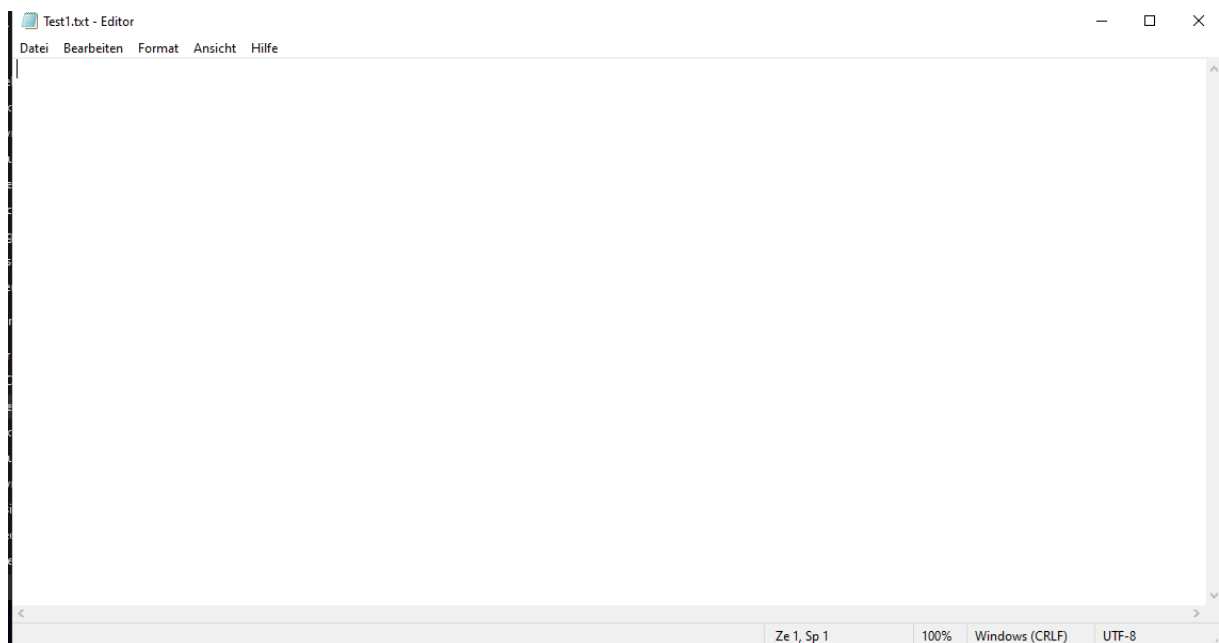
Die Ausgabe gibt die richtige Lösung aus.

Test1

Input



Output

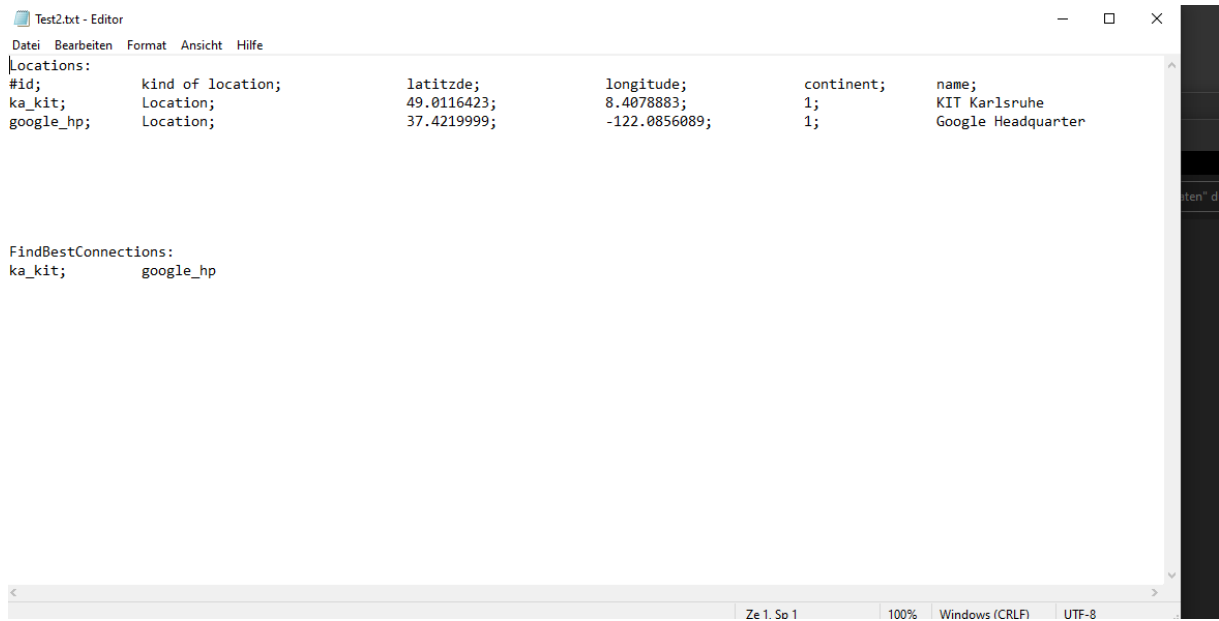


Erwartung:

Bei leerer Eingabe wird eine leere Ausgabe Datei erzeugt.

