Bundeswettbewerb Informatik 2016/2017

Dokumentation

Aufgabe 1: Rosinenpicken

und

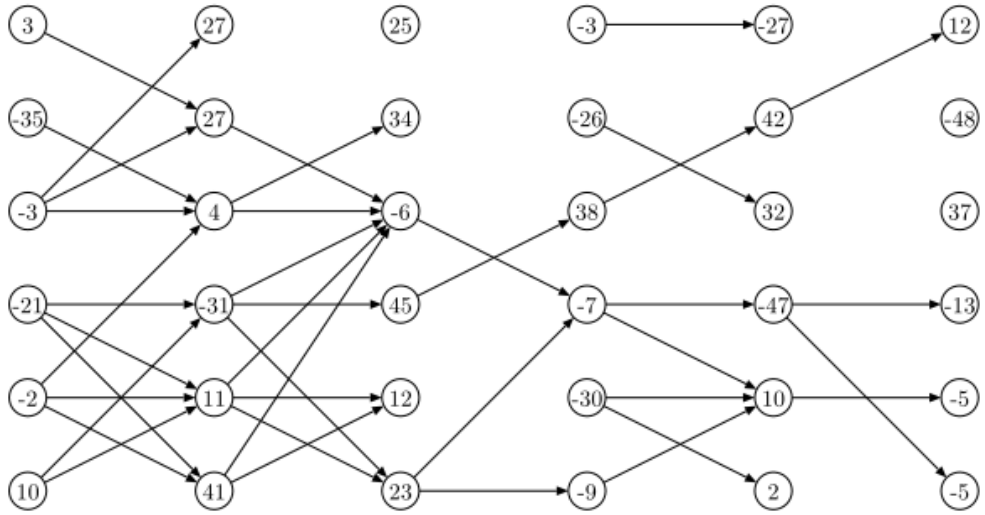
Aufageb 2: Rechtsherum in Rechthausen

von

Inhaltsverzeichnis

Aufgabe 1: Rosinen picken

Ein großes Firmenkonglomerat wird aufgelöst, und du hast die Aufgabe, aus den vielen Bruchstücken eine möglichst wertvolle Teilmenge auszusuchen. Die Werte der Einzelunternehmen sind unterschiedlich; einige sind sogar Verlustgeschäfte und haben daher einen negativen Wert. Leider ist es nicht möglich, einfach nur die Unternehmen mit positivem Wert auszuwählen, weil es noch etliche Nebenbedingungen der Form „Nimmst du A, musst du auch B nehmen“ gibt. In einem Beispielszenario könntest du nur dann die Gummibärenfabrik wählen, wenn du ebenfalls die Zuckerraffinerie und die Altreifenverwertung nimmst – auch wenn diese für sich genommen wenig attraktiv sein sollten. Juristische oder wirtschaftliche Notwendigkeiten, die du nicht beeinflussen kannst, sind für diese Nebenbedingungen verantwortlich. Graphisch können wir eine solche Situation so darstellen:



Die Kreise repräsentieren die Einzelunternehmen und sind mit deren Wert beschriftet. Ein Pfeil von A nach B bedeutet: Wenn man A auswählt, dann muss auch B gewählt werden.

Aufgabe

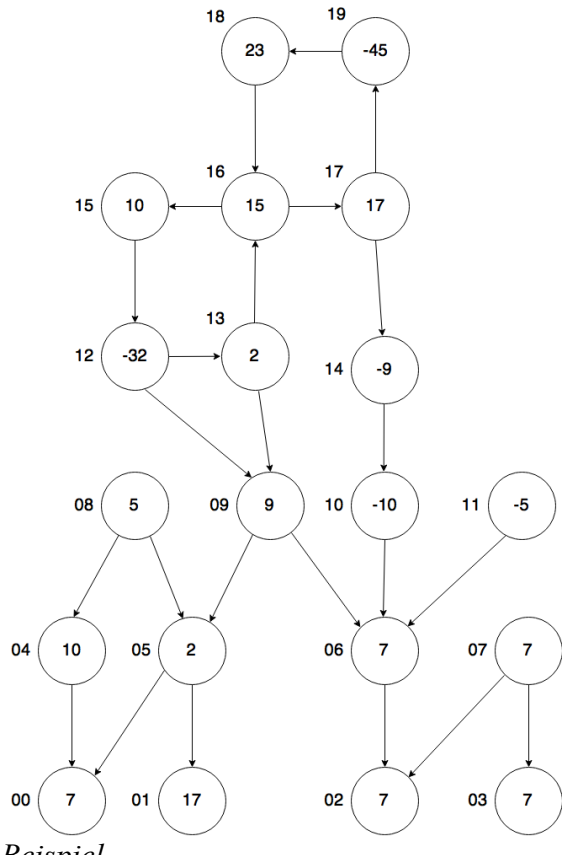
1. Überlege dir und beschreibe ein Verfahren, mit dem man eine wertvolle Teilmenge finden kann.
2. Erstelle ein entsprechendes Programm und wende es auf die Beispiele an, die du auf der BWINF-Webseite zu dieser Runde findest.
3. Findet deinVerfahren garantiert immer ein bestmögliches Ergebnis oder kommt es diesem Ziel nur nahe? Ist die Laufzeit deines Programms erträglich? Natürlich ist eine lange Laufzeit bei einer so wichtigen Entscheidung akzeptabel, wenn die Zahlen in den Kreisen für Millionen Euros stehen.

Finden von möglichst wertvollen Teilmengen

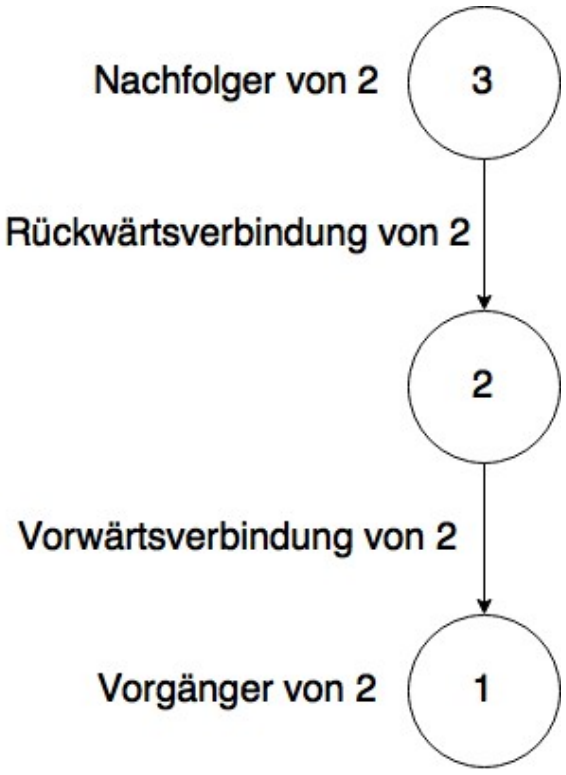
Da in der Aufgabenstellung geschrieben steht:„Leider ist es nicht möglich, einfach nur die Unternehmen mit positivem Wert auszuwählen...“ wird das Verfahren nur nach Teilmengen suchen, deren Wert > 0 beträgt. Damit wäre die wertvollste Teilmenge eine Menge von Rosinen, deren Wert > 0 beträgt und die optimale Kombination von Rosinen und all deren Nachfolgern darstellt.
Das Verfahren für das Finden von diesen möglichst wertvollen Teilmengen unterteilt sich in 5 Teilschritte:

1. Finden und Auflösen von Zyklen
2. Entfernen von allen negativen Rosinen, die keine Vorgänger haben
3. Bestimmen des Absolutwertes und der Absolutmenge aller Rosinen
4. Bestimmen der Optimalmenge und des Optimalwertes für jede Rosine
5. Ermitteln der wertvollsten Teilmenge

Das Verfahren wird schrittweise an folgendem Beispiel vollzogen.

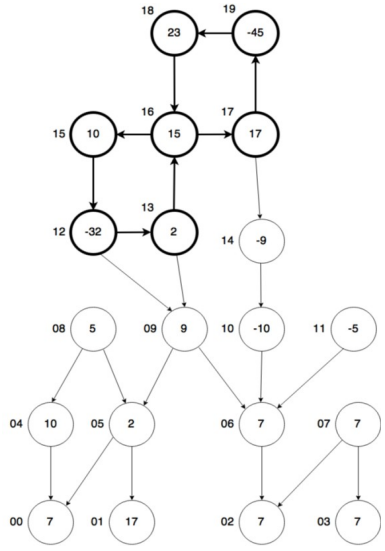


Beispiel

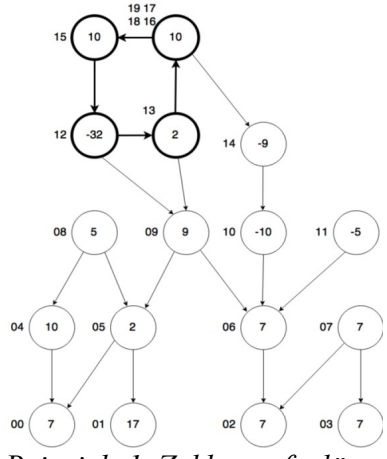


1. Finden und Auflösen von Zyklen:

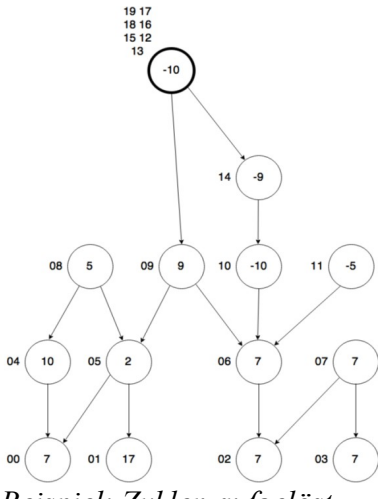
In dem Firmenkonglomerat kann es vorkommen, dass ein Zyklus entsteht. Dieser Zyklus muss aufgelöst werden, damit der Wert der Teilmenge nicht immer weiter erhöht wird, in dem der Zyklus immer wieder durchlaufen wird. Ein Zyklus kann durch die Regel:„Nimmst du A, muss du auch B nehmen“ einfach aufgelöst werden. Durch diese Regel muss der gesamte Zyklus ausgewählt werden, wenn ein Teil des Zyklus ausgewählt wurde.
Durch diese Tatsache kann der Zyklus durch eine neue Rosine ersetzt werden ohne damit die Aufgabe zu verändern. Der Wert der neuen Rosine beträgt soviel, wie die Werte aller Rosinen im Zyklus addiert ergeben. Außerdem müssen alle Verbindungen außerhalb des Zyklus, die auf eine Rosine des Zyklus zeigen oder von einer Rosine des Zyklus auf eine Rosine außerhalb des Zyklus, auf den neuen Knoten übertragen werden.
Im Beispiel gibt es zwei Zyklen zwischen den Rosinen 19→18→16→17→19 und zwischen den Rosinen 16→15→12→13→16. Diese Zyklen werden aufgelöst und jeweils eine neue Rosine mit dem Wert aller Rosinen des Zyklus erstellt. Alle Verbindungen des Zyklus, die nicht auf eine Rosine, die im Zyklus ist, zeigen, werden auch auf die neue Rosine übernommen. Diese neue Rosine ersetzt nun den Zyklus in dem Graphen.



Beispiel: mit Zyklen



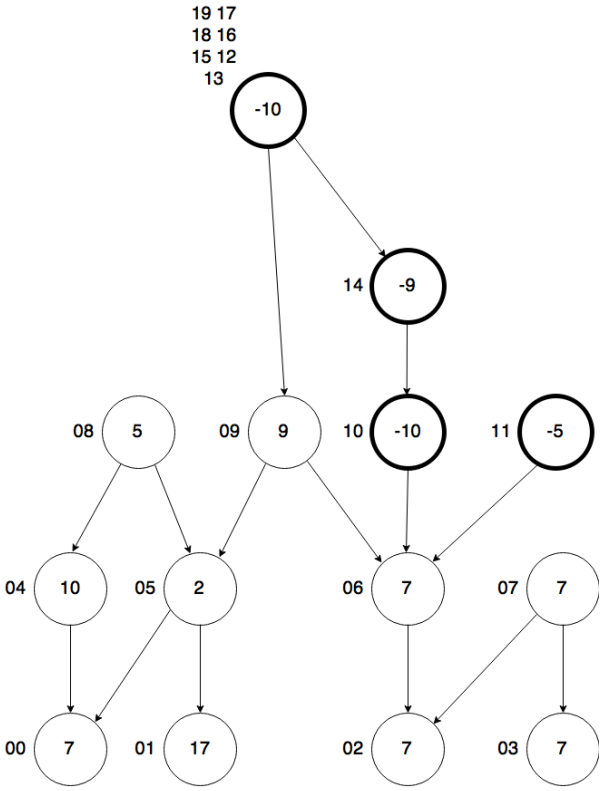
Beispiel: 1. Zyklus aufgelöst



Beispiel: Zyklen aufgelöst

2. Entfernen von allen negativen Rosinen, die keine Vorgänger haben:

Da der Graph gerichtet ist und nach einer möglichst wertvollen Teilmenge gesucht wird, können alle negativen Rosinen, die keine Vorgänger haben, aus dem Graphen entfernt werden.
Somit wird die Menge, die keine Zyklen mehr enthält, auf Rosinen untersucht, die keine oder nur negative Vorgänger haben und deren Wert < 0 beträgt.



Beispiel: negative Rosinen

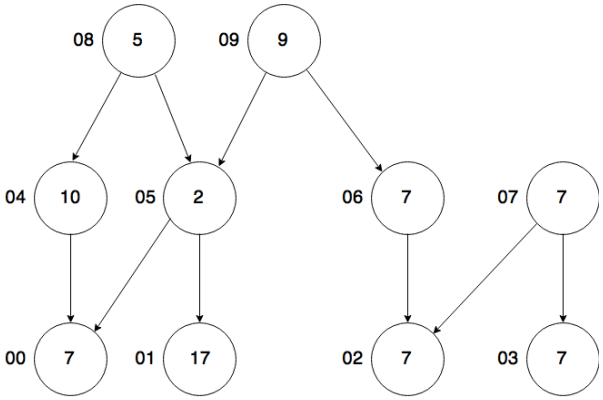
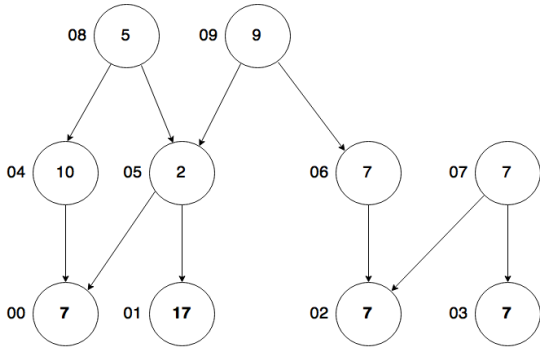


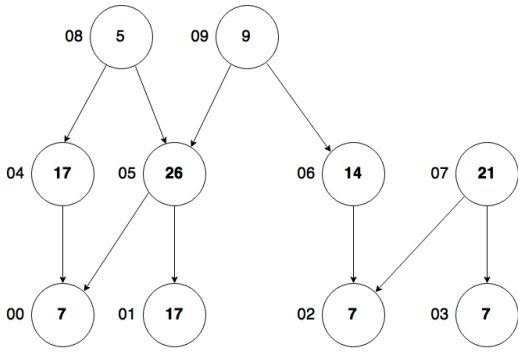
Abbildung 1: Beispiel: negative Rosinen entfernt

3. Bestimmen des Absolutwertes und der Absolutmenge aller Rosinen:

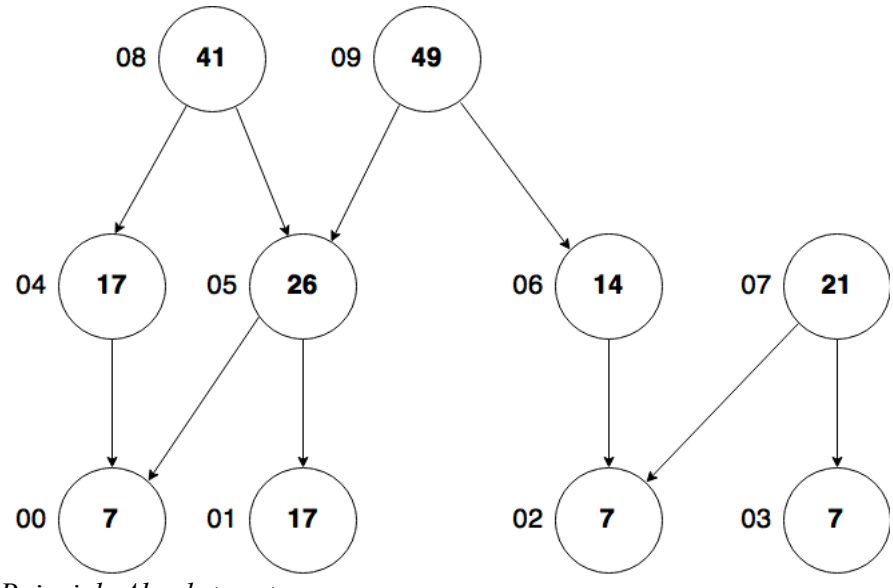
Der Gesamtwert einer Rosine setzt sich aus dem Wert der einzelnen Rosine und allen Rosinen, die mit dieser Rosine über Vorwärtsverbindungen verbunden sind (Absolutmenge), zusammen (Absolutwert).
Um diesen Absolutwert zu ermitteln, wird nun bei den Rosinen, die keine Vorwärtsverbindungen haben, angefangen und deren Wert als ihr Absolutwert festgelegt. Anschließend werden immer weiter die Rosinen untersucht, von denen der Absolutwert aller Vorgänger bekannt ist und zusammen addiert zu dem Absolutwert. Falls es bei dieser Untersuchung vorkommt, dass zwei Rosinen addiert werden, die beide in der Menge ihrer Vorgängern eine gemeinsame Rosine haben, darf diese nur einmal gewertet werden.



Beispiel: Absolutwerte bilden (1)



Beispiel: Absolutwerte bilden (2)

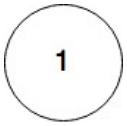


Beispiel: Absolutwerte

4. Bestimmen der optimalen Rosinenkombination für jede Rosine:

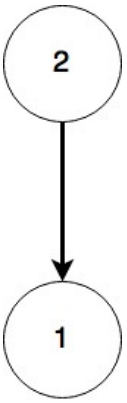
In diesem Schritt des Verfahrens werden die optimalen Kombinationen von Rosinen gesucht.
Es sollen wieder alle Rosinen so untersucht werden, dass immer alle Vorgänger der momentan zu untersuchenden Rosine schon untersucht wurden. Dadurch kann sicher gestellt werden, dass die bisherige Auswahl an Rosinen wirklich die optimale ist.
Es wird sich an jeder Rosine gemerkt, welche die optimale Kombination aus Vorgängern dieser Rosine ist(Optimalmenge). Diese Menge wird aufgebaut in dem immer der Absolutwert der Nachfolger mit der Optimalmenge des Vorgänger verglichen wird.
Bei diesem Vorgehen gibt es drei Fälle zu beachten.

