Automatisierte Komposition von wissensvermittelnden Dokumenten für das World Wide Web

-- ALGORITHMEN --

zu der von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

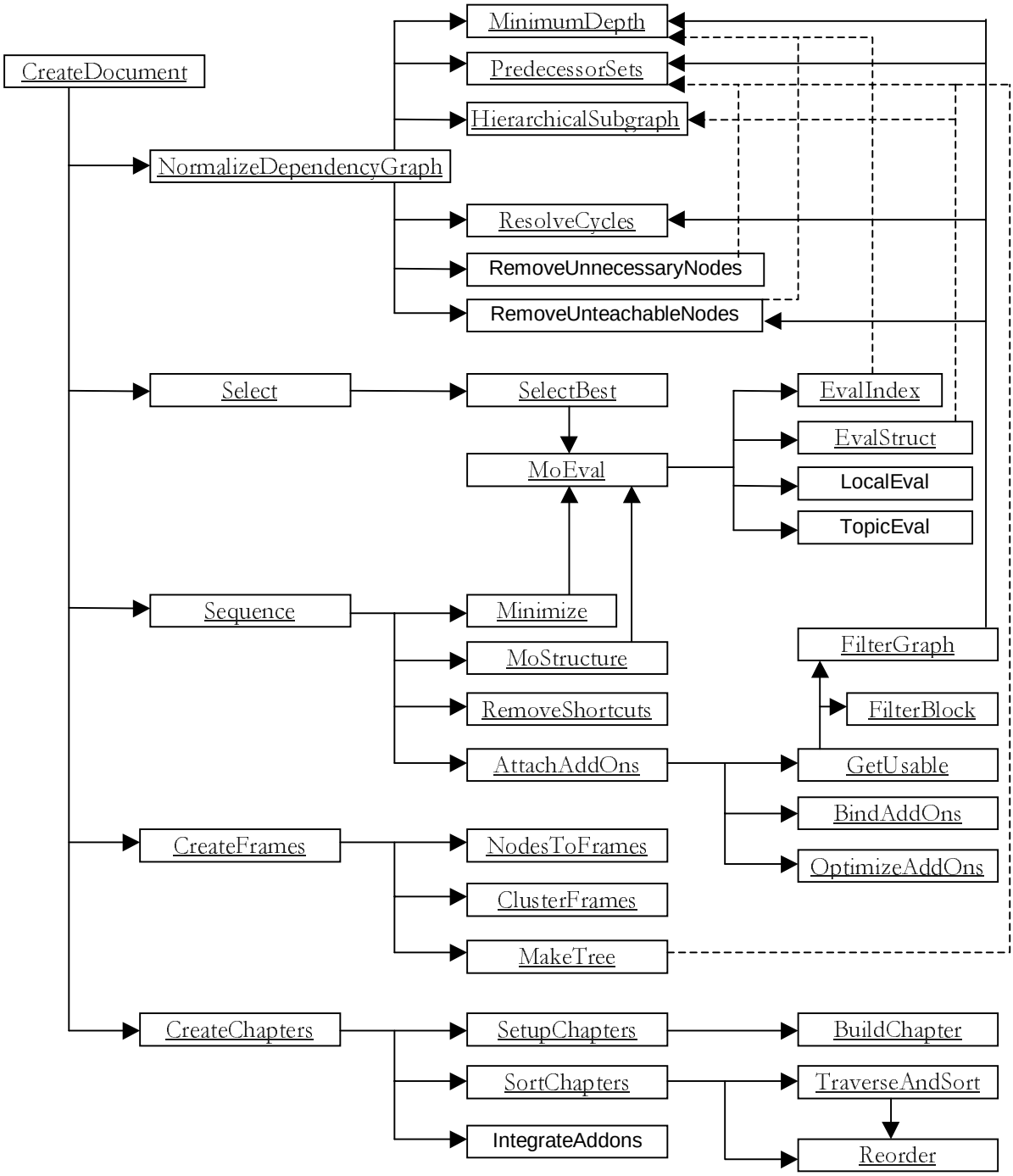
vorgelegt von

Diplom-Informatiker

Jörg Caumanns

geboren am 13. Mai 1968 in Dinslaken

Das nachfolgende Diagramm stellt die Abhängigkeiten zwischen den einzelnen Algorithmen zur Auswahl und Strukturierung von Medienobjekten dar. Durchgezogene Linien zeigen dabei direkte Aufrufe an ("CreateDocument" ruft den Algorithmus "NormalizeDependencyGraph" auf), während gestrichelte Linien auf eine starke Abhängigkeit hinweisen ("EvalStruct" benötigt das Ergebnis des Algorithmus "HierarchicalSubgraphs", ohne diesen Algorithmus selbst explizit aufzurufen).



Algorithmus MinimumDepth(G) Kapitel 6.3

Aufgabe:	Berechnung der **minimalen Tiefen** aller Medienobjekte und Stichworte eines gegebenen Abhängigkeitsgraphen
Eingabe:	Abhängigkeitsgraph G
Ausgabe:	-
Besonderheiten:	Die minimalen Tiefen der Medienobjekte und Stichworte werden in zwei "globalen" Vektoren mdepth und sdepth abgelegt. Diese Vektoren bilden eine Möglichkeit der Implementierung der in Kapitel 6.3 definierten Abbildungen $\mu_{depth}(m, G)$ und $\sigma_{depth}(s, G)$.

```
for all m ∈ M(G) do: mdepth_m ← 0        // Min. Tiefen aller Medienobjekte
for all s ∈ S(G) do: sdepth_s ← 0        // Min. Tiefen aller Stichworte
for all m ∈ M(G) do: h_m ← 0             // Hilfsvektor (s.o.)
iteration ← 0                            // Iterationszähler
nowready ← ∅                             // In der nächsten Iteration berechenbaren Objekte
ready ← M_root(G)                        // Die min. Tiefen der Wurzelknoten sind immer 0

do
  iteration ← iteration + 1
  nowready ← ∅

  for all m ∈ ready
      mdepth_m = iteration               // Die minimale Tiefe eines Medienobjekts entspricht der
                                         // Iteration, in der dieses Objekt bearbeitet wird.
      for all i ∈ μ_prv(m,I(G)) do       // Wenn die minimale Tiefe eines Medienobjektes berechnet
          s = kw(i)                      // wurde, können auch die minimalen Tiefen seiner
                                         // vermittelten Stichworte bestimmt werden.
          if sdepth_s = 0 then           // Wenn ein vermitteltes Stichwort noch nicht besucht wurde,
              sdepth_s = mdepth_m        // ist seine minimale Tiefe gleich der minimalen Tiefe des
                                         // aktuellen Medienobjekts.
              for all o ∈ σ_req(s,I(G)) do        // Überprüfen, ob durch die Bestimmung der
                  h_o = h_o + 1                   // min. Tiefe eines Stichwortes, ein davon
                  if h_o = |μ_req(o,I(G))| then   // abhängiges Medienobjekt berechenbar ist.
                      nowready ← nowready ∪ {o}
                  end if
              end for
          end if
      end for
  end for

  ready ← nowready                       // Wenn berechenbare Medienobjekte existieren, wird die
while ready ≠ ∅                          // nächste Iteration gestartet
```

Algorithmus PredecessorSets(G) Kapitel 6.5

Aufgabe:	Berechnung der **Vorgängermengen** aller Medienobjekte eines gegebenen Abhängigkeitsgraphen
Eingabe:	Abhängigkeitsgraph G
Ausgabe:	-
Besonderheiten:	Die Vorgängermengen der Medienobjekte werden in einem "globalen" Vektor ps abgelegt. Dieser Vektor bilden eine Möglichkeit der Implementierung der in Kapitel 6.5 definierten Abbildung $\mu_{ps}(m, G)$.

```
for all m ∈ M(G) do                    // Initialisierung der Vorgängermenge eines jeden
    ps_m ← ∅                           // Medienobjekts als leere Menge und
    color_m ← white                    // Markieren der Objekte als nicht besucht.
end for

for all m ∈ M(G) do                    // Sicherstellen, daß alle Medienobjekte
    if color_m = white then PS-Visit(m,G)   // von der rekursiven Tiefensuche (PS-Visit)
end for                                // erfaßt werden.
```

