Thema:

Interaktive Module zu Graphenproblemen für `autotool`

G. Martius, M. Lindemeyer, A. Kiel, J. Endrullis, M. Kreuz

Institut für Informatik - Theoretische Informatik, Universität Leipzig,
Augustusplatz 10-11, D-04109 Leipzig

Dr. Waldmann, Prof. Gerber

Fertigungsdatum: 27-05-2002

Inhaltsverzeichnis

1 Beschreibung **3**

2 Inhaltliche Beschreibung **3**

3 Softwarebeschreibung **10**

3.1 Kontrolleur 10

3.2 Kontrolleurmodule 13

3.2.1 Colorisierung 13

3.2.2 Hamilton 14

3.2.3 Clique 16

3.2.4 Graceful 18

3.2.5 PCP 21

3.3 weitere Module 23

3.3.1 Statistik 23

3.3.2 HTML 25

3.3.3 Labeling 26

3.3.4 Graph.Viz 27

4 Ergebnisbeschreibung **30**

5 Literaturangabe **31**

1 Beschreibung

Ziele und Anwendungen

Viele interessante Mengen M von Objekten sind dadurch charakterisiert, dass ein Objekt O genau dann zu M gehört, wenn ein zu O gehöriges Beweis-objekt B existiert. Oft ist leicht zu verifizieren, dass B zu O gehört, aber B selbst ist schwer zu finden. Klassische Beispiele aus der Berechenbarkeits-und Komplexitätstheorie sind ein drei-färbarer Graph, sowie eine Drei-Färbung, oder eine PCP-Instanz mit einem Lösungswort.
Wir hatten im Rahmen des Praktikums Funktionale Programmierung die Aufgabe, `autotool`-Module zur Verwaltung solcher Objekte und Beweise zu entwerfen und zu implementieren.
Insbesondere soll ein Wettbeweb ermöglicht werden, bei dem die Teilnehmer sich gegenseitig Aufgaben stellen. Man sendet ein passendes Paar (O,B) ein, das vom System geprüft wird. Daraufhin wird O (ohne B) veröffentlich, und die anderen Studenten müssen selbst dazu passende B finden. Nach bestimmten Kriterien werden Wertungspunkte auf Aufgabensteller und -löser verteilt.

Ziel ist die Erstellung eines interaktiven Moduls für das bestehende automatische Korrekturprogramm `autotool`.
Es soll dem Studenten des Grundstudiums die Möglichkeit gegeben werden, selbst Aufgaben und deren Lösungen einzusenden.

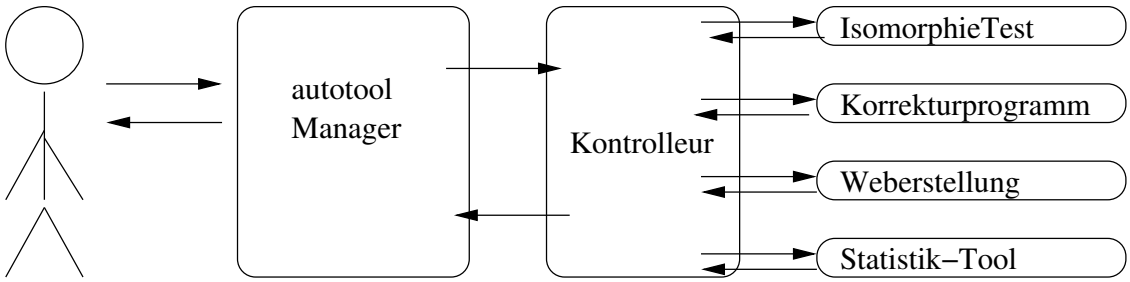
Aufgaben

Es soll ein Modul erstellt werden, welches die Koordination der Korrekturprogramme für die einzelnen Aufgabentypen übernimmt.
Die Aufgaben der Einsendungen sollen jeweils nur einmal vorkommen.
Bei den Aufgabentypen handelt es sich dabei um Hamilton, Colorisierung, Graceful Labeling, Clique und PCP. Dabei sollen weitere Aufgabentypen integriert werden können.
Dem Student soll in jedem Fall eine E-Mail geschickt werden, in der die Bewertung der Aufgabe und Lösung ersichtlich ist. Die Aufgaben und Lösungen sollen nach korrekter Einsendung im Internet präsentiert werden. Ausserdem soll eine Statistik mit geeigneter Punktverteilung erstellt werden, die ebenfalls im Internet zu veröffentlichen ist.

