TU Wien
Institut für Logic and Computation
Forschungsbereich Algorithms and Complexity

186.866 Algorithmen und Datenstrukturen VU

Programmieraufgabe P2

PDF erstellt am: 19. März 2024

1 Vorbereitung

Um diese Programmieraufgabe erfolgreich durchführen zu können, müssen folgende Schritte umgesetzt werden:

1. Laden Sie das Framework `P2.zip` aus TUWEL herunter.
2. Entpacken Sie `P2.zip` und öffnen Sie das entstehende Verzeichnis als Projekt in IntelliJ (nicht importieren, sondern öffnen).
3. Öffnen Sie die nachfolgend angeführte Datei im Projekt in IntelliJ. In dieser Datei sind sämtliche Programmiertätigkeiten durchzuführen. Ändern Sie keine anderen Dateien im Framework und fügen Sie auch keine neuen hinzu.
   `src/main/java/exercise/StudentSolutionImplementation.java`
4. Füllen Sie Vorname, Nachname und Matrikelnummer in der Methode `StudentInformation provideStudentInformation()` aus.

2 Hinweise

Einige Hinweise, die Sie während der Umsetzung dieser Aufgabe beachten müssen:

- Lösen Sie die Aufgaben selbst und nutzen Sie keine Bibliotheken, die diese Aufgaben abnehmen.
- Sie dürfen beliebig viele Hilfsmethoden schreiben und benutzen. Beachten Sie aber, dass Sie nur die oben geöffnete Datei abgeben und diese Datei mit dem zur Verfügung gestellten Framework lauffähig sein muss.

3 Übersicht

In dieser Programmieraufgabe wird ergänzend zum Dijkstra-Algorithmus der A*-Algorithmus thematisiert. Dieser ist eine Verallgemeinerung des Dijkstra-Algorithmus. Der A*-Algorithmus soll in verschiedenen Varianten implementiert und im Anschluss analysiert werden.

4 Theorie

Die grundlegende Theorie kann in den Vorlesungsfolien „Graphen“ gefunden werden. In diesen Folien wird der Dijkstra-Algorithmus vorgestellt und im Folgenden wird darüber hinaus der A*-Algorithmus erklärt.

Das Ziel des A*-Algorithmus ist es, den kürzesten Pfad von einem Knoten s (Source) zu einem Zielknoten t (Target) in einem gewichteten, gerichteten Graphen $G = (V, A)$ zu finden. Wie im Dijkstra-Algorithmus wird auch hier eine Priority Queue (Vorrangwarteschlange) benutzt. Im Dijkstra-Algorithmus erfolgt die Priorisierung eines Knotens x in dieser Queue anhand der bekannten geringsten Kosten $g(x)$, die benötigt werden, um x zu erreichen (bisher kürzester Pfad). Bei dem A*-Algorithmus werden die Kosten $g(x)$ um eine Kostenschätzung des verbleibenden Pfades von x zum Zielknoten t erweitert. Diese Schätzung wird von einer Funktion $h\colon V \to \mathbb{R}$ übernommen, welche *Heuristik* genannt wird. Die Priorisierung für den Knoten x erfolgt also nicht nur durch $g(x)$, sondern durch $g(x) + h(x)$ (Kosten des bisherigen Pfades + Abschätzung der verbleibenden Kosten).

Der Pseudocode für den A*-Algorithmus ist in Algorithmus 1 dargestellt. Hier kann auch die Ähnlichkeit zu dem Dijkstra-Algorithmus nachvollzogen werden. Damit der A*-Algorithmus Pfade mit den geringsten Kosten findet, muss die Heuristik h *zulässig* sein. D.h. die Schätzung darf die Kosten für den optimalen Restpfad nie überschätzen.

Definition 4.1 (Zulässige Heuristik)**.** Sei $G = (V, A)$ ein gewichteter, gerichteter Graph und sei $t \in V$ ein Knoten. Weiters sind $h^*(x)$ die minimalen Kosten von einem Knoten $x \in V$ nach t. Eine Heuristik $h\colon V \to \mathbb{R}$ ist *zulässig*, wenn $h(x) \leq h^*(x)\ \forall x \in V$ gilt.

