UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Algoritm polinomial bazat pe grafuri pentru compoziția serviciilor web semantice

propusă de

Ștefania-Liana Țucăr

Sesiunea: *Iulie, 2017*

Coordonator științific
Prof. Dr. Cornelius Croitoru
Drd. Paul Diac

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

Algoritm polinomial bazat pe grafuri pentru compoziția serviciilor web semantice

Ștefania-Liana Țucăr

Sesiunea: *Iulie, 2017*

Coordonator științific

Prof. Dr. Cornelius Croitoru
Drd. Paul Diac

DECLARAŢIE PRIVIND ORIGINALITATE ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licenţă cu titlul „Algoritm polinomial bazat pe grafuri pentru compoziția serviciilor web semantice" este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
- codul sursă, imaginile etc. preluate din proiecte open-source sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
- rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iaşi,
30 iunie 2017

Absolvent,
Ștefania-Liana Țucăr

_____

(semnătura în original)

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul „Algoritm polinomial bazat pe grafuri pentru compoziția serviciilor web semantice", codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza" din Iaşi să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi,
30 iunie 2017

Absolvent,
Ștefania-Liana Țucăr

(semnătura în original)

Cuprins

1 Introducere

Arhitecturile orientate pe servicii web si tehnologii bazate pe servicii web sunt foarte populare in acest moment. Avand la dispozitie un numar de servicii web, fiecare cu functionalitatea sa, putem incerca sa combinam mai multe dintre ele pentru a obtine o noua functionalitate. Daca apare o solicitare formata din doua multimi de informatii: input - informatiile oferite la de cel ce face soliticate si output - informatiile ce se cer a se afla, exista posibilitatea ca aceasta solicitare sa nu poate fi satisfacuta de un singur serviciu din repository. Aici apare necesitatea combinarii functionalitatilor serviciilor care ne sunt puse la dispozitie, pentru a produce o compozitie de servicii web care sa satisfaca solicitarea.

Un serviciu este descris de mai multi parametri, printre care si parametrii lui de input si de output. Acestia doi din urma sunt cei care ne intereseaza in problema simplificata a compozitiei serviciilor web. Un serviciu poate fi apelat doar daca ii furnizam toate informatiile necesare din multimea de inputuri si in urma apelarii ne va returna toate informatiile din multimea de outputuri.

Cum numarul de servicii web dintr-un repository poate fi foarte mare, apare necesitatea unor algoritmi rapizi care sa obtina compozitii valide si de preferat cu anumite proprietati bune.

Periodic are loc cate o competitie in acest domeniu (WSC2005, WSC2008, WSC2009). In competitia din anul 2005 o compozitie de servicii web este considerata a fi o lista de servicii web ce vor fi apelate secvential. Pentru fiecare serviciu din aceasta lista vor putea fi folosite la input toate outputurile serviciilor dinaintea sa impreuna cu toate instantele oferite de cel ce face cererea. Pentru a folosi ca input pentru un serviciu un output de la alt serviciu, aceastea trebuie sa coincida. In aceasta descriere a compozitiei, este de dorit ca solutia sa aiba un numar cat mai mic de servicii. Problema gasirii unei compozitii cu numar minim de servicii este $\mathbb{NP}$ dificila, demonstratia gasindu-se in capitolul 3. In [1] este prezentat un algortim de complexitate timp polinomiala, bazat pe euristici si cu un sistem de scor dinamic, care nu garanteaza obtinerea solutiei optime, dar care obtine rezultate foarte bune pe teste. Rezultatele acestui tip de abordare pentru problema din competitia din 2005, ne incurajeaza sa optam pentru un algoritm de complexitate timp polinomiala care sa incerce sa obtina compozitii aproape la fel de bune, dar intr-un timp mult mai scurt decat algoritmii care garanteaza compozitia optima pentru varianta problemei din competitia din 2008.

In competitia din 2008 definitia compozitiei a evoluat in doua directii. Prima diferenta este ca o compozitie nu mai este doar o lista de servicii ce pot fi apelate secvential. Apare si conceptul de apelare in paralel a serviciilor, compozitia fiind descrisa ca un proces workflow. A doua diferenta intervine in folosirea argumentelor ce pot fi date ca input unui serviciu. Inputurile si outputurile sunt descrise prin niste multimi de instante, iar fiecare instanta se incadreaza intr-un concept, conceptele formand la randul lor o ierarhie. Ierarhia de concepte impreuna cu instantele ce fac parte din fiecare concept formeaza o

taxonomie. O instanta b poate fi data ca argument input la un serviciu ce asteapta o instanta a, doar daca b este egal cu a sau daca b este o instanta mai particulara decat a. Astfel, un output obtinut de la un serviciu poate fi folosit ca input la alt serviciu daca cel dintai este macar la fel de particular ca cel din urma. In aceasta definitie a compozitiei, o solutie este de dorit sa aiba atat un numar mic de servicii, cat si un numar mic de pasi de executie (execution path).

In capitolul 2 va fi prezentata mai amanuntit problema, impreuna cu diferitele ei variante. O analiza a complexitatii problemei va fi facuta in capitolul 3, in care vom arata ca anumite variante ale acestei probleme sunt $\mathbb{NP}$ dificile. Algoritmul, cu explicatii si pseudocod este descris in capitolul 4, impreuna cu o analiza a limitei sale de eroare. In capitolul 5 sunt prezentate rezultatele algoritmului pe un set de teste publice, folosite atat in competitie cat si de [3] pentru a-si testa algoritmul. Concluziile si directiile de viitor sunt prezentate in capitolul 6.

1.1 Contribuții

Gasirea compozitiei cu numar minim de servicii in problema descrisa in competitia din 2008 este, ca si in problema anterioara, $\mathbb{NP}$ dificila, insa pentru gasirea numarului minim de pasi de executie (execution path) exista algoritm determinist polinomial.
Am implementat un algoritm pentru problema din competitia din 2008, cu o abordare determinista bazata pe grafuri, obtinand o solutie care garanteaza un numar minim de pasi de executie (exec path). Deoarece problema este $\mathbb{NP}$ dificila si am dorit ca algoritmul determinist sa fie rapid, algoritmul nu poate garanta ca obtine o compozitie cu numar minim de servicii, insa acesta a fost testat pe un repository de teste publice, unele dintre ele fiind folosite chiar in competitia oficiala si rezultatele sunt foarte bune. Algoritmul ar fi castigat competitia din 2008 si obtine solutia optima pe 7 din 8 teste, intr-un timp foarte scurt, fiind pe unele teste chiar de peste 50 de ori mai rapid decat un algoritm care garanteaza solutia optima.
In capitolul 3 se gaseste demonstratia ca problema in cauza este $\mathbb{NP}$ dificila.
Pentru a verifica compozitiile returnate de algoritm, un validator de solutii a fost implementat.
Pentru o mai buna analiza si vizualizare a solutiei oferite de algoritm, o reprezentare grafica sub forma unui graf este afisata la fiecare rulare a algoritmului.

2 Problema

Servicii Web In aceasta abordare vom considera ca un serviciu web este definit de doua multimi de parametri:

- S_{in} - multimea de inputuri.
- S_{out} - multimea de outputuri.