Algorithmus PS-Visit(G) Kapitel 6.5

Aufgabe:	Berechnung der **Vorgängermenge** eines Medienobjektes innerhalb eines nicht-zyklischen Abhängigkeitsgraphen
Eingabe:	Medienobjekt m Abhängigkeitsgraph G
Ausgabe:	Vorgängermenge von m in G
Besonderheiten:	Die Vorgängermenge des Medienobjekts wird zusätzlich in einem "globalen" Vektor ps abgelegt. Zur Berechnung der Vorgängermenge von m werden auch die Vorgängermengen aller direkten und indirekten Vorgänger von m berechnet.

```
color_m ← gray                         // Medienobjekt m als besucht markieren
ps_m ← μ_pre(m,G)                      // alle direkten Vorgänger sind Teil der Vorgängermenge

for all o ∈ μ_pre(m,G) do              // rekursives Berechnen der Vorgängermenge durch
    if color_o ≠ black then            //Zusammenfassen der Vorgängermengen der direkten
        ps_m ← ps_m ∪ PS-Visit(o,G)    // Vorgänger.
    else                               // Falls die Vorgängermenge eines Vorgängers bekannt ist,
        ps_m ← ps_m ∪ ps_o             // kann sie direkt aus ps übernommen werden.
    end if
end for

color_m = black                        // m als berechnet markieren

return ps_m
```

Algorithmus HierarchicalSubgraphs(G) Kapitel 6.6

Aufgabe:	Berechnung aller in einem Abhängigkeitsgraphen enthaltenen **hierarchischen Subgraphen**
Eingabe:	Abhängigkeitsgraph G
Ausgabe:	Menge HSG aller hierarchischen Subgraphen in G
Besonderheiten:	Jedes Element von HSG ist eine Menge von Medienobjekten, über denen anhand einer Indexmenge ein hierarchischer Subgraph aufgespannt werden kann.

```
HSG ← ∅                                  // Initialisierung der Resultatmenge

candidates ← FindSubgraphRoots( G )      // Bestimmen der Wurzelknoten von möglichen hierarch.
                                         // Subgraphen (Sequenzen der Länge 2)
for all w ∈ candidates do                // Verfolgen der gefundenen Sequenzen durch eine rekursive
                                         // Tiefensuche (FollowSubgraph)
    hsg_cand ← {w} ∪ FollowSubgraph( w, μndx(w,I(G)), G )

    if |hsg_cand| > 2 then               // ein hierarchischer Subgraph liegt vor, wenn die bei der
        HSG ← HSG ∪ hsg_cand             // Tiefensuche gefundenen Sequenzen aus mindestens drei
    end if                               // Knoten bestehen.
end for

return HSG
```

*// **Anmerkung:** Die Zweiteilung des Algorithmus in das Finden von potenziellen Wurzeln eines hierarchischen*
// Subgraphen und das anschließende Verfolgen dieser Wurzel ist allein in Performanz-Aspekten begründet:
// Das Verfolgen einer Sequenz ist sehr "teuer" und wird daher nur durchgeführt, wenn es sich auch lohnen könnte.

Algorithmus FindSubgraphsRoots(G) Kapitel 6.6

Aufgabe:	Berechnung aller in einem Abhängigkeitsgraphen enthaltenen **Sequenzen** mit einer Länge von mindestens 2 Medienobjekten, die theoretisch die Wurzel eines heirarchischen Subgraphen bilden können
Eingabe:	Abhängigkeitsgraph G
Ausgabe:	Menge *candidates* aller Medienobjekte, die Ausgangspunkt einer mindestens zwei Knoten langen Sequenz sind
Besonderheiten:	Jedes Element von HSG ist eine Menge von Medienobjekten, über denen anhand einer Indexmenge ein hierarchischer Subgraph aufgespannt werden kann.

```
candidates ← ∅

// Das notwendige Kriterium für die Wurzel eines hierarchischen Subgraphen ist, daß dieser Knoten sämtliches von
// mindestens einem seiner Nachfolger benötigte Vorwissen bereitstellt

for all m ∈ M(G) do                    // paarweiser Vergleich eines jeden Medienobjekts mit
    for all o ∈ μsucc(m,G) do          // jedem seiner Nachfolger
        all_provided ← true
        for all i ∈ μreq(o,I(G)) do    // Wenn o ein Stichwort benötigt, daß von m nicht vermittelt
                                       // oder ebenfalls verlangt wird, bilden m und o keinen HSG
            if not ∃j ∈ μndx(m,I(G)): kw(i)=kw(j) und
                                      prv(i,I(G)) > req(j,I(G)) then
                all_provided ← false
            end if
        end for

        if all_provided then
            candidates ← candidates ∪ {m}
        end if
    end for
end for

return candidates
```

Algorithmus FollowSubgraph(m, G) Kapitel 6.6

Aufgabe:	Rekursive Berechnung aller in einem Abhängigkeitsgraphen enthaltenen **Sequenzen**
Eingabe:	Medienobjekt m, von dem aus die weitere Sequenzberechnung durchgeführt wird
	Menge der Stichworte sp, die auf dem Weg von einer Wurzel zu m bereits vermittelt wurden
	Abhängigkeitsgraph G
Ausgabe:	Menge hsg aller Medienobjekte, die auf einem sequenziellen, abgeschlossenen Pfad liegen, der mit dem Knoten m beginnt
Besonderheiten:	hsg ist eine Menge von Medienobjekten, über denen anhand der Indexmenge von G ein hierarchischer Subgraph aufgespannt werden kann.

```
hsg ← {m}                                  // m ist die Wurzel des zu berechnenden Subgraphen

for all o ∈ μsucc(m,G) do                  // Für jeden Nachfolger von m wird überprüft, ob sein
    all_provided ← true                    // verlangtes Vorwissen durch das in dem bisher
    for all i ∈ μreq(o,I(G)) do            // berechneten Subgraphen vermittelte Wissen abgedeckt ist.
        if not ∃j ∈ sp: kw(i)=kw(j) und prv(i)>req(j) then
            all_provided ← false
        end if
    end for

    // Falls ein Nachfolger diese Bedingung erfüllt wird er zum hierarchischen Subgraphen hinzugefügt.
    // Dieser Nachfolger bildet nun den Startpunkt der nächsten Rekursion von FollowSubgraph
    if all_provided then
      hsg ← hsg ∪ FollowSubgraph(o,sp ∪ μprv(o,I(G)))
    end if
end for

return hsg
```

Algorithmus CreateDocument(M, S, I, u^c) Kapitel 7.1

Aufgabe:	Berechnung eines an ein vorgegebens Nutzungsszenario angepaßten hypermedialen Lehrsystems
Eingabe:	Menge M der zur Verfügung stehenden Medienobjekte
	Menge S der von diesen Medienobjekten referenzierten Stichworte
	Menge I der M mit S verknüpfenden Indexeinträge
	Zugrundeliegendes Nutzungsszenario u^c
Ausgabe:	Website
Besonderheiten:	-

```
for all b ∈ u_blocks(u_c) do          // Für jeden Block des Nutzungsszenarios werden
                                      // Medienobjekt-, Seiten- und Kapitelstruktur berechnet
    Berechne M^b, S^b, I^b            // Nutzeranpassung (siehe Kapitel 5)
                                      // Erstellen des Abhängigkeitsgraphen
    G^b ← (M^b, ⋃_{m∈M^b} μ_kw(m,I^b), ⋃_{m∈M^b} μ_req(m,I^b), ⋃_{m∈M^b} μ_prv(m,I^b))

    G^bn ← NormalizeDependencyGraph(G^b,b)      // Normalisierung

    G^b* ← Select( G^bn, b )                    // Auswahl der geeignetsten Medienobjekte
    T^b* ← Sequence( G^b*, M^b, b )             // Erstellen der Medienobjektstruktur
    (F^b*, O^b*) ← CreateFrames( T^b*, b )      // Erstellen der Seitenstruktur
    (C^b*, F^free) ← CreateChapters( F^b*, Ob* ) // Erstellen der Kapitelstruktur

    Printout( C^b*, F^free )                    // Erstellen der HTML-Seiten dieses Blocks
end for

PrintTitle( )                         // Erstellen einer übergeordneten Index-Seite
```