Theorem 4.2. Sei $G = (V, A)$ ein gewichteter, gerichteter Graph, $s \in V$ der Startknoten, $t \in V$ der Zielknoten und h eine zulässige Heuristik. Der A*-Algorithmus findet mit h immer einen Pfad mit den geringsten Kosten von s nach t in G.

Algorithmus 1: A*-Algorithmus

Input: gewichteter, gerichteter Graph $G = (V, A)$
mit $w_{ij} > 0 \; \forall (i,j) \in A$, sowie Source s und Target t
$g(s) = 0$
$g(x) = \infty \; \forall x \in V \setminus \{s\}$
$\texttt{pred}(x) = null \; \forall x \in V$
Priority Queue $Q \leftarrow \{(s, h(s))\}$
while $Q \neq \emptyset$ **do**
    $x \leftarrow$ entferne Knoten x mit minimalen Kosten $g(x) + h(x)$ aus Q
    if $x = t$ **then**
        return $g(t) + h(t) = g(t)$ und den Pfad $s, \ldots, \texttt{pred}(t), t$
    forall $v \colon (x, v) \in A$ **do**
        $g_{\text{candidate}} \leftarrow g(x) + w_{xv}$
        if $g_{candidate} < g(v)$ **then**
            $g(v) \leftarrow g_{\text{candidate}}$
            $\texttt{pred}(v) \leftarrow x$; // besserer Pfad nach v
            Füge v zu Q mit Kosten $g_{\text{candidate}} + h(v)$ hinzu oder
             reduziere die Kosten von v in Q auf $g_{\text{candidate}} + h(v)$
return kein Pfad von s nach t

5 Implementierung

Neben dem eigentlichen A*-Algorithmus müssen Sie auch zwei der verwendeten Heuristiken implementieren. Da wir im Folgenden unseren Schwerpunkt auf Graphen setzen, deren Knoten geometrische Punkte in der Ebene repräsentieren, werden Sie dabei mit der Klasse `Point` arbeiten. Diese stellt zwei Methoden zur Verfügung:

- `int getX()`: Gibt die x-Position zurück.
  Mittels `int xPosition = point.getX()` kann so beispielsweise die x-Position des Punktes `point` ausgelesen werden.
- `int getY()`: Gibt die y-Position zurück.
  Mittels `int yPosition = point.getY()` kann so beispielsweise die y-Position des Punktes `point` ausgelesen werden.

5.1 Heuristiken

Insgesamt wird der A*-Algorithmus in dieser Aufgabe mit vier unterschiedlichen Heuristiken ausgeführt:

- **Manhattan Distanz:** Für einen Knoten x ist $h(x)$ die Summe der Differenzen der jeweiligen Koordinaten von x und dem Zielknoten t. Es werden also die absolute horizontale und die absolute vertikale Distanz summiert.
- **Euklidische Distanz:** Für einen Knoten x ist $h(x)$ die euklidische Distanz zwischen x und Zielknoten t.
- **Kürzester Pfad:** Für einen Knoten x sind $h(x)$ die Kosten des kostengünstigsten Pfades von x zum Zielknoten t. Beachten Sie, dass diese Heuristik voraussetzt, dass die Kosten des kostengünstigsten Pfades bereits bekannt sind. Gleichzeitig ist deren Ermittlung Ziel des A*-Algorithmus.
- **Null:** Für alle Knoten $x \in V$ gilt $h(x) = 0$.

Die ersten beiden Heuristiken müssen Sie selbst implementieren.

5.1.1 Manhattan-Distanz

Implementieren Sie die Manhattan-Distanz in der Methode `double heuristicManhattanDistance(Point point1, Point point2)`, indem Sie die Manhattan-Distanz der beiden Punkte `point1` und `point2` zurückgeben.

Testen Sie Ihre Lösung mit der Testinstanz `manhattan-distance-heuristic-test.csv`, bevor Sie fortfahren. Informationen über das Testen können Sie im Abschnitt 6 erhalten.

5.1.2 Euklidische Distanz

Implementieren Sie die euklidische Distanz in der Methode `double heuristicEuclideanDistance(Point point1, Point point2)`, indem Sie die euklidische Distanz der beiden Punkte `point1` und `point2` zurückgeben.

Testen Sie Ihre Lösung mit der Testinstanz `euclidean-distance-heuristic-test.csv`, bevor Sie fortfahren. Informationen über das Testen können Sie im Abschnitt 6 erhalten.