De asemenea consideram ca apelarea unui serviciu web nu modifica starea sistemului. Un serviciu web S poate fi apelat daca am obtinut anterior apelarii toti parametrii din multimea S_{in}, iar in urma apelarii acesta ne va intoarce toati parametrii din multimea S_{out}.

Un repository este o multime $\{S_1, S_2, .., S_n\}$ de servicii web.

Solicitare O solicitare este formata din doua multimi de parametri: C_{in} (oferite de cel ce instantiaza solicitarea) si C_{out}(parametri pe care cel ce instantiaza solicitarea ii doreste).

Compozitie Pentru a putea satisface o solicitare, poate interveni necesitatea folosirii mai multor servicii web dintr-un repository. Putem folosi parametrii oferiti ca output de un serviciu S_1 pentru a apela un alt serviciu S_2. De asemenea, pot fi folositi parametrii oferiti ca output de mai multe servicii pentru a apela un alt serviciu.
Presupunem ca detinem o multime de parametrii $provided = \{I_1, I_2, ..., I_k\}$.

Definitie 2.1. O compozitie secventiala de servicii web care satisface o solicitate $C = (C_{in}, C_{out})$ este o insiruire (lista) de servicii web care vor fi apelate in ordinea aparitiei in lista. Ca un serviciu sa poata fi apelat, trebuie sa detinem toti parametrii sai input (ori sa fie dintre parametri oferiti, adica din multimea C_{in}, ori sa fie obtinut ca output al unui alt serviciu ce se afla inaintea sa in lista).
$\mathcal{C} = (S_{i_1}, S_{i_2}, ..., S_{i_k})$ cu $(S_{i_j})_{in} \subseteq (\cup_{q \in \{1,...,j-1\}} (S_{i_q})_{out}) \cup C_{in}$ si $C_{out} \subseteq (\cup_{q \in \{1,...,k\}} (S_{i_q})_{out})$

Compozitia descrisa mai sus este cea folosita in competitia din 2005.

Definitia problemelor

Definitie 2.2. Problema compozitiei de servicii web (WSC).
Input: un triplet $(C_{in}, C_{out}, repository)$, conform definitiilor anterioare.
Output: compozitie secventiala care sa satisfaca solicitarea $C = (C_{in}, C_{out})$.

O solutie este caracterizata de numarul de servicii web. Este de dorit ca solutia sa aiba un numar cat mai mic de servicii, acest lucru fiind incurajat si in competitia din 2005.

Definitie 2.3. Problema compozitiei minime de servicii web (MIN-WSC).
Input: un triplet $(C_{in}, C_{out}, repository)$, conform definitiilor anterioare.
Output: compozitie secventiala cu numar minim de servicii care sa satisfaca solicitarea $C = (C_{in}, C_{out})$.

Evolutia problemei Standardele de rand XML pentru interoperarea serviciilor web specifica doar interoperabilitate sintactica, fara intelesul semnatic al mesajelor. De exemplu WSDL poate preciza operatiile ce se pot face folosind un serviciu web si structura datelor primite si trimise, dar nu poate mentiona semnificatiile sau constrangerile semantice ale datelor. Din acest considerent programatorii trebuie sa stabileasca anumite conventii pentru interactiunea serviciilor web, ceea ce face compozitia automata de servicii web dificila.
Serviciile web semantice sunt construite conform unor standarde universale pentru schimbul de date semantice, ceea ce face mai usor combinarea datelor de la diferite surse si servicii fara a se pierde seminificatiile lor.
Astfel, competitia din 2008 a introdus servicii web semantice. Pentru a putea defini serviciile web semantice si compozita acestora introducem notiunile noi: taxonomie, instanta, concept si anumite relatii intre acestea.

Taxonomie Pentru problema compozitiei semantice de servicii web, instantele si conceptele sunt aranjate intr-o taxonomie. Fiecare concept poate avea subconcepte mai particulare si instante care fac parte din conceptul respectiv. Toate conceptele sunt subconcepte (directe sau indirecte) ale unui concept cel mai general. Fiecare concept apare exact o singura data in taxonomie.
Putem reprezenta ierarhia de concepte sub forma unui arbore in care fiecarui concept ii este asociat un nod, iar relatia parinte-copil din arbore reprezinta relatia de abstractizare directa dintre doua concepte.
Fie urmatoarele multimi:

- $Concepte$ = multimea tuturor conceptelor din taxonomie
- $Instante$ = multimea tuturor instantelor din taxonomie

Definitie 2.4. Fie functia $fInstante : Concepte \to \mathcal{P}(Instante)$, unde $fInstante(concept) =$ multimea tuturor instantelor ce fac parte din acel concept.

Multimile formate din rezultatul aplicarii functiei $finstante$ pe toate conceptele existente in taxonomie, formeaza o partitie a multimii tuturor instantelor din taxonomie.

Definitie 2.5. Fie functia $fSubconcepte : Concepte \to \mathcal{P}(Concepte)$

$$fSubconcepte(concept) = \begin{cases} \emptyset, daca fSubconcepteDirecte(concept) = \emptyset \\ \cup_{c \in fSubconcepteDirecte(concept)}(fSubconcepte(c) \cup c), altfel \end{cases}$$

Definitie 2.6. Definim o functie $fparinteI : Instante \rightarrow Concepte$, unde
$fparinteI(instanta) = concept \mid instanta \in fInstance(concept)$.
$fparinteI(instanta)$ este conceptul direct din care face parte instanta (adica cel mai specific concept in care se incadreaza respectiva instanta).

Definitie 2.7. Intre instante se defineste urmatoarea functie:
$subsumes : Instances \times Instances \rightarrow \{fals, adevarat\}$
$subsumes(a, b)$ e adevarat $\iff$ instanta b este mai specifica decat instanta a. Asta inseamna ca nodul asociat lui $partinteI(b)$ este un succesor (nu neaparat direct) al nodului asociat lui $parinteI(a)$ in arborele de concepte descris de taxonomie sau instantele sunt egale.
$subsumes(a, b)$ e adevarat $\iff fpartinteI(b) \in fSubconcepte(fParinteI(a))$ sau $a = b$.

Daca $subsumes(a, b)$ este adevarat, atunci b poate fi folosit ca input pentru un serviciu ce necesita instanta a. Asta inseamna ca acel serviciu necesita informatie macar cat este oferita de instanta a.

Definitie 2.8. $subsumesM : \mathcal{P}(Instances) \times \mathcal{P}(Instances) \rightarrow \{0, 1\}$

$$subsumesM(A, B) = \begin{cases} 1, & \text{daca } \forall a \in A,\ \exists b \in B | subsumes(a, b) = 1 \\ 0 & \text{altfel} \end{cases}$$

Adica pentru doua multimi de instante A si B, $subsumesM(A, B)$ este 1 ddaca pentru orice instanta a din multimea A putem gasi macar o instanta in multimea B care este mai specifica decat a. Asta inseamna ca avand la dispozitie instantele din multimea B, un serviciu ce are ca inputuri instantele din multimea A poate fi apelat, folosind multimea B.

Definitie 2.9. Un serviciu web semantic este descris de doua multimi:

- S_{in} multimea de instante input.
- S_{out} multimea de instante output.