2 Inhaltliche Beschreibung

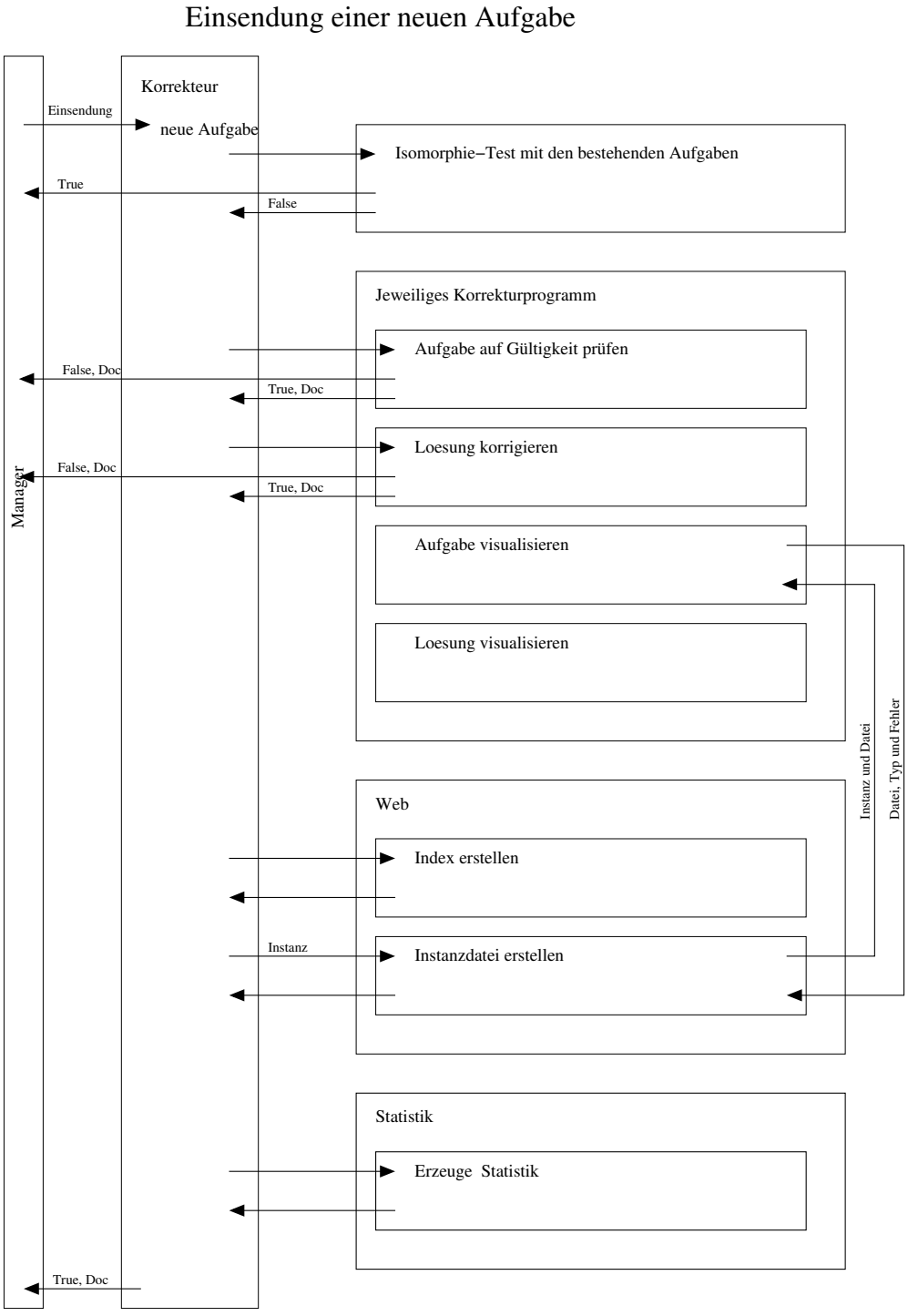
Sachverhaltbeschreibung

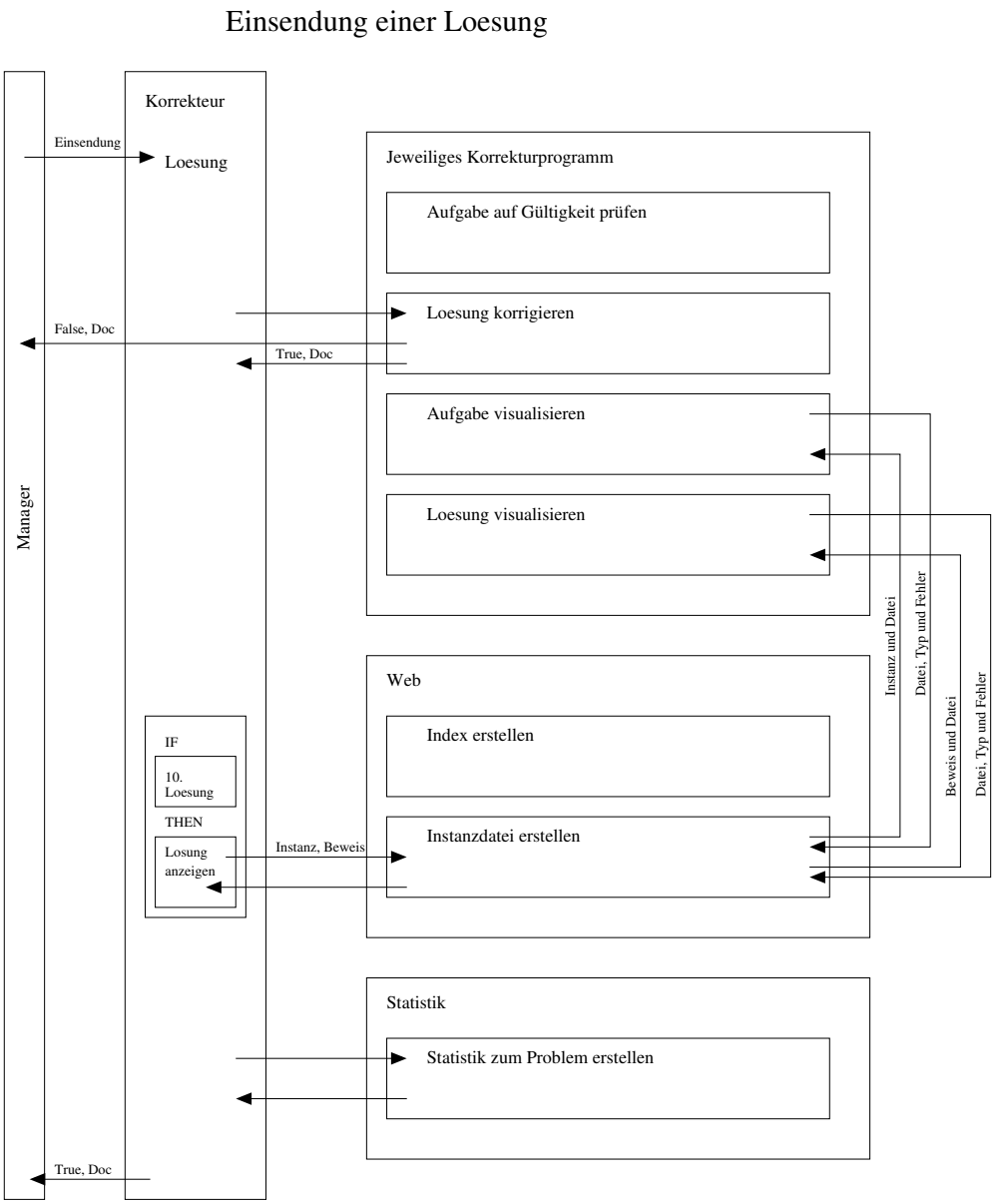
Unser Programm wird in Haskell programmiert.
Die Umgebung für das bestehende `autotool` ist Unix. Die Einsendung der Aufgaben vom Studenten erfolgt per E-Mail und wird mittels `procmail` an den bestehenden `autotool-Manager` geschickt. Dieser ruft je nach Subject ein vorhandenes Korrekturprogramm auf, in unserem Fall wird `Kontrolleur` aufgerufen, der seinerseits unsere Korrekturprogramme aufruft.


autotool
Manager
Kontrolleur
IsomorphieTest
Korrekturprogramm
Weberstellung
Statistik–Tool

Modellierung

Nachdem der Inhalt der Mail durch den `Manager` an den `Kontrolleur` gelangt, wird zunächst überprüft, ob es sich nur um eine Lösung einer schon bestehenden Aufgabe handelt, oder ob eine Aufgabe mit Lösung verschickt wurde.





Wenn eine neue Aufgabe gesendet wurde, muss ein `Isomorphie-Test` durchgeführt werden, damit gewährleistet wird, dass keine Aufgabe doppelt eingeschickt wird. Ist die Aufgabe schon vorhanden (True), wird dies dem Studenten sofort über den `Manager` mitgeteilt. Falls die Aufgabe noch nicht vorhanden war (False), wird die Aufgabe auf Gültigkeit (`valid`) in Abhängigkeit des Problems von dem jeweiligen Korrekturmodul geprüft. Ist die Aufgabe valid, wird vom `Kontrolleur` mit der Lösung normal weiter verfahren. War die Aufgabe noch nicht vorhanden, aber nicht valid wird dies dem Studenten wieder unverzüglich mitgeteilt.

Die eingesendete Lösung wird passend zu der Aufgabe überprüft. Ist die Lösung zu einer schon bestehenden Aufgabe korrekt oder falsch, wird dies dem Studenten sofort mitgeteilt. Ist die Lösung zu einer mitgeschickten Aufgabe falsch, wird dies ebenfalls dem Studenten sofort mitgeteilt. Ist die Lösung zu einer mitgeschickten Aufgabe richtig, wird die mitgeschickte Aufgabe, sowie die Lösung abgespeichert. In diesem Fall wird im Internet die `index.html` erneuert und die Aufgaben bzw. Lösungen visualisiert online verfügbar sein. Nun wird das `Statistik-Tool` aufgerufen und aktualisiert die Ausgabe. Zuletzt bekommt der Student über den `Manager` Glückwünsche der richtig gelösten Aufgabe sowie die entsprechenden Punkte.

Klassenstruktur

In Haskell sind Klassen Interfaces, Instanzen Implementierungen von Interfaces und anstatt Klassen von Objektorientierter Sprachen gibt es nur Datenstrukturen. Deswegen sprechen wir hier von Klassen und Datenstrukturen bzw. Datentypen.

Wir wollen eine Klasse `Problem` schreiben, die die Interfacedefinition der Korrekturprogramme darstellt. Diese Klasse ist eine Relation zwischen drei Datentypen. Diese sind `Problem`, `Instanz` und `Beweis`.

- `Problem` ist ein eigener Datentyp mit einem konstanten Konstruktor, welcher zur grundsätzlichen Unterscheidung der Probleme dient.
- `Instanz` ist eine beliebige Datenstruktur einer Aufgabe.
- `Beweis` ist eine beliebige Datenstruktur einer Lösung.