5.2 A*-Algorithmus

Implementieren Sie den A*-Algorithmus in der Methode `void aStar(Graph g, PriorityQueue q, Heuristic h, int source, int target, int[] path)`.

Der Parameter `Graph g` stellt den gewichteten, gerichteten Graphen dar, in dem der Pfad mit den geringsten Kosten vom Knoten mit ID `source` zum Zielknoten mit ID `target` gesucht werden soll. `Graph g` bietet einige Methoden an, die zur Implementierung des A*-Algorithmus hilfreich sein könnten:

- `int numberOfVertices()`: Gibt die Anzahl an Knoten zurück. Mittels `int number = g.numberOfVertices()` können Sie die Anzahl der Knoten im Graph `g` auslesen. Knoten werden mittels aufsteigender IDs gekennzeichnet. Angenommen es gibt insgesamt n Knoten, dann gibt es jeweils Knoten mit Knoten-ID 1, 2, ..., $n-1$ und n.

- `double getEdgeWeight(int vertexIdStart, int vertexIdEnd)`:
  Gibt das Kantengewicht einer Kante zurück.
  Das Kantengewicht einer Kante kann durch die Angabe von zwei Knoten (mittels Knoten-ID) ausgelesen werden. So kann das Kantengewicht mit den Endknoten der IDs 1 und 2 mittels `double edgeWeight = g.getEdgeWeight(1, 2)` erlangt werden. Gibt es zwischen zwei Knoten keine Kante, erhalten Sie den Wert -1 als Ergebnis.

- `int[] getSuccessors(int vertexId)`: Gibt die Nachfolger eines Knotens zurück.
  Die Nachfolger des Knoten mit der ID 1 können mittels `int[] successors = g.getSuccessors(1)` ermittelt werden. Die Nachfolger werden als Array von Knoten-IDs repräsentiert. Bei der Übergabe einer ungültigen Knoten-ID, wird `null` zurückgegeben.

Der Parameter `PriorityQueue q` enthält eine Priority Queue, die Sie zur Umsetzung des A*-Algorithmus nutzen sollen. Falls Sie eine eigene Implementierung oder Instanz nutzen, wird die für die Fragenbeantwortungen notwendige Visualisierung nicht korrekt funktionieren. Folgende Methoden werden von `PriorityQueue q` zur Verfügung gestellt:

- `void add(int vertexId, double weight)`: Fügt einen Knoten mit Priorität `weight` zur Priority Queue hinzu.
  Ein neuer Knoten kann der Priority Queue hinzugefügt werden, indem eine Knoten-ID `int vertexId` und eine Gewichtung `double weight` angegeben wird. Zum Beispiel kann der Knoten mit der ID 1 mit Priorität 20 mit `q.add(1, 20)` eingefügt werden. Sollte eine Knoten-ID bereits in der Priority Queue vorhanden sein, hat der Aufruf dieser Methode keine Folgen.

- `int removeFirst()`: Entfernt den ersten Knoten.
  Mittels `int idOfFirstVertex = q.removeFirst()` können Sie die Knoten-ID des Knotens mit den niedrigsten assoziierten Kosten (höchste Priorität) aus der Priority Queue auslesen und gleichzeitig aus dieser entfernen. Ist die Priority Queue leer, wird -1 zurückgegeben.

- `void decreaseWeight(int vertexId, double weight)`:
  Überschreibt die mit dem Knoten assoziierten Kosten mit geringeren.

Um die assoziierten Kosten eines Knotens innerhalb der Priority Queue zu senken, kann diese Methode aufgerufen werden. Einerseits muss die Knoten-ID des gewünschten Knotens (`int vertexId`) angegeben werden und andererseits die neuen, verringerten Kosten (`double weight`). Änderungen passieren nur, wenn Kosten angegeben werden, die tatsächlich kleiner sind, als die bisherigen, und wenn ein Knoten mit der Knoten-ID in der Priority Queue vorhanden ist.

- `boolean isEmpty()`: Gibt `true` zurück, wenn sich keine Knoten mehr in der Priority Queue befinden.
- `boolean contains(int vertexId)`: Gibt Ihnen Auskunft darüber, ob sich ein bestimmter Knoten in der Priority Queue befindet.
  Mittels `boolean isContained = q.contains(1)` kann überprüft werden, ob der Knoten mit der ID 1 in der Priority Queue enthalten ist.