Algorithmus NormalizeDependencyGraph(G^b, b) Kapitel 7.2

Aufgabe:	Normalisierung eines Abhängigkeitsgraphen und Berechnung aller in Kapitel 6 beschriebenen Maßzahlen
Eingabe:	Abhängigkeitsgraph G^b
	Blockbeschreibung b
Ausgabe:	Normalisierter Abhängigkeitsgraph G^{bn}
Besonderheiten:	-

```
G^bn ← ResolveCycles( G^b )          // Entfernung von Zyklen
MinimumDepth( G^bn )                 // Berechnung der minimalen Pfade
G^bn ← RemoveUnteachableNodes( G^bn ) // Entfernen nicht-erlernbarer Knoten
PredecessorSets( G^bn )              // Berechnen der Vorgängermengen
G^bn ← RemoveUnnecessaryNodes( G^bn, b) // Entfernen nicht benötigter Knoten
HierarchicalSubgraphs( G^bn )        // Berechnen der hierarchischen Subgraphen
return G^bn
```

Algorithmus ResolveCycles(G, b) Kapitel 7.2

Aufgabe:	Auflösen aller Zyklen in einem Abhängigkeitsgraphen durch Entfernen geeigneter Knoten und Kanten
Eingabe:	Abhängigkeitsgraph G
	Blockbeschreibung b
Ausgabe:	Zyklenfreier Abhängigkeitsgraph
Besonderheiten:	-

```
M' ← ∅; I' ← ∅                          // Initialisierung
Berechne SCC(G)                         // Berechnen der starken Zusammenhangskomponenten
                                        // und Analyse der dabei gefundenen Zyklen

for all m ∈ M(G) do                     // Entfernen aller Knoten, die nicht auflösbare Zyklen
    if μcycle(m,G) ≠ "deadend" then     // verursachen
        M' ← M' ∪ {m}
        I' ← I' ∪ μndx(m,I(G))
    end if
end for

G' = G(M', I')                          // Analyse des verbleibenden Graphen
Berechne SCC( G(M', I') )

for all scc ∈ SCC do
    if ∃m ∈ scc with μcycle(m,G') = "resolver" then
        // Falls ein Zyklus einen resolver enthält, werden dessen sämtliche, auf
        // dem Zyklus liegenden Ausgangskanten entfernt.
        RemoveCyclicEdges( G', scc, m )
    Else
        for all m ∈ scc do
            // Falls kein resolver existiert, müssen alle auf dem Zyklus
            // liegenden Ausgangskanten der Connectoren entfernt werden.
            if μcycle(m,G') = "connector" then
                RemoveCyclicEdges( G', scc, m )
            end if
        end for
    end if
end for

return G(M', I')                        // Erstellen eines zyklenfreien Graphen
```

Algorithmus RemoveCyclicEdges(G, scc, m) Kapitel 7.2

Aufgabe:	Entfernen aller Ausgangskanten der von dem Medienobjekt m benötigten Stichworte, die auf dem Zyklus scc liegen.
Eingabe:	Abhängigkeitsgraph G
	Starke Zusammenhangskomponente scc
	Medienobjekt m
Ausgabe:	-
Besonderheiten:	Dieser Algorithmus verändert die Kantenmenge von G

```
for all i ∈ μprv(m,I(G)) do
    s ← kw(i)          // Untersuchen der Eingangskanten jedes von m benötigten Stichwortes
    for all j ∈ σreq(s,I(G)) do
        o ← mo(j)      // Falls ein s benötigendes Medienobjekt o auf dem gleichen Zyklus liegt
                       // wie m, wird die Kante von m zu s aus dem Abhängigkeitsgraph entfernt.
        if o ∈ scc then
            Iprv(G) ← Iprv(G) \ {i}
        end if
    end for
end for
```

Algorithmus Select (G^{bn}, b) — Kapitel 7.3.1

Aufgabe:	Auswahl eines Subgraphen des Abhängigkeitsgraphen, in dem alle verlangten Lernziele erlernbar sind
Eingabe:	Normalisierter Abhängigkeitsgraph G^{bn}
	Blockbeschreibung b
Ausgabe:	Subgraph $G^{b+} \subseteq G^{bn}$ mit b_content(b) $\lozenge\lozenge$ G^{b+}
Besonderheiten:	-

```
M^b+ ← ∅                                  // Menge der ausgewählten Medienobjekte
S^prv ← ∅                                 // Menge der vermittelten Inhalte
S^req ← b_content(b) \ S^prv              // Menge der zu vermittelnden Stichworte

while S^req ≠ ∅ do                        // Iteration, bis ein abgeschlossener Subgraph gefunden ist

    m ← SelectBest( G^bn, M^b+, S^prv, S^req, b )   // Auswahl des besten Medienobjektes
    M^b+ ← M^b+ ∪ {m}                     // Erweitern von M^b+ um das ausgewählte Medienobjekt
    S^prv ← S^prv ∪ μ_kwprv(m,I(G^bn))    // Erweitern der vermittelten Stichworte
    S^req ← S^req \ μ_kwprv(m,I(G^bn))    // Reduzieren der offenen Stichworte
    newopen ← μ_kwreq(m,I(G^bn)) \ S^prv  //Erweitern der offenen Stichworte

    S^req ← S^req ∪ newopen
end while

return G(M^b+, I(G^bn))
```

Algorithmus SelectBest (G^{bn}, M^{b+}, S^{prv}, S^{req}, b) Kapitel 7.3.2

Aufgabe:	Auswahl des heuristisch best-bewerteten Medienobjekts
Eingabe:	Normalisierter Abhängigkeitsgraph G^{bn}
	Bereits ausgewählte Medienobjekte M^{b+}
	Bereits vermittelte Stichworte S^{prv}
	Menge der noch offenen Stichworte S^{req}
	Blockbeschreibung b
Ausgabe:	In Abhängigkeit von den Eingabewerten best-bewertetes Medienobjekt m^{best}
Besonderheiten:	-

```
m^best ← 0          // am besten bewertetes Medienobjekt
s^best ← 0          // am besten bewertetes Stichwort
p^best ← 0          // größte Wahrscheinlichkeit

for all s ∈ S^req do          // Suche nach Objekten, die ein offenes Stichwort exklusiv vermitteln
    if |σ_moprv(s,I(G^bn)) \ M^b+| = 1 then
        return mbest ∈ (σ_moprv(s,I(G^bn)) \ M^b+)
    end if
end for

for all s ∈ S^req do      // Bewertung aller vermittelnden Medienobjekte aller offenen Stichworte
    best_p ← 0; sum_eval ← 0; best_mo ← 0
    for all i ∈ σ_prv(s,I(G^bn)) do
        if mo(i) ∉ M^b+ then
            e_mo(i) ← MoEval(i, s, G^bn, M^b+, S^prv, S^req, b)
            sum_eval ← sum_eval + e_mo(i)
        end if
    end for

    for all i ∈ σ_prv(s,I(G^bn)) do          // Durch Normierung mit der Summe der
        if mo(i) ∉ M^b+ then                 // Bewertungen aller s vermittelnden
            p ← e_mo(i) / sum_eval           // Medienobjekte können die Bewertungen
            if p > best_p then               // als Wahrscheinlickeiten interpretiert werden
                best_p ← p: best_mo ← mo(i)
            end if
        end if
    end for

    // Relativierung der höchsten Wahrscheinlichkeit durch die minimale Tiefe des zu vermittelnden Stichworts
    best_p ← best_p * ( 1 + log10(σ_depth(s,G^bn)))

    if best_p > p^best then                  // Auswahl des bestbewerteten Medienobjekts
        p^best ← best_p: m^best ← best_mo:  s^best ← s
    end if
end for

return m^best
```

Algorithmus MoEval (i, G^{bn}, M^{b+}, S^{prv}, S^{req}, b) Kapitel 7.3.4

Aufgabe:	Bewertung eines Medienobjekts bezüglich seiner Eignung zur Vermittlung eines Stichwortes
Eingabe:	Zu bewertender Indexeintrag i
	Normalisierter Abhängigkeitsgraph G^{bn}
	Aktueller Subgraph M^{b+}
	Offene Stichworte S^{req}
	Bereits vermittelte Stichworte S^{prv}
	Blockbeschreibung b
Ausgabe:	Numerische Bewertung des Medienobjekts
Besonderheiten:	Die Gewichtungen sollten so gewählt werden, dass der Wert dieser Funktion immer zwischen 0 und 1 liegt.

```
eval^local ← LocalEval( mo(i), G^bn ) // lokale Bewertung

if s ≠ 0 then
    eval^kw ← EvalIndex( i, G^bn )   // Bewertung bezüglich des Stichworts
else
    eval^kw ← 0                      // Sonderfall: Bewertung unabhängig von einem Stichwort
end if

eval^struct ← EvalStruct( mo(i),G^bn,M^b+,S^req,S^prv,b ) // Strukturelle Bewertung

eval^doc ← TopicEval(i, G^bn, M^b+, S^req, S^prv, b)   // Anwendungsabhängige Bewertung

// Gewichten und Zusammenfassen der Bewertungen
return ((α^kw * eval^kw)+(α^struct * eval^struct)+(α^local * eval^local)+(α^doc * μ^doc))
```