Test2

Input

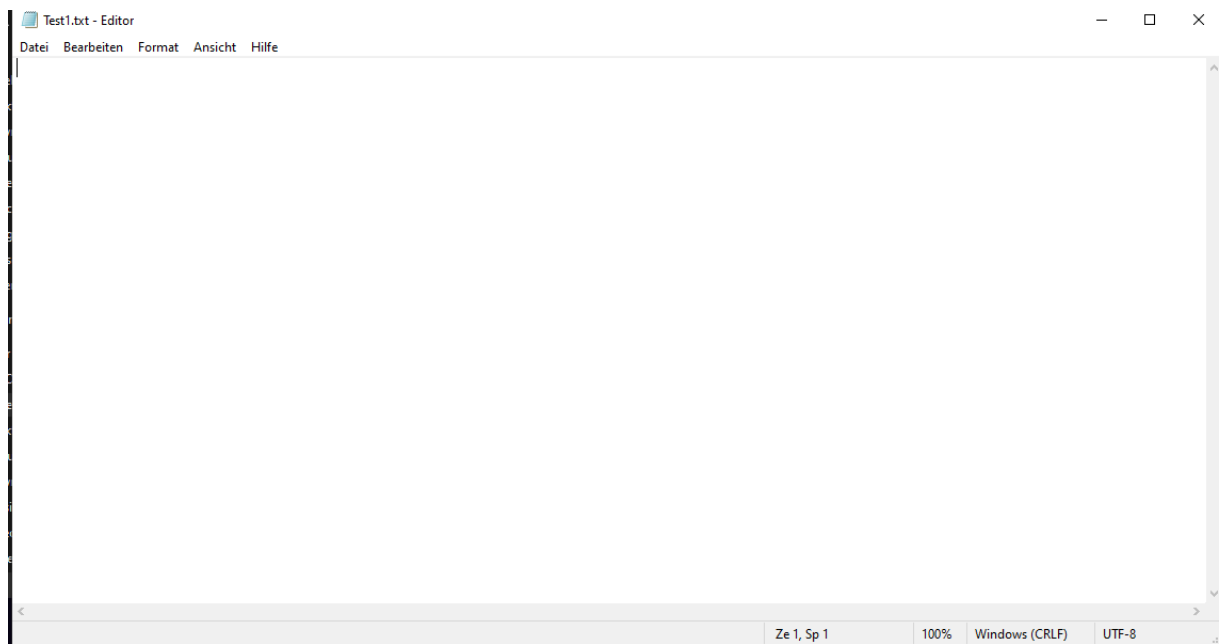


```
Test2.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe

Locations:
#id;          kind of location;    latitzde;    longitude;    continent;    name;
ka_kit;       Location;                49.0116423;  8.4078883;   1;           KIT Karlsruhe
google_hp;    Location;                37.4219999;  -122.0856089; 1;           Google Headquarter

FindBestConnections:
ka_kit;       google_hp
```

Output



```
Test1.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe


```

Erwartung:

Da die Entfernung länger als 2011 km ist, gibt es keine Lösung.

Test3

Input

```
Test3.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe

Locations:
#id;          kind of location;    latit;    longitude;    continent;    name;
csu_zen;      Location;                        48.176971; 11.5895754;   1;            CSU-Zentrale, Muenchen
reichstag;    Location;                        52.518191; 13.3751725;   1;            Reichstagsgebaeude, Berlin

FindBestConnections:
csu_zen;      reichstag
```