Die Funktionen, welche von den Instanzen implementiert werden müssen, sind:

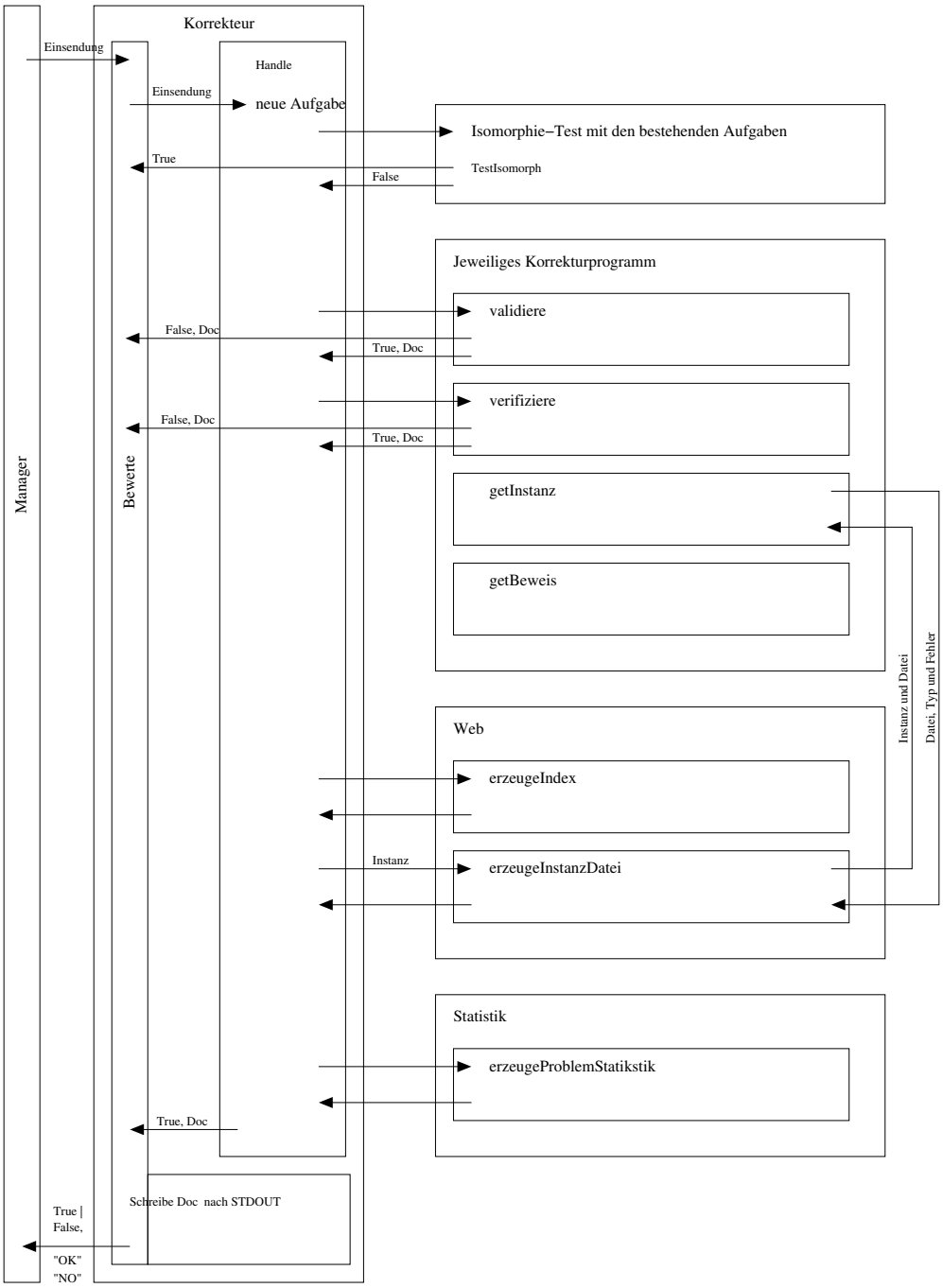
- `validiere` ist die Funktion, die die Gültikeit der Aufgabe prüft.
- `verifiziere` ist die Funktion, die die Korrektheit der Lösung überprüft.
- `getInstanz` ist die Funktion, welche die Visualizierung der Aufgabe erledigt
- `getBeweis` ist die Funktion, welche die Visualisierung der Lösung erledigt

Die Klasse wird dann für die einzelnen Probleme instantiiert. Somit entscheidet der Interpreter selbst, um welches Problem es sich handelt, und welches Korrekturprogramm benutzt wird.

Desweiteren soll die Datenstruktur `Einsendung` entweder eine Aufgabe oder eine Lösung repräsetieren, welche vom Studenten kommt. Eine Aufgabe besteht aus den Teilen `Problem`, `Instanz` und `Beweis`. Eine Lösung besteht aus den Teilen `Problem`, `Ident` und `Beweis`. Dabei sind `Problem` `Instanz` und `Beweis` die gleichen Datentypen, wie sie in der Klasse `Problem` verwendet werden. Hinzu kommt `Ident`, welcher zur Unterscheidung der einzelnen Aufgaben eines Problems genutzt wird.

Demnach ergibt sich dadurch eine exaktere Beschreibung des Programms wie folgt:

Einsendung einer neuen Aufgabe



Algorithmenbeschreibung

Isomorphietest

Es wird überprüft, ob die sortierten Knotengradlisten der beiden Graphen identisch sind. Falls sie identisch sind, wird ein weiterer Test durchgeführt, bei dem die Listen der angrenzenden Knotengradlisten auf Identität überprüft werden.

zusammenhängender Graph

Es wird überprüft, ob von jedem Knoten aus über eine Kette jeder andere beliebige Knoten erreichbar ist.

Colorisierung

Es wird überprüft, ob sich die zwei Knoten einer Kante in jeweils unterschiedlichen Klassen befinden. Die Knotenklassen können als Färbung betrachtet werden.

Hamilton

Es wird überprüft, ob die Knoten des Kreises auch in der Knotenmenge des Graphen sind. Danach wird getestet, ob zwischen zwei in der Kreis-Liste benachbarten Knoten eine Kante existiert.

Clique

Es wird überprüft, ob die Knoten in der Liste alle untereinander verbunden sind.

Graceful-Labeling

Es wird überprüft, ob das Labeling eine Abbildung von der Menge der Knoten ist, wobei das kleinste Label die Zahl 0 und das größte Label gleich der Anzahl der Kanten sein muss. Desweiteren muss das Labeling eine eineindeutige Abbildung sein und die Differenzenliste muss dicht gegenüber der natürlichen Zahlen sein.

PCP

Es wird überprüft, ob die Folge zwei identische Zeichenketten erzeugt.

Statistik

Punkte gibt es im Statistik-Modul für Aufgabenstellungen und jeweils die ersten 10 korrekten Beweise. Der Gesamtwert einer Aufgabe ist von der Anzahl der Einsendungen und der durchschnittlichen Lösungszeit abhängig. Je schwieriger eine Aufgabe ist, d.h. bei hoher Lösungszeit, desto mehr Punkte gibt es.

Anwendungsbeispiel

Student sendet neue Aufgabe und Lösung ein:

```
import Challenger
import Col.Col

student = Aufgabe{problem = Col
          , instanz = bsp_graph
          , beweis = bsp_loesung
          }
bsp_graph = Graph {
          knoten = mkSet [1,2,3,4,5]
          , kanten = mkSet [kante 1 2,
                            kante 2 3,
                            kante 3 4,
                            kante 4 5,
                            kante 5 1,
                            kante 3 5
          ]}

bsp_loesung = [[1,4],[2,5],[3]]
```

ein isomorpher Graph könnte sein:

```
bsp_graph = Graph {
          knoten = mkSet ['z','f','h','g','i']
          , kanten = mkSet [kante 'z' 'f',
                            kante 'i' 'z',
                            kante 'g' 'h',
                            kante 'f' 'g',
                            kante 'i' 'g',
                            kante 'h' 'i'
          ]}
```

3 Softwarebeschreibung - Spezifikation, Entwurf und Implementierung

3.1 Kontrolleur

Das Modul Kontrolleur ist das Hauptprogramm unseres Projektes. Es behandelt die Einsendungen der Studenten und verwaltet die Aufrufe aller weiternen Module.

Module des Kontrolleur

- Challenger: Die Definition der Klasse Problem und der Datentypen Einsendung sowie Ident.
- Instanzen: Modul um die Aufgaben zu verwalten.
- Beweise: Modul um die Loesungen zu verwalten.
- Kontrolleur: Das Koordinationsprogramm.
- Problems: Die Sammlung aller Problemtypen.