Algorithmus EvalIndex (i, G^{bn}) Kapitel 7.3.4

Aufgabe:	Bewertung eines Medienobjekts bezüglich seiner Eignung zur Vermittlung eines Stichwortes
Eingabe:	Zu bewertender Indexeintrag i
	Normalisierter Abhängigkeitsgraph G^{bn}
Ausgabe:	Numerische Bewertung des Medienobjekts
Besonderheiten:	Die Gewichtungen sollten so gewählt werden, dass der Wert dieser Funktion immer zwischen 0 und 1 liegt.

```
eval^prv ← (prv(i) / r)              // Bewertung des Grades der Vermittlung
for all j ∈ μ_prv(mo(i), I(G^bn)) do
    if prv(j)* 0.9 > prv(i) then
        eval^prv ← eval^prv * 0.9    // Abwertung für jedes stärker vermittelte Stichwort
    end if
end for

eval^pos ← (rel(i) / r)              // Bewertung der Relevanz
for all j ∈ μ_ndx(mo(i), I(G^bn)) do
    if rel(j)* 0.9 > rel(i) then
        eval^pos ← eval^pos * 0.8    // Abwertung für jedes relevantere Stichwort
    end if
end for

// Vergleich der minimalen Tiefen durch Normierung der Differenz
eval^depth ← (σ_depth(kw(i),G) - μ_depth(mo(i),G)) / σ_depth(kw(i),G)

cps ← 0                              // Anzahl Medienobjekte mit  i in ihrer Vorgängermenge
for all j ∈ σ_prv(kw(i), I(G^bn)) do
    if mo(i) ∈ μ_ps(mo(j),G) then cps ← cps + 1
end for

// Bewertung in Relation zu der Anzahl konkurrierender Medienobjekte setzen.
if |σ_prv(kw(i),I(G^bn))| > 1 then
    eval^ps ← cps / (|σ_prv(kw(i),I(G^bn))| - 1)
else
    eval^ps ← 0
end if

// Zusammenfassung und Gewichtung
e ← (α^aprv * eval^prv)+(α^apos * eval^pos)+(α^adepth * eval^depth)+(α^aps * eval^ps)

// Abwertung für Medienobjekte, die das Stichwort vertiefen
if req(i) > 0 then e ← e * 0.9

return e
```

Algorithmus EvalStruct (i, G^{bn}, M^{b+}, S^{prv}, S^{req}, b) Kapitel 7.3.4

Aufgabe:	Bewertung eines Medienobjekts bezüglich der in der Blockbeschreibung enthaltenen Präferenzen zu Struktur und Kompaktheit des zu erzeugenden Blocks
Eingabe:	Zu bewertendes Medienobjekt m
	Normalisierter Abhängigkeitsgraph G^{bn}
	Aktueller Subgraph M^{b+}
	Offene Stichworte S^{req}
	Bereits vermittelte Stichworte S^{prv}
	Blockbeschreibung b
Ausgabe:	Numerische Bewertung des Medienobjekts
Besonderheiten:	Die Gewichtungen sollten so gewählt werden, dass der Wert dieser Funktion immer zwischen 0 und 1 liegt.

```
// Ermitteln der hierarchischen Subgraphen, in denen das zu bewertende Medienobjekt enthalten ist
hsg ← { h ∈ HSG(G^bn) | m ∈ h }
eval^hsg ← |hsg|
for all h ∈ hsg do                    // Wenn bereits Teile eines m enthaltenden hierarchischen
    already_selected ← |h ∩ M^b+|     // Subgraphen ausgewählt sind, schlägt sich dies positiv
    eval^hsg ← eval^hsg + (already_selected * 0.5) // in der Bewertung von m nieder
end for
eval^hsg ← eval^hsg / |{o ∈ M^b+ | ∃h ∈ HSG: o ∈ h }|

// Berechnung der bereits ausgewählten direkten Vorgänger und Nachfolger
i^pre ← { o ∈ μ_pre(m,G^bn) ∩ M^b+ | μ_kwreq(m,I(G^bn)) ⊆ μ_kw(o,I(G^bn)) }
i^succ ← { o ∈ μ_succ(m,G^bn) ∩ M^b+ |μ_kwreq(oI(G^bn)) ⊆ μ_kw(m,I(G^bn)) }

// Bewertung anhand der Struktur von m und seinen bereits ausgewählten Vorgängern und Nachfolgern
if |i^pre| < 2 then
    eval^i ← (|i^succ| * 2) + |i^pre|   // Branch
else
    eval^i ← (|i^succ| * 2) - |i^pre|   // Merge
end if
.
m^pre ← μ_ps(m,G^bn) ∩ M^b+                  // Ausgewählte Objekte bestimmen, die „vor“
m^succ ← { o ∈ M^b+ | m ∈ μ_ps(o,G^bn) }// oder „hinter“ dem zu bewertenden Objekt liegen

// Medienobjekte, die mit Hilfe von m mit in ihrer Vorgängermenge enthaltenen Objekten verknüpft werden.
m^mid ← { o ∈ m^succ | m^pre ∩ μ_ps(o,G^bn) ≠ ∅ }
eval^m ← |m^mid|*2 + |m^succ| - (|m^pre| - |m^mid|)

return (α^hsg * eval^hsg) + (α^i * eval^i) + (α^m * eval^m)
```

Algorithmus Sequence (G^{b+}, M^b, b) Kapitel 7.4

Aufgabe:	Erstellung einer Medienobjektstruktur aus den vom Algorithmus "Select" ausgewählten Medienobjekten
Eingabe:	Abgeschlossener Subgraph G^{b+} des Abhängigkeitsgraphen
	Insgesamt verfügbare Medienobjekte M^b
	Blockbeschreibung b
Ausgabe:	Medienobjektstruktur T^{b*} des Blocks b aus dem Abhängigkeitsgraphen G
Besonderheiten:	-

```
G^b* ← Minimize( G^b+, b )          // Entfernen redundanter Knoten
T^b+ ← MoStructure( G^b* )          // Erstellen der Medienobjektstruktur
T^b+ ← RemoveShortcuts( T^b+ )      // Entfernen redundanter Kanten
T^b* ← AttachAddOns( T^b+, G^b*, M^b, b )   // Hinzufügen zusätzlicher Objekte
```