1. Eine Rosine hat keinen Vorgänger:
   Wenn eine Rosine keinen Vorgänger hat, wird der Absolutwert mit 0 verglichen.
   Wenn der Absolutwert > 0 beträgt, wird der Optimalwert und die Optimalmenge mit dem Absolutwert und der Absolutmenge gleichgesetzt.
   Wenn der Absolutwert < 0 beträgt, wird die Optimalmenge mit einer leeren Menge gleichgesetzt und der Optimalwert = 0 gesetzt.



1. Fall

2. Eine Rosine hat genau einen Vorgänger:
   In diesem Fall wird der Absolutwert dieser Rosine mit dem Optimalwert der Vorgängerrosine verglichen.
   Ist der Optimalwert >= Absolutwert, dann wird der Optimalwert und die Optimalmenge der Rosine mit dem Optimalwert und der Optimalmenge der Vorgängerrosine gleichgesetzt.
   Andernfalls wird der Optimalwert und die Optimalmenge mit dem Absolutwert und der Absolutmenge der Rosine gleichgesetzt.



2. Fall

3. Genau eine Rosine hat beliebig viele Vorgänger:

   In diesem Fall müssen die Optimalmengen der direkten Vorgängern miteinander vereint werden. Falls es Schnittmenge gibt, darf jedes Element daraus nur einfach gewertet werden.

   Der dabei entstehende Wert der Menge wird mit dem Absolutwert der Rosine verglichen.

   Wenn der Wert der dabei entstandenen Vereinigungsmenge > dem Absolutwert der Rosine ist, wird der Optimalwert dem Wert der Vereinigungsmenge gleichgesetzt und die Optimalmenge mit der Vereinigungsmenge gleichgesetzt.

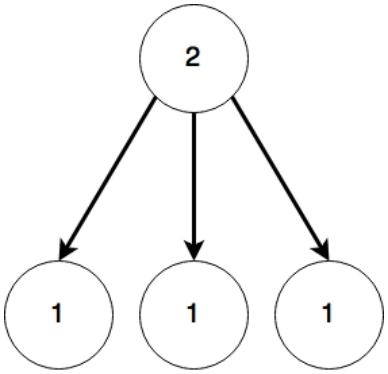
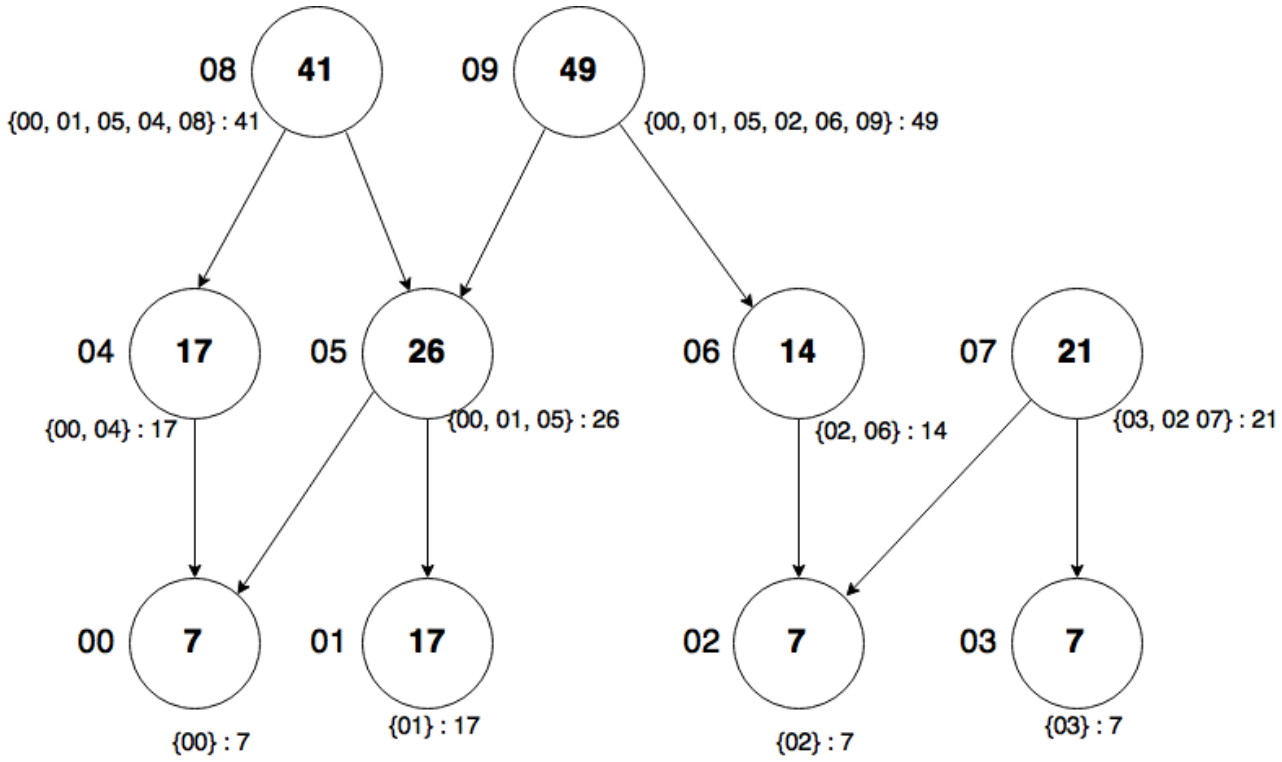


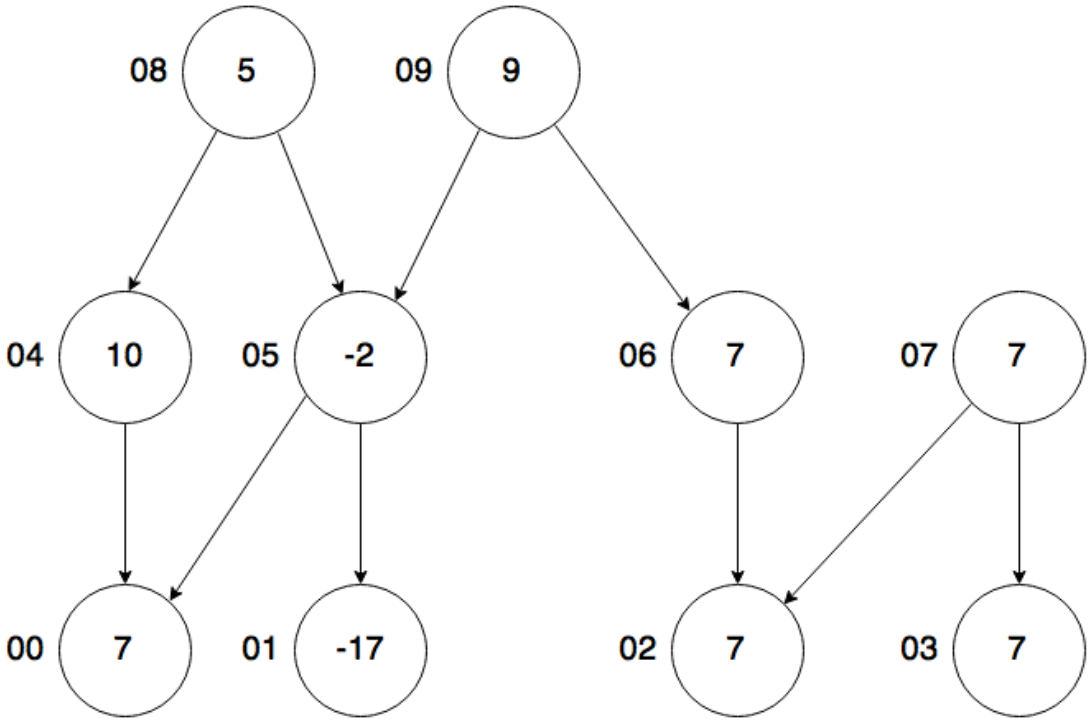
Abbildung 2: 3. Fall

   Wenn der Wert der der dabei entstandenen Vereinigungsmenge < dem Absolutwert der Rosine ist, wird der Optimalwert mit dem Absolutwert und die Optimalmenge mit der Absolutmenge gleichgesetzt.

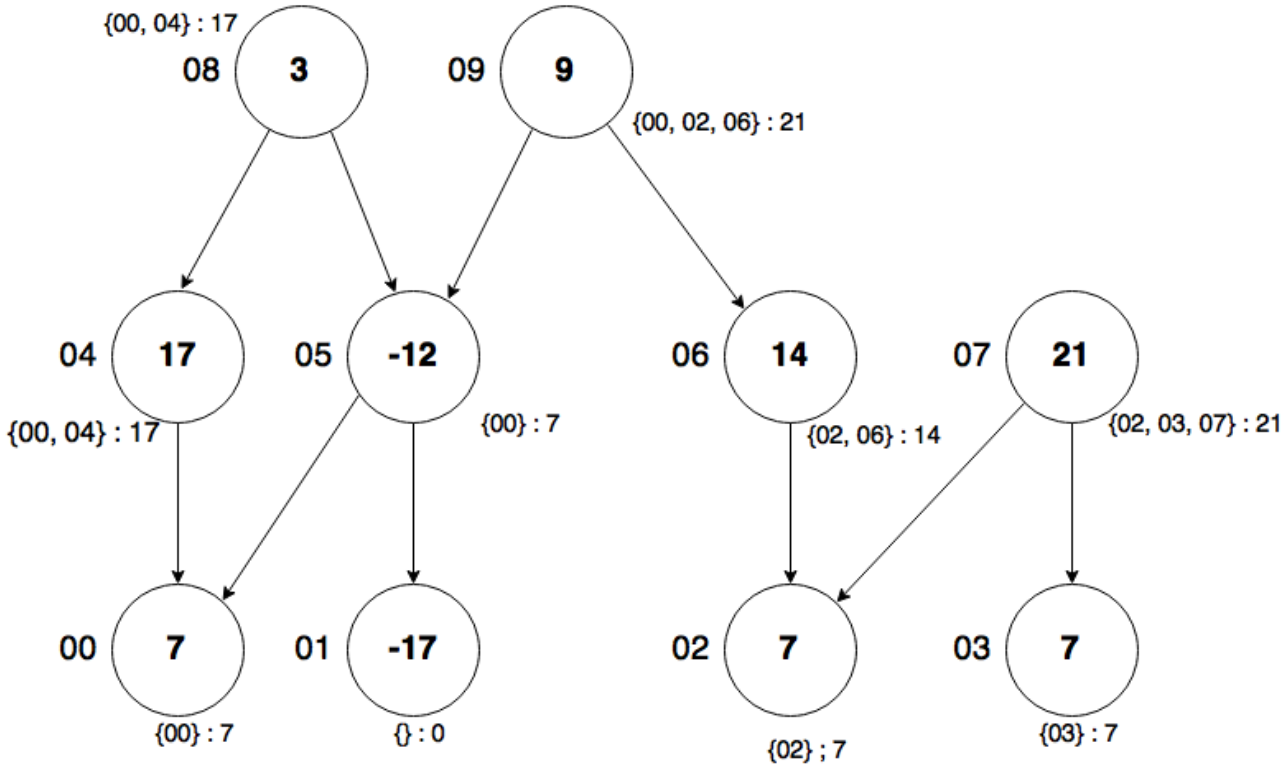
Absolutwerte mit Optimalmengen und Optimalwerten



modifizierter Ausgangsgraph:



Absolutwerte mit optimal Mengen und Optimalwert:



5. Ermitteln der wertvollsten Teilmenge:

Um letztendlich die wertvollste Teilmenge auszusuchen, wird die Optimalmenge von jeder Rosine, die keinen Nachfolger hat, untersucht.
Dabei sollen alle Optimalmengen dieser Rosinen vereinigt werden, wobei die Elemente der Schnittmenge nur einmal zusammengeführt werden dürfen.
Dabei entsteht die optimale Rosinenkombination.
Beispiel:

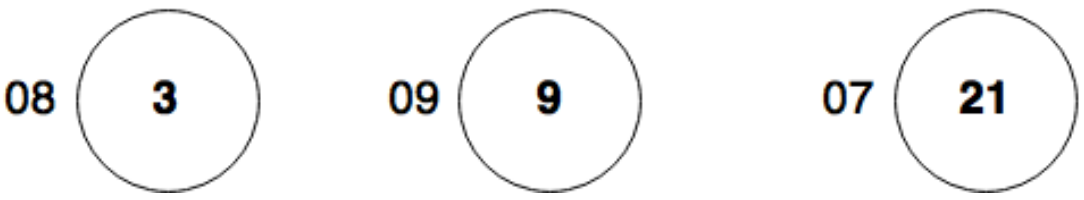
{00, 01, 02, 03, 04, 05, 06 ,07, 08, 09} : 78

{00, 04, 01, 05, 08} : 41 {00, 01, 05, 02, 06, 09} : 49 {02, 03, 07} : 21

modifiziertes Beispiel:

{00, 04, 02, 06, 03, 07} : 45

{00, 04} : 17 {00, 02, 06} : 21 {02, 03, 07} : 21



Umsetzung

Die Lösungsidee wurde in c# implementiert. In der Dokumentation der Umsetzung werden die Schritte zum Einlesen der Datei, zum Finden und Auflösen von Zyklen, zum Bestimmen des Absolutwertes und der Absolutmenge aller Rosinen und zum Bestimmen der Optimalmenge und des Optimalwertes für jede Rosine beschrieben.

1. Einlesen der Datei:

Beim Einlesen der Datei wird für jede Rosine eine Instanz der Klasse 'Raisin' [I.] (Referenz auf Quellcode S. 10) erstellt. In dieser Klasse werden jeweils der Name [II.], eine eindeutige ID [IV.], der Wert [III.], alle Vorwärtsverbindungen [V.] und Rückwärtsverbindungen [VI.] gespeichert. Außerdem werden Variablen für das Finden von Zyklen [VII.], dem Optimalwert und der Optimalmenge [VIII.] gespeichert.
Außerdem hat die Klasse zwei Methoden *NextForwardConnection* [IX.] und *GetDefintivevalueOfRaisin* [X.].
Die erste Methode *NextForwardConnection* gibt bei jedem Aufruf nacheinander die Vorwärtsverbindungen, als Funktionswert zurück.
Die zweite Methode *GetDefintivevalueOfRaisin* bestimmt den Absolutwert für diese Rosine.

```
I.  class Raisin
    {
        public enum Sets { Start, Stack, End };


        //VARIABLES CREATED BY READING THE FILE
        #region
II.     public string raisinName;
        //the value read from file
III.    public float value;
        //the number this raisin was read from the file
IV.     public string ID;
        //the dependencies from this raisin
V.      public List<Raisin> forwardConnections;
VI.     public List<Raisin> backwardsConnections;
        #endregion

VII.    //VARIABLES FOR FINDING CYCLES
        #region
        //if I select this raisin all the raisins I have to take too(SUCCESSORS)
        public List<Raisin> successors;
        //if the raisin is part of a loop or not
        public bool inLoop;
        //the set where this raisin is currently
        public Sets currentSet;
        //from which raisin this raisin was introduced
        public Raisin introducedBy;
        //if this raisins consists of multiple raisins that are in a cycle
        public bool isCycle = false;
        #endregion
VIII.
        //a List with all Raisin-Names i have to take if i take this raisin
        public HashSet<Raisin> raisinsIncluded = new HashSet<Raisin>();
        //the value read from file + the value of all successors from this raisin
        public float definitiveValue;

        //All Raisins that are the optimal-set
        //can only include raisins below this raisin
        public HashSet<Raisin> optimalSet = new HashSet<Raisin>();
        //the value of all Raisins in the optimal-set
        public float optimalValue = 0f;


IX.     public Raisin NextForwardConnection()


X.      public void GetDefintivevalueOfRaisin()
    }
```

2. Finden und Auflösen von Zyklen:

Für das Finden von Zyklen werden drei Mengen (Listen) erstellt. Die erste Liste ist die Startliste in der alle Rosinen am Anfang sind. Von der Startliste wird das erste Element herausgenommen, untersucht und dann in die zweite Liste, den Stack, verschoben. Wenn alle Vorgänger einer Rosine untersucht wurden, wird die Rosine von dem Stack in die dritte Liste, die Endliste, verschoben. Dieser Vorgang wird für alle Elemente aus der Startliste solange wiederholt, bis alle aus der Startliste in die Endliste verschoben wurden.
Um alle Zyklen in dem Firmenkonglomerat(Graph) zu finden wird die Funktion FindAllLoops [I.] (Referenz auf Quellcode S. 12) aufgerufen.
In dieser Funktion werden zunächst alle eingelesenen Rosinen in die Startliste geschoben [II.]. Daraufhin wird eine While-Schleife solange ausgeführt, bis es keine Elemente mehr in der Startliste gibt [III.]. Nun wird die erste Rosine aus der Startliste untersucht [IV.]. Wenn diese Rosine schon einmal untersucht wurde und festgestellt wurde, das sie Teil eines Zyklus ist und deswegen zu einer neuen Rosine zusammengefasst wurde, wird sie übersprungen [V.].
Sie wird auch übersprungen, wenn sie sich schon in der Endliste befindet [VI.].
Wenn es hingegen der Fall ist, dass die Rosine keine Vorgänger mehr hat, kann sie in die Endliste verschoben werden, weil sie kein Teil eines Zyklus sein kann [VII.].
Wenn dieser Fall nicht zutrifft wird die Rosine in den Stack verschoben [VIII.] und eine While-Schleife solange ausgeführt, bis keine Elemente mehr im Stack sind [IX.].
Als erster Schritt der Schleife wird der nächste Vorgänger des ersten Elementes im Stack untersucht [X.]. Wenn schon alle Vorgänger dieser Rosine untersucht wurden, wird die Rosine in die Endliste verschoben [XI.]. Ansonsten wird bei der Rosine gespeichert, von wem sie als Vorgänger bestimmt wurde, um bei dem Auftreten eines Zyklus feststellen zu können, welche Rosinen alle im Zyklus sind [XII.]. Jetzt wird überprüft, ob die Rosine sich schon im Stack befindet [XIII.]. Wenn dies der Fall ist wurde ein Zyklus gefunden und die Funktion substituedCycle aufgerufen [XIV.]. Diese Funktion löst den Zyklus auf und ersetzt ihn mit einer neuen Rosine, die den Wert und die Namen aller Rosinen im Zyklus erhält. Diese neu erstellt Rosine wird wieder in die Startliste geschoben [XV.].
Wenn die momentan zu untersuchende Rosine noch nicht im Stack ist, sich aber in der Startliste befindet, wird sie in den Stack verschoben [XVI.].
Andernfalls wird sie in den Stack verschoben [XVII.].

Die Endliste ist so geordnet, dass, wenn durch sie iteriert wird, immer alle Vorgänger bekannt sind.