Output

```
Test3.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe

CSU-Zentrale, Muenchen-->Reichstagsgebaeude, Berlin
-----
Strecke Luftline: 499,608820675903

Schnellste Verbindungen
CSU-Zentrale, Muenchen-- IndividualVerbindung-- > Reichstagsgebaeude, Berlin
Dauer: 06:28:07.1010000, CO2-Emission: 113,122280529295

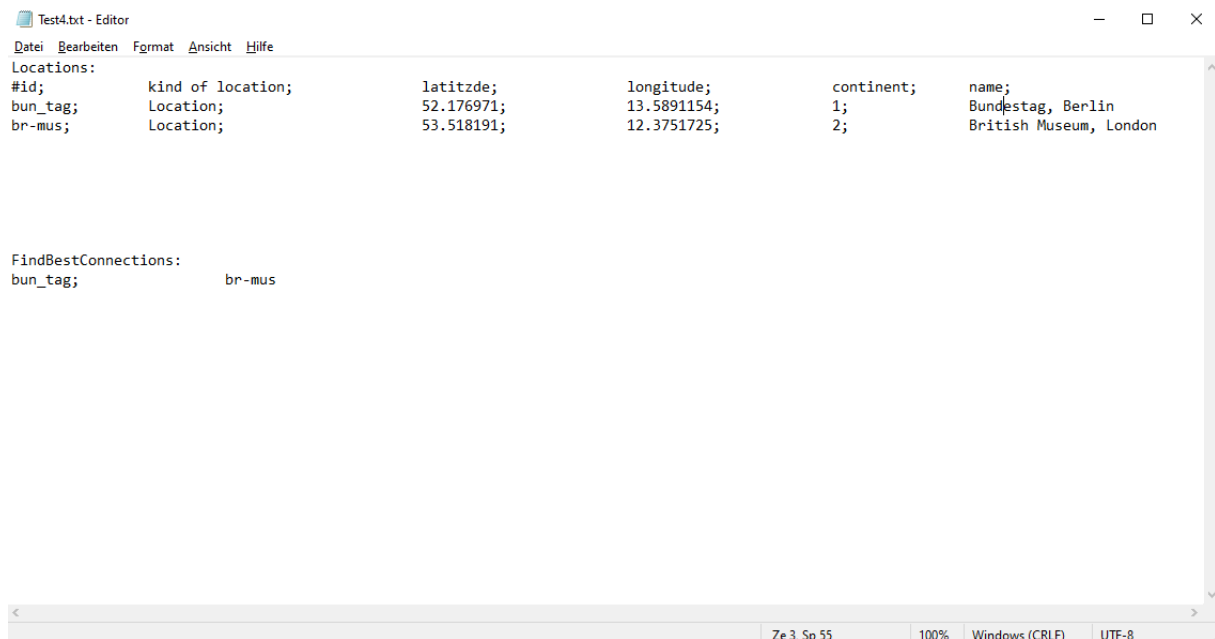
Verbindungen mit der geringsten CO2-Emission:
CSU-Zentrale, Muenchen-- IndividualVerbindung-- > Reichstagsgebaeude, Berlin
Dauer: 06:28:07.1010000, CO2-Emission: 113,122280529295
```

Erwartung:

Da die Entfernung unter 2011 km beträgt, gibt es eine Lösung.

Test4

Input



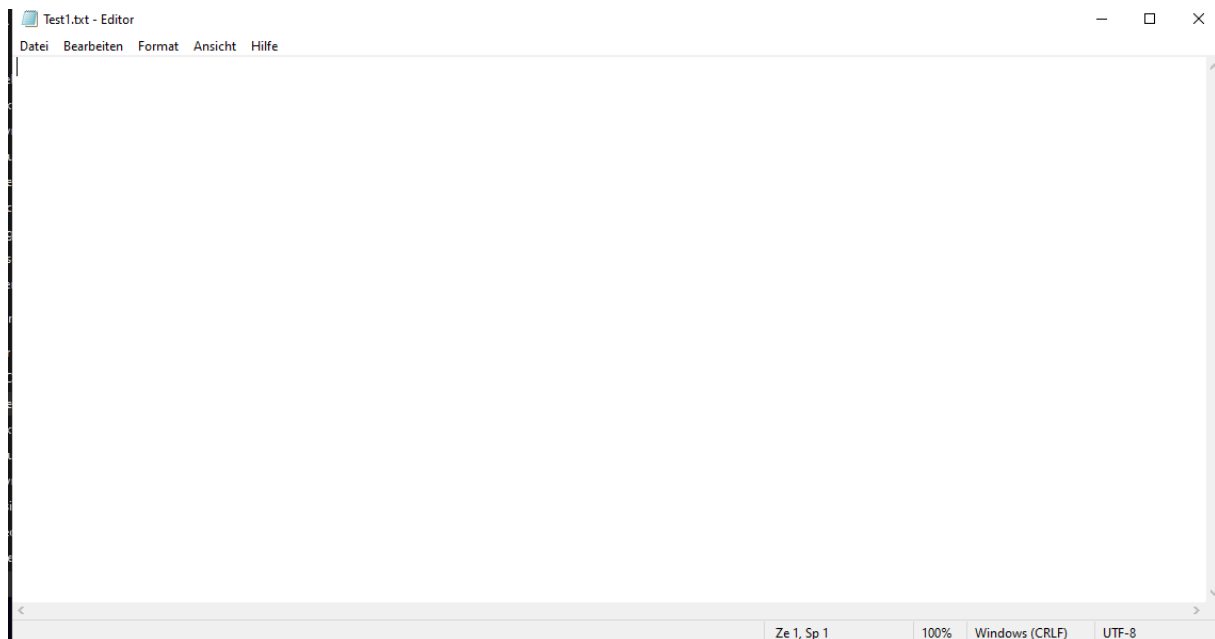
```
Test4.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe

Locations:
#id;          kind of location;    latitzzde;    longitude;    continent;    name;
bun_tag;      Location;                  52.176971;    13.5891154;    1;            Bundestag, Berlin
br-mus;       Location;                  53.518191;    12.3751725;    2;            British Museum, London

FindBestConnections:
bun_tag;      br-mus
```