Fata de problema anterioara, compozitia evolueaza in sensul ca nu mai este doar o lista de servicii care pot fi executate secvential, ci apare si posibilitatea executarii in paralel a unor servicii sau portiuni din compozitie.

Definitie 2.10. Compozitie care pleaca de la o multime *provided* de instante.

- Daca $S = (S_{in}, S_{out}) \in repository$ si $subsumesM(provided, S_{in})$, atunci $Comp = S$ este compozitie care pleaca de la *provided* si $Comp_{out} = S_{out}$
- Daca $C_1, C_2, ..., C_n$ compozitii: C_i pleaca de la $(\cup_{j \in \{1,...,i-1\}}(C_j)_{out}) \cup provided$, $1 \leq i \leq n$ atunci $secv(C_1, C_2, ..., C_n)$ este compozitie care pleaca de la *provided* si $Comp_{out} = (\cup_{i \in \{1,...,n\}}(C_i)_{out}$

- Daca $C_1, C_2, ..., C_n$ compozitii: C_i pleaca de la *provided*, $1 \leq i \leq n$ atunci $paralel(C_1, C_2, ..., C_n)$ este compozitie care pleaca de la *provided* si $Comp_{out} = (\cup_{i \in \{1,...,n\}}(C_i)_{out}$

Definitie 2.11. Problema compozitiei de servicii web semantice (SEM-WSC).
Input: $(C_{in}, Cout, repository, taxonomy)$, conform definitiilor anterioare.
Output: compozitie valida care sa satisfaca solicitarea $C = (C_{in}, C_{out})$.

O astfel de compozitie de servicii web are doua proprietati calitative:

- Lungimea compozitiei: numarul total de servicii din compozitie
- Eficienta compozitiei: presupunand ca executia fiecarui serviciu ar dura o unitate de timp, care este timpul minim in care se poate executa toate compozitia

Definitie 2.12. Lungimea (nrs) si eficienta ($execpath$) compozitiei:

- Daca $Comp = S$, atunci $nrs(Comp) = 1$ si $execpath(Comp) = 1$
- Daca $Comp = secv(C_1, C_2, ..., C_n)$, atunci $nrs(Comp) = \sum_{i \in \{1,...,n\}} nrs(C_i)$ si $execpath(Comp) = \sum_{i \in \{1,...,n\}} execpath(C_i)$
- Daca $Comp = paralel(C_1, C_2, ..., C_n)$, atunci este $nrs(Comp) = \sum_{i \in \{1,...,n\}} nrs(C_i)$ si $execpath(Comp) = \max_{i \in \{1,...,n\}} execpath(C_i)$

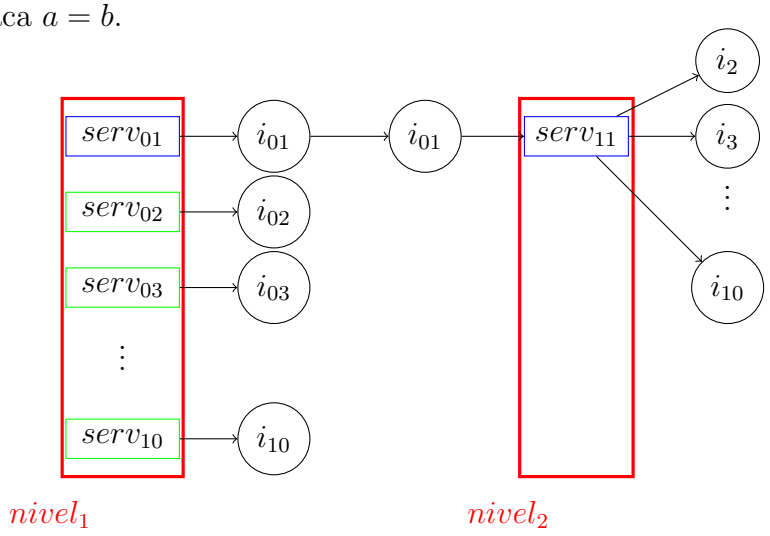
Definitie 2.13. Problema compozitiei de servicii web semantice cu numar minim de servicii(MINS-SEM-WSC).
Input: un triplet $(C_{in}, Cout, repository)$, conform definitiilor anterioare.
Output: compozitie alida cu numar minim de servicii care sa satisfaca solicitarea $C = (C_{in}, C_{out})$.

Definitie 2.14. Problema compozitiei minime de servicii web semantice cu numar minim de pasi de executie(execution path) (MINP-SEM-WSC).
Input: un triplet $(C_{in}, Cout, repository)$, conform definitiilor anterioare.
Output: compozitie valida cu numar minim de pasi de executie (exec path) care sa satisfaca solicitarea $C = (C_{in}, C_{out})$.

Definitie 2.15. Problema compozitiei minime de servicii web cu numar minim de pasi de executie (exec path) si cu numar minim de servicii pentru numarul minim de pasi de executie (MINPS-SEM-WSC).
Input: un triplet $(C_{in}, Cout, repository)$, conform definitiilor anterioare.
Output: compozitie valida cu numar minim de pasi de executie exec path care sa satisfaca solicitarea si cu numar minim de servicii pentru acest numar de pasi. $C = (C_{in}, C_{out})$.

Observatie: problemele MINS-SEM-WSC si MINP-SEM-WSC difera. Fie urmatorul exemplu de instanta a problemei:$(repository, provided, required, taxonomy)$, unde

$repository = \{serv_1 = (\emptyset, \{i_{01}\}), serv_2 = (\emptyset, \{i_{02}\}), ..., serv_{10} = (\emptyset, \{i_{10}\}), serv_{11} = (\{i_{01}\}, \{i_{02}, i_{03}, ..., i_{10}\}\}$; $provided = \emptyset$; $required = \{i_{02}, i_{03}, ..., i_{10}\}$;in taxonomie exista un singur concept si toate instantele fac parte din el. Deci $subsumes(a, b)$ e adevarat ddaca $a = b$.



Cele doua compozitii valide sunt:
$Comp_1 = secv(serv_1, serv_{11})$ si $Comp_2 = paralel(serv_{02}, serv_{03}, ..., serv_{10})$.
Numarul de servicii din prima compozitie este 2, iar eficienta compozitiei este tot 2.
Numarul de servicii din a doua compozitie este 9, iar eficienta compozitiei este 1.

3 Complexitatea problemei

In acest capitol urmeaza demonstratia ca atat problema compozitiei minime de servicii web cat si cea a compozitiei minime de servicii web semantice sunt $\mathbb{NP}$ dificile. Intai vom demonstra ca problema compozitiei minime de servicii web este $\mathbb{NP}$ dificila, reducand SET-COVER la aceasta. Apoi vom arata ca si cea de-a doua problema in discutie (adica varianta semantica) este $\mathbb{NP}$ dificila, aratand ca aceasta este o generalizare a problemei tocmai demonstrate a fi $\mathbb{NP}$ dificila (cea nesemantica).