Algorithmus Minimize (G^{b+}, b) — Kapitel 7.4.1

Aufgabe:	Entfernung redundanter Knoten aus einem Abhängigkeitsgraph
Eingabe:	Abgeschlossener Subgraph G^{b+} des Abhängigkeitsgraphen
	Blockbeschreibung b
Ausgabe:	Abhängigkeitsgraph $G^{b*} \subseteq G^{b+}$, mit b_content(b) ◊◊! G^{b*}

```
nextrun: // Label für die Neuberechnung der redundanten Medienobjekte nach Entfernen eines Knotens
  for all s ∈ S(G^b+) do          // Berechnen des maximalen Grades an Vorwissen, mit dem jedes
      sreqmax_s ← 0               //  in G^b+ enthaltene Stichwort von einem ebenfalls in G^b+
      for all i ∈ σ_req(s,I(G^b+)) do   // enthaltenen Medienobjekt benötigt wird
          if req(i) > sreqmax_s then  sreqmax_s ← req(i)
      end for
                                  // Für jedes in G^b+ enthaltene Stichwort wird ermittelt, welche
      sprv_s ← ∅                  // ebenfalls in G^b+ enthaltene Medienobjekte dieses Stichwort mit
      for all i ∈ σ_prv(s,I(G^b+)) do   // ausreichendem Grad (≥ sreqmax_s ) vermitteln.
          if prv(i) + ε ≥ sreqmax_s then sprv_s ← sprv_s ∪ {mo(i)}
      end for
  end for

  for all m ∈ M(G^b+) do          // Für jedes in G^b+ enthaltene Medienobjekt wird berechnet,
      mprv_m ← ∅                  // welche der in G^b+ enthaltenen Stichworte von diesem Objekt mit
      for all i ∈ μ_prv(m,I(G^b+)) do   // ausreichendem Grad vermittelt werden.
        if (sreqmax_kw(i) > 0) or (kw(i) ∈ b_content(b)) then
             mprv_m ← mprv_m ∪ { kw(i) }
        end if
      end for
  end for

  red ← ∅                         // Ein Medienobjekt ist echt redundant, wenn es keines der in G^b+
  for all m ∈ M(G^b+) do          // enthaltenen Stichworte mit ausreichendem Grad vermittelt.
      if mprv_m = ∅ then red ← red ∪ {m}
  end for

  for all m ∈ red do              // Entfernen der echt redundanten Medienobjekte
      M(G^b+) ← M(G^b+) \ {m}
      I(G^b+) ← I(G^b+) \ μ_ndx(m,I(G^b+))
  end for
  if red ≠ ∅ then goto nextrun // Neuberechung

  red ← ∅                         // Ein Medienobjekt ist abhängig redundant, wenn jedes von ihm
  for all m ∈ M(G^b+) do          // mit ausreichendem Maß vermittelte Stichwort auch von
      redundant ← True            // mindestens einem anderen Medienobjekt in G^b+ mit
      for all s ∈ mprv_m do       // ausreichendem Maß vermittelt wird
          if |sprv_s| = 1 then
              redundant = false
              exit for
          end if
      end for

      if redundant then red ← red ∪ {m}
  end for
```

```
if red ≠ ∅ then                 // Da abhängig redundante Medienobjekte in Konkurrenz
    mworst ← 0                  //  zueinander stehen können, muß über eine
    evalworst ← MAX_EVAL+1      // Bewertungsfunktion entschieden werden, welches
    for all m ∈red do           // Medienobjekt zuerst aus G^b+ entfernt wird
        e ← MoEval2(m, G^b+, red)
        if e < evalworst then
            mworst ← m
            evalworst ← e
        end if
    end for

    // Entfernen des schlechtesten Medienobjektes und Neuberechnung
    M(G^b+) ← M(G^b+) \ {mworst}
    I(G^b+) ← I(G^b+) \ μ_ndx(mworst,I(G^b+))
    goto nextrun
end if
```

Algorithmus MoStructure (G^{b+}) Kapitel 7.4.2

Aufgabe:	Überführung eines Abhängigkeitsgraphen in eine Medienobjektstruktur
Eingabe:	Minimal abgeschlossener Abhängigkeitsgraph G^{b+}
Ausgabe:	Aus G^{b+} abgeleitete Medienobjektstruktur T^{b+}
Besonderheiten:	-

```
nodes^t ← ∅                      // Knoten der Medienobjektstruktur
edges^t ← ∅                      // Kanten der Medienobjektstruktur
links^t ← ∅                      // Hyperlinks der Medienobjektstruktur

for all m ∈ M( G^b* ) do
    links_m ← ∅                  // Für jedes Medienobjekt werden drei (temporäre) Vektoren
    pre_m ← ∅                    // angelegt, in denen Eingangskanten, Vorgänger und
    mt_m ← (m, ∅, true)          // Medienobjektknoten des Objekts verwaltet werden.
End for

for all m ∈ M( G^b* ) do         // Berechnung der Verknüpfungen
    for all i ∈ μ_req(m,I(G^b*)) do  // zu jeder Eingangskante wird die am besten passende
        j ← BestPrv( i, G^b* )       // Ausgangskante ermittelt
        links_m ← links_m ∪ {(i,j)} // und sowohl als Kante als auch als Hyperlink in
        pre_m ← pre_m ∪ {mo(j)}      // die Medienobjektstruktur übernommen
    end for
end for

for all m ∈ M( G^b* ) do         // Erstellen der Medienobjektstruktur aus den zuvor berechneten
    nodes^t ← nodes^t ∪ {mt_m}   // Knoten, Kanten und Hyperlinks
    for all o ∈ pre_m do
        edges^t ← edges^t ∪ {(mt_o, mt_m)}
    end for
    links^t ← links^t ∪ links_m
end for

return (nodes^t, edges^t, links^t)
```

Algorithmus RemoveShortcuts (T^{b+}) Kapitel 7.4.3

Aufgabe:	Entfernung von Abkürzungen aus einer Medienobjektstruktur
Eingabe:	Medienobjektstruktur T^{b+}
Ausgabe:	T^{b+} ohne redundante Kanten
Besonderheiten:	-

```
edges^t ← t_edges(T^b+)

for all t ∈ T^b+ do
    for all v ∈ τ_pre(t,T^b+) do
        for all w ∈ τ_pre(t,T^b+) do
            if v ∈ τ_ps(w,T^b+) then
                edges^t ← edges^t \ {(v,t)}
                exit for
            end if
        end for
    end for
end for

return ( t_nodes(T^b+), edges^t, t_links(T^b+) )
```

Algorithmus AttachAddOns(T^{b+}, G^{b+}, M^b, b) Kapitel 7.5

Aufgabe:	Erweitern einer gegebenen Medienobjektstruktur um geeignete Zusatzobjekte
Eingabe:	Medienobjektstruktur T^{b+}
	Abhängigkeitsgraph der Medienobjektstruktur G^{b*}
	Insgesamt verfügbare Medienobjekte M^b
	Blockbeschreibung b
Ausgabe:	Um Zusatzobjekte erweiterte Medienobjektstruktur T^{b+}
Besonderheiten:	Dieser Algorithmus verändert die als Eingabe gegebene Medienobjektstruktur

// Ermitteln der Zusatzobjekte, die sowohl zu der Blockbeschreibung als auch zu der Medienobjektstruktur passen
$(M^a, G^{a*}) \leftarrow$ `GetUsable`(G^{b*}, M^b, b)

// Binden der Zusatzobjekte an die inhaltlich "nächsten" Medienobjekte
$(B, A, F) \leftarrow$ `BindAddOns`$(M(G^{b*}), M^a, G^{a*})$

// Ranking und "Ausdünnen" der Bindungen
$T^{b*} \leftarrow$ `OptimizeAddOns`$(G^{a*}, B, A, F, M^a, T^{b+}, b)$

Algorithmus GetUsable (G^{b+}, M^b, b) — Kapitel 7.5.1

Aufgabe:	Ermitteln aller zu einer gegebenen Blockbeschreibung kompatiblen Zusatzobjekte, die sich in eine ebenfalls gegebene Medienobjektstrktur integrieren lassen
Eingabe:	Abhängigkeitsgraph der Medienobjektstruktur G^{b*}
	Insgesamt verfügbare Medienobjekte M^b
	Blockbeschreibung b
Ausgabe:	In die Medienobjektstruktur integrierbare Medienobjekte M^a
	Abhängigkeitsgraph G^{a*} aus Basis- und Zusatzobjekten
Besonderheiten:	-

// Herausfiltern aller Medienobjekte, die auf die in der Blockbeschreibung angegebenen Zusatzobjektfilter passen
$M^{ab} \leftarrow$ FilterBlock(M^b, b)

// Herausfiltern aller Zusatzobjekte, die innerhalb der Medienobjektstruktur erlernbar wären
$G^{a*} \leftarrow$ FilterGraph(G^{b*}, M^{ab})

$M^a \leftarrow M^{ab} \cap M(G^{a*})$
return (M^a, G^{a*})

Algorithmus FilterBlock (M^b, b) — Kapitel 7.5.1

Aufgabe:	Ermiteln aller Medienobjekte, deren Genre mit einem in der Blockbeschreibung für Zusatzobjekte festgelegten Genre übereinstimmen
Eingabe:	In die Medienobjektstruktur integrierbare Medienobjekte M^a
	Abhängigkeitsgraph G^{a*} aus Basis- und Zusatzobjekten
Ausgabe:	Der Blockbeschreibung entsprechende mögliche Zusatzobjekte M^{ab}
Besonderheiten:	Der Algorithmus unterstützt keine Hierarchien von Genres (Subgenres, etc.)

```
M^ab ← ∅                                  // Initialisierung

For all z ∈ b_additional(b) do            // Paarweiser Vergleich der Medienobjektgenres mit
    for all m ∈ M^b do                    // den in der Blockbeschreibung angegebenen
        if genre(m) = z_genre(z) then     // Zusatzobjekt-Genres
            M^ab ← M^ab ∪ {m}
        end if
    end for
end for

return M^ab
```

