# Aufgabe 1: ADT Graph

In dieser Aufgabe werden wir eine ADT Graph implementieren. Ziel ist, diese ADT in den nächsten drei Aufgaben zu verwenden. Die Vorgaben für die ADT sind so ausgelegt, dass sie mit anderen Teams zu jeder Zeit (ohne zusätzlichen Implementierungsaufwand) getauscht werden kann und über alle Aufgaben nach außen gleich bleibt.

## Aufgabenstellung

Da es in dieser Aufgabe darum geht, die Datenstrukturen selbst zu implementieren, sind "Übergaben" an Datenstrukturen von Erlang OTP untersagt. Etwa die Verwendung von z.B. `dict` oder `sets`. Lediglich die Basis-Strukturen Liste (`lists`) und Tupel (`tuple`) dürfen eingesetzt werden. Algorithmen auf diesen Basisstrukturen müssen aber selbst implementiert werden. Sonst wäre damit erfolgreich am Lerneffekt vorbei implementiert.

Zur Fehlerbehandlung: Sollten nicht vorhandene Elemente gelöscht werden etc. ist die Fehlerbehandlung soweit möglich durch „Ignorieren“ durchzuführen, d.h. es wird so gehandelt, als hätte die Operation nicht stattgefunden. Aus Dokumentationsgründen können log-Dateien erstellt werden.

Eine **ADT Graph** ist zu implementieren:

**Vorgabe**:
Funktional (nach außen)

1. vertex: ist eine ganze positive Zahl als eindeutige ID einer Ecke. Der Anwender bekommt auf die Struktur eines vertex keinen Zugriff. Er arbeitet immer nur mit dieser eindeutigen vertex-ID!
2. vertexlist: werden hier Kanten erwartet, so identifizieren jeweils zwei aufeinander folgende Ecken als Paar eine Kante, d.h. bei [v1,v2,v2,v4,...] identifizieren v1,v2 und v2,v4 jeweils eine Kante, nicht jedoch v2,v2. Werden nur Ecken erwartet, zählt jede Ecke für sich. Die Reihenfolge in den Listen hat keine Bedeutung!
3. Attribute: sind eindeutig im Namen (name) innerhalb aller Kanten zu einer bestimmten Ecke und aller Ecken. Hier können Vorgaben bzgl. der Namen gemacht werden.
4. Der Graph ist entweder gerichtet oder ungerichtet.
5. Es gibt keine Mehrfachkanten.

Technisch (nach innen)

1. Listen: (vertexlist) hier ist der Datentyp `lists` von Erlang zu verwenden. Ein Anwender kann dann frei auf diese Liste zugreifen, Änderungen beziehen sich aber nur auf diese Liste, nicht auf den Graphen!
2. Ob ein Graph gerichtet ist (`directed`) oder nicht (`nondirected`) wird in der globalen Variablen `yndirected` gespeichert (siehe util.erl). Diese Variable ist von der ADT selbst zu verwalten.
3. Das Standard-Attribut für die nachfolgenden Aufgaben heisst `weight`.
4. Der Standard-Fehlerwert ist `nil`.
5. importG ist im Sinne des *single-point-of-control* mittels addEdge und setAtE zu realisieren.
6. Dateinamen und Namen von Attributen werden als Atome verwendet (keine Zeichenketten!).
7. addEdge fügt ggf. die Ecken in den Graphen ein, d.h. addEdge erfordert nicht addVertex.

8. Graphen sind über Dateien (*.graph) ein- bzw. auszulesen. Dies dient der Möglichkeit, spezielle Graphen einfach austauschen zu können. Kodierung: UTF-8 ohne BOM. Dabei ist folgendes Format zu verwenden:
   [<Name Ecke1>,<Name Ecke2>,<Wert des Kantenattribut weight ggf. nil>,...]
   Andere Attribute werden nicht im- oder exportiert.
9. Die zugehörige Datei heißt `adtgraph.erl`. Weitere Dateien neben `util.erl` sind nicht zulässig.
10. `filename` ist der Name der Datei ohne Endung. Die jeweilige Endung (.graph oder .dot) wird automatisch angehängt, d.h. z.B. importG('graph_06',d) oder printG(Graph,'graphprint') als Aufruf. In util.erl gibt es mit attachEnding bzw. attachStamp dazu Hilfsfunktionen.