Der Parameter `Heuristic h` ist ein Wrapper, der zur Berechnung der Heuristik dient. Je nach Testinstanz nutzt dieser Wrapper eine andere Heuristik, unter anderem auch jene, die von Ihnen implementiert wurden. `Heuristic h` hat nur eine Methode:

- `double evaluate(int vertex)`: Gibt den Wert der Heuristik für einen Knoten zurück.
  Mittels `double heuristicValue = h.evaluate(1)` wird die Heuristik für den Knoten mit ID 1 angewandt.

Das Ergebnis Ihrer A*-Algorithmus-Implementierung muss im Parameter `int[] path` abgelegt werden. Einen expliziten Rückgabewert gibt es nicht. Legen Sie den Pfad als Auflistung der Knoten vom Startknoten `source` bis zum Zielknoten `target` im Array `path` in einem Stück ab, sodass der Zielknoten an der letzten Stelle des Arrays liegt. Mögliche nicht benötigte Stellen zu Beginn des Arrays müssen Sie unverändert lassen.

Angenommen, der Startknoten `source` hat ID 1, der Zielknoten `target` hat ID 7 und der kürzeste Pfad ist 1, 5, 2, 7. Die entsprechende Abbildung des Pfades als Array kann in Abbildung 1 gefunden werden.

Findet Ihre Implementierung keinen Pfad, lassen Sie das Array `int[] path` unverändert.

Zum Testen Ihrer Implementierung werden zwei Graphen angeboten. Diese können jeweils via den Testinstanzen `a-star-grid.csv` und

Index	0	1	2	3	4	5	6
Knoten-IDs	0	0	0	1	5	2	7

Abbildung 1: Gefundener Pfad als Array

`a-star-street.csv` zur Überprüfung Ihrer Lösung herangezogen werden. Bedenken Sie allerdings, dass die Ursache für Fehler auch in den Umsetzungen der Heuristiken liegen könnte. Informationen über das Testen können Sie im Abschnitt 6 erhalten.

6 Testen

Führen Sie zunächst die `main`-Methode in der Datei `src/main/java/framework/Exercise.java` aus.

Anschließend wird Ihnen in der Konsole eine Auswahl an Testinstanzen angeboten, darunter befindet sich zumindest `abgabe.csv`:

```
Select an instance set or exit:
[1] abgabe.csv
[0] Exit
```

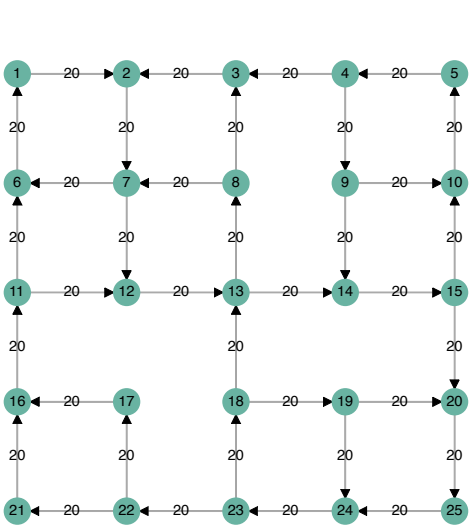
Durch die Eingabe der entsprechenden Ziffer kann entweder eine Testinstanz ausgewählt werden oder das Programm (mittels der Eingabe von `0`) verlassen werden. Wird eine Testinstanz gewählt, dann wird der von Ihnen implementierte Programmcode ausgeführt. Kommt es dabei zu einem Fehler, wird ein Hinweis in der Konsole ausgegeben.

Relevant für die Abgabe ist das Ausführen der Testinstanz `abgabe.csv`.