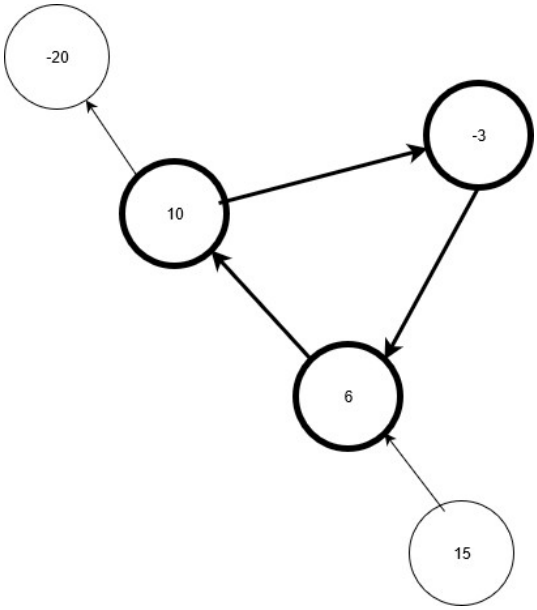
```
I.   Public void FindAllLoops(){
                //add all raisins to the start set
II.             for (int i = 0; i < raisins.Count; i++){
                    this.Move_To(raisins.Values[i], this.start, Raisin.Sets.Start);
                }

                //loop over all raisins in this file
III.            while(this.start.Count != 0)
                {
IV.                 Raisin item = this.start[0];

                    //if the raisin is in a loop
V.                  if (item.inLoop)
                        continue;

                    //if this raisin was checked and is in the END-set
VI.                 if (item.currentSet == Raisin.Sets.End)
                        continue;

                    //Raisin is an END-raisin
VII.                if(item.forwardConnections.Count == 0)
                    {
                        this.MoveFrom_To(item, this.start, this.end, Raisin.Sets.End);
                        continue;
                    }
                    //move raisin from start to stack
VIII.               this.MoveFrom_To(item, this.start, this.stack, Raisin.Sets.Stack);

                    //loop unitl stack is empty
IX.                 while(stack.Count != 0)
                    {
                        //get the next successor of the raisin
X.                      Raisin tmp = this.stack[this.stack.Count - 1].NextForwardConnection();

                        //if successor is null --> move to endset
XI.                     if (tmp == null)
                        {
                            this.MoveFrom_To(stack[this.stack.Count - 1], this.stack, this.end,
                            Raisin.Sets.End);
                            continue;
                        }
                        //set the predeccessor ot the succesor
XII.                    tmp.introducedBy = this.stack[this.stack.Count - 1];

                        //check if the successor of the raisin is in the stack
                        //--> LOOP
XIII.                   if (tmp.currentSet == Raisin.Sets.Stack)
                        {
                            //Found cycle
XIV.                        Raisin substituedCycle = this.SubstituteCycle(tmp);
XV.                         substituedCycle.currentSet = Raisin.Sets.Start;
                            this.start.Add(substituedCycle);

                            continue;
                        }
XVI.                    if(tmp.currentSet == Raisin.Sets.Start)
                        {
                            this.MoveFrom_To(tmp, this.start, this.stack, Raisin.Sets.Stack);
                            continue;
                        }

XVII.                   this.Move_To(tmp, this.stack, Raisin.Sets.Stack);
                    }
                }
            }
```

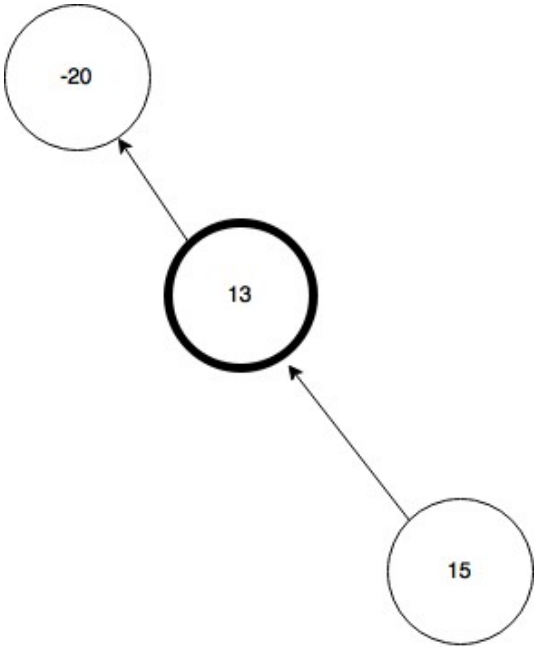
Die Beispieldateien Zyklus_01 – Zyklus_07 werden hier dargestellt mit den Lösungen, welche der Algorithmus ausgibt.

Ausgabeformat: *Namen der Rosinen im Zyklus* || value = *der Wert der Rosinen addiert*