**Objektmengen:** vertex, graph, name, value, filename, file, vertexlist
**Operationen**: (semantische Signatur)

createG: [d|ud] → graph
/`createG(<d|ud>)`
addVertex: graph × vertex → graph
/`addVertex(<graph>,<vertex>)`
deleteVertex: graph × vertex → graph
/`deleteVertex(<graph>,<vertex>)`
addEdge: graph × vertex × vertex → graph /`addEdge(<graph>,<vertex>,<vertex>)`
deleteEdge: graph × vertex × vertex → graph
/`deleteEdge(<graph>,<vertex>,<vertex>)`
setAtE: graph × vertex × vertex × name × value → graph /`setAtE(<graph>,<vertex>,<vertex>,<name>,<value>)`
setAtV: graph × vertex × name × value → graph /`setAtV(<graph>,<vertex>,<name>,<value>)`
getValE: graph × vertex × vertex × name → value /`getValE(<graph>,<vertex>,<vertex>,<name>)`
getValV: graph × vertex × name → value /`getValV(<graph>,<vertex>,<name>)`
getIncident: graph × vertex → vertexlist
/`getIncident(<graph>,<vertex>)`
getAdjacent: graph × vertex → vertexlist
/`getAdjacent(<graph>,<vertex>)`
getTarget: graph × vertex → vertexlist
/`getTarget(<graph>,<vertex>)`
getSource: graph × vertex → vertexlist
/`getSource(<graph>,<vertex>)`
getEdges: graph → vertexlist
/`getEdges(<graph>)`
getVertices: graph → vertexlist
/`getVertices(<graph>)`
importG: filename × [d|ud] → graph
/`importG(<filename>,<d|ud>)`
exportG: graph × filename → file /`exportG(<graph>,<filename>)`
printG: graph × filename → dot /`printG(<graph>,<filename>)`

Führen Sie mit Ihrer ADT eine **Zeitmessung** durch (aufg1zeit.beam, Aufruf `aufg1zeit:zeitmessung()`) und geben die damit erzeugte Datei mit ab. Desweiteren erstellen Sie einen **Druck** eines Graphen (Aufruf `aufg1zeit:graphdrucken()`) und **geben die dadurch erzeugten Dateien mit ab**. Für diese beiden Tests werden die folgenden Graphen (im gleichen Ordner wie die beam Dateien) benötigt: testaufg1.graph und graph_06.graph. Zudem sind für

die Funktionen getTarget, getEdges und setAtE **Laufzeittests** so durchzuführen, dass Sie die **Laufzeitkomplexität** (z.B. Trendlinie in Excel oder libreoffice Calc) bestimmen können (**Einheit: ms!**).

# Abnahme

**Bis Donnerstag Abend 20:00 Uhr** vor Ihrem Praktikumstermin ist ein **Entwurf** für die Aufgabe **als *.pdf** Dokument (Dokumentationskopf nicht vergessen!) mir per E-Mail (mit cc an den/die Teampratner_in) zuzusenden. Der Entwurf muss so gestaltet sein, dass er als einziges Dokument für eine Implementierung (auch für nicht Teammitglieder_innen) ausreichend ist.

**Am Tag vor dem Praktikumstermin** bis **20:00 Uhr** (Mittwoch): bitte finaler Stand (als *.zip) zusenden, der in der Vorführung des Praktikums eingesetzt wird und alle Vorgaben erfüllen muss.

Am Tag des Praktikums findet eine Besprechung mit Teams statt. Die **Besprechung muss erfolgreich absolviert werden**, um weiter am Praktikum teilnehmen zu können. Bei der Besprechung handelt es sich nicht um die Abnahme.