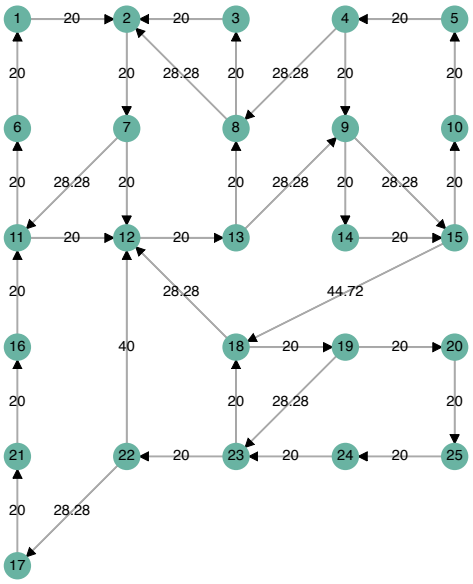
Die weiteren Testinstanzen `manhattan-distance-heuristic-test.csv`, `euclidean-distance-heuristic-test.csv`, `a-star-grid.csv` und `a-star-street.csv` sind nur zum jeweiligen Testen der einzelnen Unteraufgaben gedacht. Grid-Instanzen nutzen dabei den Grid-Graphen aus Abbildung 2a und Street-Instanzen nutzen den Street-Graphen aus Abbildung 2b.

7 Evaluierung

Wenn der von Ihnen implementierte Programmcode mit der Testinstanz `abgabe.csv` ohne Fehler ausgeführt werden kann, dann wird nach dem



(a) Grid-Graph



(b) Street-Graph

Abbildung 2: Graphen, die in den Testinstanzen verwendet werden.

Beenden des Programms im Ordner `results` eine Ergebnis-Datei mit dem Namen `solution-abgabe.csv` erzeugt.

Die Datei `solution-abgabe.csv` beinhaltet Zeitmessungen der Ausführung der Testinstanz `abgabe.csv`, welche in einem Web-Browser visualisiert werden können. (Auch Ergebnis-Dateien anderer Testinstanzen können zu Testzwecken visualisiert werden.) Öffnen Sie dazu die Datei `visualization.html` in Ihrem Web-Browser und klicken Sie rechts oben auf den Knopf *Ergebnis-Datei auswählen*, um `solution-abgabe.csv` auszuwählen.

Beantworten Sie basierend auf der Visualisierung die Fragestellungen aus dem folgenden Abschnitt.

8 Fragestellungen

Öffnen Sie `solution-abgabe.csv` und bearbeiten Sie folgende Aufgaben- und Fragestellungen. **Die Kantenkosten beider Graphen (Grid- und Street-Graphen) entsprechen der euklidischen Distanz zwischen den inzidenten Knoten.**

1. Mit welcher der vier vorgestellten Heuristiken ist das Vorgehen im A*-Algorithmus ident zum Dijkstra-Algorithmus? Begründen Sie Ihre Antwort. Ist diese Heuristik im Allgemeinen zulässig?

2. Vergleichen Sie die Ausführungen des A*-Algorithmus in den Instanzen *540* und *1165* mithilfe entsprechender Auswahl im Dropdown links oben. Möglicherweise können Sie die Suche nach den Instanzen vereinfachen, indem Sie nach dem Klick auf das Dropdown die Instanznummer tippen. Um den Vergleich zu vereinfachen, können Sie auch `solution-abgabe.csv` zweifach in einem Browser-Fenster öffnen, oder `visualization.html` in zwei separaten Browser-Fenstern öffnen.

   Nach der Auswahl einer Instanz erscheint der jeweilige Graph. Mithilfe der Knöpfe über dem Graphen können Sie nun die Ausführung des A*-Algorithmus Schritt für Schritt verfolgen. Sämtliche Knoten und Kanten, die Ihr Algorithmus bis zu einem bestimmten Schritt betrachtet hat, werden farbig dargestellt. Kanten, die nicht mehr auf einem kürzesten Pfad zu einem Knoten liegen, werden dabei nicht eingefärbt. Knoten und Kanten, die im jeweiligen Schritt betrachtet werden, sind rot markiert. Dementsprechend ist ein rot umrandeter Knoten jener Knoten v, der in diesem Schritt aus der Priority Queue entnommen wurde und rot markierte Kanten sind jene Kanten, die zu einem Nachfolger von v führen und diesen als neuen Vorgänger führen. Im letzten Schritt wird der kürzeste Pfad speziell hervorgehoben.

   Wird in beiden Instanzen der gleiche Pfad gefunden? Beantworten Sie auch für jeden der Pfade, ob dieser optimal ist, indem Sie argumentieren, dass die jeweilige Heuristik in diesem Graphen zulässig/nicht zulässig ist. Die benutzte Heuristik können Sie aus Ihrer Auswahl im Dropdown auslesen.