Algorithmus FilterGraph (G^{b+}, M^a) Kapitel 7.5.1

Aufgabe:	Ermitteln aller Medienobjekte, die in einem um eine Menge von Zusatzobjekten erweiterten Medienobjektgraph erlernbar sind
Eingabe:	Abhängigkeitsgraph der Medienobjektstruktur G^{b*}
	Der Blockbeschreibung entsprechende Zusatzobjekte M^a (siehe FilterBlock)
Ausgabe:	Normalisierter Abhängigkeitsgraph G^{a*} aus Basis- und Zusatzobjekten
Besonderheiten:	-

```
G^addon ← G(M^a, I^b)                    // Abhängigkeitsgraph der Zusatzobjekte erstellen

G^a ← ( M(G^b*) ∪ M(G^addon), I^b )      // Erstellen eines aggregierten Abhängigkeitsgraphen aus
                                         // Basis- und Zusatzobjekten

G^a' ← ResolveCycles( G^a )              // Normalisieren des aggregierten Abhängigkeitsgraphen
MinimumDepth( G^a' )                     // (siehe Algorithmus NormalizeDependencyGraph)

G^a* ← RemoveUnteachableNodes( G^a' )
PredecessorSets( G^a* )

Return G^a*
```

Algorithmus BindAddOns (M^t, M^a, G^{a+}) Kapitel 7.5.2

Aufgabe:	Binden der Zusatzobjekte an sämtliche inhaltlich passenden Medienobjekte
Eingabe:	Medienobjekte der Medienobjektstruktur M^t
	verwendbare Zusatzobjekte M^a
	aggregierter Abhängigkeitsgraph G^{a*}
Ausgabe:	Vektor der zu den Basisobjekten passenden Zusatzobjekten B
	Vektor der zu den Zusatzobjekten passenden Basisobjekte A
	Vektor der spezielles Vorwissen verlangenden Zusatzobjekte F
Besonderheiten:	Die Ergebnisvektoren sind initial mit leeren Mengen belegt

```
for all z ∈ M^a do        // Berechnen der passenden Paare aus Basis- und Zusatzobjekten
    M^z ← μ_pre(z,G^a*)
    for all s ∈ μ_kwprv(z,I(G^a*)) do        // Alle Basisobjekte, die Nachbarn des aktuell
        M^z ← M^z ∪ (σ_moprv(s,I(G^a*)) \ M^a)  // betrachteten Zusatzobjekts z sind, werden
    end for                                  // zusammengefaßt

    for all m ∈ M^z do
        // Alle Vorgänger eines Zusatzobjekts müssen in der Vorgängermenge des Basisobjekts enthalten
        // sein, damit die beiden Objekte aneinander gebunden werden können.
        if (μ_pre(z,G^a*) \ M^a) ⊆ μ_ps(m,G^a*) then
            f ← True
            for all o ∈ μ_pre(z,G^a*) ∩ M^a do
                if not μ_pre(o,G^a*) ⊆ μ_ps(m,G^a*) then
                    f ← False
                    exit for
                end if
            end for

            if f = True then                 // Wenn m und z zueinander "passen" werden sie über
                A_z ← A_z ∪ {m}              // die Vektoren A und B aneinander gebunden.
                B_m ← B_m ∪ {z}
                for all o ∈ μ_pre(z,G^a*) ∩ M^a do
                    F_o ← F_o ∪ {z}          // Vorwissen vermittelnde Zusatzobjekte
                end for
            end if
        end if
    end for
end for

return (B, A, F)
```

Algorithmus OptimizeAddOns (G^{a+}, B, A, F, T^{b+}, b) Kapitel 7.5.3

Aufgabe:	Überführung eines Abhängigkeitsgraphen in eine Medienobjektstruktur
Eingabe:	Aggregierter Abhängigkeitsgraph aus Basis- und Zusatzobjkten G^{a*}
	Vektor der zu den Basisobjekten passenden Zusatzobjekte B
	Vektor der zu den Zusatzobjekten passenden Basisobjekte A
	Vektor der spezielles Vorwissen verlangenden Zusatzobjekte F
	Medienobjektstruktur T^{b+}
	Blockbeschreibung b
Ausgabe:	Um Bindungen erweiterte Medienobjektstruktur T^{b*}
Besonderheiten:	-

```
for z = 1 to |F| do: F*_z ← ∅        // zur Vermittlung von Vorwissen benötigte Zusatzobjekte
nodes^t ← ∅                          // Knoten der endgültigen Medienobjektstruktur
links^t ← t_links(T^b+)              // Links der endgültigen Medienobjektstruktur
edges^t ← t_edges(T^b+)              // Kanten der endgültigen Medienobjektstruktur

for all a ∈ b_additional(b) do
    Pairs ← ∅          // zueinander passenden Paare aus Basis- und Zusatzobjekt (incl. Gewichtung)
    for z in A do: A*_z ← ∅: end for          // als passend ausgewählten Basisobjekte

    for all m ∈ B do                 // Alle zueinander passenden Paare der aktuell betrachteten
       B*_m ← ∅                      // Klasse von Zusatzobjekten werden gewichtet
       for all z ∈ B_m do
          if genre(z) = genre(a) then
             e ← w_edge(m,z,G^a*)
             Pairs  ← Pairs ∪ {(m, z, e)}
          end if
       end for
    end for

    SPair ← Sort(Pairs)              // Sortieren der Paare nach absteigender Bewertung

    for i = 1 to |SPair| do
       (m, z, e) ← SPair_(i)

       // Sicherstellen, daß weder zu viele Zusatzobjekte eines Genres noch zu viele Basisobjekte für ein
       // Zusatzobjekt ausgewählt werden
       if (|B*_m|< z_freq_genre(a)) and |A*_z|< z_freq_obj(a) then
          Mz ← μ_pre(z,G^a*) ∩ M^a  // Berechnen der Vorwissen vermittelnden Zusatzobjekte
          B*_m ← B*_m ∪ {(z, a, Mz)}          // Gegenseitige Referenzierung von
          A*_z ← A*_z ∪ {m}                   // Zusatz- und Basisobjekt
          for all x ∈ μ_pre(z,G^a*) ∩ M^a do  // Zusätzlich benötigte Zusatzobjekte
             F*_x ← F*_x ∪ {z}
          end for
       end if
    end for
end for
```

```
for z = 1 to |F*| do                          // Erstellung von Medienobjektknoten aus
    if F*_z ≠ ∅ then                          // ungebundenen Zusatzobjekten
        links^z ← ∅
        for all i ∈ μ_req(m,I(G^a*)) do
            j ← BestPrv( i, G^a* )
            links^z ← links^z ∪ {(i,j)}
        end for
        t ← (z, ∅, false)
        links^t ← links^t ∪ links^z
        nodes^t ← nodes^t ∪ {t}
    end if
end for

for all t ∈ t_nodes(T^b+) do                  // Integration der gebundenen Zusatzobjekte
    addon ← ∅                                 // an t gebundene Zusatzobjekte
    for all (z, a, rz) ∈ B*_mo(t) do
        links^z ← ∅


        for all i ∈ μ_req(z,I(G^a*)) do       // gebundenes Zusatzobjekt in die Medienobjekt-
            j ← BestPrv( i, G^a* )            // struktur integrieren
            links^z ← links^z ∪ {(i,j)}
        end for


        tz ← (z, ∅, true)                     // Zusatzobjekt in einen Medienobjektknoten
        links^t ← links^t ∪ links^z           // konvertieren
        addon ← addon ∪ { (tz, a, rz) }
    end for


    t* ← (mo(t),addon,true)                   // um die ausgewählten Zusatzobjekte bereicherten
    nodes^t ← nodes^t ∪ {t*}                  // Knoten zur Medienobjektstruktur hinzufügen
end for

return (nodes^t, edges^t, links^t)
```

Algorithmus CreateFrames (T^{b*}, b) Kapitel 7.6

Aufgabe:	Überführen der endgültigen Medienobjektstruktur in eine Seitenstruktur
Eingabe:	Medienobjektstruktur T^{b*}
	Blockbeschreibung b
Ausgabe:	Aus der Medienobjektstruktur T^{b*} abgeleitete Seitenstruktur F^{b*}
	Menge von für die Kapitelsequenzierung relevanten Ordering Informationen O
Besonderheiten:	-

```
F^b+ ← NodesToFrames( T^b* )      // 1:1 Abbildung der Medienobjektstruktur auf eine initiale
                                  // Seitenstruktur

F^b+ ← ClusterFrames( F^b+ )      //Zusammenfassen und Strukturieren

O ← ∅                             // Schrittweiser Übergang zur Baumstruktur
while ∃ f ∈ F^b+: |φ_pre(f,F^b+)| > 1 do
    (F^b+, O) ← MakeTree( F^b+, O ) // Nach jeder Kantenentfernung wird überprüft, ob dadurch
    F^b+ ← ClusterFrames( F^b+ )    // weitere Knoten zusammengefaßt werden können.
end while

return (F^b+, O)
```