Zyklus_01.txt

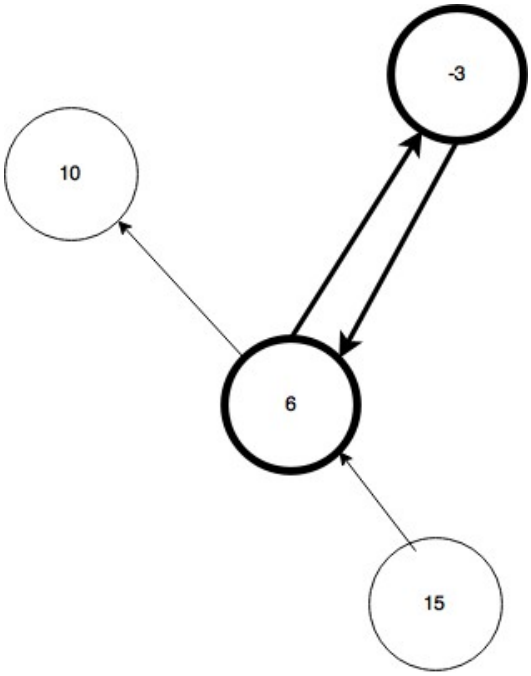
2 3 1 || value = 13

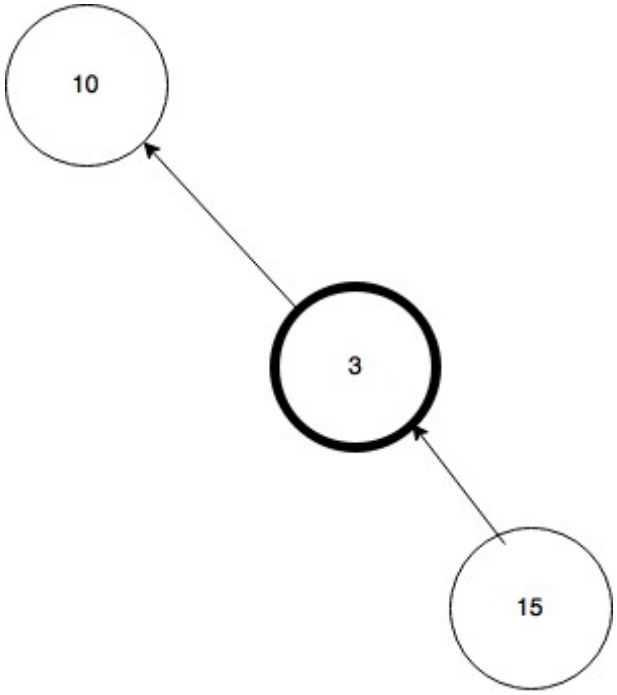




Zyklus_02.txt

2 1 || value = 3

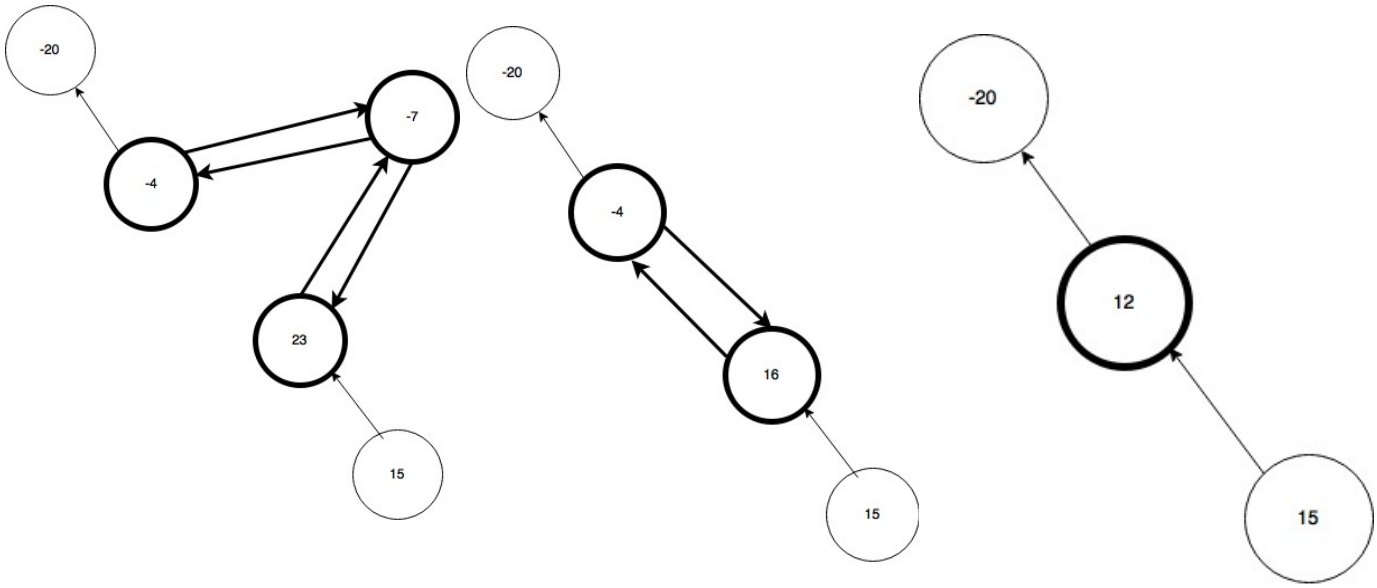




Zyklus_03.txt

2 1 || value = 16

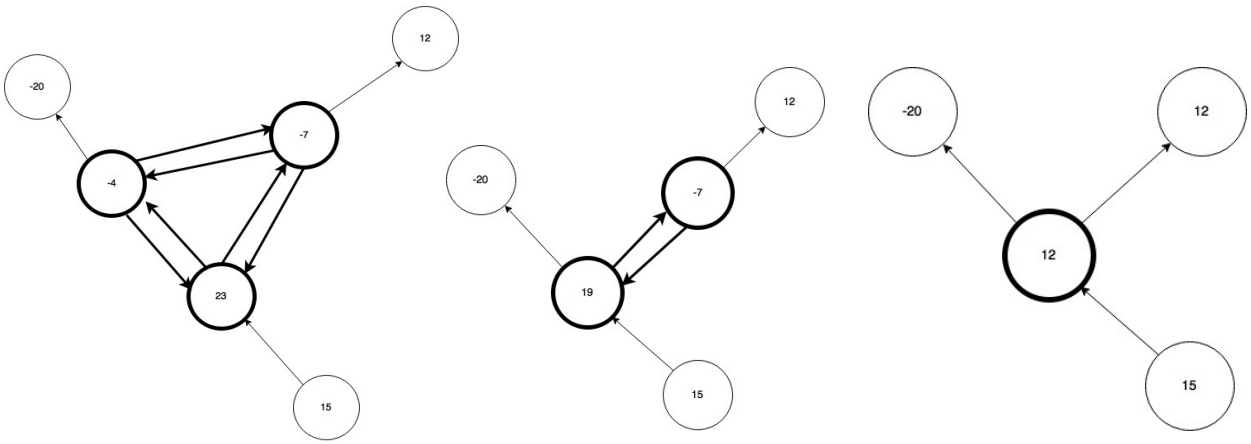
3 2 1 || value = 12



Zyklus_04.txt

2 1 || value = 19

3 2 1 || value = 12



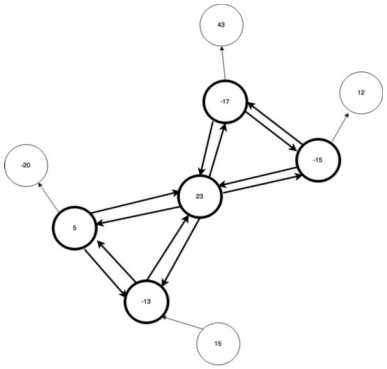
Zyklus_05.txt

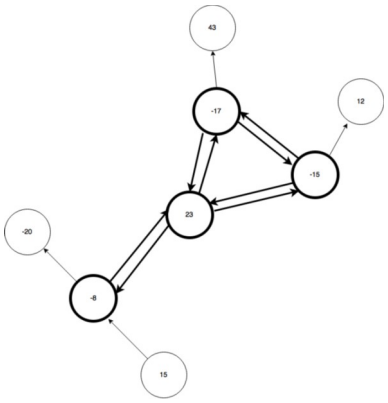
2 1 || value = -8

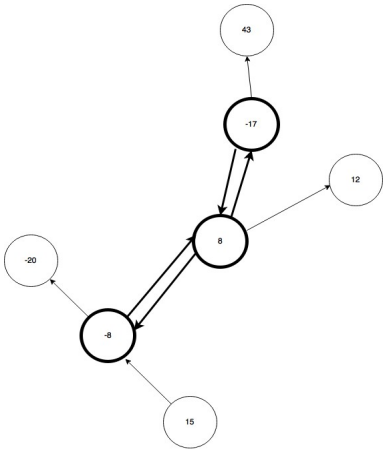
5 4 || value = 8

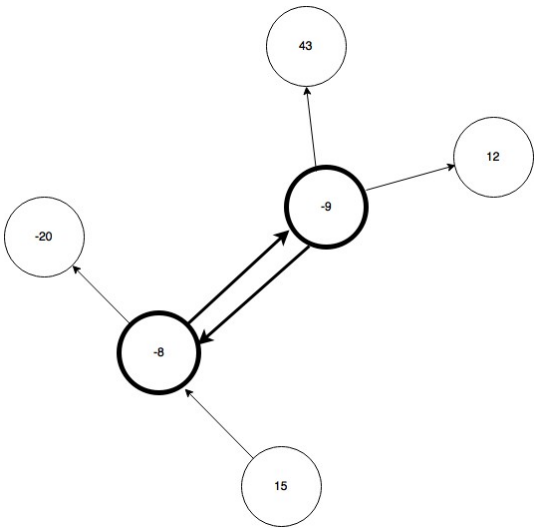
6 5 4 || value = -9

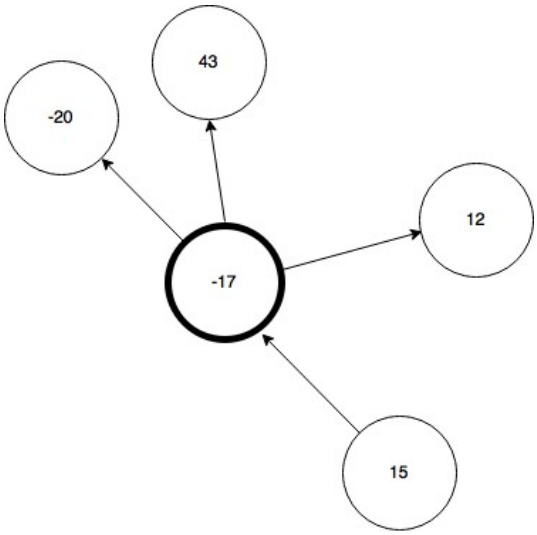
6 5 4 2 1 || value = -17







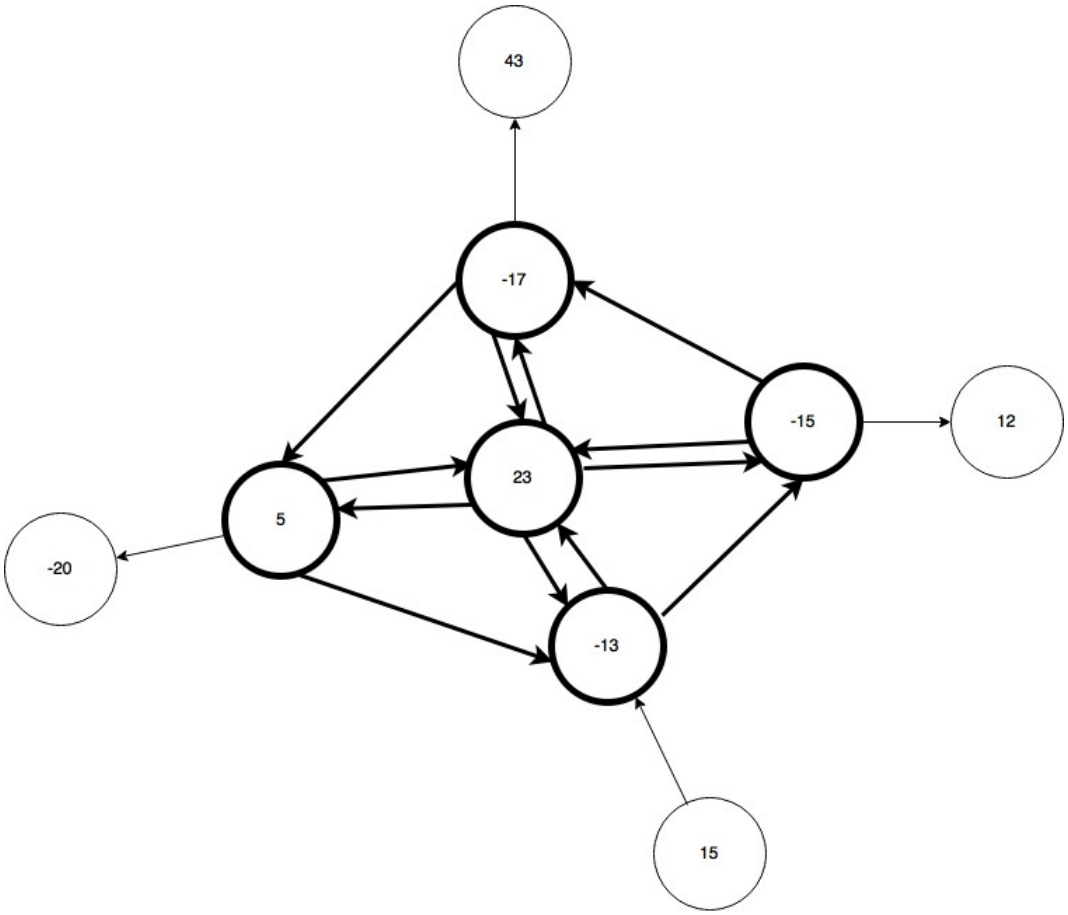


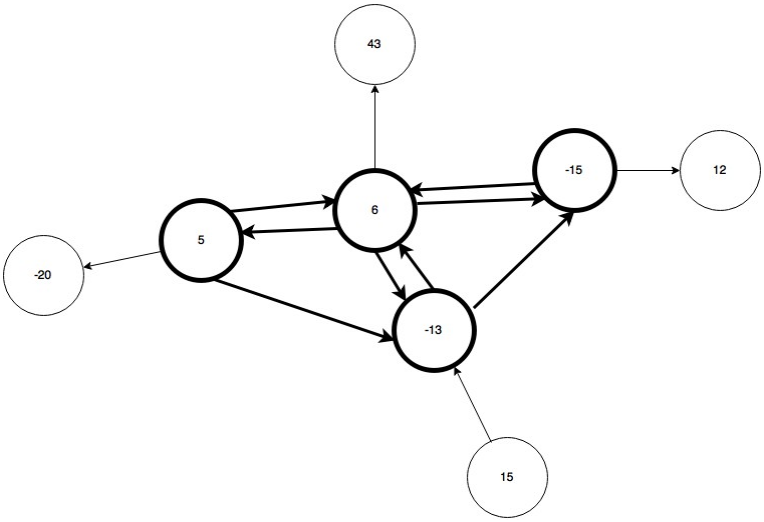


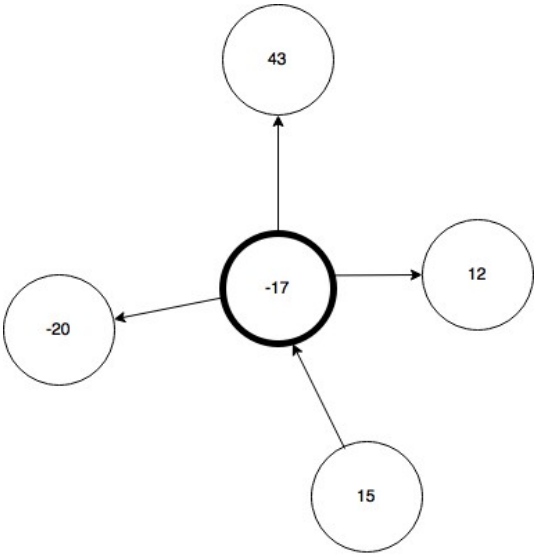
Zyklus_06.txt

4 0 || value = 6

4 0 3 2 1 || value = -17







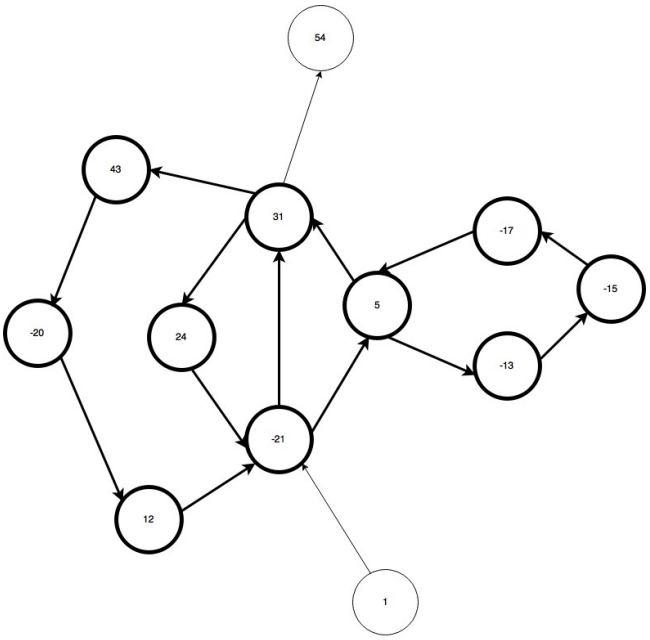
Zyklus_07.txt

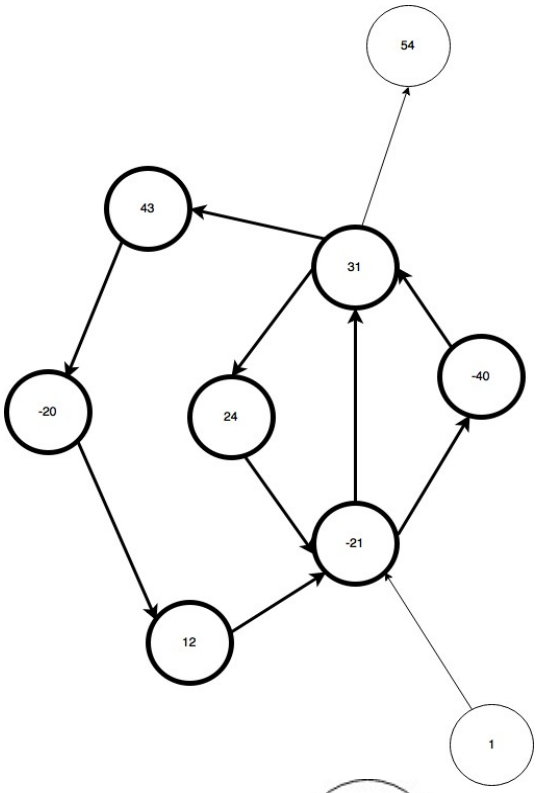
5 6 7 4 || value = -40

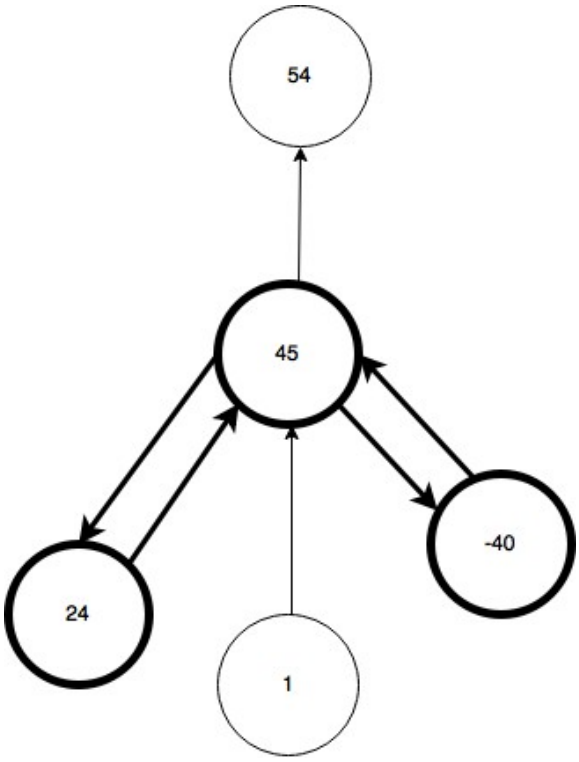
8 3 2 1 0 || value = 45

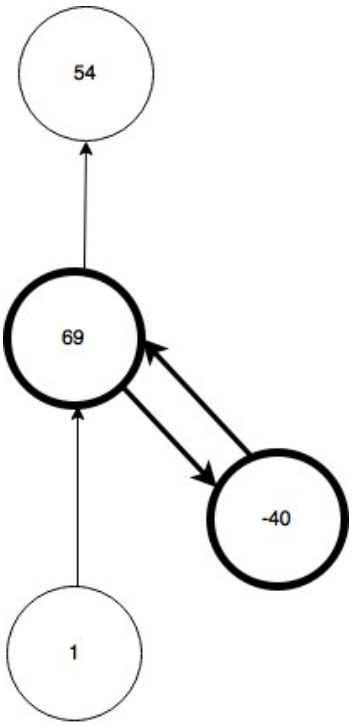
9 8 3 2 1 0 || value = 69

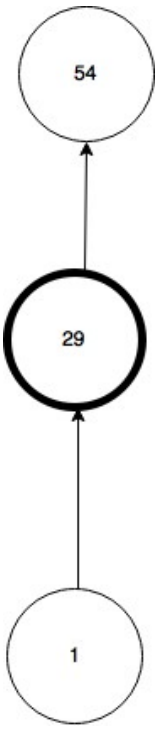
5 6 7 4 9 8 3 2 1 0 || value = 29











3. Bestimmen des Absolutwertes und der Absolutmenge aller Rosinen:

Um den Absolutwert und die Absolutmenge jeder Rosine zu bestimmen, wird die Methode *GetDefintivevalueOfRaisin* [I.] (Referenz auf Quellcode S. 19) jeder Rosine gerufen.
Die Funktion erstellt zunächst eine Schlange und fügt die Rosine als erstes Element der Schlange hinzu [II.]. Außerdem wird eine Menge (HashSet) erstellt in dem alle Rosinen gespeichert werden, die mit dieser Rosine zusammen genommen werden müssen [III.].
Nun wird eine While-Schleife solange ausgeführt, bis keine Elemente mehr in der Schlange sind [IV.]. In dieser Schleife wird zunächst immer das erste Element aus der Schlange untersucht [V.]. Wenn die Absolutmenge der zu untersuchende Rosine mehr als 0 Rosinen enthält [VI.] wurde die Absolutmenge schon festgelegt und jede dieser Rosinen [VII.] wird daraufhin untersucht, ob sie schon in dem HashSet ist, mit den Rosinen, die mit der im Moment zu untersuchenden Rosinen genommen werden müssen [VII.]. Wenn sie noch nicht in dem HashSet ist, wird sie hinzugefügt. Wenn die Absolutmenge noch nicht festgelegt wurde, werden alle Vorwärtsverbindungen dieser Rosinen untersucht [IX.]. Wenn eine dieser Vorwärtsverbindungen schon in dem HashSet steht, wird sie übersprungen [X.]. Ansonsten wird sie der Schlange angefügt [XI.]. Im Anschluss wird die untersuchte Rosine dem HashSet hinzugefügt [XII.].
Wenn alle Rosinen aus der Schlange entfernt wurden und die Schlange somit keine Element mehr enthält, werden alle Elemente aus dem HashSet [XIII.] als Absolutmenge festgelegt und deren addierte Werte ergeben den Absolutwert [XIV.].

```
I.    public void GetDefintivevalueOfRaisin()
          {
II.           Queue<Raisin> raisinsToAdd = new Queue<Raisin>();
              raisinsToAdd.Enqueue(this);

III.          HashSet<Raisin> raisinToTake = new HashSet<Raisin>();

              //loop over all predecessors of this Raisin
IV.           while(raisinsToAdd.Count != 0)
              {
                  //take first raisin from the queue
V.                Raisin tmp = raisinsToAdd.Dequeue();

VI.               if(tmp.raisinsIncluded.Count != 0)
                  {
VII.                  foreach (Raisin item in tmp.raisinsIncluded)
                      {
VIII.                     if (!raisinToTake.Contains(item))
                          {
                              raisinToTake.Add(item);
                          }
                      }

                      continue;
                  }

                  //add all forward connections of this raisin to the queue
IX.               for (int i = 0; i < tmp.forwardConnections.Count; i++)
                  {
X.                    if (raisinToTake.Contains(tmp.forwardConnections[i]))
                          continue;

XI.                   raisinsToAdd.Enqueue(tmp.forwardConnections[i]);
                  }

XII.              raisinToTake.Add(tmp);
              }

XIII.         foreach (Raisin item in raisinToTake)
              {
XIV.              this.raisinsIncluded.Add(item);
                  definitiveValue += item.value;
              }
          }
```

4. Bestimmen der Optimalmenge und des Optimalwertes für jede Rosine:

Um die Optimalmenge und den Optimalwert für jede Rosine festzulegen, wird die Funktion *PickBestRaisinSetForEveryRaisin* (Referenz auf Quellcode S. 22) aufgerufen.
In dieser Funktion wird jedes Element aus der Endliste untersucht [II.] und damit garantiert, dass alle Vorgänger der momentan zu untersuchenden Rosine schon untersucht wurden. Als erster Schritt wird in der Schleife immer ein Element aus der Endliste genommen [III.] und untersucht. Wenn diese Rosine keine Vorwärtsverbindung hat, wird wie nach Fall 1 der Strategie gehandelt [IV.]. Damit wird die Rosine als Optimalmenge[VII.] und ihr Wert als Optimalwert angenommen[VI.], wenn ihr Wert > 0 beträgt [V.].
Wenn sie genau eine Verbindung hat, wird wie nach Fall 2 der Strategie gehandelt [VIII.]. Somit wird der Optimalwert des Vorgängers mit dem Absolutwert der Rosine verglichen.
Wenn der Optimalwert >= dem Absolutwert der Rosine ist [IX.], werden der Optimalwert und die Optimalmenge des Vorgängers auf die Rosine übertragen.
Wenn hingegen der Absolutwert > dem Optimalwert [X.] ist, wird der Absolutwert und die Absolutmenge als neuer Optimalwert und neue Optimalmenge der Rosine festgelegt.
Wenn die Rosine mehr als eine Vorwärtsverbindung hat, wird wie nach dem 3. Fall der Strategie gehandelt [XI.].
Dann wird eine neue Menge (HashSet) erstellt. Daraufhin werden von jedem Vorgänger [XII.] alle Rosinen aus deren Optimalmenge [XIII.], wenn sie noch nicht in dem neu erstellten HashSet stehen, [XIV.] hinzugefügt [XV.].
Im Anschluss werden die Werte aller Rosinen aus dem HashSet [XVI.] addiert [XVII].
Falls der dabei enstandene Wert >= dem Absolutwert der moment zu untersuchenden Rosinen ist, wird das neu entstandene HashSet als Optimalmenge und der addierte Wert als Optimalwert für diese Rosine festgelegt.
Wenn der dabei entstandene Wert < dem Absolutwert [XIX.] ist, dann wird der Absolutwert und die Absolutmenge als Optimalwert und die Optimalmenge festgelegt.

```
I.    public void PickBestRaisinSetForEveryRaisin(){
II.             for (int i = 0; i < this.end.Count; i++)
                {
                    //the raisin for that we want to find the optimal set
III.                Raisin tmp = this.end[i];

                    if (tmp.optimalSet.Count > 0)
                        Console.WriteLine("A BUG OCCURED! RAISIN ALREADY HAS A OPTIMAL VALUE");

IV.                 if(tmp.forwardConnections.Count == 0)
                    {
V.                      if(tmp.definitiveValue > 0)
                        {
VI.                         tmp.optimalValue = tmp.definitiveValue;
VII.                        tmp.optimalSet = tmp.raisinsIncluded;
                        }
                    }
VIII.               else if(tmp.forwardConnections.Count == 1)
                    {
                        //optimal-value(previous node) > definitive-value(this node)
IX.                     if (tmp.forwardConnections[0].optimalValue >= tmp.definitiveValue)
                        {
                            tmp.optimalValue = tmp.forwardConnections[0].optimalValue;
                            tmp.optimalSet = tmp.forwardConnections[0].optimalSet;
                        }
                        //optimal-value(previous node) < definitive-value(this node)
X.                      else if (tmp.forwardConnections[0].optimalValue < tmp.definitiveValue)
                        {
                            tmp.optimalValue = tmp.definitiveValue;
                            tmp.optimalSet = tmp.raisinsIncluded;
                        }
                    }
XI.                 else if(tmp.forwardConnections.Count > 1)
                    {
                        HashSet<Raisin> tmpSet = new HashSet<Raisin>();
XII.                    for (int frwC = 0; frwC < tmp.forwardConnections.Count; frwC++)
                        {
                            //find all Raisins that are included in the optimal sets
                            //from the predecessors
XIII.                       foreach (Raisin item in tmp.forwardConnections[frwC].optimalSet)
                            {
                                //if it wasn't in one of the other sets...
XIV.                            if (!tmpSet.Contains(item))
                                {
                                    //Add it to the set
XV.                                 tmpSet.Add(item);
                                }
                            }
                        }
                        //calculate the REAL optimal value till this Raisin
                        float realOptimalValue = 0f;
XVI.                    foreach (Raisin item in tmpSet)
                        {
XVII.                       realOptimalValue += item.value;
                        }
                        //compare the REAL optimal-value with the Absolute-value
XVIII.                  if(realOptimalValue >= tmp.definitiveValue)
                        {
                            //take all of the optimal values
                            tmp.optimalSet = tmpSet;
                            tmp.optimalValue = realOptimalValue;
                        }
XIX.                    else if(realOptimalValue < tmp.definitiveValue)
                        {
                            //take all of the raisins that are connected to this one
                            tmp.optimalSet = tmp.raisinsIncluded;
                            tmp.optimalValue = tmp.definitiveValue;
                        }
                    }
                }
            }
```

Ergebnis des Algorithmus

Der Algorithmus ist kein heuristischer Algorithmus, sondern ein deterministischer Algorithmus. Deswegen findet er auch immer eine eindeutige und optimale Lösung.

Laufzeit des Algorithmus

Datei (*.txt)	Laufzeit
Kreis-8	00:00:00.0040086
Quadrat-6	00:00:00.0028088
Quadrat-8	00:00:00.0026844
Quadrat-13	00:00:00.0031217
Zufall-7	00:00:00.0022988
Zufall-40	00:00:00.0022743
Zufall-100	00:00:00.0028792
200	00:00:00.0030083
250	00:00:00.0030613
500	00:00:00.0150364
750	00:00:00.0430964
1000	00:00:00.1025181
10000	00:01:57.2258690

Testsystem:

Betriebssystem: Windows 10

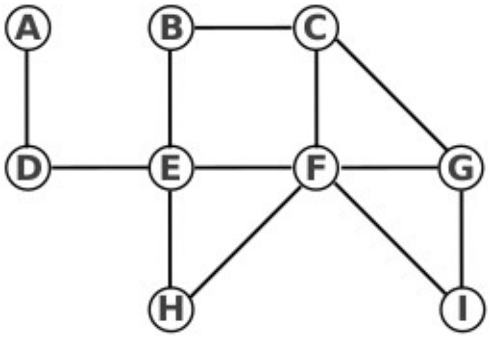
Prozessor: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, 2601 MHz, 4 Kern(e)

RAM: 16GB

Aufgabe 2: Rechtsrum in Rechthausen

Eine Statistik der Polizei von Rechthausen hat gezeigt, dass eine Vielzahl von Autounfällen durch in korrektes Linksabbiegen verursacht wird. Der Bürgermeister von Rechthausen hat deshalb beschlossen, das Linksabbiegen per Dekret zu untersagen. Autofahrer sollen in Rechthausen nur noch geradeaus fahren oder rechts abbiegen dürfen. Diese Entscheidung führt zu viel Unmut der Bewohner von Rechthausen. Sie befürchten, sich dann in der Stadt nicht mehr zurechtzufinden.

Du möchtest Abhilfe schaffen und ein Navigationssystem erstellen, das nur Routen vorschlägt, bei denen nicht links abgebogen werden muss. Dein System erhält als Eingabe ein Straßenverzeichnis. Dieses besteht aus einer MengeV von Kreuzungen mit ihren jeweiligen Koordinaten sowie der Menge E von den jenigen Paaren von Kreuzungen, die direkt durch Straßenabschnitte verbunden sind. Da Rechthausen eine moderne Stadt ist, haben die Straßen keinerlei Kurven. Das Straßennetz von Rechthausen könnte zum Beispiel so aussehen (alles, was in Rechthausen als Kreuzung angesehen wird, ist durch einen Buchstaben gekennzeichnet):



Aufgabe

1. In der üblichen Situation, in der sich zwei Straßen im rechten Winkel kreuzen,ist es klar, was unter Linksabbiegen zu verstehen ist. Formuliere eine allgemeinere Definition des Linksabbiegens. Sie soll auch dann benutzt werden können, wenn die Anzahl d der Richtungen, aus denen man sich einer Kreuzung nähern kann, nicht vier ist oder der Winkel zwischeneinerRichtungundderRichtungunmittelbarrechtsvonihrnichtimmer90◦ ist. Am Wichtigsten ist, dass die Definition in jeder Situation angewendet werden kann und eine eindeutige Entscheidung herbeiführt, ob es sich beim Fahren von einem Straßenabschnitt über eine Kreuzung in einen nächsten Straßenabschnitt um Linksabbiegen handelt oder nicht. Versuche außerdem, mit deiner Definition die ursprüngliche Bedeutung des Linksabbiegens als besonders unfallträchtiges Verkehrsmanöver zu erhalten. Deine

Definition soll insbesondere in den Fällen d = 1 (das Ende einer Sackgasse) und d = 2 sinnvolle Antworten liefern. Im obigen Beispiel sollten beispielsweise die Routen DADEF sowie GCB und FEBC erlaubt sein, die Routen DEB und IGF dagegen nicht.