Challenger

Dieses Modul muss von jeder Datei eingebunden werden. Hier steht die Definition der Klasse Problem mit den benötigten Bedingungen. Eine Besonderheit die hier erwähnt werden muß, ist die "functional dependency" von dem Typ der Instanz auf den Typ des Beweises. Bei der Instanzierung der Klasse Problem für ein Problemtyp ist deshalb für den Beweis zwingend ein eigener Datentyp notwendig. `Doc` ist dabei ein formatierter Text, der eine Beschreibung über die Korrektur liefert. In den Funktionen `getInstanz` und `getBeweis` ist der vierte Parameter ein Dateiname (ohne Endung), in dem die Funktionen ihre formatierte Ausgabe abspeichern. Der Rückgabewert ist der vollständige Dateiname, der Dateityp (Endung der Datei) und der Fehlercode des Systemaufrufs. Nach dem Typ der Datei wird entschieden, ob ein Link angelegt wird oder der Inhalt der Datei direkt in die HTML Datei eingebettet wird. Hier der Quelltext:

```
class (Show p, ToDoc i, Show i, Read i, Iso i, ToDoc b, Show b, Read b)
    => Problem  p i b | b -> i where
       validiere :: p -> i -> b -> (Bool, Doc)
       verifiziere :: p -> i -> b -> (Bool, Doc)
       getInstanz :: p -> i -> b -> String -> IO(String, String, ExitCode)
       getBeweis  :: p -> i -> b -> String -> IO(String, String, ExitCode)
data Einsendung p i b
    = Aufgabe { problem::p, instanz::i, beweis::b}
    | Loesung { problem::p, ident :: Ident , beweis::b}
data Ident = Ident {aufgabe :: Int } deriving (Show, Read)
```

Instanzen

Diesen Modul bietet Funktionen zur Verwaltung der Instanzen (Aufgaben). Dazu gehört die Speicherung auf der Festplatte, sowie das Laden von Instanzen und Überblickslisten. Zu jedem Problem gehört ein Verzeichnis. In diesem Verzeichnis wird eine Datei angelegt Names: `instanzen.list`. Darin stehen zeilenweise Records vom Typ `Instanz`. Zu jeder Instanz gibt es eine Datei `XXXX.instanz`, wobei XXXX die Nummer der Instanz ist, in der die Instanz als Haskellstruktur abgespeichert ist.
Die Datenstruktur für eine Instanz:

```
data Instanz = Instanz
    { ident :: Ident
    , korrekt :: Bool
    , autor :: String
    , kommentar :: String
    , flag :: Int  -- unused now
    , datum :: CalendarTime
    } deriving (Show, Read)
```

Das Interface:

- speichere: Speichert eine Instanz
- lade: Lädt eine Instanz
- ladeAlle: Lädt alle Instanzen
- getAlle: Lädt die Liste der Instanzen
- getAlleNachString: Lädt die Liste der Instanzen, aber mit String als Problem.
- addToAlle: Fügt eine Instanz zur Liste der Instanzen hinzu
- nurKorrekte: Filtert die Liste der Instanzen auf korrekte Instanzen.

Beweise

Diesen Modul bietet Funktionen zur Verwaltung der Beweise (Lösungen). Dazu gehört die Speicherung auf der Festplatte, wowie das Laden von Beweisen und Überblickslisten. In dem Problem-Verzeichnis wird zu jeder Instanz eine Datei angelegt Names: `xxxx_beweise.list`. Darin stehen zeilenweise Records vom Typ `Beweis`. Zu jedem Beweis gibt es eine Datei `XXXX_YYYY.beweis`, wobei XXXX für Instanznummer und YYYY für Beweisnummer steht, in der der Beweis als Haskellstruktur abgespeichert ist.

Die Datenstruktur für einen Beweis:

```
data Beweis = Beweis
    { instanz :: Ident
    , ident :: Int
    , korrekt :: Bool
    , autor :: String
    , kommentar :: String
    , datum :: CalendarTime
    } deriving (Show,Read)
```

Das Interface:

- speichere: Speichert einen Beweis
- lade: Lädt einen Beweis
- ladeAlle: Lädt alle Beweise zu einer Instanz
- getAlle: Lädt die Liste der Beweise zu einer Instanz
- getAlleNachString: Lädt die Liste der Beweise zu einer Instanz, aber mit String als Problem.
- nurKorrekte: Filtert die Liste der Bewiese auf korrekte Beweise.

Kontrolleur

Hierbei handelt sich im das Kernstück dieses Projektes. Von diesem Modul werden alle anderen Module aufgerufen. Vom `autotool-Manager` wird eine Datei angelegt, in der das Kontrolleur Modul eingebunden ist und die `bewerte` Funktion mit der Einsendung aufgerufen wird. Dann wird wie in der Abbildung aus Section 2. vorgangen.

Problems

Dieses Modul sammelt alle Korrekteur-Module auf. Wenn ein neues Korrektur-Programm für einen neuen Problemtypen geschrieben wird, so muß dies nur hier eingebunden werden und in die Exportliste eingetragen werden. Dieses Modul wird von allen Modulen eingebunden, die Funktionen der Klasse Problem benutzten und damit die instanziierten Funktionen aufrufen.

3.2 Kontrolleurmodule

3.2.1 Colorisierung

Das Modul Colorisierung behandelt das Problem der Faerbung.

Problembeschreibung Die Menge der Knoten wird in k Klassen zerlegt. Der eingesendete Graph heisst colorisiert, wenn je zwei benachbarte Knoten des Graphen zu verschiedenen Klassen gehoeren. Eingesendet wird die Colorisierung des Graphen als Liste, wobei ein Listenelement eine Liste der Knoten einer Klasse (einer Faerbung) ist.

benötigte Module Folgende Module werden von Col benötigt:

- `Graph.Type` Datenstruktur und Klasseninstanzen eines Graphen
- `Graph.Util` Utilities für Graphen
- `Challenger` Hauptmodul, welches Col.hs aufruft und in dem die Klasse Problem definiert ist
- `ToDoc` Fehlermeldungen sind im Doc format
- `Set` fuer das Arbeiten mit Mengen
- `Sort` fuers Sortieren
- `Monad (guard)` fuer Fallunterscheidungen

Instanz der Klasse Problem Die drei Teile der Klasse Problem (Problem, Instanz, Beweis) sind (Col, Graph, Klassenliste).

Haskellcode

```
instance (ToDoc (Graph a), Show (Graph a), Read (Graph a), Iso (Graph a)
         , ToDoc (Ergeb a),Show (Ergeb a), Read (Ergeb a), Ord a, ToDoc a
         , Show a)
 => Problem Col (Graph a) (Ergeb a) where

    validiere Col g lsg
    verifiziere Col g (Ergeb lsg)
    getInstanz Col g lsg dateiName
    getBeweis Col g lsg dateiName
```

Verifiziere-Funktion Es wird geprueft, ob sich Knoten einer Kante in einer Klasse befinden. Ist dies so, handelt es sich nicht um eine Colorisierung. Ist dies fuer alle Kanten nicht so, handelt es sich um eine Colorisierung.

Darstellung der Instanz Der Graph wird normal mit Knotennamen aber ohne Färbung dargestellt.

Darstellung des Beweises Die Knoten werden je nach Klasse verschiedene Farben bekommen.

3.2.2 Hamilton

Das Modul Hamilton ist ein Kontrolleurmodul, welches das Problem Hamiltonkreis behandelt.

Problembeschreibung Hamiltonkreis behandelt das Problem, zu einem gegebenen Graphen einen Weg zu finden, bei dem jeder Knoten genau einmal durchlaufen wird und Start- und Endpunkt identisch sind.

benötigte Module Folgende Module werden von Hamilton benötigt:

- `Graph.Type` Datenstruktur und Klasseninstanzen eines Graphen
- `Graph.Util` Utilities für Graphen
- `Graph.Viz` zur Darstellung der Instanz und des Beweises als Bild
- `Challenger` Hauptmodul, welches Graceful aufruft und in dem die Klasse Problem definiert ist
- `ToDoc` Fehlermeldungen sind im Doc format
- `Set` stellt die Datenstruktur Menge zur Verfügung

Instanz der Klasse Problem Die drei Teile der Klasse Problem (Problem, Instanz, Beweis) sind (Hamilton, Graph, Kreis).