**Abgabe**: Unmittelbar am Ende des Praktikums ist von **allen Teams** für die **Abgabe** der erstellte und sinnvoll dokumentierte Code abzugeben. Zu dem Code gehören die Sourcedateien und eine **Readme.txt** Datei, in der beschrieben wird, wie das System zu starten ist! Des weiteren ist der aktuelle Dokumentationskopf abzugeben. **In den Sourcedateien ist auf den Entwurf zu verweisen**, um die Umsetzung der Vorgaben zu dokumentieren. Alle Dateien sind als **ein *.zip Ordner** (mit cc an den/die Teampratner_in) per E-Mail abzugeben. Die Abgabe gehört zu den PVL-Bedingungen und ist einzuhalten, terminlich wie auch inhaltlich!

**Beachten Sie die Regularien zum Praktikum!**

# Graphentheoretische Konzepte und Algorithmen


## Aufgabe 1



1



## ADT Graph

**Objektmengen**: vertex, graph, name, value, filename, file, vertexlist

**Operationen**:

- **createG**: [d|ud] → graph
- **addVertex**: graph × vertex → graph
- **deleteVertex**: graph × vertex → graph
- **addEdge**: graph × vertex × vertex → graph
- **deleteEdge**: graph × vertex × vertex → graph
- **setAtE**: graph × vertex × vertex × name × value → graph
- **setAtV**: graph × vertex × name × value → graph
- **getValE**: graph × vertex × vertex × name → value
- **getValV**: graph × vertex × name → value
- **getIncident**: graph × vertex → vertexlist
- **getAdjacent**: graph × vertex → vertexlist

Konstruktoren & Mutatoren

Selektoren



2

# ADT Graph

**Objektmengen**: vertex, graph, name, value, filename, file, vertexlist

**Operationen**:

- **getTarget**: graph × vertex → vertexlist
- **getSource**: graph × vertex → vertexlist
- **getEdges**: graph → vertexlist
- **getVertices**: graph → vertexlist
- **importG**: filename × [d|ud] → graph
- **exportG**: graph × filename → file
- **printG**: graph × filename → dot

Selektoren





# Vorgaben

**Funktional** (nach außen)

**vertex**: ist eine ganze positive Zahl als eindeutige ID einer Ecke. Der Anwender bekommt auf die Struktur eines vertex keinen Zugriff. Er arbeitet immer nur mit dieser eindeutigen vertex-ID!

**vertexlist**: werden hier Kanten erwartet, so identifizieren jeweils zwei aufeinander folgende Ecken als Paar eine Kante, d.h. bei [v1,v2,v3,v4,...] identifizieren v1,v2 und v3,v4 eine Kante, nicht jedoch v2,v3.

**Attribute**: sind eindeutig im Namen (name) innerhalb aller Kanten zu einer bestimmten Ecke und aller Ecken. Hier können Vorgaben bzgl. der Namen gemacht werden.

Der **Graph** ist entweder gerichtet oder ungerichtet.

Es gibt keine **Mehrfachkanten**.

# Vorgaben

**Technisch** (nach innen)

**Listen**: (vertexlist) hier ist der Datentyp lists von Erlang zu verwenden. Ein Anwender kann dann frei auf diese Liste zugreifen, Änderungen beziehen sich aber nur auf diese Liste, nicht auf den Graphen!

Ob ein Graph **gerichtet** ist (directed) oder nicht (nondirected) wird in der globalen Variablen yndirected gespeichert (siehe util.erl)

Das **Standard-Attribut** für die nachfolgenden Aufgaben heißt weight.

Der **Standard-Fehlerwert** ist nil.

importG ist im Sinne des single-point-of-control mittels addEdge und setAtE zu realisieren.

Dateinamen und Namen von Attributen werden als Atome verwendet (keine Zeichenketten!).

addEdge fügt ggf. die Ecken in den Graphen ein, d.h. addEdge erfordert nicht addVertex

Die zugehörige Datei heißt **adtgraph.erl**. Weitere Dateien sind neben util.erl nicht zulässig.