   Springen Sie anschließend in den Visualisierungen zum letzten Schritt und erstellen Sie Screenshots der beiden Graphen, sodass auch der jeweilige gefundene Pfad sichtbar ist.

3. Vergleichen Sie nun die Instanzen *3040* und *3665*.

   Wird in beiden Instanzen der gleiche Pfad gefunden? Beantworten Sie auch für jeden der Pfade, ob dieser optimal ist, indem Sie argumentieren, dass die jeweilige Heuristik in diesem Graphen zulässig/nicht zulässig ist. Die benutzte Heuristik können Sie aus Ihrer Auswahl im Dropdown auslesen. Falls Sie in Ihrer Analyse zu

unterschiedlichen Ergebnissen als in der vorhergehenden Frage kommen, argumentieren Sie die Ursache dafür.

Springen Sie anschließend in den Visualisierungen zum letzten Schritt und erstellen Sie Screenshots der beiden Graphen, sodass auch der jeweilige gefundene Pfad sichtbar ist.

4. Vergleichen Sie nun die Instanzen *3040* und *4915*.

   Welchen Nachteil erkennen Sie in der Benutzung der Heuristik in Instanz *4915* gegenüber der Heuristik in *3040*? Geben Sie den Grund dafür an.

   Springen Sie anschließend in der Visualisierung von Instanz *4915* zum letzten Schritt und erstellen Sie einen Screenshot des Graphen, sodass auch der gefundene Pfad sichtbar ist.

5. Vergleichen Sie nun die Instanzen *3040* und *4290*.

   Welchen Vorteil erkennen Sie in der Benutzung der Heuristik in Instanz *4290* gegenüber der Heuristik in *3040*? Geben Sie den Grund dafür an. Halten Sie die Heuristik aus Instanz *4290* abseits von theoretischen Überlegungen sinnvoll? Nennen Sie knapp einen Anwendungsfall, der Ihrer Meinung nach in der Praxis nützlich sein könnte.

   Springen Sie anschließend in der Visualisierung von Instanz *4290* zum letzten Schritt und erstellen Sie einen Screenshot des Graphen, sodass auch der gefundene Pfad sichtbar ist.

Fügen Sie Ihre Antworten in einem Bericht gemeinsam mit den sechs erstellten Screenshots der Visualisierungen der Testinstanz `abgabe.csv` zusammen.

9 Abgabe

Laden Sie die Datei `src/main/java/exercise/StudentSolutionImplementation.java` in der TUWEL-Aktivität *Hochladen Source-Code P2* hoch. Fassen Sie diesen Bericht mit den anderen für das zugehörige Abgabegespräch relevanten Berichten in einem PDF zusammen und geben Sie dieses in der TUWEL-Aktivität *Hochladen Bericht Abgabegespräch 1* ab.

10 Nachwort

Der A*-Algorithmus hat eine große Bedeutung in der klassischen Routenplanung, um einen schnellsten oder kürzesten Weg auch in sehr großen Straßen- bzw. Wegenetzwerken zu finden. Geht man beispielsweise von einem Land wie Österreich aus, ist ein Graph mit Millionen von Knoten und Kanten für ein detailliertes Straßennetzwerk nicht außergewöhnlich. Dijkstras klassischer Algorithmus zur Wegfindung wäre hier in der Praxis viel zu langsam. Mit einer guten Schätzfunktion gelingt es mittels des A*-Algorithmus einen schnellsten oder kürzesten Weg in einer Zeit zu finden, die praktisch nur linear von der Anzahl der Kanten dieses Weges abhängt.

Abgesehen von der Routenplanung besitzt der A*-Algorithmus aber auch eine sehr große Bedeutung in der kombinatorischen Optimierung und künstlichen Intelligenz, um schwierige Planungs- und Optimierungsprobleme oder auch Puzzles zu lösen. Hier beschreibt der Graph, in dem ein kürzester Pfad gesucht wird, allgemeiner einen Zustandsraum, in dem es gilt, von einem Ausgangszustand einen bestimmten Zielzustand zu erreichen.

In unserer weiterführenden Lehrveranstaltung „Algorithmics“ behandeln wir den A*-Algorithmus ausführlicher. Auch bieten wir immer wieder Themen für Bachelorarbeiten, Projektarbeiten und Diplomarbeiten an, die auf diesem wichtige Verfahren aufbauen.