3.1 SET-COVER

Dandu-se un univers $\mathcal{U}$ si o familie $\mathcal{S}$ de submultimi ale universului $\mathcal{U}$, o acoperire este o subfamilie $\mathcal{C} \subseteq \mathcal{S}$ de multimi ale caror reuniune este $\mathcal{U}$.
In varianta decizionala a problemei SET-COVER, inputul este format din perechea ($\mathcal{U}$,$\mathcal{S}$) si un numar intreg k, intrebarea fiind daca exista o acoperire a universului $\mathcal{U}$ folosind multimi din familia $\mathcal{S}$ de submultimi, astfel incat acoperirea sa aiba dimensiunea (cardinalul) mai mica sau egala cu k.
In varianta de optimizare a problemei acoreririi, inputul este format din perechea ($\mathcal{U}$,$\mathcal{S}$) si se cere o acoperire a universului $\mathcal{U}$ folosind multimi din familia $\mathcal{S}$ de submultimi care sa aiba dimensiunea (cardinalul) minim.
Varianta decizionala a problemei SET-COVER este NP-completa, iar varianta de optimizare este NP-dificila.
Daca fiecarei multimi ii este asociat un cost, problema devine weighted set cover problem.

3.2 Reducere SET-COVER la problema compozitiei minime de servicii web (MIN-WSC)

Vom face o reducere Turing/Cook a problemei SET-COVER in varianta de oprimizare la problema compozitiei de servicii web vom arata ca si aceasta din urma este NP-dificila. Reducerea se va face in complexitate timp polinomiala in raport cu dimensiunea problemei SET-COVER.

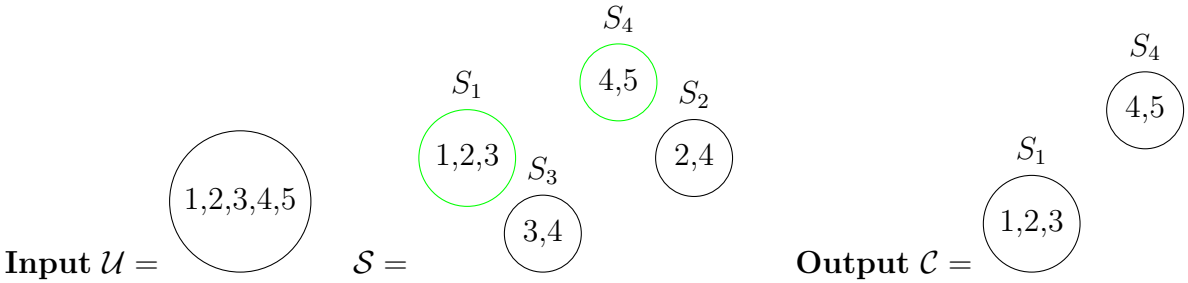
Propozitie 3.1. Problema compozitiei de servicii web este in $\mathbb{P}$ (algoritmul ce va fi descris in capitolul 4 rezolva problema in timp polinomial). $\Rightarrow$ Problema compozitiei de servicii web este in $\mathbb{NP}$.

Vom defini o functie f ce asociaza fiecarei instante a problemei SET-COVER in varianta de optimizare o instanta a problemei compozitiei de servicii web.
Fie $\mathcal{S} = \{S_1, S_2, .., S_n\}$ (cu $\cup_{S_i \in \mathcal{S}} = \mathcal{U}$), $f(\mathcal{U}, \mathcal{S}) = (provided, requested, repository) = (\emptyset, \mathcal{U}, \mathcal{R})$, unde $\mathcal{R} = \{serv_1, serv_2, .., serv_n | serv_i = (\emptyset, S_i), i \in \{1, 2, ..., n\}\}$.

Functia f asociaza unei instante a problemei SET-COVER o instanta in problema compozitiei de servicii web astfel: cel ce face solicitarea nu ofera nici o instanta si doreste sa obtina cate o instanta pentru fiecare element din universul $\mathcal{U}$, iar repository-ul este format doar din tot atatea servicii cate multimi contine familia $\mathcal{S}$; fiecare serviciu cu indicele i nu necesita nici o instanta pentru a fi apelat si ofera la output o multime de instante corespunzatoare pentru multimii $S_i \in \mathcal{S}$.

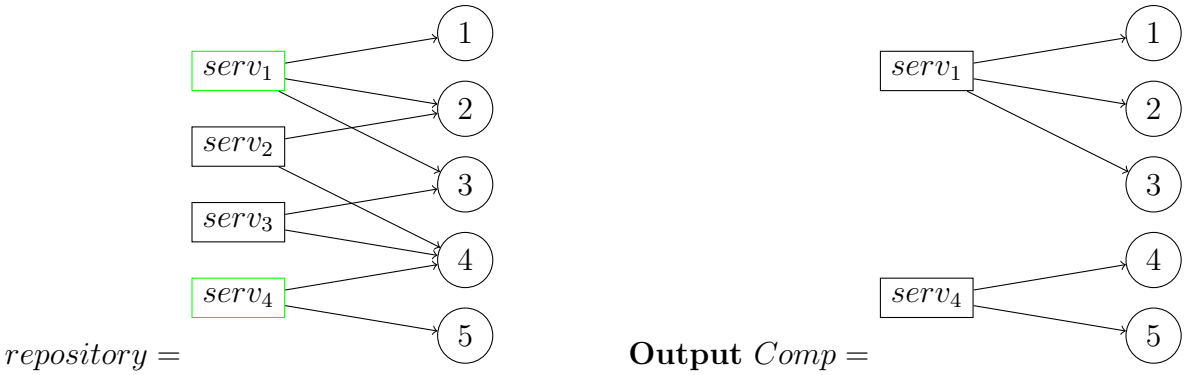
Exemplu 3.1. Fie: $\mathcal{U} = \{1, 2, 3, 4, 5\}$ si $\mathcal{S} = \{\{1, 2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}\}$.
Acoperirea de cardinal minim a universului $\mathcal{U}$ este $\mathcal{C} = \{\{1, 2, 3\}, \{4, 5\}\}$. Rezultatul aplicarii functiei f peste instanta din exemplu a problemei SET-COVER este:
$(\emptyset, \{1, 2, 3, 4, 5\}, \{(\emptyset, \{1, 2, 3\}), (\emptyset, \{2, 4\}), (\emptyset, \{3, 4\}), (\emptyset, \{4, 5\})\})$. Outputul corespunzator este $Comp = (serv_1, serv_4)$

O reprezentare grafica a instantei problemei SET-COVER in care am reprezentat multimile ca niste cercuri cu elementele sale in interior



O reprezentare grafica a instantei problemei acoperirii de servicii web in care serviciile sunt reprezentate ca niste noduri dreptunghiulare si instantele ca niste noduri in forma circulara. Arcele de la un nod corespunzator unui serviciu la un nod corespunzator unei instante reprezinta faptul ca serviciul respectiv ofera la output acea instanta. La output am reprezentat si instantele(desi ele nu fac parte din output), pentru o mai clara intelegere.

Input $provided = \emptyset, wanted = \{1, 2, 3, 4, 5\}$,



Propozitie 3.2. Functia f ruleaza in complexitate timp polinomiala in raport cu dimensiunea instantei problemei SET-COVER.