# Vorgaben

Graphen sind über Dateien (*.graph) ein- bzw. auszulesen. Dies dient der Möglichkeit, spezielle Graphen einfach austauschen zu können. Kodierung: UTF-8 ohne BOM. Dabei ist folgendes Format zu verwenden:
[<Name Ecke1>,<Name Ecke2>,<Wert des Kantenattribut weight ggf. nil>,...]
Andere Attribute werden nicht im- oder exportiert.

`filename` ist der **Name der Datei ohne Endung**. Die jeweilige Endung (.graph oder .dot) wird automatisch angehängt, d.h. z.B. importG('graph_06',d) oder printG(Graph,graphprint) als Aufruf. In util.erl gibt es mit attachEnding bzw. attachStamp dazu Hilfsfunktionen.

Da es in dieser Aufgabe darum geht, die Datenstrukturen selbst zu implementieren, sind "Übergaben" an Datenstrukturen von Erlang OTP untersagt. Etwa die Verwendung von z.B. **dict** oder **sets**. Lediglich die Basis-Strukturen Liste (lists) und Tupel (tuple) dürfen eingesetzt werden.

Algorithmen auf diesen Basisstrukturen müssen selbst implementiert werden. Sonst wäre damit erfolgreich am Lerneffekt vorbei implementiert. Zur Unterstützung steht die Datei util.erl zur Verfügung.

# Fehlerbehandlung

Sollten

- nicht vorhandene Elemente gelöscht werden,
- die gleiche Kante mehrfach eingefügt werden,
- etc.

ist die Fehlerbehandlung durch „Ignorieren“ durchzuführen, d.h. es wird so gehandelt, als hätte die Operation nicht stattgefunden.

Aus Dokumentationsgründen können log-Dateien erstellt werden.

Falls ein Wert erwartet wird, wird nach Möglichkeit nil als Fehlerwert zurück gegeben.

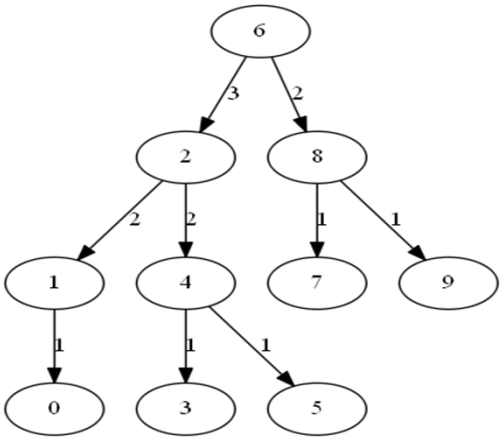




# *Graphviz*: Graph Visualization Software

```
digraph tree
  {
  6 -> 2 [label = 3];
  6 -> 8 [label = 2];
  2 -> 1 [label = 2];
  2 -> 4 [label = 2];
  1 -> 0 [label = 1];
  4 -> 3 [label = 1];
  4 -> 5 [label = 1];
  8 -> 7 [label = 1];
  8 -> 9 [label = 1];
  }
```

dot -Tsvg graph.dot > graph.svg
dot -Tpng graph.dot > graph.png



# Durchzuführende Tests

```
2> aufg1zeit:zeitmessung().
        addVertex
        deleteVertex
        >>--<<
        addEdge
        importG ud
```