Algorithmus NodesToFrames (T^{b*}) Kapitel 7.6

Aufgabe:	Erstellen einer Seitenstruktur aus einer Medienobjektstruktur durch 1:1 Abbildung von Medienobjekten auf Seiten
Eingabe:	Medienobjektstruktur T^{b*} eines Blocks b
Ausgabe:	Initiale Seitenstruktur F^{b+} dieses Blocks
Besonderheiten:	-

```
nodes^f ← ∅                          // Knoten der initialen Seitenstruktur
edges^f ← ∅                          // Kanten der initialen Seitenstruktur
links^f ← t_links(T^b*)              // Hyperlinks der initialen Seitenstruktur

for all t ∈ t_nodes( T^b* ) do
    h ← ∅                            // Hyperlinks der betrachteten Seite
    for all (i,j) ∈ links^f with mo(i) = tm_mo(t) do
        h ← h ∪ {(i,j)}              // Binden der Hyperlinks aus der Medienobjektstruktur an
        links^f ← links^f \ {(i,j)}  // einzelne Seiten der Seitenstruktur
    end for
    nodes^f ← nodes^f ∪ {({t},h,Tree)}          // Seite mit einem Medienobjekt anlegen
end for

for all (t, u) ∈ t_edges( T^b* ) do
    f ← Seite in nodes^f mit ft_nodes(f) = {t} // Kanten der Medienobjektstruktur in
    g ← Seite in nodes^f mit ft_nodes(g) = {u} //Kanten der Seitenstruktur umwandeln
    edges^f ← edges^f ∪ {(f,g, w_edge(tm_mo(t),tm_mo(u)))}
end for

return (nodes^f, edges^f, links^f)          //Initiale Seitenstruktur erstellen
```

Algorithmus ClusterFrames (F^{b+}) Kapitel 7.6.2

Aufgabe:	Zusammenfassung von sequentiell strukturierten Seiten und Auslagerung von schwach angebundenen Wurzelknoten aus einer Seitenstruktur
Eingabe:	Seitenstruktur F^{b+}
Ausgabe:	Optimierte Seitenstruktur F^{b+}
Besonderheiten:	-

```
nodes^f ← f_nodes(F^b+)          // verbleibende Knoten
edges^f ← f_edges(F^b+)          // verbleibende Kanten
changed ← false

do
  for all f ∈ nodes^f do         // Identifizieren von potentiell zu isolierenden Wurzelseiten
      if |φ_succ(f,F^b+)| > 0 and |φ_pre(f,F^b+)| = 0 and
              |ft_nodes(f)| = 1 and μ_standalone(ft_nodes(f)_(1)) = True then
          for all g ∈ φ_succ(f,F^b+) do
              // Für jeden Nachfolger einer Wurzelseite muß überprüft werden, ob andere Seiten noch
              // schwächer angebunden sind als die betrachtete Wurzelseite.
              min_weight ← 1
              this_weight ← 0

              for all (x,y,e) ∈ φ_in(g,F^b+) do
                  if e < min_weight then min_weight ← e
                  if x = f then this_weight ← e // "Merken" der Bindungsstärke
              end for

              if min_weight = this_weight then
                  // Wenn der Nachfolger keine schwächere Verbindung hat als zu der betrachteten
                  // Wurzelseite, wird diese Verbindung durch einen Hyperlink ersetzt.
                  edges^f ← edges^f \ (f,g,this_weight)
                  ft_tree(f) ← Linked
                  changed ← True
              end if

          end for
      end if
  end for

 // vor der Sequenzierung müssen zunächst die komplexen Szenarien betrachtet werden
  if changed then return (nodes^f, edges^f, f_links(F^b+))

  for all f ∈ nodes^f do
      // Überprüfen der für die Zusammenfassung benachbarter Seiten zu erfüllenden Kriterien
      if |φ_succ(f,F^b+)| ≤ 1 and |φ_pre(f,F^b+)| ≤ 1 then
          (x,g,e) ← f' ∈ φ_out(f,F^b+)
          if |φ_succ(g,F^b+)| ≤ 1 and |φ_pre(g,F^b+)| ≤ 1 then
              if e > 0.5 and GenreFits(f,g) and LayoutFits(f,g) then
                  nodes^f ← nodes^f \ {f}     // Ersetzen der einzelnen Seiten durch die
                  nodes^f ← nodes^f \ {g}     // zusammengefaßte Seite
                  fg_n ← ft_nodes(f) ∪ ft_nodes(g)
                  fg_l ← ft_links(f) ∪ ft_links(g)
```

```
                nodes^f ← nodes^f ∪ { (fg_n, fg_l) }

                // Anpassen der Kanten der Seitenstruktur
                for all (x,y,z) ∈ ϕ_in(f,F^b+) do
                    edges^f ← edges^f \ {(x,y,z)}
                    edges^f ← edges^f ∪ {(x,fg,z)}
                end for

                for all (x,y,z) ∈ ϕ_out(g,F^b+) do
                    edges^f ← edges^f \ {(x,y,z)}
                    edges^f ← edges^f ∪ {(fg,y,z)}
                end for

                changed ← True
            end if
        end if
    end if
  end for
while( changed )

return (nodes^f, edges^f, f_links(F^b+))
```

Algorithmus MakeTree (F^{b+}, O^b) — Kapitel 7.6.3

Aufgabe:	Umwandlung einer beliebig strukturierten Seitenstruktur in eine hierarchisch strukturierte Seitenstruktur
Eingabe:	Seitenstruktur F^{b+}
	Zu der Seitenstruktur gehörige Ordering Informationen O^b
Ausgabe:	Hierarchisch strukturierte Seitenstruktur F^{b+}
	Erweiterte Ordering Informationen O^b
Besonderheiten:	-

```
nodes^f ← f_nodes(F^b+)
edges^f ← f_edges(F^b+)
eval_worst ← MAX_EVAL

// wenn keine Seite mit mehr als einem Vorgänger existiert, ist keine weitere Strukturierung notwendig
pre2 ← ∅        // Menge aller Seiten mit mehr als einem Vorgänger
for all f ∈ nodes^f do
    if |φ_pre(f,F^b+)| > 1 then  pre2 ← pre2 ∪ {f}
end for
if pre2 = ∅ then return (F^b+, ∅)

// Szenario 3 - Abgesicherte Hyperlinks
for all f ∈ pre2 do
     // paarweise Analyse aller Vorgänger
    for all (g,f,e) ∈ φ_in(f,F^b+) do
        for all h ∈ φ_pre(f,F^b+) \ {g} do
            // wenn die Kante g→f durch h→f abgesichert ist, wird sie aus der Seitenstruktur entfernt
            if φ_ps(g,F^b+) ⊆ φ_ps(h,F^b+) then
                edges^f ← edges^f \ {(g,f,e)}
                F^b* ← (nodes^f, edges^f, f_links(F^b))
                O^b* ← O^b ∪ {(g,f,True)}
                return (F^b+, O^b+)
            end if
        end for
    end for
end for

// Szenario 4 - Nicht abgesicherte Hyperlinks
for all f ∈ pre2 do
    fsucc ← ∅ // von einem Nachfolger von f referenzierte Seiten

    for all g ∈ φ_ss(g,F^b+) do       // Bestimmen der von Nachfolgern referenzierten Seiten
        for all t ∈ ft_nodes(g) do
            for all (i,j) ∈ t_links(t) do
                h ← μ_frame(mo(i))    // μ_frame(m) liefert alle Seiten, die m enthaltent.
                fsucc ← fsucc ∪ {h}
            end for
        end for
    end for
```

```
    // Suche nach für f interessanten Ordering Informationen
    conflict ← ∅ // zu beachtende Ordering Informationen
    for all (f1, f2, a ) ∈ O^b do
        // Eine Ordering Information ist für die Bewertung eines Knotens relevant, wenn einer der verbundenen
        // Knoten in der Vorgänger- oder Nachfolgermenge des Knotens enthalten ist.
        if f1 ∈ (φ_ps(f,F^b+) ∪ φ_ss(f,F^b+)) then
            conflict ← conflict ∪ {f2}
        end if
        if f2 ∈ (φ_ps(f,F^b+) ∪ φ_ss(f,F^b+)) then
            conflict ← conflict ∪ {f1}
        end if
    end for

    for all (g,f,e) ∈ φ_in(f,F^b+) do
        // Die Bewertung einer Kante setzt sich zusammen aus:
        //   1. Kantengewichtung
        //   2. Der Kante "folgende" Hyperlinks
        //   3. Die Kante "schneidende" Ordering Informationen
        eval ← e + (α^succ * |fsucc ∩ φ_ps(g,F^b+)|) -
        (α^conflict *|conflict ∩ (φ_ps(g,F^b+) ∪ φ_ss(g,F^b+))|)
        if eval ≤ eval_worst then
            eval_worst ← eval
            link_worst ← (g,f,e)
        end if
    end for

    (g,f,e) ← link_worst      // Entfernen der am schlechtesten bewerteten Kante
    edges^f ← edges^f \ {(g,f,e)}
    F^b* ← (nodes^f, edges^f, f_links(F^b+))
    O^b* ← O^b ∪ {(g,f,True)}
    return (F^b*, O^b*)
end for

return (F^b+, O^b)
```