Haskellcode

```
instance (ToDoc (Graph a), Show (Graph a), Read (Graph a), Iso (Graph a)
         , ToDoc (Kreis a), Show (Kreis a), Read (Kreis a), Ord a, ToDoc a
         , Show a)
    => Problem Hamilton (Graph a) (Kreis a) where

    validiere Hamilton graph kreis
    verifiziere Hamilton graph kreis
    getInstanz Hamilton graph kreis dateiName
    getBeweis Hamilton graph kreis dateiName
```

Implementierungsdetails Bei der Validierung und Verifizierung werden verschiedene Tests abgearbeitet. Weitere Hilfsfunktionen führen die einzelnen Tests durch. Zu beachten ist, das der Graph als gerichteter Graph angesehen wird, und man bei Bedarf zwei Kanten für Hin- und Rückrichtung einfügen muss.

Validierung Beim Validieren wird überprüft, ob alle Kanten des Graphen auf vorhandene Knoten zeigen und ob der Graph zusammenhängend ist.

Verifizierung Beim Verifizieren wird geprüft ob zwischen zwei im Kreis aufeinander folgende Knoten eine Kante existiert. Der Aufwand dafür ist $O(Kantenanzahl \cdot Knotenanzahl)$.

Darstellung der Instanz Der Graph wird mit den Knotennamen dargestellt. Außerdem wird das Graphviz Tool `neato` genutzt und die Grafik im `png` Format ausgegeben.
Transformationsstruktur:

```
instanzTrans :: ShowText knoten => GVTrans knoten
instanzTrans = GVTrans
{ getGVProg = Neato
, getGVFormat = "png"
, isGVDirected = True
, getGVNID = showText -- Graphviz Knoten ID ist einfach showText knoten
, getGVNName = showText -- Knotenname auch
, getGVNLabel = Nothing
, getGVNColor = Nothing
, getGVNXAtts = Nothing
, getGVELabel = Nothing
, getGVEXAtts = Nothing
}
```

Darstellung des Beweises Der Beweis wird als Graph dargestellt bei dem die Knoten entsprechend dem Kreis durchnummeriert sind. Das Format ist ebenfalls `png`.

3.2.3 Clique

Das Modul Clique ist ein Kontrolleurmodul, welches das Problem Clique behandelt.

Problembeschreibung k-Clique behandelt das Problem, zu einem gegebenen Graphen eine Clique der Grösse k zu finden. Eine Clique ist ein vollverbundener Teilgraph.

benötigte Module Folgende Module werden von Clique benötigt:

- `Graph.Type` Datenstruktur und Klasseninstanzen eines Graphen
- `Graph.Util` Utilities für Graphen
- `Graph.Viz` zur Darstellung der Instanz und des Beweises als Bild
- `Challenger` Hauptmodul, welches Graceful aufruft und in dem die Klasse Problem definiert ist
- `ToDoc` Fehlermeldungen sind im Doc format
- `Set` stellt die Datenstruktur Menge zur Verfügung

Instanz der Klasse Problem Die drei Teile der Klasse Problem (Problem, Instanz, Beweis) sind (Clique, (Graph a, Int), TGraph).

Haskellcode

```
instance (ToDoc (Graph a), Show (Graph a), Read (Graph a), Iso (Graph a)
         , ToDoc (TGraph a), Show (TGraph a), Read (TGraph a), Ord a, ToDoc a
         , Show a)
    => Problem Clique (Graph a) (TGraph a) where


     validiere Clique graph tgraph
     verifiziere Clique graph tgraph
     getInstanz Clique graph tgraph dateiName
     getBeweis Clique graph tgraph dateiName
```

Implementierungsdetails Bei der Validierung und Verifizierung werden verschiedene Tests abgearbeitet. Weitere Hilfsfunktionen führen die einzelnen Tests durch. Zu beachten ist, das der Graph als gerichteter Graph angesehen wird, und man bei Bedarf zwei Kanten für Hin- und Rückrichtung einfügen muss.

Validierung Beim Validieren wird überprüft, ob alle Kanten des Graphen auf vorhandene Knoten zeigen und ob der Graph zusammenhängend ist.

Verifizierung Beim Verifizieren wird geprüft ob ein Knoten des TGraph mit allen anderen Knoten des TGraph verbunden wird (Hin- und Rückrichtung). Der Aufwand dafür ist $O(Cliquengroesse \cdot Kantenanzahl)$.

Darstellung der Instanz Der Graph wird mit den Knotennamen dargestellt. Außerdem wird das Graphviz Tool `neato` genutzt und die Grafik im `png` Format ausgegeben.
Transformationsstruktur:

```
instanzTrans :: ShowText knoten => GVTrans knoten
instanzTrans = GVTrans
{ getGVProg = Neato
, getGVFormat = "png"
, isGVDirected = True
, getGVNID = showText -- Graphviz Knoten ID ist einfach showText knoten
, getGVNName = showText -- Knotenname auch
, getGVNLabel = Nothing
, getGVNColor = Nothing
, getGVNXAtts = Nothing
, getGVELabel = Nothing
, getGVEXAtts = Nothing
}
```

Darstellung des Beweises Der Beweis wird als gefärbter Graph dargestellt. Die Knoten der Clique werden rot, die anderen Knoten werden schwarz dargestellt. Das Format ist ebenfalls `png`.

3.2.4 Graceful

Das Modul Graceful ist ein Kontrolleurmodul, welches das Problem Graceful Labeling behandelt.

Problembeschreibung Bei Graceful geht es darum, für einen Graphen ein graceful (schönes, hübsches) Labeling zu finden. Ein Labeling ist eine Abbildung von der Menge der Knoten in die Menge der natürlichen Zahlen.
Ein Labeling ist graceful, wenn gilt:

- Das kleinste Label ist die Zahl 0.
- Das grösste Label ist die Zahl e, wobei e die Anzahl der Kanten ist
- Wenn sich das Label einer Kante als Absolutwert der Differenz der beiden durch die Kante verbundenen Knoten definiert,
  - muss das kleinste Label die Zahl 1 sein und
  - die Menge der Label muss gleich der Menge {1,2,..,e} sein.

benötigte Module Folgende Module werden von Graceful benötigt:

- `Graceful.Labeling` der Beweis/die Lösung des Problems ist ein Labeling
- `Graph.Type` Datenstruktur und Klasseninstanzen eines Graphen
- `Graph.Util` Utilities für Graphen
- `Graph.Viz` zur Darstellung der Instanz und des Beweises als Bild
- `Challenger` Hauptmodul, welches Graceful aufruft und in dem die Klasse Problem definiert ist
- `ToDoc` Fehlermeldungen sind im Doc format
- `Sort` fürs Sortieren
- `Maybe` Maybe halt

Instanz der Klasse Problem Die drei Teile der Klasse Problem (Problem, Instanz, Beweis) sind (Graceful, Graph, Labeling).

Haskellcode

```
instance
( Ord knoten, Show knoten, ShowText knoten, ToDoc knoten
, ToDoc (Graph knoten), Show (Graph knoten), Read (Graph knoten)
, ToDoc (Labeling knoten), Show (Labeling knoten), Read (Labeling knoten)
)
=> Problem Graceful (Graph knoten) (Labeling knoten) where

validiere Graceful graph labeling
verifiziere Graceful graph labeling
getInstanz Graceful graph labeling dateiName
getBeweis Graceful graph labeling dateiName
```

Implementierungsdetails Bei der Validierung und Verifizierung werden verschiedene Tests abgearbeitet. Die Funktion `testeAlles` arbeitet diese Tests der Reihe nach ab und gibt entweder die Fehlermeldung des Ersten fehlgeschlagenen Tests oder eine Positivmeldung zurück. Der Typ der Funktion ist also:

```
testeAlles :: Doc -> [(Bool, Doc)] -> (Bool, Doc)
```

Jeder Test ist eine Funktion mit dem Rückgabewert `(Bool, Doc)`.