Benötigt: testaufg1.graph

```
addVertex: 173ms, im Durchschnitt: 0.017300ms
deleteVertex: 2365ms, im Durchschnitt: 0.23650
addEdge: 1063ms, im Durchschnitt: 0.106300ms
deleteEdge: 590ms, im Durchschnitt: 0.059000ms
setAtE: 204ms, im Durchschnitt: 0.020400ms
getValE: 31ms, im Durchschnitt: 0.003100ms
setAtV: 126ms, im Durchschnitt: 0.012600ms
getValV: 46ms, im Durchschnitt: 0.004600ms
getIncident: 93ms, im Durchschnitt: 0.009300ms
getAdjacent: 0ms, im Durchschnitt: 0.000000ms
getTarget: 15ms, im Durchschnitt: 0.001500ms
getSource: 0ms, im Durchschnitt: 0.000000ms
getVertices: 16ms, im Durchschnitt: 0.001600ms
getEdges: 0ms, im Durchschnitt: 0.000000ms
importG mit d: 43765ms, im Durchschnitt: 0.437650ms
importG mit ud: 49656ms, im Durchschnitt: 0.496560ms
Gesamtzeit über alle Tests: 197143ms
```

measurements724000.log





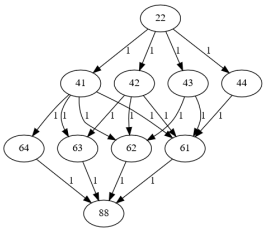


# Durchzuführende Tests

```
2> aufg1zeit:graphdrucken().
        importG
        exportG
Exportiere Graph mit 10 Ecken und 18 Kanten nach 'export_graph.graph'.
        printG
        dot -Tsvg
        done
ok
3>
```

Benötigt: graph_06.graph

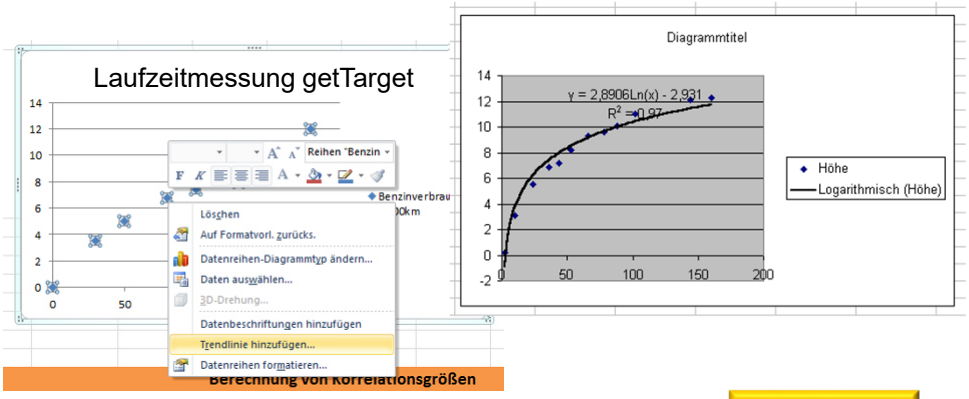
export_graph.svg
export_graph.dot
export_graph.graph




```
digraph gerichtet
{
22 -> 41 [label = 1];
22 -> 42 [label = 1];
22 -> 43 [label = 1];
22 -> 44 [label = 1];
41 -> 61 [label = 1];
41 -> 62 [label = 1];
41 -> 63 [label = 1];
41 -> 64 [label = 1];
42 -> 61 [label = 1];
42 -> 62 [label = 1];
42 -> 63 [label = 1];
43 -> 61 [label = 1];
43 -> 62 [label = 1];
44 -> 61 [label = 1];
61 -> 88 [label = 1];
62 -> 88 [label = 1];
63 -> 88 [label = 1];
64 -> 88 [label = 1];
}
```

```
[22,41,1,22,42,1,22,43,1,22,44,1,41,61,1,41,62,1,41,63,1,41,64,1,42,61,1,42,62,1,42,63,1,43,61,1,43,6
```

# Laufzeitmessung




Einheit: ms!

```
Start = erlang:timestamp(),
      Aufruf Algorithmus,
Ende = erlang:timestamp(),
Diffms = util:float_to_int(timer:now_diff(Ende,Start)/1000),
```