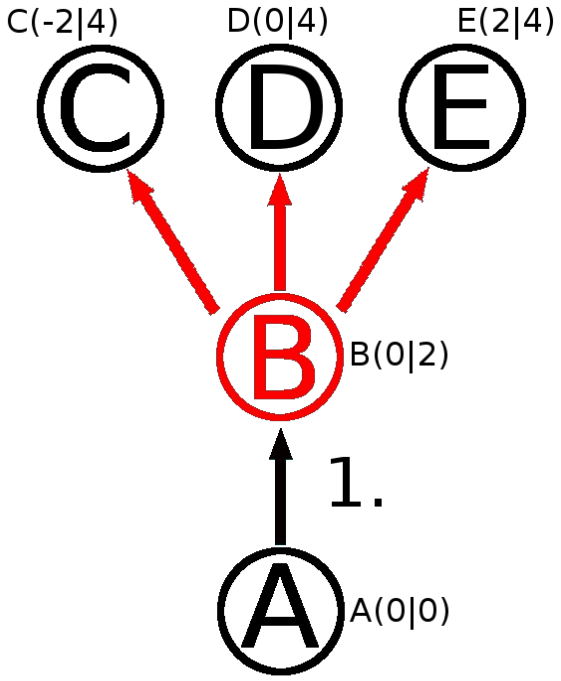
2. Veranschauliche deine Definition durch geeignete Beispiele.
3. Schreibe ein Programm, das einen kürzesten Weg ohne Linksabbiegen von einer Startkreuzung S zu einer Zielkreuzung T auf beliebigen Straßen findet, falls T von S aus überhaupt erreicht werden kann. Dabei sollst du die Weglänge auf zwei verschiedene Arten messen: zum einen nach der Anzahl, zum anderen nach der Gesamtlänge der auf dem Weg durchfahrenen Straßenabschnitte. Im obigen Beispiel: Was ist ein kürzester Weg ohne Linksabbiegen von A nach C? Und von H nach A?
4. Erweitere dein Programm so, dass es für ein gegebenes Straßenverzeichnis ermittelt, ob jede Kreuzung von jeder anderen Kreuzung aus ohne Linksabbiegen erreicht werden kann.
5. Erweitere dein Programm so, dass es ein Paar von Start- und Zielkreuzung ermittelt, für das das Verbot des Linksabbiegens die Weglänge um den größtmöglichen Faktor erhöht. Wende auch hier beide Weglängenmaße an.
6. Wende dein Programm auf die Beispiele an, die du auf der BWINF-Webseite zu dieser Runde findest.

Definition des Linksabbiegens

Meine Definition für das Linksabbiegen beinhaltet einen allgemeinen Fall und drei Sonderfälle.

Allgemeiner Fall:

Wenn in Rechthausen ein Fahrer an einer Kreuzung losfährt, ist es ihm erlaubt, in alle Richtungen zu fahren. Erst wenn er eine Kreuzung hinter sich gelassen hat, lässt sich ein Linksabbiegen bestimmen. Um das Linksabbiegen zu bestimmen, werden zwei Vektoren gebildet. Der eine zeigt von der letzten Kreuzung, an der der Fahrer gewesen, ist auf die, an der er im Moment ist.
Der zweite zeigt von der Kreuzung, an der der Fahrer steht, zu der Kreuzung, zu der er als nächstes fahren möchte.
Das Kreuzprodukt von Vektoren liefert einen Vektor, der senkrecht auf den beiden Vektoren steht. Deswegen wird das Kreuzprodukt sowie der Winkel zwischen diesen beiden Vektoren bestimmt. Da in Rechthausen alle Vektoren in der Ebene liegen, beträgt die Z-Komponente 0. Deswegen unterscheidet sich nur die Z-Komponente des Kreuzproduktes von 0.
Ist das Kreuzprodukt der beiden Vektoren >= 0, ist es als Linksabbiegen anzusehen. Ist es der Fall, dass alle gebildeten Werte für die Kreuzprodukte > 0 betragen, dann tritt der 3. Sonderfall in Kraft.
Wenn ein Fahrer in dem Beispiel '1' von der Kreuzung 'A' zu der Kreuzung 'B' fährt, gibt es nun drei Kreuzungen, die ihm zur Auswahl stehen. Welche von diesen sind nun aber als Rechtsabbiegen zu betrachten?



Beispiel 1

von\zu	A (0 | 0)	B (0 | 2)	C (-2 | 4)	D (0 | 4)	E(2 | 4)
A (0 | 0)	(0 | 0)	(0 | 2)	---	---	---
B (0 | 2)	(0 | -2)	(0 | 0)	(-2 | 2)	(0 | 2)	(2 | 2)
C (-2 | 4)	---	(2 | -2)	(0 | 0)	---	---
D (0 | 4)	---	(0 | -2)	---	(0 | 0)	---

E(2 | 4)	---	(-2 | -2)	---	---	(0 | 0)

Kreuzprodukt zweier Vektoren: $\vec{a} \times \vec{b} = \begin{pmatrix} a_x \\ a_y \end{pmatrix} \times \begin{pmatrix} b_x \\ b_y \end{pmatrix} = a_x \cdot b_y - b_x \cdot a_y$

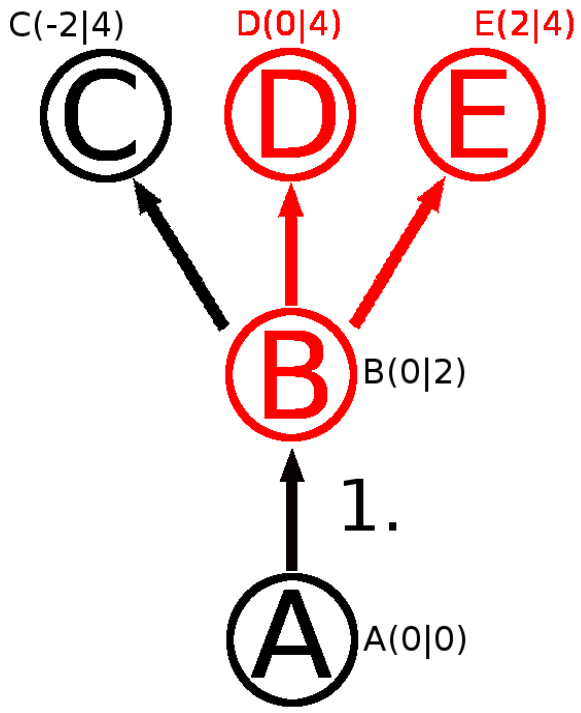
$$\overrightarrow{AB} \times \overrightarrow{BC} = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \times \begin{pmatrix} -2 \\ 2 \end{pmatrix} = 0 \cdot 2 - (-2) \cdot 2 = 4$$

$$\overrightarrow{AB} \times \overrightarrow{BD} = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \times \begin{pmatrix} 0 \\ 2 \end{pmatrix} = 0 \cdot 2 - 0 \cdot 2 = 0$$

$$\overrightarrow{AB} \times \overrightarrow{BE} = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \times \begin{pmatrix} 2 \\ 2 \end{pmatrix} = 0 \cdot 2 - 2 \cdot 2 = -4$$

Daraus ergibt sich, dass der Weg 'A' → 'B' → 'C' nicht als Rechtsabbiegen anzusehen ist, weil sein Kreuzprodukt > 0 beträgt.

Die beiden Wege 'A' → 'B' → 'D' und 'A' → 'B' → 'E' sind beide als Rechtsabbiegen anzusehen, weil das Kreuzprodukt >= 0 beträgt.

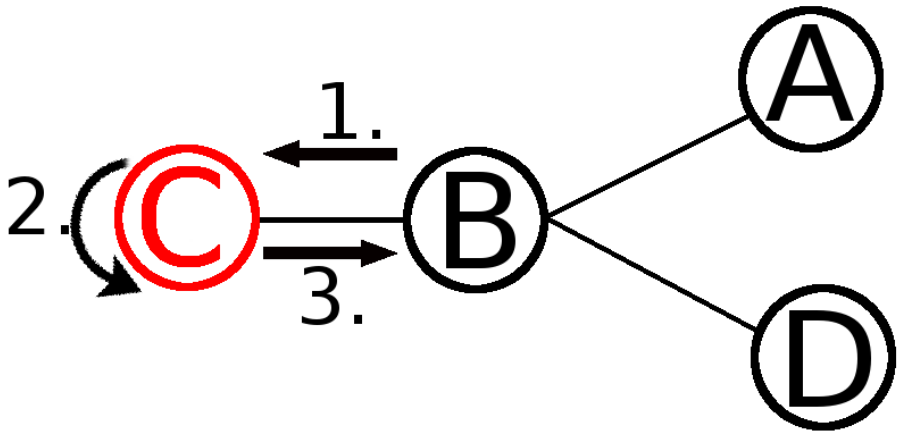


Beispiel 1

1. Sonderfall:

Der erste Sonderfall trifft zu, wenn eine Kreuzung genau eine Straße hat, die von ihr abgeht (eine Kreuzung nur mit einer anderen Kreuzung verbunden ist). Dann darf der Fahrer an der Kreuzung, an der es nur einen Weg zurück gibt, wenden.

In dem 'Beispiel 2' wäre die Kreuzung 'C' ein Sonderfall 1. Wenn die Startkreuzung 'B' gewählt wird und nun zu der Kreuzung 'C' gefahren wird, landet der Fahrer in einer Sackgasse. Jetzt soll es möglich sein, an 'C' (der Sackgasse) zu 'wenden' und weiter nach 'B' zurückzufahren. So ist es möglich, sich in einer Sackgasse einmal zu drehen.

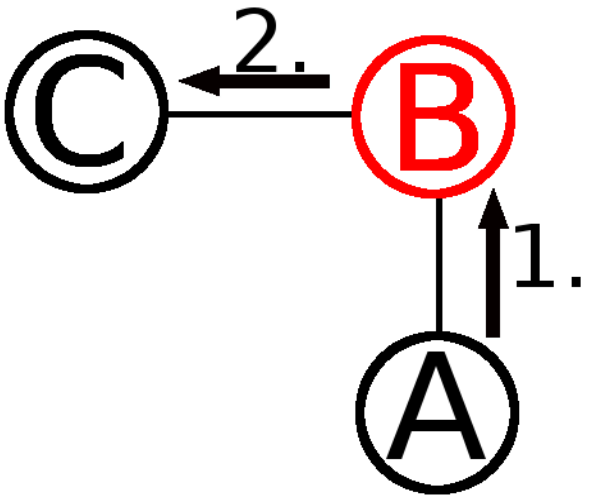


Beispiel 2

2. Sonderfall:

Der zweite Sonderfall tritt ein, wenn eine Kreuzung nur zwei Verbindungen (mit zwei anderen Kreuzungen verbunden ist) hat. In diesem Fall ist das Linksabbiegen erlaubt, da es sonst keine Möglichkeiten für den Fahrer gibt.

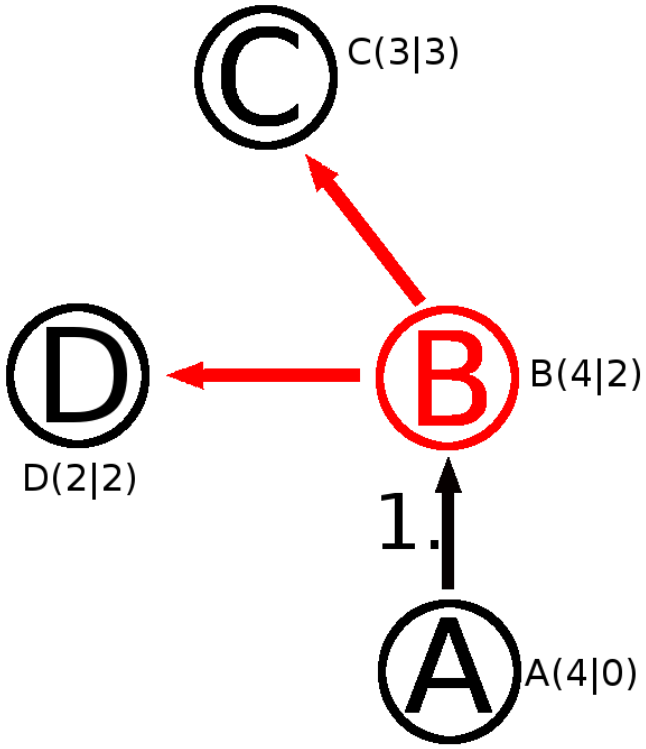
In dem 'Beispiel 3' würde dieser Sonderfall zutreffen, wenn der Fahrer an Kreuzung 'A' losfährt und an Kreuzung 'B' ankommt. Hier würde es nur den Weg nach links geben. Diesen Weg darf der Fahrer dann auch einschlagen, da es sonst keine Möglichkeit gibt, von dieser Kreuzung weiter zu kommen.



Beispiel 3

3. Sonderfall:

Der dritte Sonderfall tritt ein, wenn eine Kreuzung nur Verbindungen zu anderen Kreuzungen hat, bei denen die beiden Vektoren (letzte Kreuzung → jetzigem Standpunkt | jetziger Standpunkt → nächste Kreuzung) ein Kreuzprodukt > 0 bilden. In diesem Fall wird der Winkel zwischen den Vektoren ermittelt und die Kreuzung mit dem kleinsten Winkel als einzige Möglichkeit des Weiterfahrens festgelegt.



Beispiel 4

von\zu	A (4 | 0)	B (4 | 2)	C (3 | 3)	D (2 | 2)
A (4 | 0)	(0 | 0)	(0 | 2)	---	---
B (4 | 2)	(0 | -2)	(0 | 0)	(-1 | 1)	(-2 | 0)
C (3 | 3)	---	(1 | -1)	(0 | 0)	---
D (2 | 2)	---	(2 | 0)	---	(0 | 0)

In dem 'Beispiel 4' sind die Kreuzprodukte aller Verbindungen < 0, wenn der Fahrer von 'A' kommt. Deswegen werden die Winkel zwischen den Vektoren verglichen.

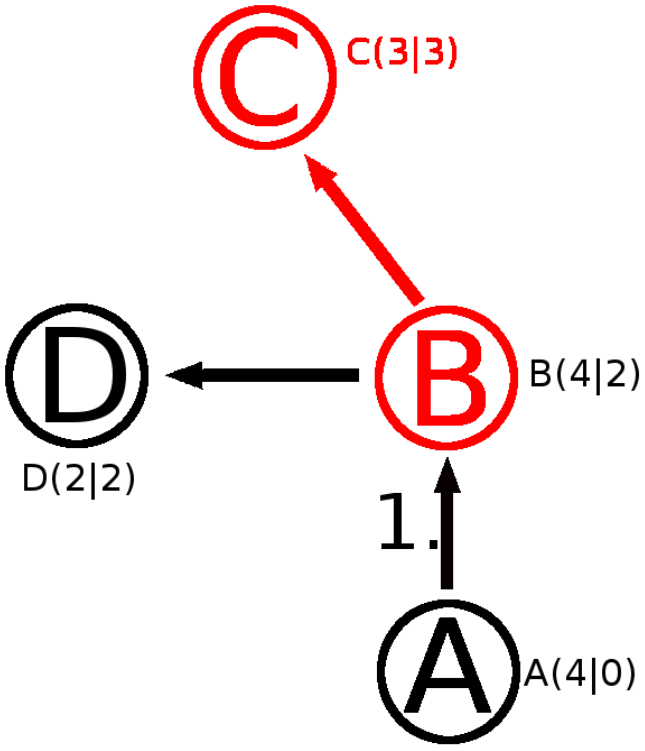
$$\overrightarrow{AB}=\overrightarrow{OB}-\overrightarrow{OA}=\begin{pmatrix}0\\2\end{pmatrix};\ \overrightarrow{BC}=\overrightarrow{OC}-\overrightarrow{OB}=\begin{pmatrix}-1\\1\end{pmatrix};\ \overrightarrow{BD}=\overrightarrow{OD}-\overrightarrow{OC}=\begin{pmatrix}-2\\0\end{pmatrix}$$

$$\textit{Winkel zwischen zwei Vektoren}:\cos(\varphi)=\frac{\vec{u}\circ\vec{v}}{|\vec{u}||\vec{v}|}$$

$$\overrightarrow{AB}\circ\overrightarrow{BC}=\begin{pmatrix}0\\2\end{pmatrix}\cdot\begin{pmatrix}-1\\1\end{pmatrix}=0\cdot-1+2\cdot1=0+2=2$$
$$|\overrightarrow{AB}||\overrightarrow{BC}|=\sqrt{0^2+2^2}\cdot\sqrt{(-1)^2+1^2}=\sqrt{2}\cdot2$$
$$\cos(\varphi)=\frac{\overrightarrow{AB}\circ\overrightarrow{BC}}{|\overrightarrow{AB}||\overrightarrow{BC}|}=\frac{2}{\sqrt{2}\cdot2}$$
$$\varphi=\arccos\left(\frac{2}{\sqrt{2}\cdot2}\right)$$
$$\underline{\varphi=45\,°>0}$$

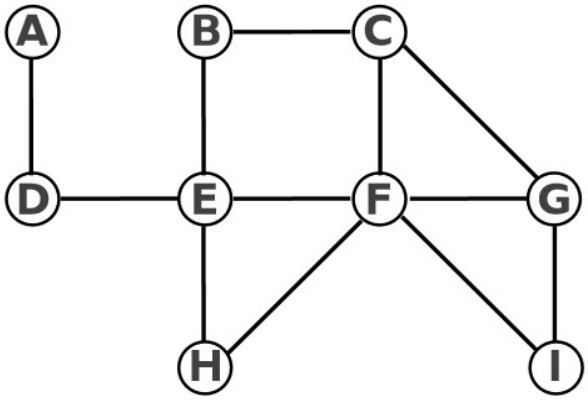
$$\cos(\varphi)=\frac{\overrightarrow{AB}\circ\overrightarrow{BD}}{|\overrightarrow{AB}||\overrightarrow{BD}|}=\frac{0}{\sqrt{2}\cdot2}$$
$$\underline{\varphi=90\,°>0}$$

Jetzt wird die Verbindung gewählt, deren Vektoren den kleinsten Winkel bilden.
In diesem Fall ist es die Verbindung 'A' → 'B' → 'C'. Diese Verbindung ist die einzig erlaubte Verbindung, wenn der Fahrer von 'A' bei 'B' ankommt.



Beispiel 4

Im 'Beispiel 5' welches auch auf der Wettbewerbsseite gefunden werden kann sollen die Routen D → A → D → E → F, G → C → B und F → E → B →C erlaubt sein, die Routen D → E → B und I → G → F hingegen nicht.