Demonstraţie. Avem urmatoarele:

- $|provided| = 0$
- $|requested| = |\mathcal{U}|$
- $|repository| = |\mathcal{S}|$ si $|serv_{input}| = 0, |serv_{output}| \leq |\mathcal{U}|, \forall serv \in \mathcal{S}$
- fiecare element din multimile de mai sus se creaza in timp constant

Din acestea de mai sus rezulta ca complexitatea timp a functie de transformare f este polinomiala in raport cu dimensiunea instantei problemei SET-COVER.

□

Propozitie 3.3. Dupa aplicarea functiei f peste inputul problemei SET-COVER, rezultatul este o instanta valida a problemei compozitiei de servicii web. Aceasta prorpietate este asigurata din modul de constructie al functiei.

Propozitie 3.4. Fie $(\mathcal{U}, \mathcal{S})$ inputul problemei SET-COVER. Solutia problemei compozitiei minime de servicii web pentru inputul $f(\mathcal{U}, \mathcal{S})$ poate fi procesata in timp polinomial (in dimensiunea instantei) pentru a se obtine un output valid pentru SET-COVER.

Demonstraţie. $f(\mathcal{U}, \mathcal{S}) = (provided, requested, repository) = (\emptyset, \mathcal{U}, \mathcal{R})$, unde
$\mathcal{R} = \{serv_1, serv_2, .., serv_n | serv_i = (\emptyset, S_i), i \in \{1, 2, ..., n\}\}$.
Fie $Comp = (serv_{i_1}, serv_{i_2}, .., serv_{i_k})$ solutia problemei compozitiei minime de servicii web pentu input descris mai sus. Solutia pentru SET-COVER va fi $\mathcal{C} = \{S_{i_1}, S_{i_2}, ..., S_{i_k}\}$.
Cum $Comp$ este solutie, rezulta ca:

1. $requested \subseteq \cup_{z \in 1,..,k} (serv_{i_z})_{out}$
2. $\forall Comp'$ solutie, $|Comp'| \geq |Comp|$

Din 1. rezulta ca $\mathcal{U} \subseteq \cup_{z \in 1,..,k} S_{i_z}$. Rezulta ca $\mathcal{C}$ este acoperire valida.
Mai ramane de aratat ca aceasta acoperire are intr-adevar cardinal minim.
Presupunem prin reducere la absurd $\exists \mathcal{C}' = \{S_{j_1}, S_{j_2}, ..., S_{j_q}\}$ acoperire a universului $\mathcal{U}$ cu $|\mathcal{C}'| < |\mathcal{C}|$ (adica $q < k$).
Cum $\mathcal{C}'$ este solutie $\Rightarrow \mathcal{U} \subseteq \cup_{z \in 1,..,q} S_{j_z}$. $\Rightarrow requested \subseteq \cup_{z \in 1,..,q} (serv_{j_z})_{out} \Rightarrow$
$(serv_{j_z})_{in} = \emptyset, \forall z \in \{1, ..., q\}$
Putem crea compozitia $Comp' = (serv_{j_1}, serv_{j_2}, .., serv_{j_q})$ valida cu $|Comp'| = q < k < |Comp| \Rightarrow$ Exista o compozitie valida de cardinal mai mic decat cardinalul lui $Comp$. Contradictie (deoarece $Comp$ este solutia problemei compozitiei de servicii web) $\Rightarrow$ Presupunerea facuta este falsa. $\Rightarrow$ Acoperirea $\mathcal{C}$ este de cardinal minim posibil, deci este solutie a problemei de acoperire a universului $\mathcal{U}$ cu multimi din familia $\mathcal{S}$.

Procesarea se face in timp polinomial cu dimensiunea inputului pentru problema compozitiei de servicii web, deoarece pentru fiecare serviciu din compozitia rezultata trebuie doar sa adaugam multimea corespunzatoare (cea cu acelasi indice) la acoperire. □

Din toate cele prezentate anterior rezulta ca problema compozitiei de servicii web este $\mathbb{NP}$ dificila.

3.3 Problema compozitiei minime de servicii web semantice este o generalizare a problemei MIN-WSC

Putem transforma in timp polinomial un input al problemei compozitiei de servicii web in input pentru problema compozitiei de servicii web semantice.
Fie $(provided, request, repository)$, (unde $repository = \{serv_1, serv_2, ..., serv_n\}$) inputul pentru problema compozitiei de servicii web. Fie urmatoarea taxonomie:
$taxonomie = (concepte, instante, ierarhieConcepte, relatieConceptInstanta)$, unde:

- $concepte = r$
- $instante = \cup_{i \in 1,...,n}((serv_i)_{input} \cup (serv_i)_{output})$
- $ierarhieConcepte = \emptyset$
- $relatieConceptInstanta = \{(r, inst) | inst \in instante\}$

$(provided, request, repository, taxonomie)$ este input valid (din modul cum am construit) pentru problema compozitiei minime de servicii web semantice.
Din faptul ca $ierarhieConcepte = \emptyset$ rezulta ca $subsumes(a, b)$ e adevarata ddaca $a = b$. De aici rezulta ca solutia oferita de algoritmul pentru rezolvarea problemei de servicii web semantice pentru acest input coincide exact cu solutia problemei compozitiei web pentru inputul $(provided, request, repository)$.
Cum problema compozitiei minime de servicii web este $\mathbb{NP}$ dificila si aceasta este o generalizare a problemei compozitiei minime de servicii web semantice rezulta ca si problema compozitiei de servicii web semantice este $\mathbb{NP}$ dificila.

Concluzie si directii de abordare ale problemei Cum problema compozitiei de servicii web este $\mathbb{NP}$ dificila si problema compozitiei de servicii web semantice este o problema mai generala decat aceasta rezulta ca ambele probleme sunt $\mathbb{NP}$ dificile.
Dupa aceasta analiza se pot deduce doua abordari diferite ale rezolvarii problemei folosind un algoritm determinist. Putem implementa un algoritm determinist care sa ofere mereu solutia optima, insa acesta avand complexitate timp exponentiala in dimensiunea instantei. Aceasta abordare ducand la algoritmi ale caror executie dureaza mult, care cel mai probabil vor merge foarte incet pentru instante mari si este deci nescalabil si probabil de multe ori ineficient in practica in repository-uri de dimensiuni relativ mari.

A doua abordare evidenta este aceea de a face un algoritm determinist de complexitate timp polinomiala, care insa nu garanteaza oferirea unei solutii optime pe orice instanta a problemei.
Algoritmul ce urmeaza a fi prezentat se incadreaza in cea de-a doua abordare, obtinand rezultate foarte bune pe teste, obtinand solutia optima pe sapte din opt teste intr-un timp foarte scurt.

4 Algoritmul

Ideea generala Ideea de baza a algoritmului este ca vom gasi o solutie valida (o compozitie care sa satisfaca solicitarea) parcurgand serviciile din repository (putem spune ca parcurgem de la "stanga la dreapta") si apoi analizam aceasta solutie (parcurgand de la "dreapta la stanga") pentru elimina (atat din solutie cat si din repository) serviciile inutile si pentru a elimina din serviciile "utile" instantele "inutile" pe care acestea le ofera la output. Alternam aceste doua parcurgeri pana cand obtinem de doua ori consecutiv aceeasi solutie in prima parcurgere (de la "stanga la dreapta").