# Laufzeitmessung

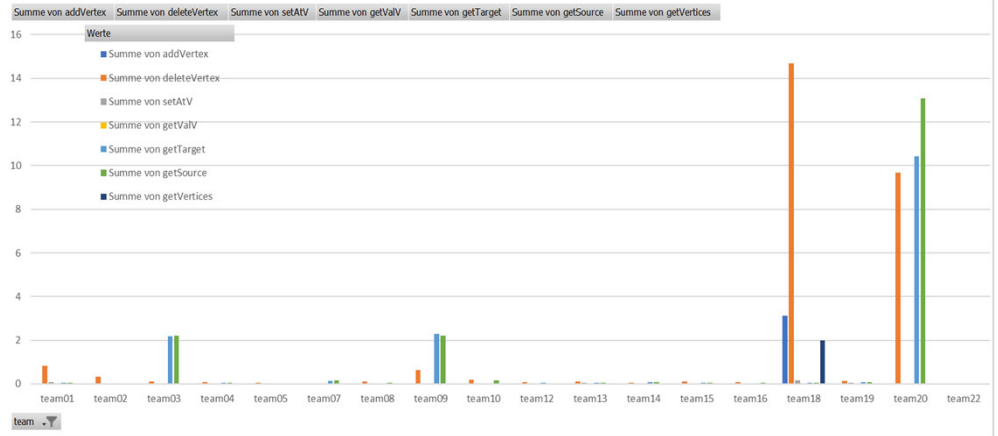
Deskriptive Methode, verfolgt den gleichen Ansatz, den Wissenschaftler verfolgen, um die Natur zu begreifen:

- **Beobachten** Sie bestimmte Eigenschaften der Software, im Allgemeinen begleitet von genauen Zeitmessungen.
- **Erstellen Sie ein hypothetisches Modell der Laufzeit**, das mit Ihren Beobachtungen übereinstimmt.
- **Prognostizieren** Sie mithilfe dieser Hypothese das Verhalten bei z.B. großen Eingabemengen.
- **Verifizieren** Sie Ihre Vorhersagen durch Betrachtung des Codes, insbesondere der Schleifen.
- **Fahren Sie fort**, indem Sie obige Schritte wiederholen, bis die Hypothese mit Ihren Beobachtungen übereinstimmt.

Informatik

Hochschule für angewandte Wissenschaften Hamburg

# Ergebnisse



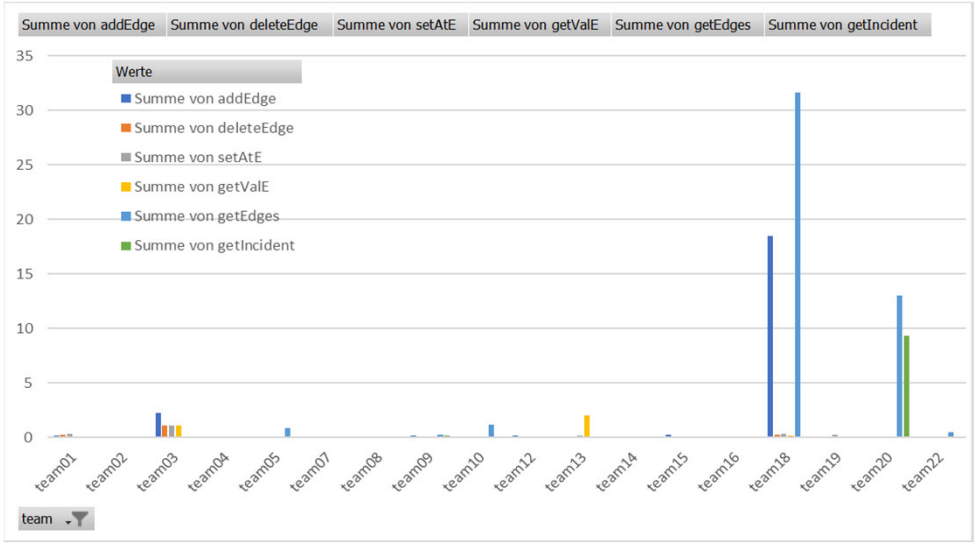

13

13

Informatik

Hochschule für angewandte Wissenschaften Hamburg

# Ergebnisse




14

14