Ze 3, Sp 55 100% Windows (CRLF) UTF-8

Output



```
Test1.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe
```

Ze 1, Sp 1 100% Windows (CRLF) UTF-8

Erwartung:

Da die beiden Orte auf unterschiedlichen Kontinenten liegen, gibt es keine individuelle Verbindung zwischen den beiden Orten und die Ausgabe Datei ist leer.

Test5

Input

```
Test5.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe

Locations:
#id;          kind of location;    latitide;    longitude;    continent;    name;
csu_zen;      Location;                        48.176971;   11.5895754;   1;            CSU-Zentrale, Muenchen
mum_hbf;      PublicTransportStop;             48.140235;   11.559417;   1;            Muenchen Hbf
mum_flug;     Airport;                         48.35333;    11.770723;   1;            Muenchen Flughafen
reichstag;    Location;                        52.518191;   13.3751725;   1;            Reichstagsgebäude, Berlin
ber_flug;     Airport;                         52.553625;   13.2901544;   1;            Berlin Flughafen Tegel
ber_hbf;      PublicTransportStop;             52.524195;   13.3693013;   1;            Berlin Hbf


FlightSchedule:
#id1;         id2;          (stops;    domestic flight)
mum_flug;     ber_flug;     0;         True

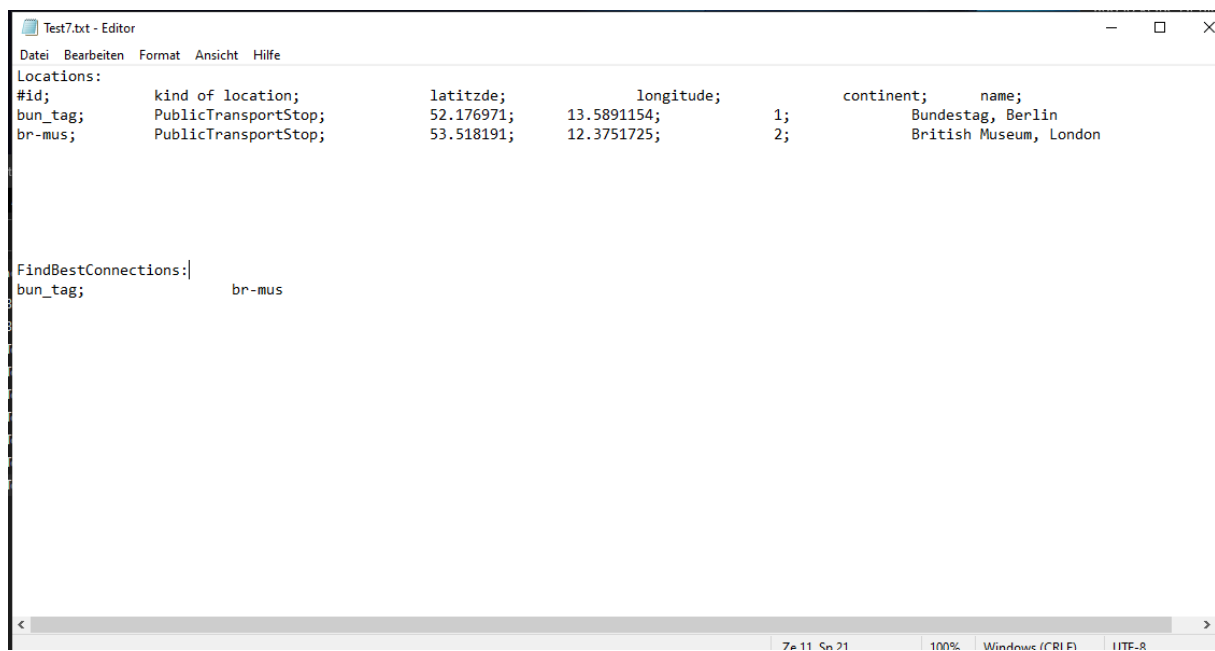

FindBestConnections:
```

Output

Es gibt keine Ausgabe da die Start- und Zielknoten leer sind.