Algorithmus CreateChapters (F^{b+}, O^{b+}) Kapitel 7.7

Aufgabe:	Überführung einer Seitenstruktur in eine geordnete logische Struktur (Kapitelstruktur)
Eingabe:	Hierarchische Seitenstruktur F^{b+}
	Bei der Kapitelstrukturierung zu berücksichtigende Ordering Informationen O^{b+}
Ausgabe:	Kapitelstruktur C^{b+}
	Menge der nur über Hyperlinks an die Kapitelstruktur angebundenen Seiten F^{free}
Besonderheiten:	-

```
C^b+ ← SetupChapters( F^b+ )      //Zusammenfassen von Seiten zu Kapiteln

C^b+ ← SortChapters( C^b+, O^b+ ) // Ordnen der Unterkapitel

C^b* ← IntegrateAddOns( C^b+ )    // Einbetten der gebundenen Zusatzobjekte

                                  // Ermitteln der nicht eingebundenen Seiten:
F^free ← ∅                        // Alle Seiten, die nicht in das hierarchische Gerüst der Kapitelstruktur
for all f ∈ ft_nodes(F^b*) do     // angebunden sind, müssen für die Erzeugung der HTML Seiten
    if ft_tree(f) = Linked then   // gesondert vermerkt werden, da sie nicht durch Traversieren der
        F^free ← F^free ∪ {f}     // Kapitelstruktur erreichbar sind.
    end if
end for

return (C^b*, F^free)
```

Algorithmus SetupChapters (F^{b+}) Kapitel 7.7.1

Aufgabe:	Umwandlung einer Seitenstruktur in eine Kapitelstruktur durch Konvertierung von Seiten-Clustern in Kapitel
Eingabe:	Seitenstruktur F^{b+}
Ausgabe:	Kapitelstruktur C^{b+}
Besonderheiten:	Die Ordnung von Geschwisterknoten der erzeugten Kapitelstruktur ist willkürlich, da bei der Umwandlung keine Ordering Informationen beachtet werden.

```
C^b+ ← ∅

// Ermitteln der Wurzelseiten, die die Startseiten der Top-Level Kapitel bilden
F^root ← ∅
for all f ∈ ft_nodes(F^b*) do
    if φ_pre(f,F^b*) = ∅ and ft_tree(f) = "Tree" then
        F^root ← F^root ∪ {f}
    end if
end for

// Ausgehend von den Wurzelseiten wird durch Traversieren der Nachfolgerseiten die initiale Kapitelstruktur aufgebaut
i ← 1
for all f ∈ F^root do
    c ← BuildChapter( f, F^b*, nil, nil, i )
    C^b+ ← C^b+ ∪ {c}
    i ← i + 1
end for

return C^b+
```

Algorithmus BuildChapter (f, F^{b+},p, flast, ii) Kapitel 7.7.1

Aufgabe:	Rekursive Erstellung einer Kapitelstruktur aus einer hierarchischen Seitenstruktur
Eingabe:	Top-Level Seite f des zu erstellenden Kapitels
	Seitenstruktur F^{b+}
	dem zu erzeugenden Kapitel übergeordnetes Kapitel p
	letzte Seite flast des übergeordneten Kapitels
	initiale Ordnungsnummer ii des Kapitels
Ausgabe:	Kapitel c
Besonderheiten:	-

```
c ← ( ∅, p, ∅, ii, f )              // Initialisierung des zu erstellenden Kapitels
pre ← flast                         // das übergeordnete Kapitel stellt den Vorgänger
succ ← nil

// Solange Seiten mit nur einem Nachfolger folgen, werden diese zum aktuellen Kapitel hinzugefügt.
while |φsucc(f,F^b*)| = 1 do
    succ ← frame ∈ φsucc(f,F^b*)    // Verkettung der Seiten
    c_frames(c) ← c_frames(c) ∪ {(f, pre, succ)}
    pre ← f
    f ← succ
    succ ← nil
end while

c_frames(c) ← c_frames(c) ∪ {(f, pre, succ)}

// Falls der Algorithmus beim Verfolgen der Seitenstruktur auf eine Gabelung trifft, wird "BuildChapter" rekursiv
// für alle von dieser Gabelung ausgehenden Pfade aufgerufen.
i ← 1
for all g ∈ φsucc(f,F^b*) do
    cg ← BuildChapter( g, c, f, i )
    c_children(c) ← c_children(c) ∪ {cg}
    i ← i + 1
end for
return c
```

Algorithmus SortChapters (C^{b+}, O^{b+}) Kapitel 7.7.2

Aufgabe:	Sortierung der Kapitelstruktur anhand der beigefügten Ordering Informationen
Eingabe:	Initiale Kapitelstrukur C^{b+}
	Ordering Informationen O^{b+}
Ausgabe:	Geordnete Kapitelstruktur C^{b+}
Besonderheiten:	-

```
Reorder( C^b+, O^b* )              // Ordnen der Kapitel oberster Hierarchiestufe

for all c ∈ C^b+ do
    TraverseAndSort( c, O^b* ) // Traversieren und Ordnen der Unterkapitel
end for

return C^b+
```

Algorithmus TraverseAndSort (c, O^{b+}) Kapitel 7.7.2

Aufgabe:	Traversieren und Ordnen der Unterkapitel eines gegebenen Kapitels
Eingabe:	Kapitel c, dessen Unterkapitel geordnet werden sollen
	Die Ordnung beschreibende Ordering Informationen O^{b+}
Ausgabe:	-
Besonderheiten:	-

```
Reorder( c_children(c), O^b*)       // Ordnen der Unterkapitel

for all cc ∈ c_children(c) do
    TraverseAndSort( cc, O^b* )     // Traversieren der Unterkapitel
end for
```

Algorithmus Reorder (sc, O^{b+}) Kapitel 7.7.2

Aufgabe:	Ordnen einer Menge von Geschwisterknoten anhand vorgegebener Ordering Informationen
Eingabe:	Menge von Geschwisterknoten sc
	Ordering Informationen O^{b+}
Ausgabe:	-
Besonderheiten:	-

```
ord ← ∅      // Menge der für die Ordnung von sc relevanten Ordering Informationen

for all (f1, f2, s) ∈ O^b* do
    c1 ← nil              // Eine Ordering Information ist für die Ordnung von sc relevant, wenn sie
    c2 ← nil              // Abhängigkeiten zwischen  Nachfolgerseiten von verschiedenen
    for all c ∈ sc do     // in sc enthaltenen Kapiteln beschreibt
        if f1 ∈ χframes(c) then c1 ← c
        if f2 ∈ χframes(c) then c2 ← c
    end for
    if (c1 ≠ nil) and (c2 ≠ nil) then
        ord ← ord ∪ { (c2, c1, s) }
    end if
end for

// Umsortierung durch Vertauschen
for all s in [True, False] do
    for all (c1, c2, f) ∈ ord with f = s do
        if c_order(c1) > c_order(c2) then
            for all c ∈ sc do
                if c_order(c) > c_order(c2) and
                              c_order(c) < c_order(c1) then
                    c_order(c) = c_order(c) + 1
                end if
            end for
            c_order(c1) = c_order(c2)
            c_order(c2) = c_order(c2) + 1
        end if
    end for
end for
```