4.1 Parcurgerea de la "stanga la drepta"

Vom gasi o compozitie valida care sa satisfaca cerea (C_{in}, C_{out}) in urmatorul mod: construim solutia nivel cu nivel.

- Primul nivel este format din toate serviciile ce pot fi apelate folosind doar instantele oferite ce cel ce face solicitarea (C_{in}).
- Al doilea nivel va fi format din toate serviciile care ofera la output macar o instanta noua si care pot fi apelate folosind instantele din multimea C_{in} impreuna cu instantele obtinute in urma apelarii tuturor serviciilor de pe primul nivel. O instanta a este numita "instanta noua" daca nu exista o alta instata b din cele obtinute anterior care sa fie egala cu ea sau care sa fie mai specifica decat ea.
- Cotinuam in aceeasi maniera (pentru fiecare serviciu dintr-un nou nivel i vom folosi ca input instantele din multimea C_{in} impreuna cu cele obtinute ca output de la serviciile de pe nivelele $\{1, .., i-1\}$).

Aceasta constructie se opreste atunci cand am obtinut (ca output al serviciilor apelate) toate instantele din multimea C_{out} sau cand nu mai sunt servicii ce pot fi apelate folosind toate instantele detinute. In cazul in care constructia s-a oprit cu prima conditie (cand am obtinut toate instantele dorite), atunci am construit o compozitie valida. Aceasta compozitie are execution path minim.
Daca constructia s-a oprit in cel de-al doilea caz (cand nu mai putem apela alte servicii si nu avem toate instantele cerute), inseamna ca nu exista nicin o compozitie valida care sa satisfaca solicitarea.

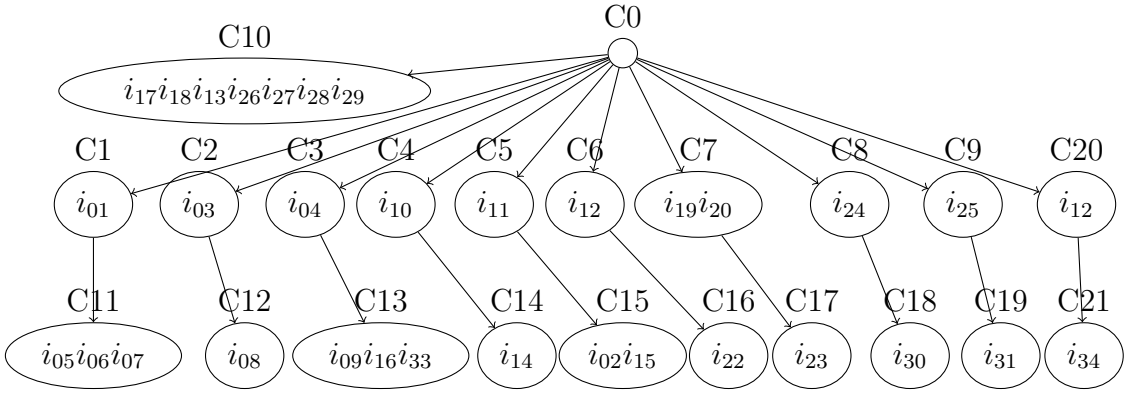
Definitie 4.1. Formal fiecare nivel este o multime de servicii cu urmatoarele proprietati:

- $nivel_1 = \{serv \in repository | serv_{in} \subseteq \mathcal{C}_{in}\}$
- $nivel_i = \{serv \in repository | serv_{in} \subseteq owned, \exists newout \in serv_{out}\ a.i. \nexists out \in owned$ cu $subsumes(newout, out)$ adevarat, unde $owned = (\cup_{serv' \in nivel_j, j=\{1,..,i-1\}} serv'_{out}) \cup provided\}, i > 1$

Exemplu 4.1. Fie $(C_{in}, C_{out}, repository, taxonomy)$ input al problemei unde:

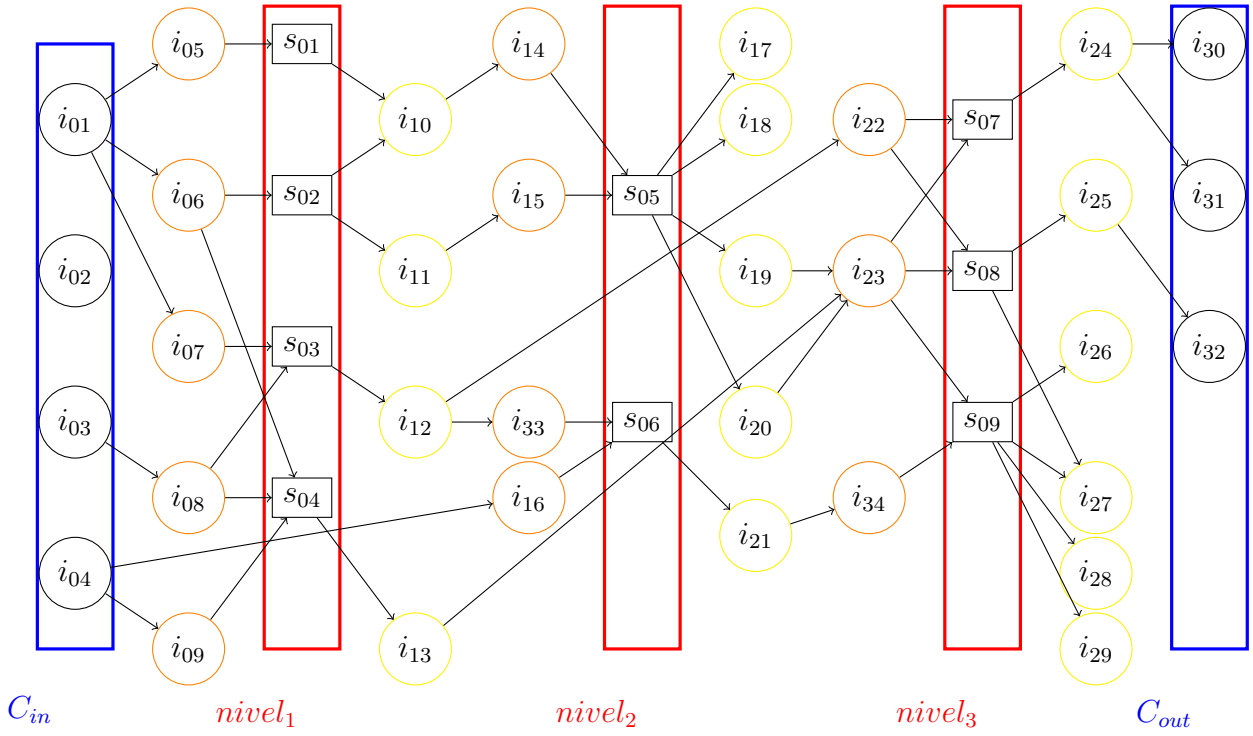
- $C_{in} = \{i_{01}, i_{02}, i_{03}, i_{04}\}$
- $C_{out} = \{i_{30}, i_{31}, i_{33}\}$
- $repository = \{s_{01} = (\{i_{05}\}, \{i_{10}\}), s_{02} = (\{i_{06}\}, \{i_{10}, i_{11}\}), s_{03} = (\{i_{07}, i_{08}\}, \{i_{12}\}), s_{04} = (\{i_{06}, i_{08}, i_{09}\}, \{i_{13}\}), s_{05} = (\{i_{14}, i_{15}\}, \{i_{17}, i_{18}, i_{19}\}), s_{06} = (\{i_{33}, i_{16}\}, \{i_{20}, i_{21}\}), s_{07} = (\{i_{22}, i_{23}\}, \{i_{24}\}), s_{08} = (\{i_{22}, i_{23}\}, \{i_{25}, i_{27}\}), s_{09} = (\{i_{23}\}, \{i_{26}, i_{27}, i_{28}, i_{29}\})\}$
- $taxonomy = (concepte, instante, ierarhieConcepte, relatieConceptInstanta)$, unde:
  - $concepte = \{C_0, C_1, C_2, .., C_{18}\}$
  - $instante = \{i_{01}, i_{02}, .., i_{33}\}$
  - $ierarhieConcepte = \{(C1, C2), (C3, C4), (C5, C6), (C7, C8), (C9, C10), (C11, C12), (C13, C14), (C15, C16), (C17, C18), (C0, C1), (C0, C2), ...(C0, C18)\}$
  - $relatieCOnceptInstanta = \{(C1, i_{01}), (C2, i_{03}), (C3, i_{04}), (C4, i_{10}), (C5, i_{11}), (C6, i_{12}), (C7, i_{19}), (C7, i_{20}), (C8, i_{24}), (C9, i_{25}), (C10, i_{26}), (C10, i_{27}), (C10, i_{28}), (C10, i_{29}), (C11, i_{05}), (C11, i_{06}), (C11, i_{07}), (C12, i_{08}), (C13, i_{09}), (C13, i_{16}), (C13, i_{33}), (C14, i_{14}), (C15, i_{02}), (C15, i_{15}), (C16, i_{22}), (C17, i_{23}), (C18, i_{30}), (C19, i_{31})\}$