Validierung Beim Validieren wird überprüft, ob alle Kanten des Graphen auf vorhandene Knoten zeigen und ob der Graph zusammenhängend ist.
Ausgaben:

- `Es gibt Kanten, die zu dem Knoten ... zeigen, den es im Graph nicht gibt.`
- `Der Graph ist nicht zusammenhängend.`
- `Der Graph ist valid.`

Verifizierung Beim Verifizieren wird fogendes geprüft:

- Ist das Labeling eine Abbildung von der Menge der Knoten?
- Ist das kleinste Label die Zahl 0?
- Ist das größte Label gleich der Anzahl der Kanten?
- Ist das Labeling eine eineindeutige Abbildung?
- Ist die Differenzenliste dicht gegenüber der natürlichen Zahlen?

Ausgaben:

- `Das Labeling ist keine Abbildung von der Menge der Knoten.`
- `Das kleinste Label muss 0 sein.`
- `Das gößte Label muss mit der Anzahl der Kanten (...) übereinstimmen.`
- `Das Labeling ist keine eineindeutige Abbildung.`
- `Die Menge der Differenzen ist nicht dicht gegenüber den natürlichen Zahlen.`
- `Die Lösung ist richtig.`

Darstellung der Instanz Der Graph wird mit den Knotennamen aber ohne Labels dargestellt. Außerdem wird das Graphviz Tool `neato` genutzt und die Grafik im `PNG` Format ausgegeben. Der Graph ist natürlich ungerichtet.
Transformationsstruktur:

```
instanzTrans :: ShowText knoten => GVTrans knoten
instanzTrans = GVTrans
{ getGVProg = Neato
, getGVFormat = "png"
, isGVDirected = True
, getGVNID = showText -- Graphviz Knoten ID ist einfach showText knoten
, getGVNName = showText -- Knotenname auch
, getGVNLabel = Nothing
, getGVNColor = Nothing
```

```
, getGVNXAtts = Nothing
, getGVELabel = Nothing
, getGVEXAtts = Nothing
}
```

Darstellung des Beweises Der Graph wird mit Knotennamen, Knoten- und Kantenlabels dargestellt. Wobei die Label der Knoten die vom Labeling gegebenen und die der Kanten die Differenzen sind. Die Formate sind wie bei der Instanz.
Transformationsstruktur:

```
getBeweisTrans :: (ShowText knoten, Ord knoten)
=> (Labeling knoten) -> (GVTrans knoten)
getBeweisTrans labeling = GVTrans
{ getGVProg = Neato
, getGVFormat = "png"
, isGVDirected = True
, getGVNID = showText
, getGVNName = showText
, getGVNLabel = Just (getNLabel labeling)
, getGVNColor = Nothing
, getGVNXAtts = Nothing
, getGVELabel = Just (getELabel labeling)
, getGVEXAtts = Nothing
}
```

3.2.5 PCP

Das Modul PCProblem ist ein Kontrolleurmodul, welches das Postsche Korrespondezproblem behandelt.

Problembeschreibung Beim PCP geht es darum, aus mehreren geordneten Paaren von Wortfragmenten eine Folge zu finden, die jeweils zwei gleiche Wortketten erzeugt. Das Problem ist sehr komplex, bei drei Wortpaaren mit Alphabet {0,1} und Wortfragmenten mit Länge kleiner gleich 3 kann die minimale Lösungsfolge grösser als 50 sein.

benötigte Module Folgende Module werden von PCProblem benötigt:

- `Challenger` Hauptmodul, welches Graceful aufruft und in dem die Klasse Problem definiert ist
- `FiniteMap` stellt die Datenstruktur FiniteMap zur Verfügung
- `PCP.Type` stellt die Datenstruktur PCP zur Verfügung
- `System` wird zur Ausgabe in Dateien benötigt
- `Set` stellt die Datenstruktur Menge zur Verfügung
- `ToDoc` Fehlermeldungen sind im Doc Format

Instanz der Klasse Problem Die drei Teile der Klasse Problem (Problem, Instanz, Beweis) sind (PCProblem, PCP, Folge).

Haskellcode

```
instance (ToDoc (PCP a), Show (PCP a), Read (PCP a), Iso (PCP a)
         , ToDoc (Folge a), Show (Folge a), Read (Folge a), Ord a, ToDoc a, Show a)
    => Problem PCProblem (PCP a) (Folge a) where

validiere PCProblem  pcp folge
verifiziere PCProblem pcp folge
getInstanz PCProblem pcp folge dateiName
getBeweis PCProblem pcp folge dateiName
```

Implementierungsdetails Es wurden die oben aufgeführten Funktionen überladen. Weiterhin wurden einige Hilfsfunktionen geschrieben, die die Validierung und Verifizierung ermöglichen, sowie die Zeichenketten zur Visualisierung erzeugen.

Validierung Die Eingabegrössen werden auf Korrektheit geprüft:

- Ist das PCP nicht leer
- Ist die Folge nicht leer
- Sind die Elemente der Folge im PCP enthalten

Verifizierung Beim Verifizieren wird nur geprüft, ob die beiden erzeugten Ketten gleich sind. Der Aufwand hierfür ist linear. Es wird nicht geprüft, ob die Lösung die kürzeste ist, da der Rechenaufwand hierfür enorm werden kann.

Darstellung der Instanz Das PCP wird in einer Tabelle im HTML-Format abgelegt und dann auf der entsprechenden Seite eingebunden.

Darstellung des Beweises Der Beweis wird ebenfalls im HTML-Format abgelegt. Es wird die Lösungsfolge und und die erzeugte Zeichenkette abgespeichert.

3.3 weitere Module

3.3.1 Statistik

Das Statistik Modul erstellt eine Statistik und vergibt Punkte für die Einsendungen. Punkte gibt es für Aufgabenstellungen und jeweils die ersten 10 korrekten Beweise. Der Gesamtwert einer Aufgabe ist von der Anzahl der Einsendungen und der durchschnittlichen Lösungszeit abhänig. Es gibt mehr Punkte, umso schwieriger eine Aufgabe ist, d.h. bei hoher Lösungszeit. Aber die Aufgaben sollen nicht unlösbar sein, deshalb steigt der Wert einer Aufgabe mit der Anzahl der korrekten Lösugen.
Der Aufgabensteller erhält einen Teil des Gesamtwertes seiner Aufgabe. Die Lösungen bekommen die Punkte gestaffelt nach Platz und Lösungszeit.

Die Grundlage des Statistik-Moduls bilden die Korrekturergebnisse der Einsendungen, dabei werden folgende Module verwendet:

- Web.Html: für die Erzeugung der Webseite mit der Statistik
- Web.StatistikKonfiguration: die Bewertungsfunktionen und Konfiguration für das Statistik-Modul
- Web.Konfiguration: allgemeine Konfiguration, vor allem Ausgabeverzeichnis für die Webseiten
- Challenger, Problems, Beweise, Instanzen: für das Einlesen der Einsendungen und der Korrekturergebnisse, auf deren Grundlage die Statistik erstellt wird
- Locale, Time, Zeit: für die Bestimmung der Zeitdifferenzen zwischen den Einsendungen

Die wichtigste Funktion des Statistik-Modul ist "problemStatistik" String-> IO (). Dieser Funktion wird der Name des Problems als String übergeben. Die Funktion benutzt Instanzen.getAlleNachString, um alle Instanzen dieses Problems einzulesen. Dann wird in der Funktion instanzStatistik für jede Instanz eine einzelne Statistik mit Hilfe von Beweise.getAlleNachString erstellt. Hier findet auch die Bewertung der Einsendungen entsprechend deren Zeitpunkt statt. Für genauere Informationen siehe Statistik.hs.