Beispiel 5

<u>D → A → D → E → F:</u>

Wenn der Fahrer von 'D' nach 'A' fährt, kommt er in eine Sackgasse und somit in den 2. Sonderfall. Deswegen kann er dort drehen und zurück zu 'D' fahren. Bei 'D' angekommen wird der zweite Sonderfall ausgelöst und deswegen darf er nach links zu 'E' abbiegen und anschließend weiter nach 'F' geradeaus fahren.

<u>G → C → B:</u>

Wenn 'G' als Startpunkt angenommen wird, darf der Fahrer erst einmal zu allen anliegenden Kreuzungen fahren.
Wenn er also zu 'C' fährt, darf er zu 'B' weiter geradeaus fahren.

<u>F → E → B →C:</u>

Dieser Fahrweg ist erlaubt, weil jedes Abbiegen im rechten Winkel nach rechts stattfindet. Deswegen ist das Kreuzprodukt immer < 0 und dieser Weg erlaubt.

<u>D → E → B:</u>

Wenn der Fahrer von 'D' über 'E' zu 'B' fährt, ist dies nicht erlaubt, weil er von 'E' zu 'B' die Möglichkeit hat nach rechts zu 'H' abzubiegen oder geradeaus nach 'F' zu fahren und das Kreuzprodukt von D → E → B > 0 beträgt.

<u>I → G → F:</u>

Für den Fall, dass der Fahrer bei 'I' losgefahren und bei 'G' angekommen ist, tritt der 3.Sonderfall in Kraft, da es nur Vektoren gibt, die ein Kreuzprodukt > 0 bilden. Dem Fahrer ist es nur gestattet, nach 'C' zu fahren, da der Winkel C → G und G → I kleiner ist als zwischen F → G und G → I.

Lösungsidee

Meine Lösungsidee unterteilt sich in mehrere einzelne Lösungsschritte und basiert auf meiner Definition für das Rechtsabbiegen.

1. Zunächst soll das Straßenverzeichnis aus einer Datei ausgelesen werden. Dabei sollen alle Kreuzungen mit deren Koordinaten und die Verbindungen zwischen den Kreuzungen jeweils gespeichert werden.
   Das hierbei entstehende Straßenverzeichnis kann auch als Graph angesehen werden. Dieser Graph ist ungerichtet, weil es möglich ist in beide Richtungen zu fahren, wenn die Bedingung des Rechtsabbiegens eingehalten wird.
   Somit können in dem Graphen auch Zyklen auftreten.
   Außerdem ist der Graph als ungewichtet und auch als gewichtet anzusehen, weil die Distanz zwischen den Kreuzungen mittels Pythagoras ermittelt werden kann und so eine Wichtung entstehen kann. Als ungewichtet anzusehen ist er, wenn der kürzeste Weg nach Kreuzungsanzahl gesucht wird.
   Daraus folgt auch, dass der Graph keine negativen Kantengewichte haben kann.

2. Im nächsten Schritt soll das Rechtsabbiegen für alle Verbindungen festgelegt werden. Dies soll wie in meiner Definition für das Rechtsabbiegen geschehen und für alle Verbindungen von Kreuzungen erfolgen.

3. Anschließend soll das Start- und Zielkreuzungspaar ermittelt werden, für das das Verbot des Linksabbiegens die Weglänge um den größtmöglichen Faktor erhöht hat. Dafür soll ein unveränderter Dijkstra-Algorithmus auf alle Kreuzungen angewandt werden. Dieser soll auch wieder für beide Wegmaße verwendet werden.

4. In diesem Schritt soll der kürzeste Weg zwischen allen Kreuzungen ermittelt werden.
   Hier für soll ein angepasster Dijkstra-Algorithmus verwendet werden. Der Algorithmus soll so angepasst werden, dass eine Kreuzung erst als optimal angesehen wird, wenn alle von ihr ausgehenden Verbindungen einmal durchlaufen worden sind. Durch dieses Verfahren wird garantiert, dass jede Möglichkeit für das Rechtsabbiegen genutzt wird.
   Dieser Algorithmus soll anschließend auf alle Kreuzungen angewandt werden, damit alle kürzesten Wege von allen Kreuzungen zu allen anderen Kreuzungen gefunden werden können.
   Außerdem ist es so möglich, dabei zu ermitteln, ob alle Kreuzungen von allen anderen erreicht werden können, indem überprüft wird, ob es für jede Zielkreuzungs-/Startkreuzungskombination einen kürzesten Weg gibt.

5. Eine grafische Ausgabe des Straßenverzeichnisses soll anschließend erfolgen, in der eine Startkreuzung und eine Zielkreuzung bestimmt werden kann und der kürzeste Weg zwischen diesen dargestellt werden soll. Die Darstellung soll einmal den kürzesten Weg nach der Distanz darstellen und einmal den kürzesten Weg nach der Anzahl der Kreuzungen.

Umsetzung

Die Lösungsidee wurde in c# implementiert. In der Dokumentation der Umsetzung werden nur die in der Lösungsidee benannten 5 Schritte näher erläutert.

1. Die Datei auslesen:

Bei dem Auslesen der Datei wird ein Dictionary erstellt. Dieses Dictionary speichert als Key den Namen der Kreuzung und speichert als Value eine Instanz der Klasse 'Crossing'.
Jede Instanz dieser Klasse wird bei dem Auslesen der Datei initialisiert. Dabei werden die Koordinaten, der Name, die Nummer der Kreuzung und alle Verbindungen zu dieser Kreuzung gesetzt. Die Nummer der Kreuzung ist ein Integer, der für jede Kreuzung um eins erhöht wird und so alle Kreuzungen nach dem Platz in der Eingabedatei nummeriert.
Die Klasse 'Crossing' wird verwendet, um eine Kreuzung mit allen ihren Verbindungen, Koordinaten und Eigenschaften zu speichern.
Diese Klasse 'Crossing' hat 8 Eigenschaften und 4 Methoden.
Es werden jeweils die Koordinaten der Kreuzung gespeichert [1.](Referenz auf Quellcode S. 10), der Name der Kreuzung [2.], die wievielte Kreuzung es ist (es wird beim Einlesen der Datei für jede Kreuzung mitgezählt) [3.], alle angrenzenden Kreuzungen [4.], die Entfernungen zu allen angrenzenden Kreuzungen [5.] und die Kreuzungen, die von einer bestimmten Kreuzung aus als Rechtsabbiegen anzusehen sind [6.].
Diese Verbindungen werden in einem Dictionary gespeichert. Als momentaner Standpunkt wird hierbei die Kreuzung genommen, bei der dieses Dictionary gespeichert wurde. Als Key soll der Name der Kreuzung angegeben werden, die als letztes von dem Fahrer besucht wurde. Der Value zu diesem Key ist dann eine Liste mit allen Verbindungen, die als Rechtsabbiegen zu betrachten sind.
Die Abstände zu allen angrenzenden Kreuzungen werden in einem Dictionary gespeichert. So kann als Key der Name der Zielkreuzung eingegeben werden und als Value wird der dazu gehörige Abstand als Float gespeichert.
Mit den erlaubten Kreuzungen durch das Verbot des Linksabbiegens ist es ähnlich. Sie werden auch in einem Dictionary gespeichert. Als Key wird der Name der Vorgängerkreuzung angegeben und als Value wird eine Liste mit allen Kreuzungen, die von der Vorgängerkreuzung über die aktuelle Kreuzung als Rechtsabbiegen anzusehen sind, ausgegeben.
CalculateLengthToAllConnectedCrossings [7.] berechnet die Distanz zu allen anderen Kreuzungen,

die mit dieser Kreuzung verbunden sind.

Die Methode FindAllowedConnectionsForThisCrossing [8.] ermittelt alle Kreuzungen, die erlaubt sind trotz des Verbot des Linksabbiegens und speichert sie in der Eigenschaft allowedConnections [6.].

```
1.  readonly public float x;
    readonly public float y;
2.  readonly public string name;
    //the number of this crossing in the dictionary
3.  readonly public int number;
    // All connections to this crossing
4.  public List<Crossing> connections = new List<Crossing>();
    // the distances to all connections to this crossing
    // KEY(Char) -- the name of the crossing distance is measured to
    // VALUE(double) -- the distance to crossing
5.  public Dictionary<String, float> distances = new Dictionary<String, float>();
    // The connections that are allowed to drive
    // Key(Char) of the Dictionary is the name of the Crossing we are coming from
    // List<Crossing> are all allowed directions to drive to coming from the key(Char)
6.  public Dictionary<String, List<Crossing>> allowedConnections = new Dictionary<String,
    List<Crossing>>();
    //Finds all allowed connections to crossings from this node
    //calculates the distance to all Crossings connected to this crossing
    //AND for all crossings that are allowed to drive
7.  public void CalculateLengthToAllConnectedCrossings()

8.  public void FindAllowedConnectionsForThisCrossing()
```

2. Bestimmen des Rechtsabbiegens

Um alle erlaubten Kreuzungen zu finden, wird zunächst eine For-Schleife über alle Verbindungen, die es zu dieser Kreuzung gibt, ausgeführt [I.] (Referenz auf Quellcode S. 11-12).
Hier werden auch die Sonderfälle beachtet.
Wenn es nur eine Verbindung gibt [II.], tritt der erste Sonderfall in Kraft und diese Kreuzung wird als einzige Möglichkeit des Weiterfahrens betrachtet.
Wenn es genau zwei Verbindungen gibt [III.], tritt der zweite Sonderfall in Kraft und die Kreuzung, welche nicht als letztes von den beiden Kreuzungen besucht wurde, wird als einzige Weiterfahr-Möglichkeit ausgewählt.
Wenn es mehr als zwei Verbindungen gibt [IV.], wird nach dem allgemeinen Fall gehandelt.
Deswegen wird noch eine For-Schleife über alle Verbindungen zu dieser Kreuzung ausgeführt [V.].
Es wird immer die Iteration übersprungen, wenn die letzte Kreuzung und die nächste Kreuzung die gleiche Kreuzung sind.
Andernfalls wird zunächst das Kreuzprodukt zwischen den beiden Vektoren gebildet und der Winkel zwischen den Vektoren berechnet [VII.]. Die Kreuzung wird sich gemerkt falls dieser Winkel der bisher kleinste Winkel in dieser Iteration ist [VIII.]. Um später, falls es keine Verbindung gibt, die als Rechtsabbiegen anzusehen ist, die Verbindung mit dem kleinsten Winkel als einzige Möglichkeit festzulegen [XI.].
Wenn das Kreuzprodukt zwischen den beiden Vektoren in dieser Iteration der inneren For-Schleife <= 0 beträgt [IX.], wird diese Verbindung temporär gespeichert, da die Verbindung als Rechtsabbiegen anzusehen ist.
Wenn die innere For-Schleife zum Ende gekommen ist, wird überprüft, ob eine Kreuzung gefunden wurde, die als Rechtsabbiegen zu betrachten ist, wenn nicht wird der dritte Sonderfall angewandt [XI.].

```
public void FindAllowedConnectionsForThisCrossing(){
                //loop over all the connections from this Crossing
                //--> coming from all possible Crossings to this one
    I.          for (int coming = 0; coming < connections.Count; coming++)
                {
                    //List with all allowed connections
                   //from a specified crossing to a specified Crossing over this Crossing
                    List<Crossing> tmpAllowedCrossings = new List<Crossing>();
                    //the name of the crossing we're coming from
                    String comingName = connections[coming].name;
                    //the greatest angle coming from a specified crossing
                    //over this crossing to the destination crossing
                    //if no allowed connection was found
                    //--> take the one with the greates angle
                    double smallestAngle = 360;
```

```
                Crossing smallestAngleGoingTo = new Crossing("", 0, 0, 0);

                //if one street is connected with this crossing
                // --> go back the way you are coming from
II.             if (this.connections.Count == 1)
                {
                      allowedConnections.Add(comingName, new List<Crossing> { connections[coming] });
                }
                //if two streets are connected with this crossing
III.            else if (this.connections.Count == 2)
                {
                    if(coming == 0) allowedConnections.Add(comingName, new List<Crossing>
    { connections[1] });
                    else if (coming == 1) allowedConnections.Add(comingName, new List<Crossing>
    { connections[0] });
                }
                //if more then two streets are connected with this crossing
IV.             else if (this.connections.Count > 2)
                {
                    //loop over all the connections from this Crossing
                    //--> going to all possible crossings
V.                  for (int going = 0; going < connections.Count; going++)
                    {
                          //if the node we're coming from is the same we're going to
VI.                       if (coming == going) continue;

                          //Vector pointing from the STARTING crossing
                          //to this crossing
                          Vector comingVec = new Vector((this.x - connections[coming].x),
                                                        (this.y - connections[coming].y));
                          //Vector pointing from THIS crossing
                          //to the crossing that must be checked
                          Vector goingVec = new Vector((connections[going].x - this.x),
                                                       (connections[going].y - this.y));
                          //calculate the crossproduct and the angle between the Vectors
VII.                      double crossProduct = Vector.CrossProduct(comingVec, goingVec);
                          double angleBetween = Vector.AngleBetween(comingVec, goingVec);

                          //remember the one with the smallest angle
VIII.                     if (angleBetween < smallestAngle)
                          {
                                smallestAngle = angleBetween;
                                smallestAngleGoingTo = connections[going];
                          }

                       //it's is an allowed direction(turn right)
IX.                    if (crossProduct <= 0)
                       {
                              //add this direction to the allowed List
                              tmpAllowedCrossings.Add(connections[going]);
                       }
                              //it isn't an allowed direction(turn left)
                       else continue;
                }
                //if (a) possible direction(s) was(were) found
X.              if (tmpAllowedCrossings.Count > 0)
                {
                     //adds the connections to the dictionary
                     allowedConnections.Add(comingName, tmpAllowedCrossings);
                }
                //if all connections are turning left
XI.             else if (tmpAllowedCrossings.Count == 0)
                {
                    //add the one with the smallest angle
                    allowedConnections.Add(comingName, new List<Crossing> { smallestAngleGoingTo });
                }
                //if some Bug occured an you can't go anywhere!
                else Console.WriteLine("Coming From " + comingName + " You are at " + this.name + "
    Something went wrong! You can't go anywerhe!!!");
            }
    }
```

3. Ermitteln des größtmöglichen Faktors, um den die Weglänge erhöht wird:

Um ein Paar von Start- und Zielkreuzung zu ermitteln, für das das Verbot des Linksabbiegens die Weglänge um den größtmöglichen Faktor erhöht, wird wieder für alle Kreuzungen der Dijkstra-Algorithmus angewandt. Dieses Mal wird der Algorithmus aber unverändert verwendet und auf alle Kreuzungen in dem Straßenverzeichnis angewandt. Außerdem wird er wieder für die beiden unterschiedlichen Wegmaße benutzt, um den größtmöglichen Faktor für beide Maße ermitteln zu können. Im Anschluss wird die Weglänge von jedem Start- und Zielkreuzungspaar mit Linksabbiegen und ohne Linksabbiegen und mit den beiden Wegmaßen verglichen.
Hierfür wird über das Jagged-Array mit allen Werten für die kürzesten Wege mit einer verschachtelten For-Schleife iteriert [I.] (Referenz auf Quellcode S. 13) und die Werte für alle kürzesten Wege verglichen [III.][IV.]. Jedes Mal, wenn das Verhältnis der beiden Werte für den kürzesten Weg größer als das bisher gespeicherte Verhältnis ist, wird sich das neue Start- und Zielkreuzungspaar mit deren Verhältnis gemerkt.
Außerdem wird während dieser verschachtelten For-Schleife auch überprüft, ob jede Kreuzung von jeder anderen erreicht werden kann. Falls es passiert, dass ein Wert aus dem Jagged-Array gleich dem maximalen Wert ist [II.], ist es nicht möglich, diese beiden Kreuzungen von einander zu erreichen.

```
I.   foreach (KeyValuePair<String, Crossing> item in this.crossings){
                    foreach (KeyValuePair<String, Crossing> item2 in this.crossings){
                        //if a pair of crossings doesnt have a shortest path
                        //--> Not all crossings are reachable from all crossings
II.                     if (this.distances[i][j] == float.MaxValue / 2)
                            this.allNodesAreReachable = false;
                        //check if a new greatest factor was found
                        //measurement -- DISTANCE
III.                    if (this.distances[i][j].Item1 / this.distancesWITH_LEFT[i][j].Item1 >
    factorMeasurementCrossing)
                        {
                            factorMeasurementDistance = this.distances[i][j] /
    this.distancesWITH_LEFT[i][j];
                            factorMeasurementDistanceNames = "From " + item.Key + " To " +
    item2.Key + " Was Increased by the factor of ";
                        }
                        //check if a new greatest factor was found
                        //measurement -- CROSSINGS
IV.                     if (this.distancesCrossingMeasured[i][j].Item1 /
    this.distancesCrossingMeasuredWITH_LEFT[i][j].Item1 > factorMeasurementCrossing)
                        {
                            factorMeasurementCrossing = this.distancesCrossingMeasured[i][j] /
    this.distancesCrossingMeasuredWITH_LEFT[i][j];
                            factorMeasurementCrossingNames = "From " + item.Key + " To " +
    item2.Key + " Was Increased by the factor of ";
                        }
                        j++;
                    }
                    j = 0; i++;
                }
```