Test6

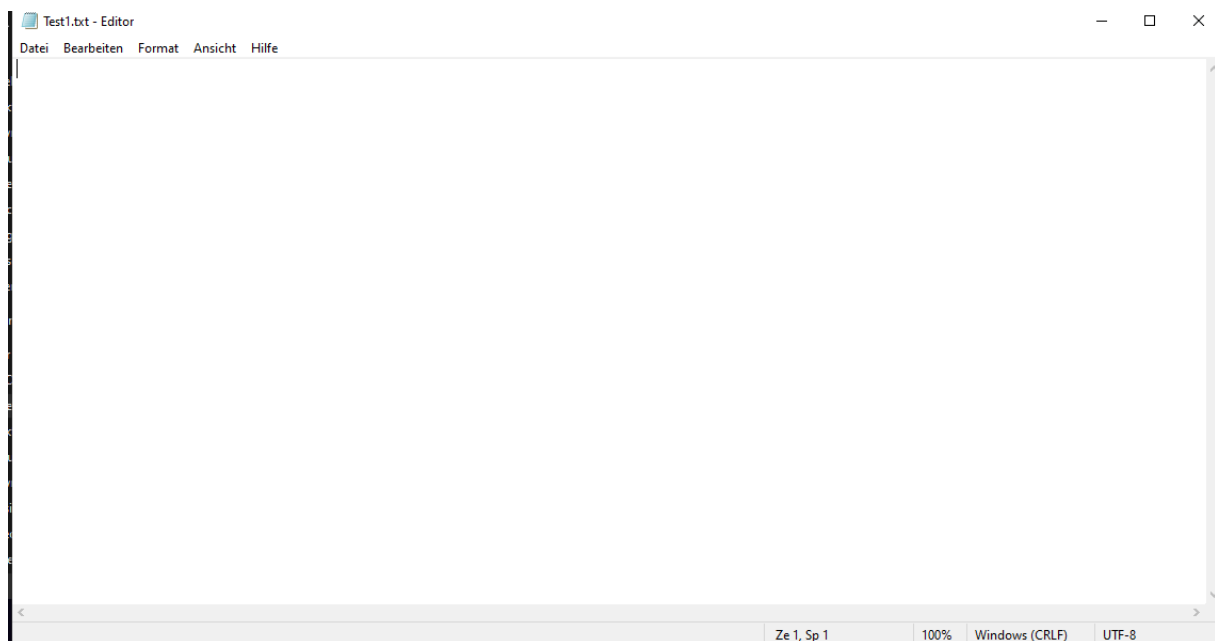
Input



```
Test7.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe
Locations:
#id;          kind of location;      latitnde;      longitude;      continent;      name;
bun_tag;      PublicTransportStop;      52.176971;      13.5891154;      1;      Bundestag, Berlin
br-mus;      PublicTransportStop;      53.518191;      12.3751725;      2;      British Museum, London

FindBestConnections:|
bun_tag;          br-mus
```

Output



```
Test1.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe
```

Erwartung:

Da die beiden Orte auf unterschiedlichen Kontinenten liegen, gibt es keine öffentliche Verbindung zwischen den beiden Orten und die Ausgabe Datei ist leer.

Technische Umgebung

Das Programm wurde in der objektorientierten Programmiersprache C# mit den Microsoft Visual Studio Community 2019 Version 16.7.6 Editor geschrieben. Für die Ausführung des Programms wird das .NET Framework 4.7.2 benötigt.

Die Klassendiagramme, Prozessablaufpläne und das Sequenzdiagramm sind mithilfe der Website <https://draw.io/> erstellt worden. Die Dokumentation wurde mit Microsoft Word 2019 erstellt.

Eigenhändigkeitserklärung

Ich versichere durch meine Unterschrift, dass ich das Prüfungsprodukt und die eingereichten Dokumentationen selbstständig angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche ausgewiesen sind. Die Arbeit hat in dieser Form keiner anderen Prüfungsinstitution vorgelegen.

04.12.2021, Hennigsdorf

Datum, Ort

Unterschrift Prüfungsteilnehmer