Web.StatistikKonfiguration Das StatistikKonfiguration-Modul enthalt Funktionen zur Konfiguratio des Statistik-Moduls:

- punktePro_Aufgabe:: Int
  gibt den maximalen Wert einer Aufgabe an
- punktePro_Einsendung:: Int
  gibt die maximalen Punkte für eine Einsendung an, d.h. alle Punktvergaben über dieser werden gekappt
- wert_Ungeloest:: Int
  Wert einer bisher ungelösten Aufgabe (Vorschlag: 0 - kein Wert, weil Aufgaben lösbar bleiben sollen)
  alle Werte werden in Prozent von 0 bis 100 angegeben ([0..100])
- wert_SteigerungProLoesung:: Int
  mehr Lösungen bringen mehr Wert (Aufgaben sollen nicht unlösbar sein)
  d.h. bei n korrekten Lösugen ergibt sich der ist der Aufgabenwert aus n * wert_SteigerungProLoesung * punktePro_Aufgabe

- wert_einfache_Aufgabe:: Int
  Wert einer sehr einfachen Aufgabe
  d.h. Wert einer Aufgabe die weit unter der Zeit einer schweren Aufgabe gelöst wurde
- zeit_schwere_Aufgabe:: Int
  durchschnittliche Lösungszeit für eine schwere Aufgabe
  d.h. ab dieser Zeit wird eine Aufgabe als schwer gewertet und bekommt den vollen Wert: wert_schwere_Aufgabe
- wert_schwere_Aufgabe:: Int
  Wert einer schweren Aufgabe, d.h. ab der zeit_schwere_Aufgabe gibt es diesen Prozentsatz von Aufgabenwert
- berechnePunktVerteilung:: [CalendarTime]-> [Int]
  berechnet die Punktverteilung für Einsendung und [Lösungen] anhand der Zeiten der Einsendungen
  in der übergebenen Liste entspricht das erste Feld dem Zeitpunkt der Aufgabenstellung, die folgenden Felder sind Lösungen
  hier werden die Punkte indirekt proportional zu Platz und Zeit vergeben, der Aufgabensteller erhält 1/5 des Aufgabenwertes
  (denkbar wäre sicherlich auch eine reine Platz- bzw. Zeitbasierte Funktion)
- berechneAufgabenWert:: [Int]-> Int
  berechnet den Wert einer Aufgabe anhand der Zeitabstände zwischen den Einsendungen
  übergeben werden die Lösungszeiten einer Aufgabe in Sekunden
  der Wert steigt mit der Anzahl der Lösungen und einer hohen Lösungszeit, da diese auf anspruchsvolle Aufgaben hinweist
  (denkbar wäre auch ein fester Aufgabenwert)
- berechneWichtung:: Int->Int->Int
  berechnet einen gewichteten Wert anhand der Position in der Kette
  diese Funktion wird verwendet, weil die einzelnen Beweise in den Berechnungen unterschiedlich stark gewichtet werden
  d.h. der erste Beweis zählt am meisten, die folgenden gehen immer schwächer in die Bewertung ein
  bisher wird indirekt proportional zum Platz gewichtet
- werte_Platz:: Int-> Int
  Bewertung für den Platz der Lösung
  anhand des Platzes wird hier eine relative Punktzahl für die Lösung berechnet
  "relativ" heisst, die Punktzahl wird noch entsprechend der insgesamt zu vergebenden Punkte skaliert
- werte_Zeit:: Int-> Int
  Bewertung der Zeit der Lösung
  anhand der Zeit wird eine relative Punktzahl für die Zeit der Lösung berechnet
  "relativ" heisst, die Punktzahl wird noch entsprechend der insgesamt zu vergebenden Punkte skaliert
- gewichteterDurchschnitt:: (Int->Int->Int)->[Int]-> Int
  berechnet einen gewichteten Durchschnitt
  d.h. es wird der Durchschnitt der Liste berechnet, nachdem auf jeden Punkt die Platz-Wichtungsfunktion angewendet wird
- summeMitWichtung:: (Int->Int->Int)->[Int]-> Int
  berechnet eine gewichtete Summe (analog zu gewichteterDurchschnitt)
- berechneZeiten:: CalendarTime->[CalendarTime]->[Int]
  berechnet die Zeit von Aufgabenstellung bis zu jeder Lösung in Sekunden

3.3.2 HTML

Das Html-Module stellt Funktionen zur Erzeugung von HTML-Dokumenten zur Verfügung.

Datenstrukturen

```
data HTMLElement =
  HTMLDokument String [HTMLElement]
  | Image String String
  | Link String String
  | Section HTMLElement [HTMLElement]
  | SubSection HTMLElement [HTMLElement]
  | List [HTMLElement]
  | Tabelle [[HTMLElement]]
  | Text String
  | TextPlain String
  | TextVerbatim String
```

Diese Datenstruktur stellt den Kern des HTML-Moduls dar. Sie stellt Konstruktoren für alle wichtigen HTML-Elemente zur Verfügung. Kompliziertere Strukturen können durch die Schachtelung der Elemente erreicht werden. D.h. die Elemente HTMLDokument, Section, usw. können aus mehreren Child-Elementen bestehen.

Funktionen

- toHTML:: HTMLElement-> String
  Der Funktion toHTML wird ein HTML-Element übergeben. Die Funktion wandelt dieses in einen zur HTML-Spezifikation konformen String. Die Funktion arbeitet rekursiv und ruft sich mit allen Child-Elementen selbst auf.
- toFile:: String->HTMLElement-> IO()
  Der Funktion toFile wird der Name der Zieldatei und ein HTMLElement übergeben. Die Funktion schreibt die Ausgabe von toHtml in die angegebene Datei. Es wird also eine HTML-Datei erzeugt.
- erzeugeAnker :: [Anker] -> HTMLElement
  Der Funktion wird eine Liste von Ankern übergeben und es wird eine Indexstruktur auf diese erzeugt, ähnlich einem Inhaltsverzeichnis.

3.3.3 Labeling

Das Modul Labeling stellt eine Abbildung von einem beliebigen Typ in die Menge der natürlichen Zahlen zur Verfügung.
Speziell wird ein Labeling als eine Abbildung von Knoten in die Menge der natürlichen Zahlen benutzt.

benötigte Module Folgende Module werden von Graceful benötigt:

- `Graph.Graph` wegen der Datenstruktur `(Graph knoten)`. Wird nur bei der Funktion `getDiff` gebraucht. Es ist zu überlegen das Labeling komplett unabhängig vom Graph zu machen.
- `FiniteMap` weil das Labeling mit einer `FiniteMap` implementiert ist.
- `ReadFM` das Labeling muss auch gelesen werden können
- `ToDoc` für die `ToDoc` Instanz des Labeling
- `Maybe` Maybe halt
- `Sort` sortieren halt

Datenstruktur

```
newtype Labeling knoten = Labeling
    ( FiniteMap knoten Integer
    ) deriving (Show, Read)
```

Funktionen Die Menge der Funktionen ist nicht als vollständig zu betrachten. Es wurden die Benötigten implementiert.

Konstruktion

- `emptyLabeling :: Labeling knoten`
- `unitLabeling :: (knoten, Integer) -> Labeling knoten`
- `mkLabeling :: Ord knoten => [(knoten, Integer)] -> Labeling knoten`

Einsicht

- `knotenSet :: Ord knoten => Labeling knoten -> Set knoten`
- `labelSet :: Ord knoten => Labeling knoten -> Set Integer`
- `getLabel :: Ord knoten => Labeling knoten -> knoten -> Integer`
- `getSortedLabelList :: Labeling knoten -> [Integer]`
- `getDiff :: Ord knoten => Labeling knoten -> Kante knoten -> Integer`
- `sizeL :: Labeling knoten -> Int`
- `lToList :: Labeling knoten -> [(knoten, Integer)]`

3.3.4 Graph.Viz

Das Modul Graph.Viz stellt einen Graph mittels Graphviz dar.
Die Erstellung der Ausgabedatei mit dem Graphen ist dreistufig:

- Transformation `(Graph a) -> (Graphviz a)` mittels `GVTrans`
- erzeugen der Eingabedatei für das Graphviztool
- Aufruf des Graphviztools, welches die Ausgabedatei erzeugt