4. Ermitteln der kürzesten Wege mit einem angepassten Dijkstra-Algorithmus:

Um den kürzesten Weg zu finden, wird ein angepasster Dijkstra-Algorithmus verwendet. Da Dijkstra nur alle kürzesten Wege von einer Kreuzung zu allen anderen findet, muss der Algorithmus auf alle Kreuzungen angewandt, werden. Weswegen Dijkstra so oft angewandt werden muss, wie es Kreuzungen in der Karte gibt.
Der Algorithmus geht nach dem Prinzip vor, dass immer der Weg weiter geführt wird, welcher im Moment die geringsten Kosten aufweist.
Diese Knoten werden in einer SortedList <distance, Node> gespeichert, um effektiv danach suchen zu können.
Anstatt das jede Kreuzung als optimal angenommen wird, wenn sie aus der Liste genommen wurde und bearbeitet wurde, wird eine Kreuzung erst als optimal angenommen, wenn alle abgehenden Verbindungen einmal genutzt wurden. Somit wird sicher gestellt, dass alle Kreuzungen von allen angrenzenden Kreuzungen erreicht werden. Somit ist sicher, dass alle Möglichkeiten für das Rechtsabbiegen genutzt werden und dadurch alle kürzesten Wege gefunden werden.
Zunächst iteriert eine For-Schleife über alle Verbindungen von der Kreuzung, für die alle kürzesten Wege gesucht werden (Startkreuzung) [I.] (Referenz auf Quellcode S. 16-17). In der Schleife wird die Distanz zu der momentanen Kreuzung gesetzt [II.] und eine unabhängige Kopie der Kreuzung erstellt [III.] und initialisiert. Bei dem Initialisieren werden die Distanz und der Vorgänger der Kreuzung gesetzt. Der Vorgänger wird benötigt, um ein Rechtsabbiegen bestimmen zu können.
Die unabhängige Kopie dient dazu, den Prozess parallelisieren zu können. Im nächsten Schritt der Schleife wird diese Verbindung von der Startkreuzung aus als 'visited' markiert [IV.], damit wenn alle Verbindungen von der Startkreuzung aus besucht wurden, diese als bearbeitet markiert wird.
Nun wird nur noch die neu entstandene Kreuzung der Liste mit allen noch zu untersuchenden Kreuzungen hinzugefügt [V.].
Wenn diese For-Schleife zum Ende gekommen ist, wird überprüft, ob alle Verbindungen der Startkreuzung genutzt wurden und anschließend wird die Startkreuzung als bearbeitet markiert [VI.]. Als nächster Schritt wird eine While-Schleife solange ausgeführt, bis keine Kreuzungen mehr untersucht werden müssen [VII.].
In jeder Iteration dieser While-Schleife wird die erste Kreuzung von allen noch abzuarbeitenden Kreuzungen als die momentan zu untersuchende Kreuzung gesetzt [VIII.].
Nun wird wieder eine For-Schleife über alle Verbindungen dieser Kreuzung, die als Rechtsabbiegen

bestimmt wurden, ausgeführt [IX.]. Durch diese Schleife werden nur die Kreuzungen untersucht, die von der vorhergehenden Kreuzung aus als Rechtsabbiegen anzusehen sind.
Der erste Schritt ist es die Zielkreuzung festzulegen [X.]. Wenn diese Kreuzung aber schon als abgearbeitet markiert wurde, wird diese Iteration der For-Schleife übersprungen [XI.]. Ansonsten wird die Zielkreuzung zu den genutzten Verbindungen der momentanen Kreuzung hinzugefügt [XII.]. Im Anschluss wird noch überprüft, ob ein neuer kürzerer Weg zu der Zielkreuzung gefunden wurde [XIII.].
Wenn der Weg kürzer ist als der bisherige, werden die Distanz und die Vorgänger der Zielkreuzung neu gesetzt [XIV.]. Anschließend wird die Zielkreuzung nur noch der Liste mit den noch zu überprüfenden Kreuzungen hinzugefügt [XV.].
Wenn hingegen die Abfrage, ob ein neuer kürzester Weg gefunden wurde, nicht zutrifft [XVI.], wird eine unabhängige Kopie der Zielkreuzung erstellt [XVII.] und mit dem Vorgänger von der momentan zu untersuchenden Kreuzung initialisiert [XVIII.].
Diese Kopie wird erstellt, damit wirklich alle Möglichkeiten des Abbiegens genutzt werden. Im normalen Dijkstra-Algorithmus wäre es in diesem Fall so, dass die Kreuzung als optimal angenommen wird. Dies kann aber durch das Rechtsabbiegen nicht so einfach angenommen werden. Ein Knoten darf nicht mehr weiter überprüft werden, wenn alle Verbindungen von ihm einmal genutzt wurden. Da nur auf diese Weise garantiert werden kann, dass jede Kreuzung von jeder möglichen Richtung besucht wurde und so alle Möglichkeiten für das Rechtsabbiegen überprüft wurden.
Diese Kopie wird nun noch mit den Vorgängern der momentanen Kreuzung initialisiert [XVIII.]. Es wird für die Kopie auch die bisher genutzten Verbindungen der momentanen Kreuzung übernommen [XIX.]. Zum Schluss wird die Kopie noch der Liste mit den abzuarbeitenden Kreuzungen hinzugefügt [XX.].
Als letzter Schritt der For-Schleife wird noch überprüft, ob alle Verbindungen der momentan zu überprüfenden Kreuzung genutzt wurden und wenn dies zu trifft, wird die Kreuzung als abgearbeitet markiert [XXI.].
Wenn die For-Schleife ihr Ende erreicht hat, wird die momentan abzuarbeitende Kreuzung aus der Liste mit den noch abzuarbeitenden Kreuzungen entfernt. Nach dem Beenden der While-Schleife wird nur noch ein Float-Array mit allen kürzesten Wegen von der Startkreuzung aus als Funktionswert zurückgegeben. Diese Funktionswerte setzten dann nach und nach das Jagged-Array mit allen kürzesten Weg-Distanzen von jeder Kreuzung zu jeder anderen zusammen.

```
private Tuple<float, List<string>>[] dijkstra(Crossing start)
        {
            //the List with all nodes that need to be worked off
            SortedList<float, Node> queue = new SortedList<float, Node>(new NodesComparer<float>());
            //List with all nodes in ths map
            Dictionary<string, Node> allNodes = initializeListForDijkstra();
            //the costs to reach all other nodes from this one
            Tuple<float, List<string>>[] costs = new Tuple<float, List<string>>[crossingsCnt];

            //set the cost for the startnode to zero
            allNodes[start.name].cost = 0;

                            //add all crossings connected with the start cossing
                            //--> at the start their is no turning right
                            //----> INITIALIZING THE LIST
I.                          for (int i = 0; i < start.connections.Count; i++)
                            {
                                //add the queue the nodes to work off
                                //the direct connections from the start node(no turning right from the start
    crossing)
II.                             float dist = start.distances[start.connections[i].name];
III.                            Node currentNode = allNodes[start.connections[i].name];
                                //set the dist to this node
                                currentNode.cost = dist;
                                //set the start node as the predecessor
                                currentNode.predecessor.Add(start.name);
                                //add the node to the visited nodes list from the start node
IV.                             allNodes[start.name].visitedNodes.Add(currentNode.name);
                                //add the new node reached from the start node to the queue
V.                              queue.Add(dist, currentNode);
                            }
                            //set the start node as worked off
VI.                         if(visitedNodesEqual_To_AllNodes(allNodes[start.name]))
                                allNodes[start.name].visited = true;


VII.                        while (queue.Count != 0)
                            {
                                //the current node to work with
VIII.                           Node currentNode = queue.Values[0];

                                //add all allowed(turning right) connections from the current node
IX.                             for (int i = 0;  i <
    crossings[currentNode.name].allowedConnections[currentNode.predecessor[currentNode.predecesso
    r.Count - 1]].Count; i++)
                                {
                                    //the crossing we want to go to
X.                                  Crossing nextCrossing =
    crossings[currentNode.name].allowedConnections[currentNode.predecessor[currentNode.predecesso
    r.Count - 1]][i];

XI.                                 if (allNodes[nextCrossing.name].visited ||
    currentNode.visitedNodes.Contains(nextCrossing.name))
                                        continue;


                                   //add the next Crossing to the visited ones from this node
XII.                                currentNode.visitedNodes.Add(nextCrossing.name);

                                    //checks the distance to the next node
                                    //--> if less then current value override it
                                    //--> and add the node to the queue
                                    float currentDist =
    crossings[currentNode.name].distances[nextCrossing.name] + currentNode.cost;
XIII.                               if (currentDist < allNodes[nextCrossing.name].cost)
                                    {
                                        //set predecessor and cost of the next node
XIV.                                    allNodes[nextCrossing.name].predecessor = new
    List<String>(currentNode.predecessor);
                                        allNodes[nextCrossing.name].predecessor.Add(currentNode.name);
                                        allNodes[nextCrossing.name].cost = currentDist;
```

```
                            //add the node that was the next-node in this iteration
                            //to the queue
XV.                         queue.Add(currentDist, allNodes[nextCrossing.name]);
                        }
                        //if the distance is greater
                        //--> create a new node with the greater distance(Set predecessor, and
    the new distance)
                        //--> and add this one to the list
XVI.                    else
                        {
XVII.                       Node tmp = new Node(nextCrossing.name);
                            tmp.cost = currentDist;

XVIII.                      tmp.predecessor = new List<string>(currentNode.predecessor);
                            tmp.predecessor.Add(currentNode.name);


XIX.                        tmp.visitedNodes = allNodes[nextCrossing.name].visitedNodes;

XX.                         queue.Add(currentDist, tmp);
                        }

                        //check if all connection from this node were used
XXI.            if (visitedNodesEqual_To_AllNodes(currentNode))
                        {
                            currentNode.visited = true;

                        }


                    }

                    //remove the node from the queue
XXII.                       queue.RemoveAt(0);
                }

                int o = 0;
XXIII.                  foreach (KeyValuePair<String, Node> item in allNodes)
                {
                    costs[o] = new Tuple<float, List<String>>(item.Value.cost,
    item.Value.predecessor);
                    o++;
                }
                return costs;
    }
```

Laufzeit Verbesserungen des Algorithmus

1. Der Algorithmus wurde so angepasst, dass er unabhängig ist und somit eine Parallelisierung möglich ist. Das bedeutet, dass er nur die Startkreuzung übergeben bekommt und sich eine unabhängige Kopie von allen Kreuzungen erstellt.
   Durch die Parallelisierung ist es anschließend möglich, die einzelnen Dijkstra-Algorithmen auf unterschiedliche Prozessorkerne auszulagern, und somit die Laufzeit je nach Prozessoranzahl deutlich zu verringern.
   Wenn hingegen nur ein Prozessorkern in dem System gefunden wird, wird der Prozess ohne Parallelisierung gestartet, da so nicht mehrere Threads gestartet werden müssen. Da das Eröffnen von Threads extra Zeit benötigt und die Laufzeit merklich verringern würde, wenn nur ein Prozessor vorhanden ist.
2. Da bei einer Liste die Zeiten für das Einfügen und das Entfernen von Elementen sehr hoch sind, wäre hierfür die Verwendung eines Fibonacci-Heap möglich. Ein Fibonacci-Heap ist eine Priority-Queue. In einem Fibonacci-Heap können die Elemente mit einer festgelegten Priorität in beliebiger Reihenfolge effizient abgespeichert werden.
   Bei dieser Datenstruktur beträgt der Aufwand für das Einfügen eines neuen Elements O(1), der Aufwand für das Entfernen beträgt O(log n) und der Aufwand, um das minimale Element auszugeben beträgt O(log n). Da diese Operationen die meist genutzten bei dem Dijkstra-Algorithmus sind, würde es eine Laufzeit Verbesserung geben.

Durch die Verbesserungen konnte ich folgende Verbesserungen in der Laufzeit erreichen:

Beispielname	Single Core	Multi Core
wettbewerb_bsp.txt	00:00:00,0001	00:00:00,0006
00595x00815.txt	00:00:01,2837	00:00:00,3910
01417x01889.txt	00:00:07,5207	00:00:02,6190
01910x04719.txt	00:00:36,9997	00:00:12,4840
11467x29342.txt	> 30min	00:12:19,0984

Testsystem:

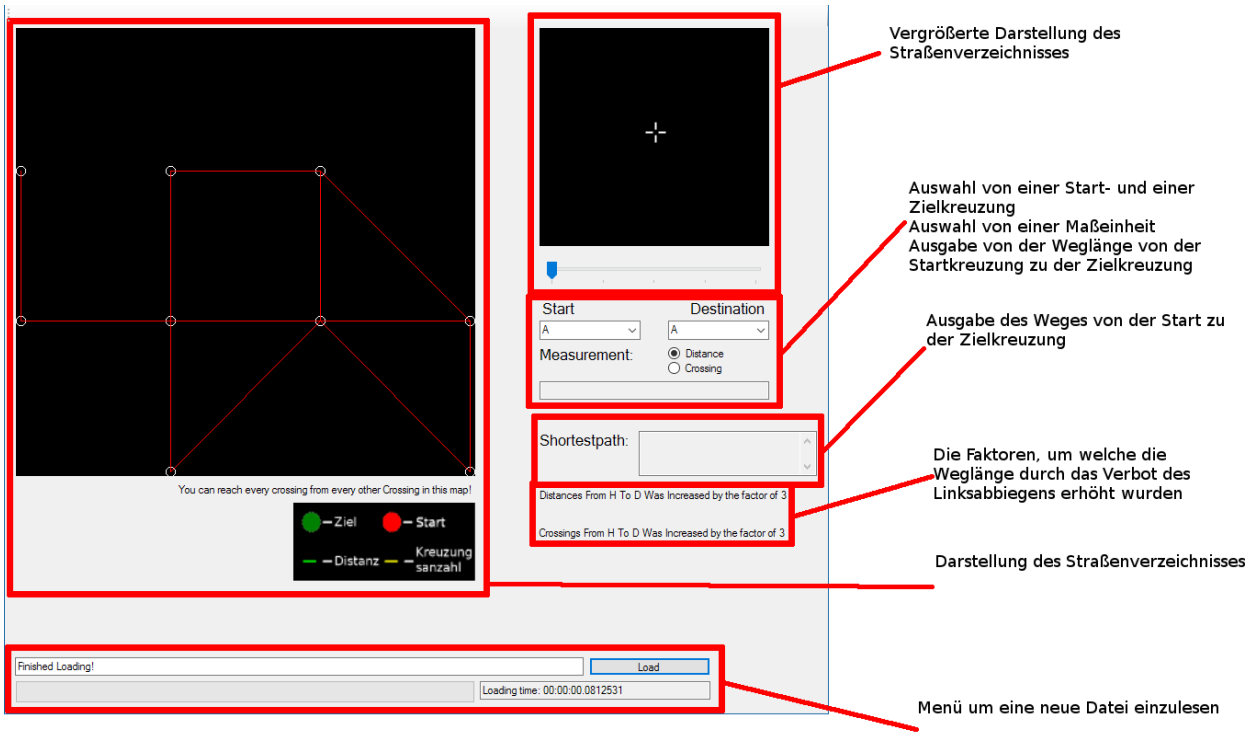
Betriebssystem: Windows 10

Prozessor: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, 2601 MHz, 4 Kern(e)

RAM: 16GB

5. Grafische Darstellung:

Die Ausgabe des Programms findet über eine GUI statt.



Mit Hilfe dieser GUI kann dem Benutzer sofort das Straßenverzeichnis grafisch dargestellt werden. Außerdem können die kürzesten Wege auch direkt grafisch in der Benutzeroberfläche dargestellt werden.

Wenn eine Datei geladen wurde und eine Startkreuzung sowie eine Zielkreuzung ausgewählt wurde, wird der kürzeste Weg in das Straßenverzeichnis eingezeichnet. Wenn als Maßeinheit die Distanz gewählt wurde, wird der kürzeste Weg in grün dargestellt. Falls Kreuzungen als Maßeinheit gewählt wurden, wird der kürzeste Weg gelb dargestellt.

In der GUI wird auch der kürzeste Weg durch die Namen der nacheinander folgenden Kreuzungen ausgegeben.

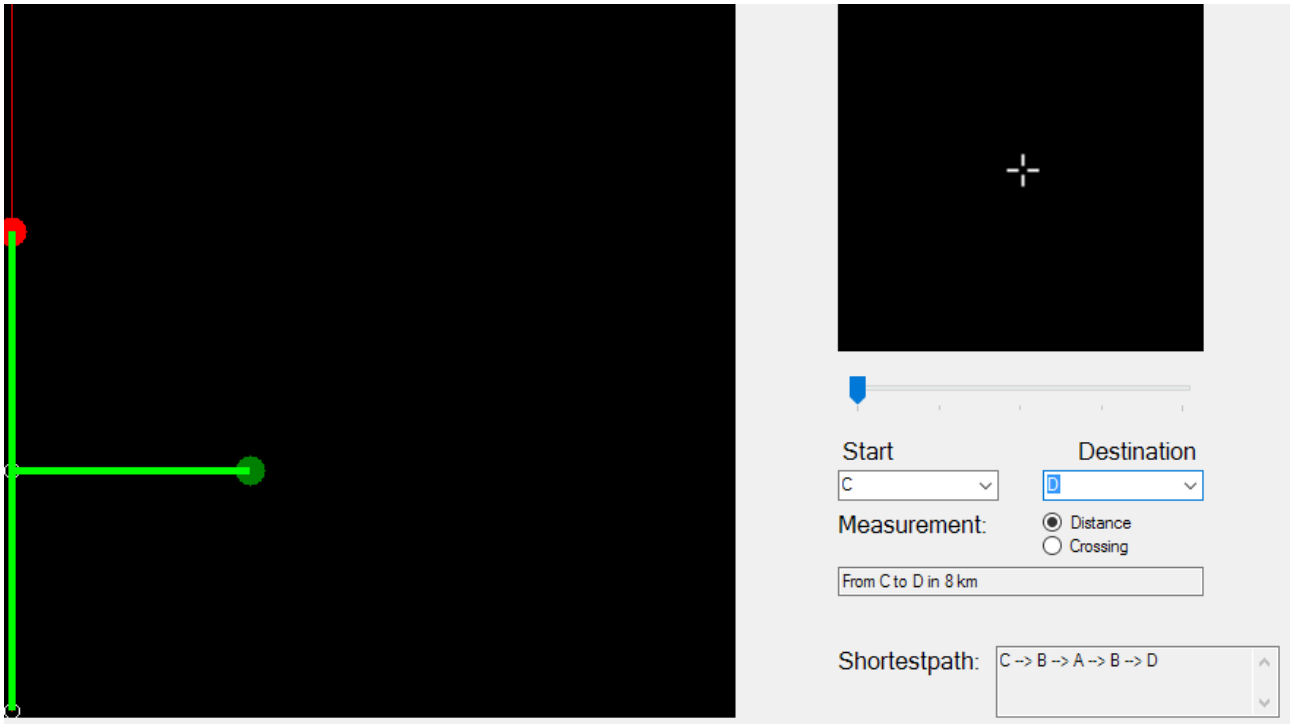
Der größtmögliche Faktor, der durch das Verbot des Linksabbiegens entstanden ist, wird auch in der GUI dargestellt.

Direkt unter der Darstellung des Straßenverzeichnisses wird ausgegeben, ob alle Kreuzungen von allen anderen erreichbar sind.

Beispiele

Für das Programm wurden zwölf Beispiel Eingaben erstellt. Diese Beispiele sind in dem Ordner zu finden, in dem sich auch diese Dokumentation befindet.
5 Beispiele sind dazu gedacht, die Laufzeit des Algorithmus zu testen.
6 Beispiele sind dafür gedacht, den Algorithmus auf mögliche Schwachstellen zu prüfen, da das Straßenverzeichnis in jedem dieser Beispiele eine ganz bestimmte Schwierigkeit aufweist.
Das letzte Beispiel ist das Beispiel von der Website des Bundeswettbewerbs.

Schwierigkeit_01.txt:
Dieses Beispiel(Schwierigkeit_01) dient dazu den Algorithmus darauf zu überprüfen, dass er nicht den direkten Weg von C→D(C→B →D) nimmt denn dieser Weg wäre mit einem Linksabbiegen verbunden. Es soll der Weg C→B→A→B→D herausgefunden werden. Bei diesem Weg wird das Linksabbiegen durch das Wenden bei 'A' umgangen.
Der Algorithmus erkennt das Linksabbiegen richtig und wendet bei 'A'.

