# Entwurfsdokument

*Verteilter Algorithmus zur Bestimmung eines minimalen Spannbaumes*

Ziel dieser Aufgabe ist es, einen verteilten Algorithmus zur Berechnung eines minimalen Spannbaumes zu entwickeln. Die Implementierungssprache ist Erlang.

## Vorraussetzungen des Algorithmus

- Der Graph ist ungerichtet und gewichtet
- Jede Kante hat eine eindeutige Gewichtung
- Jeder Knoten kennt seine inzidenten Kanten und deren Gewichtung
- Anfangs ist jeder Knoten in einem ruhenden Zustand und wird entweder spontan geweckt oder wird durch eine eingehende Nachricht eines anderen Knoten geweckt
- Nachrichten können in beide Richtungen einer Kante geschickt werden

## Eigenschaften des Algorithmus

- asynchron
- Nachrichtenbasiert

## Arten von Kanten (Kantenspezifiezierung)

- **Branch:** Kanten die als Teil des minimalen Spannbaums bestätigt wurden
- **Rejected:** Kanten die bereits vom Spannbaum ausgeschlossen wurden.
- **Basic:** Noch zu klassifizierende Kanten

## Fragmentlevel

Zu Beginn des Algorithmus ist jeder Knoten ein einzelnes Fragment. Jedes dieser Fragmente besitzt das Level 0.
Das Level eines Fragments wird durch das zusammenführen zweier Fragmente gleichen Levels erhöht. Dies bedeutet, sollten zwei Fragmente mit dem Level *n* zusammengeschlossen werden erhält das Ausgangsfragment das Level *n+1*.
2 Fragmente unterschiedlichen Levels erhöhen das Level des neuen Fragments nicht, das Ausgangsfragmentlevel entspricht dem höheren Level der beiden Eingangsfragmente.
Die minimale Anzahl an Knoten lässt sich aus den Fragmentlevel herleiten:

$$min(V) = 2^n \; mit \; n : Fragmentlevel$$

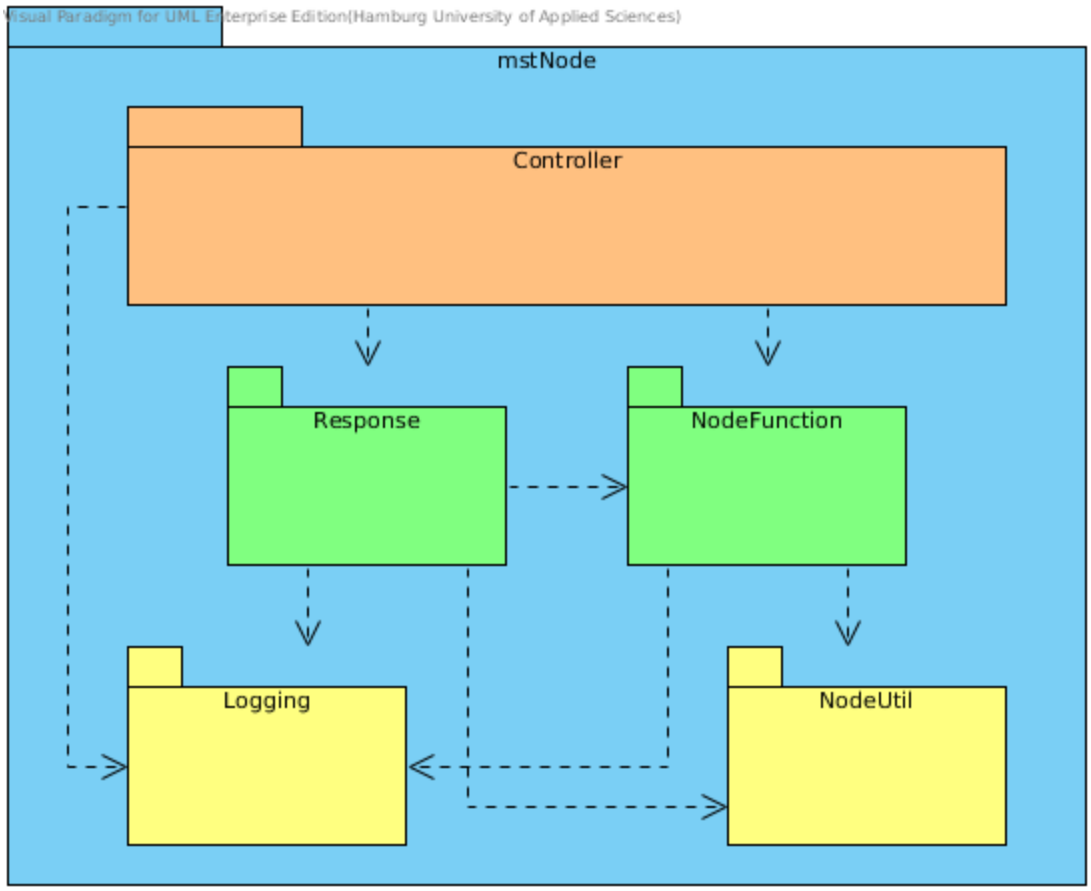
## Fragmentidentität

Jedes Fragment besitzt eine Identität. Die Identität entspricht der Gewichtung des Kerns des Fragmentes. Fragmente mit dem Level 0 haben dementsprechend keine Identität.

## Knotenzustände

- *sleeping* - initialer Zustand jedes Knotens
- *find* - Zustand eines Knoten bei der Suche einer minimalen ausgehenden Kante
- *found* - Kante gefunden, keine *test*-Nachrichten werden verschickt

## Implementierungsdetails




- Jedes Package ist ein Erlangmodule
- Es werden insgesamt zwei Prozesse verwendet
  - Prozess 1: MstNode, Controller, Response, NodeFunction, NodeUtil
  - Prozess 2: Logging
- Die Pfeile zeigen die möglichen Aufrufrichtungen der Modulfunktionen auf

- Das *MstNode-Package* selbst ist unser Modul, um eine Node zu starten
- Im *Controller-Package* läuft die *receive*-Funktion ab und ruft dann die entsprechenden Module auf
- Jedes Package hat die Möglichkeit über das *Logging-Package* Nachrichten in einer .log-Datei zu loggen
- Das Package *NodeUtil* bietet Hilfsfunktionen für die anderen Packages an
- Im Package *Response* sind alle Funktionen zur Beantwortung einer eingegangen Nachricht enthalten
    - accept(), connect(), initiate(), reject(), report(), test()
- Im Package *NodeFunction* sind alle Funktionen zur Bearbeitung einer eingegangen Nachricht enthalten
    - changeRoot(), report(), test(), wakeup()

## Messages

➔ {initiate, Level, FragName, NodeState, Edge}
- Startet die Suche nach einer *akmg*. Wird von den Knoten die sich direkt am *core* befinden (nach der Bildung eines neuen *core*) zu allen Knoten des Fragments verschickt.

➔ {test, Level, FragName, Edge}
- Bei der *akmg* Bestimmung wählen die außenliegenden Knoten eine *basic* Kante aus und verschicken eine *test* Nachricht, erwartet wird danach als Antwort entweder eine *reject* oder *accept* Nachricht

➔ {accept, Edge}
- Falls das Level des Fragments, das die *tes*t Nachricht empfangen hat, größer ist als das Level des Fragments das die Nachricht verschickt hat, schickt der Knoten die *accept* Nachricht

➔ {reject, Edge}
- Wird verschickt wenn der Knoten bereits im eigenen Fragment beinhaltet ist

➔ {report, Weight, Edge}
- Nachricht um die *akmg* (lokale beste Kante) an die innenliegenden Knoten weiterzureichen

➔ {changeroot, Edge}
- Wird verschickt wenn die *core*-Knoten eine *report* Nachrichten ausgetauscht haben und die neue *akmg* bestimmt wurde

➔ {connect, Level, Edge}
- *connec*t wird über die *akmg* verschickt, um den neuen Knoten in das Fragment einzuverleiben

# Definitionen:

**NodeState**:

- sleeping | find | found

**Level**:

- Ebene des Fragments eine ganze Zahl

**FragName**:

- Name des Fragments, das sich aus dem Gewicht des Cores ergibt

**Edge**:

- {Weight, NodeX, NodeY} eine Kante zwischen den Knoten Nodex und Nodey mit Gewicht Weight. Nodex und Nodey sind logische Namen von Knoten, die systemweit registriert sein müssen

**TestEdge**:

- {Weight, NodeX, NodeY} siehe *Edge*

**BestEdge**:

- {Weight, NodeY, NodeX} umgekehrte Edge-Struktur

**InBranch**:

- {Weight, NodeY, NodeX} umgekehrte Edge-Struktur

**EdgeOrddict**:

- *Key*: EdgeWeight, *Value*: {NodeName, EdgeState}