Datenstruktur Graphviz

- ist eine Datenstruktur eines Graphen, der alle für die Darstellung benötigten Informationen enthält
- im Gegensatz dazu enthält ein normaler Graph keine Informationen zur Darstellung
- die Informationen werden aus der Transformationsdatenstruktur `GVTrans` entnommen
- wenn man lediglich seinen Graphen darstellen will, braucht einen der Graphviz nicht zu interessieren - viel wichtiger ist `GVTrans`

```
data Graphviz = Graphviz
    { nodesGV :: GVNodeMap
    , edgesGV :: Set GVEdge
    , directedGV :: Bool
    } deriving (Show)

data GVNode = GVNode
    { nameGVN :: GVName
    , labelGVN :: Maybe GVLabel
    , colorGVN :: Maybe GVColor
    , xattsGVN :: Maybe GVXAtts
    } deriving (Show)

data GVEdge = GVEdge
    { idGVN1 :: GVNodeID
    , idGVN2 :: GVNodeID
    , labelGVE :: Maybe GVLabel
    , xattsGVE :: Maybe GVXAtts
    } deriving (Show, Eq, Ord)
```

Datenstruktur GVTrans Die Datenstruktur `GVTrans` ist Träger der Informationen, welche für die Ausgabe eines Graphen benötigt werden. Die einzelnen Felder sind Funktionen oder Maybe Funktionen. Falls man eine optionale Angabe nicht machen will schreibt man einfach `Nothing` hin.

```
data GVTrans a = GVTrans
    { getGVProg :: GVProg
    , getGVFormat :: GVFormat
    , isGVDirected :: Bool
    , getGVNID :: a -> GVNodeID
    , getGVNName :: a -> GVName
    , getGVNLabel :: Maybe (a -> GVLabel)
    , getGVNColor :: Maybe (a -> GVColor)
```

```
 , getGVNXAtts :: Maybe (a -> GVXAtts)
 , getGVELabel :: Maybe (Kante a -> GVLabel)
 , getGVEXAtts :: Maybe (Kante a -> GVXAtts)
 }
```

Die einzelnen Funktionen

- `getGVProg`
  - ist entweder Default, Dot oder Neato
  - beschreibt welches Graphviz Tool genutzt werden soll
  - Empfehlungen kann ich zur Zeit noch nicht geben
- `getGVFormat`
  - Format in dem die Ausgabe sein soll.
  - Möglich sind alle Formate, die Graphviz unterstützt
  - Beispiele: `ps` `gif` `png`
  - Bitte achte auf gültige Angaben.
- `isGVDirected`
  - True, falls der Graph gerichtet ist; False, falls nicht
- `getGVNID`
  - soll eine möglichst kurze eindeutige ID eines Knotens zurückgeben
  - idealerweise `Kx` wobei x eine laufende Nummer ist
  - diese ID wird nur intern verwendet
  - Ich wollte die eigentlich selber erzeugen. Nur wird es ziemlich komplex, wenn ich die Knoten erst durchnummeriere und dann bei jeder Kante schauen muss, welche ID ich dem Knoten gegeben habe. Ich werd das später vieleicht mal implementieren und die externe ID nicht mehr nutzen.
  - für den Anfang reicht übrigends ein `showText`, falls der Knoteninhalt nicht zu groß ist.
- `getGVNName`
  - Der Name des Knotens, der neben dem möglichen Label im Knoten steht. Es sind `\n`'s erlaubt.
  - Normalerweise sollte hier nur `showText` stehen.
- `getGVNLabel`
  - Optional, sonst wie Name, steht auch im Knoten.
  - Wurde aus Gründen der möglicherweise anderen Darstellung extra gemacht.
- `getGVNColor`
  - Hintergrundfarbe des Knotens. Hier sind übliche englische Farbbezeichnungen oder was das hier `#%2x%2x%2x` RGB auch immer heißen mag.
    `http://www.research.att.com/~erg/graphviz/info/colors.html`
- `getGVNXAtts`
  - Beliebige Graphviz Attribute des Knotens. Siehe Graphviz Docu.

- `getGVELabel`
  - Das Label der Kante.
- `getGVEXAtts`
  - Beliebige Graphviz Attribute der Kante. Siehe Graphviz Docu.

Einfaches Beispiel

```
myTrans :: Show a => GVTrans a  -- Typangabe, falls Knotentyp frei
myTrans = GVTrans
    { getGVProg :: Default
    , getGVFormat :: "gif"
    , isGVDirected :: False
    , getGVNID :: show
    , getGVNName :: show
    , getGVNLabel :: Nothing
    , getGVNColor :: Nothing
    , getGVNXAtts :: Nothing
    , getGVELabel :: Nothing
    , getGVEXAtts :: Nothing
    }
```

Beispiel, wenn man Informationen hineinbringen will

```
myTrans :: (Show knoten, Ord knoten)
    => (Labeling knoten) -> (GVTrans knoten)
myTrans labeling = GVTrans
    { getGVProg :: Default
    , getGVFormat :: "gif"
    , isGVDirected :: False
    , getGVNID = show
    , getGVNName = show
    , getGVNLabel = Just (getNLabel labeling)
    , getGVNColor = Nothing
    , getGVNXAtts = Nothing
    , getGVELabel = Just (getELabel labeling)
    , getGVEXAtts = Nothing
    }
```

`getNLabel` und `getELabel` müssen natürlich noch implementiert werden.

4 Ergebnisbeschreibung

Grenzen

Visualisierungen werden mit `graphviz` gemacht. Dieses Tool stellt komplexe Graphen unübersichtlich dar. Die Isomorphie für Graphen ist nicht vollständig implementiert, sondern lediglich ausreichend. Beim PCP ist die Funktion zum Isomorphietest noch nicht implementiert. Eventuell ist die Bewertungsfunktion der Statistik noch nicht ausgewogen, sie könnte nach einem ausgiebigen Test bei Bedarf angepasst werden.

Erweiterungsmöglichkeiten

Es ist relativ einfach, neue Korrekturmodule zu schreiben und einzubinden. Es müssen lediglich die Funktionen des Interface (Problem) überladen werden und das Modul in der Datei `Problems.hs` registriert werden. Weiterhin könnte man noch Funktionen schreiben, die das Problem der Graphisomorphie genauer prüfen. Diese wären jedoch für entsprechende Strukturen sehr rechenintensiv.

Hard/Softwarevorraussetzungen

Unix als Betriebssystem, Haskell Sprachspezifikation 98, Haskellinterpreter nach Dez 2001 (Modulkonzept).
Ein Server, auf dem das Autotool läuft, welches die Umgebung für Challenger darstellt.
Auf dem Server muss außerdem ein Webserver laufen, der die erzeugten Webseiten darstellen kann. Eine der Anzahl der Einsendungen entsprechende Performance des Servers muss für den reibungslosen Betrieb vorhanden sein.

Zusammenfassung

Bei diesem Projekt kamen zwei Faktoren zum Tragen. Zum einen eine ausgeprägte Teamarbeit mit fünf Leuten und zum anderen die Integration in eine bestehende Umgebung. Dabei war das Erlernen der Strukturen und Konzepte des `autotool` und die sinnvolle Verteilung der einzelnen Programmieraufgaben auf die einzelnen Mitwirkenden eine grundlegende Vorraussetzung.
Es ist uns einerseits gelungen, das `autotool` um die Fähigkeit der Annahme benutzerspezifischer Aufgaben zu erweitern und anderseits einige nützliche Kontrolleurmodule zu integrieren, welche sofort von Studenten eingesetzt werden können. Durch diese Dokumentation und die durchaus durchdachte Struktur hoffen wir, dass möglichst einfach zahlreiche neue Kontrolleurmodule hinzukommen werden.

5 Literaturangabe

Bücher

Computer and intractability
A Guide to the Theory of NP-Completeness
M.R.Garey/D.S.Johnson; NY, Freeman and Company 1979

Internet

www.haskell.org
www.cs.uu.nl/research/projects/generic-haskell
www-i2.informatik.rwth-aachen.de/Forschung/FP/Haskell
www.cs.chalmers.se/~nordland/ohaskell