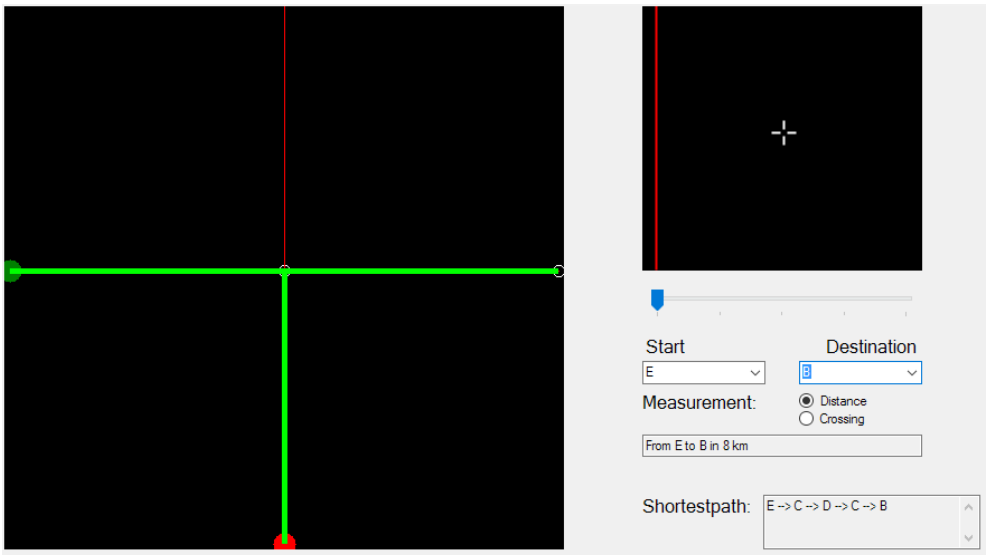


Ausgabe Schwierigkeit_01.txt

(kürzester Weg: C→B→A→B→D)

Schwierigkeit_02.txt:

Durch Schwierigkeit_02 soll etwas ähnliches wie bei Schwierigkeit_01 überprüft werden. Es wurde noch eine Kreuzung dem Beispiel hinzugefügt, um zu zeigen, dass der Algorithmus trotz dieser zusätzlichen Kreuzung immer noch ohne links abzubiegen den kürzesten Weg von 'B' nach 'E' findet und sich nicht von der zusätzlichen Kreuzung beirren lässt.

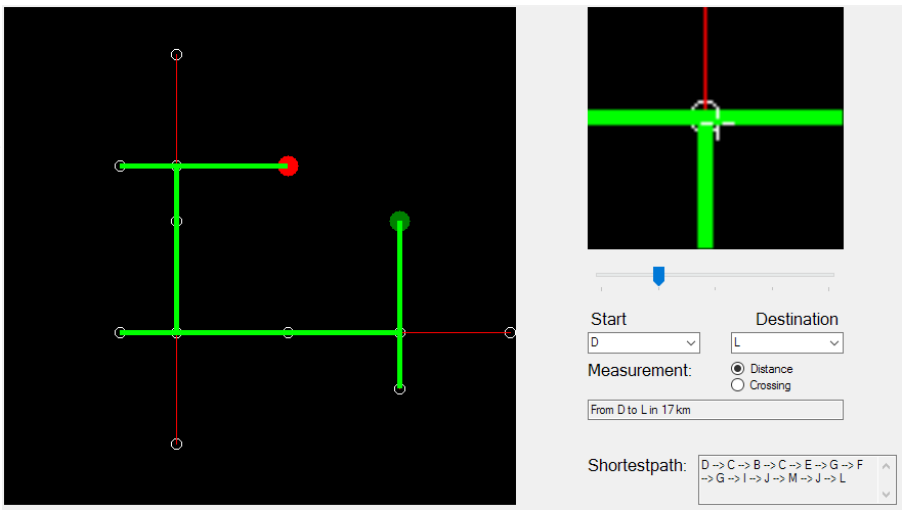


Ausgabe Schwierigkeit_02.txt

(kürzester Weg: E→C→D→C→B)

Schwierigkeit_03.txt:

Schwierigkeit_03 zeigt, dass der Algorithmus immer das Linksabbiegen durch Wenden in einer Sackgasse verhindert. Bei dem Weg von D→L wird dies drei mal gefordert und der Algorithmus erkennt jede von diesen Situationen. Außerdem findet er den kürzesten Weg, weil es immer zwei Möglichkeiten für das Wenden gibt, wobei die eine Möglichkeit immer eine größere Distanz hat als die andere. Der Algorithmus wählt sich aber immer die Möglichkeit zum Wenden mit der geringeren Distanz.



Ausgabe Schwierigkeit_03.txt

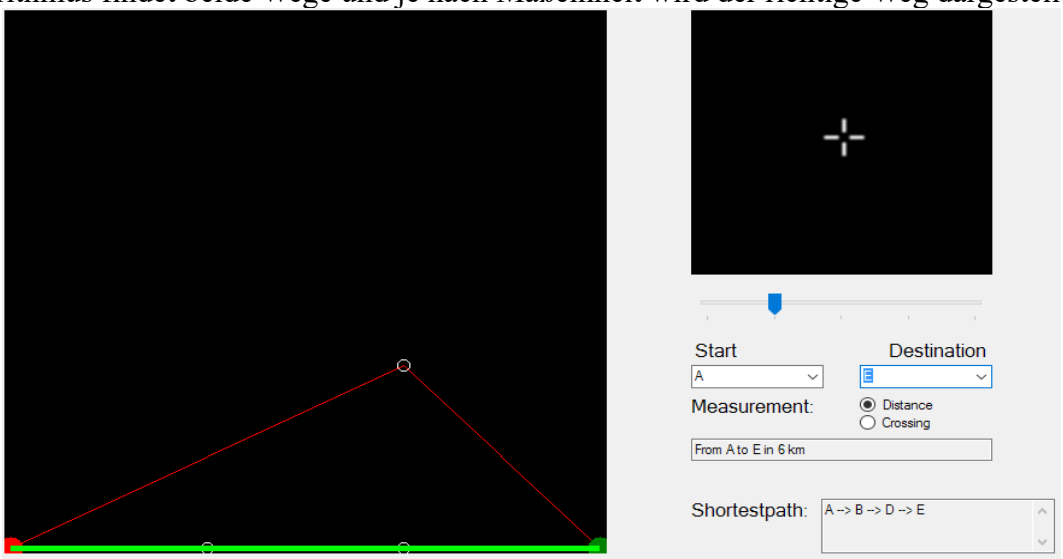
(kürzester Weg: D→C→B→C→E→G→F→G→I→J→M→J→L)

Schwierigkeit_04.txt:

In Schwierigkeit_04 wird der Algorithmus darauf überprüft, ob die beiden Maßeinheiten (Kreuzungen, Distanz) richtig ermittelt werden.

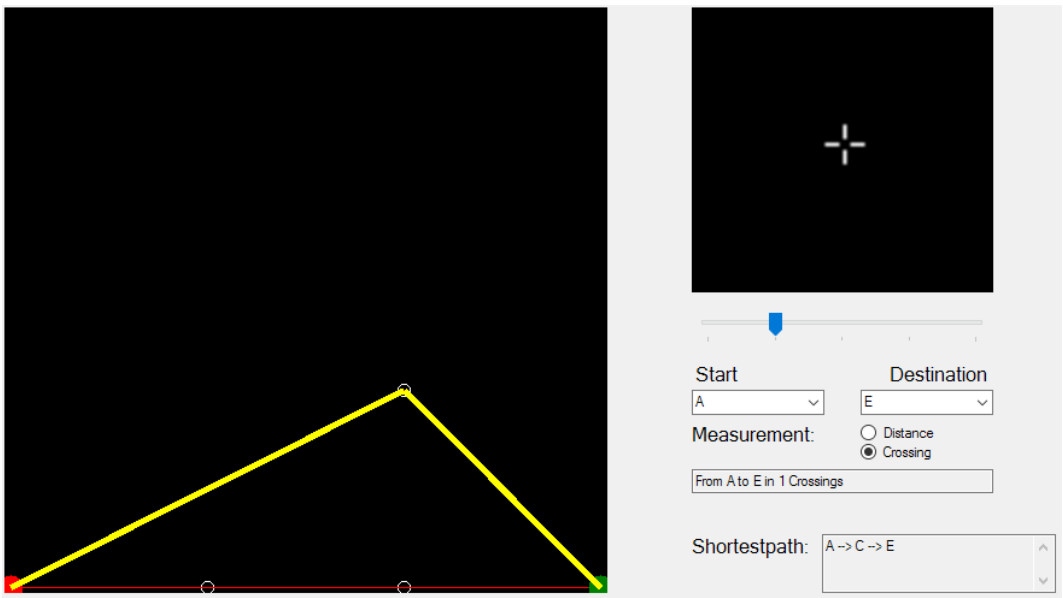
Der Weg von A→E sollte zwei unterschiedliche kürzeste Weg je nach Maßeinheit ergeben. Wenn die Distanz als Maßeinheit verwendet wird, sollte das Ergebnis A→B→D→E lauten. Wenn hingegen die Anzahl von Kreuzungen als Maßeinheit verwendet wird, muss der kürzeste Weg A→C→ E lauten.

Der Algorithmus findet beide Wege und je nach Maßeinheit wird der richtige Weg dargestellt.



Ausgabe Schwierigkeit_04.txt (Distanz)

(kürzester Weg: A→B→D→E)



Ausgabe Schwierigkeit_04.txt (Kreuzungen)

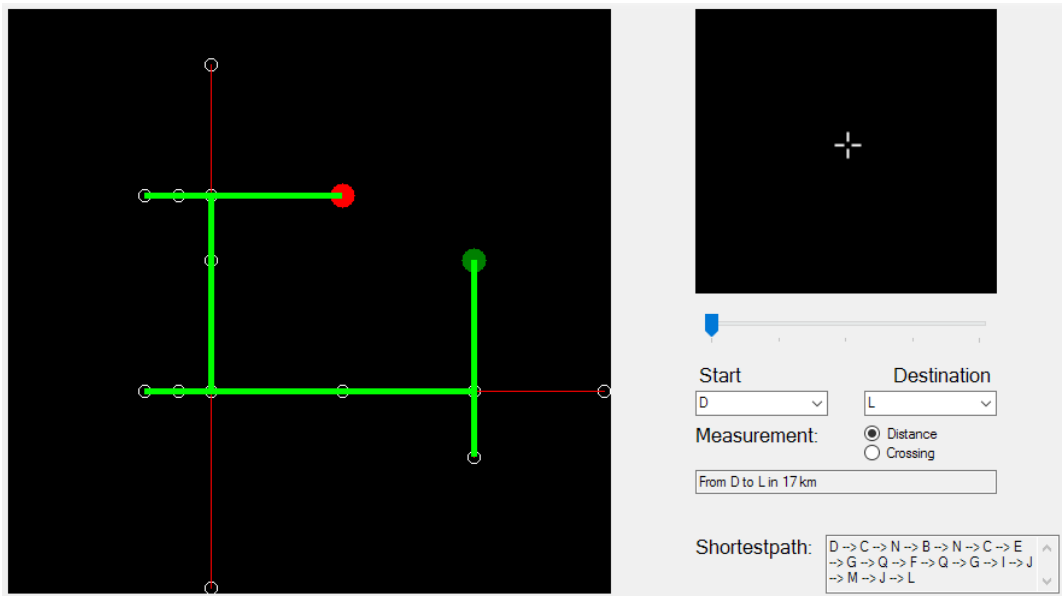
(kürzester Weg: A →C→E)

Schwierigkeit_05.txt:

Schwierigkeit_05 weist fast die gleiche Schwierigkeit auf wie Schwierigkeit_03.txt bis auf die zusätzlichen Kreuzungen zwischen B – C und F – G.

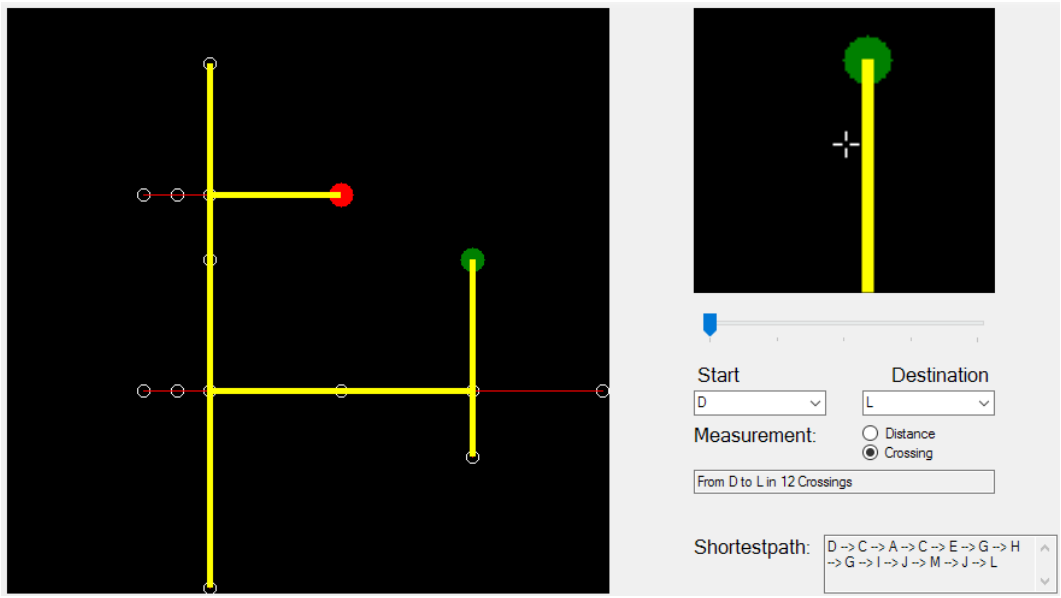
Durch diese Veränderungen des Straßenverzeichnisses wird wieder die Erkennung der unterschiedlichen Maßeinheiten des Algorithmus auf die Probe gestellt.

Der Algorithmus erkennt aber je nach Maßeinheit trotzdem den richtigen Weg.



Ausgabe Schwierigkeit_05.txt (Distanz)

(kürzester Weg: D→C→N→B→N→C →E→G→Q→F→Q→G→I→J→M→J→L)



Ausgabe Schwierigkeit_07.txt (Kreuzungen)

(kürzester Weg: D→C→A→C→E→G→H→G→I→J→M→J→L)

Schwierigkeit_06.txt:

Schwierigkeit_06 zeigt, dass der Algorithmus richtig erkennt, wenn eine Kreuzung von einer anderen Kreuzung nicht erreichbar ist und das dieser Fall richtig in der GUI ausgegeben wird.

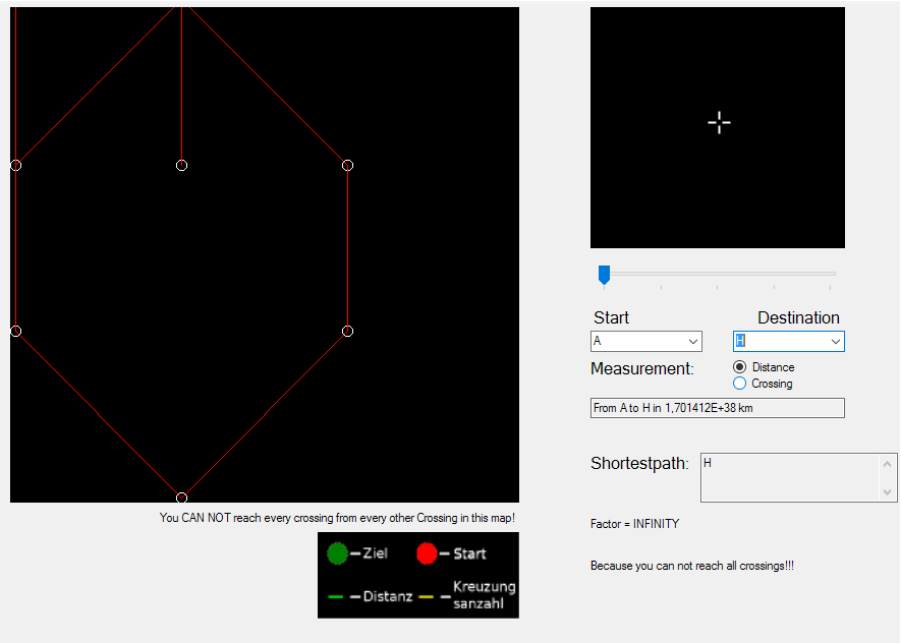


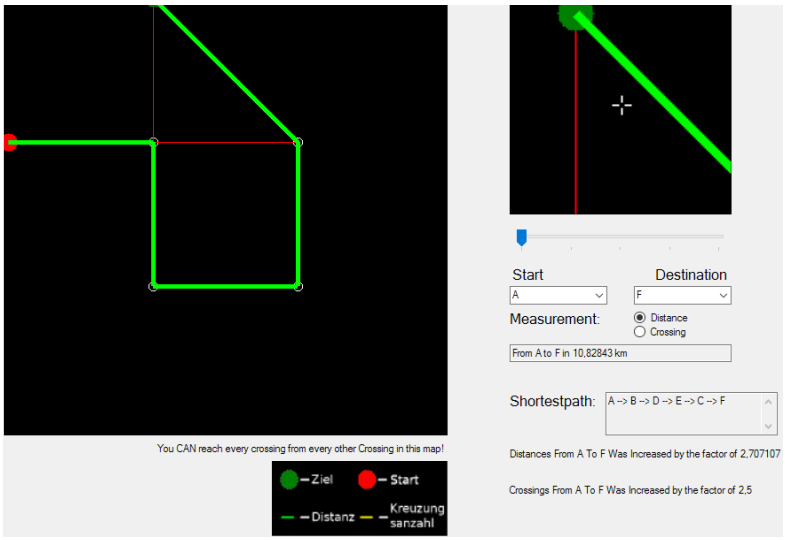
Abbildung 3: Ausgabe Schwierigkeit_06.txt

(kürzester Weg: D→C→A→C→E→G→H→G→I→J→M→J→L)

Schwierigkeit_07.txt

Diese Beispieldateie zeigt, dass der Algorithmus das Paar von Start- und Zielkreuzung, für das das Verbot des Linksabbiegens die Weglänge um den größtmöglichen Faktor erhöht, richtig ermittelt. Der Weg von 'A' zu 'F' ist ohne das Verbot 4km (2 Kreuzungen) und mit dem Verbot 10,82km (5 Kreuzungen) lang.

Damit ergeben sich die Faktoren 2,7 (Distanz) und 2,5 (Kreuzungen). Diesen Wert ermittelt auch der Algorithmus.



Ausgabe Schwierigkeit_07.txt

Laufzeiten des Programms

Beispiel Name	Laufzeit des Programms
schwierigkeit_01.txt	00:00:00.0198234
schwierigkeit_02.txt	00:00:00.0276278
schwierigkeit_03.txt	00:00:00.0208596
schwierigkeit_04.txt	00:00:00.0206573
schwierigkeit_05.txt	00:00:00.0206609
wettbewerb_bsp.txt	00:00:00.0212096
00595x00815.txt	00:00:01.1227512
01417x01889.txt	00:00:07.1265651
01910x04719.txt	00:00:29.8656624
03299x06305.txt	00:01:19.6849607
11467x29342.txt	00:36:25.4678854

Testsystem:

Betriebssystem: Windows 10

Prozessor: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, 2601 MHz, 4 Kern(e)

RAM: 16GB