In imaginea urmatoare este reprezentata taxonomia in forma arborescenta unde fiecarui concept ii corespunde un nod. In interiorul nodului corespunzator unui concept am notat toate instantele ce se incadreaza direct in acel concept. Daca o instanta b se gaseste "in interiorul" unui nod de pe drumul de la nodul in care se afla instanta a la radacina, atunci $subsumes(a, b)$ este adevarat si b poate fi dat ca parametru de intrare pentru un serviciu ce asteapta instanta a. De exemplu instanta $subsumes(i_{04}, i_{16})$ e adevarat.



Imaginea urmatoare este o reprezentare grafica a unei compozitii valide pentru inputul descris mai sus impreuna cu impartirea pe nivele a serviciilor din compozitie. Serviciile sunt reprezentate cu noduri in forma dreptunghiulara, in timp ce instantele sunt reprezentate in noduri in forma circulara.
Semnificatia areclor este urmatoare:

- De la o instanta i la un serviciu s :instanta i este input al serviciului s.
- De la un serviciu s la o instanta i: instanta i este output al serviciului s
- De la o instanta i_1 la o instanta i_2 : i_1 este instanta output a unui serviciu s_1 sau este din C_{in}, i_2 este instanta input a unui serviciu s_2 sau este din C_out, s_1 este pe un nivel mai mic ca $s2$ si $subsumes(i_2, i_1)$ e adevarat, adica instanta i_1 poate fi folosita ca input in locul instaintei i_2 si i_1 poate fi returnata in locul lui i_2 pentru instantele cerute (cele din C_{out})



Observatii pe baza exemplului: Daca ne uitam la serviciile s_{06} si s_{07} observam ca o parte din instantele input pot fi obtinute si de pe orice nivele anterioare, nu neaparat cel imediat anterior, sau chiar din cele oferite in solicitare(C_{in}). Cu toate acestea un serviciu este pus pe un nivel doar cand toate inputurile sale sunt deja obtinute (de ex. s_{07}).
Putem observa ca sunt instante obtinute ca output care nu sunt folosite in compozitie (de ex. i_{13},i_{21}) si mai mult decat atat in compozitie pot exista servicii care sa returneze doar instante nefolosite (de ex. s_09).
Pentru a imbunatati compozitia eliminand ce este inutil din ea facem urmatoarea **parcurgere de la "dreapta la stanga"**.
Daca ne uitam pe $nivel_1$ putem observa ca exista serviciul s_{01} care ofera la output instanta i_{10} si serviciul s_{02} care ofera aceeasi instanta si in plus instanta i_{11}. Ar fi fost suficient sa avem doar s_02 in acest nivel, dar conform celor discutate pana acum s_{01} nu este considerat "servicu inutil". Pentru acest tip de situatie se va face un **sistem de scor dinamic** descris intr-o sectiune urmatoare.

4.2 Parcurgere de la "dreapta la stanga"

In acest pas vom scoate anumite elemente "inutile" din solutie si pentru o viteza mai mare a algoritmului si din repository.
In repository pastram doar serviciile ce se regasesc in compozitia de la pasul anterior.
Parcurgem acum solutia gasita de la "dreapta la stanga" (adica de la ultimul nivel la primul) pentru a vedea care dintre serviciile de pe un nivel (si care outputuri ale lor) sunt "utile".
Vom nota cu $nrNivele$ numarul de nivele din compozitia obtinuta.

Definitie 4.2. Spunem despre un serviciu dintr-o compozitie ca este util pentru o solicitare daca printre instantele returnate la output exista macar o instanta utila in acea compozitie pentru acea solicitare.

Definitie 4.3. Spunem despre o instanta output returnata de un serviciu (de pe nivelul i) dintr-o compozitie ca este utila pentru o solicitare (C_{in}, C_{out}) daca:

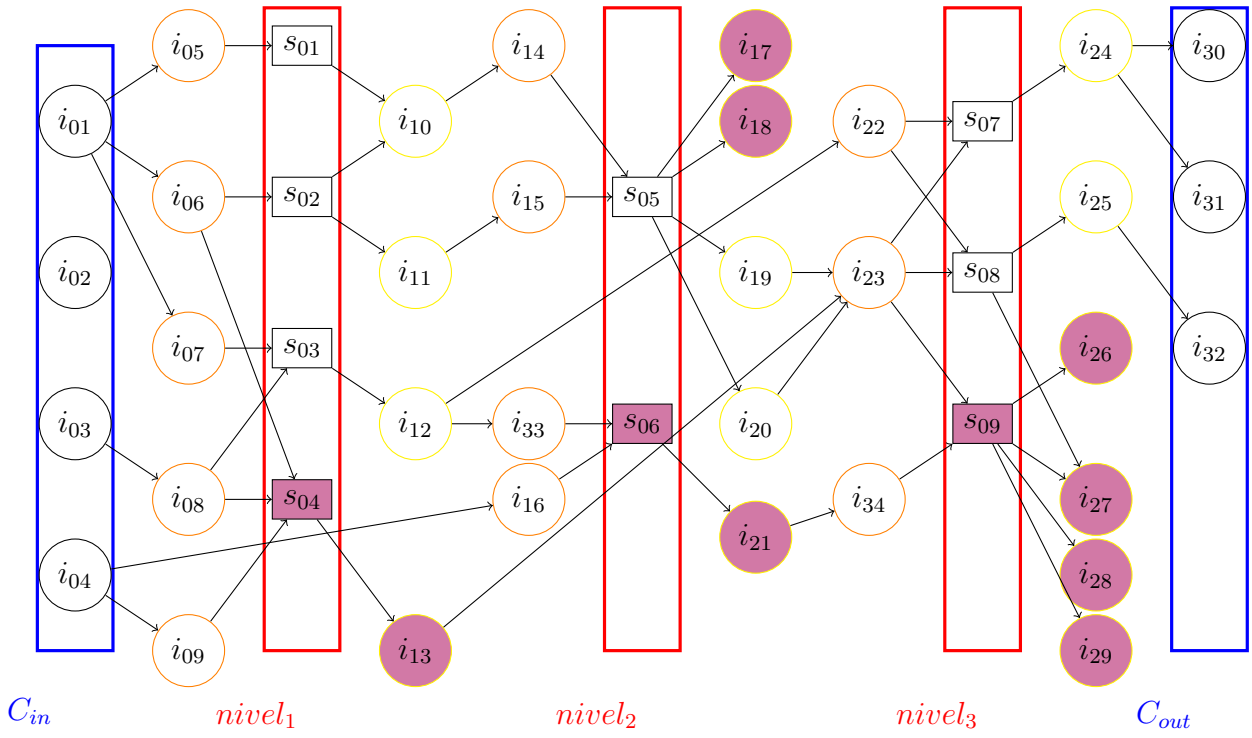
- este egala cu una dintre instantele C_{out} sau cu o particularizare a acesteia si in plus nu exista printre instantele C_{in} o instanta egala sau mai particulara ca aceasta, pentru serviciile de pe ultimul nivel, adica $i = nrNivele$

- 1. este egala cu o instanta $inst \in C_{out}$ sau cu o particularizare a acesteia SI nu exista un serviciu pe un nivel urmator $k, i < k$ care sa returneze la output instanta $inst$ sau o particularizare a acesteia
  2. instanta este egala cu o instanta $inst$ care este instanta input sau o particularizare a unei instante input a unui serviciu util de pe un nivel urmator $j, j > i$ SI in plus nu exista un serviciu pe un nivel intermediar $k, i < k < j$ care sa returneze la output instanta $inst$ sau o particularizare a acesteia
  3. nu exista printre instantele C_{in} o instanta egala sau mai particulara ca aceasta

  Daca are loc (1. sau 2.) si 3., pentru serviciile de pe orice nivel $i < nrNivele$

Pastram pe fiecare nivel doar serviciile utile (conform definitiei de mai sus) in compozitie (cea construita la pasul anterior) si pentru solicitarea data ca input la algoritm.
La finalul parcurgerii pastram pentru fiecare serviciu $serv$ in multimea sa de outputuri $serv_{out}$ doar instantele utile in compozitie pentru solicitare (utile pentru macar un serviciu din compozitie, nu neaparat pentru $serv$) .

Exemplu 4.2. In imaginea urmatoare sunt marcate prin umplere nodurile corepunzatoare serviciilor si instantelor inutile din exemplul de la pasul anterior.



Observatii pe baza exemplului: Toate instantele din care nu pleaca nici un arc sunt marcate (de ex. i_{13}, i_{17}, i_{29}).
Serviciile din care pleaca doar instante inutile sunt marcate (de ex. s_{09}), iar cele din care pleaca macar o instanta utila raman nemarcate (de ex. s_{05}).
Daca ne uitam la instanta i_{13} care este output al serviciului s_{04} de pe nivelul $i = 1$ putem remarca ca este marcata ca inutila, desi din ea pleaca un arc spre o instanta input a unui serviciu util (i_{23} input al lui s_{08} de pe nivelul $j = 3$). Instanta i_{13} este considerata inutila, deoarece exista instante de la care pleaca un arc spre i_{23} (de ex. i_{19}) care sunt returnate de un serviciu de pe un nivel intermediar (s_{05} de pe nivelul $k = 2$ $i = 1 < k = 2 < j = 3$).
Se observa necesitatea parcurgerii nivelelor in ordine descrescatoare, deoarece pentru a sti despre un serviciu sau despre o instanta daca sunt utile trebuie sa stim aceste informatii despre nivelele urmatoare. De exemplu pentru serviciu s_{06} de pe nivelul 2 putem spune ca e inutil dupa ce am aflat despre s_{09} de pe nivelul 3 ca este inutil.

4.3 Scor

Scorul unui serviciu este egal cu numarul de instante output pe care acesta le returneaza si nu le avem deja. Pentru eficienta, acest scor trebuie calculat in timpul constructiei solutiei. Atunci cand obtinem o instanta noua, aflam care sunt toate instantele din taxonomie care sunt mai generale decat aceasta instanta noua si pe care nu le avem deja si pentru fiecare dintre aceastea parcurgem toate serviciile care au acea instanta ca output si decrementam valoarea scorului cu valoarea 1.

4.4 Subsumes

Pentru a vedea daca o instanta b este mai particulara decat o instanta a trebuie sa calculam valorea functiei $subsumes(a, b)$.
Pentru a calcula in complexitate timp $O(1)$ valoare functiei $subsumes\,(a, b)$ (adica daca instanta b poate fi folosita ca input pentru un serviciu ce are nevoie de instanta a) am facut o preprocesare de complexitate timp $O(N)$, unde N este numarul de concepte din taxonomie. Am parcurs in adancime taxonomia si pentru fiecare nod (concept) am retinut timpul de intrare, respectiv timpul de iesire din el.
Pentru fiecare nod, aceste doua numere sunt egale cu prima, respectiv ultima pozitie a aparitiei nodului intr-o liniarizare in care adaugam fiecare nod exact de 2 ori: o data cand intram in el si o data cand iesim din el (adica atunci cand toti succesorii sai au fost parcursi si adaugati in liniarizare).
Intre prima si ultima aparitie ale unui nod in liniarizare se afla exact succesorii sai, fiecare aparand de exact 2 ori.
Mai formal definim doua functii tintrare si tiesire ale caror valori reprezinta cele doua numere descrise mai sus. Pentru definirea functiei *tintrare* consideram ca succesorii directi ai unui nod sunt ordonati: $pozitie : Concepte \to \mathbb{N}$, aceasta fiind functia care defineste ordinea, numerotarea incepand de la 1.

Definitie 4.4. $tintrare : Concepte \to \mathbb{N}$

$$tintrare(concept) = \begin{cases} 1, \text{daca concept=radacina arborelui} \\ tintrare(parinteI(concept)) + 1, \text{daca concept} \neq \text{radacina arborelui} \\ \qquad\qquad \text{si } pozitie\,(concept) = 1 \\ tiesire\,(frate) + 1, \text{unde } pozitie(frate) + 1 = pozitie(concept) \\ \qquad\qquad \text{si } parinteI(frate) = parinteI(concept), \text{altfel} \end{cases}$$

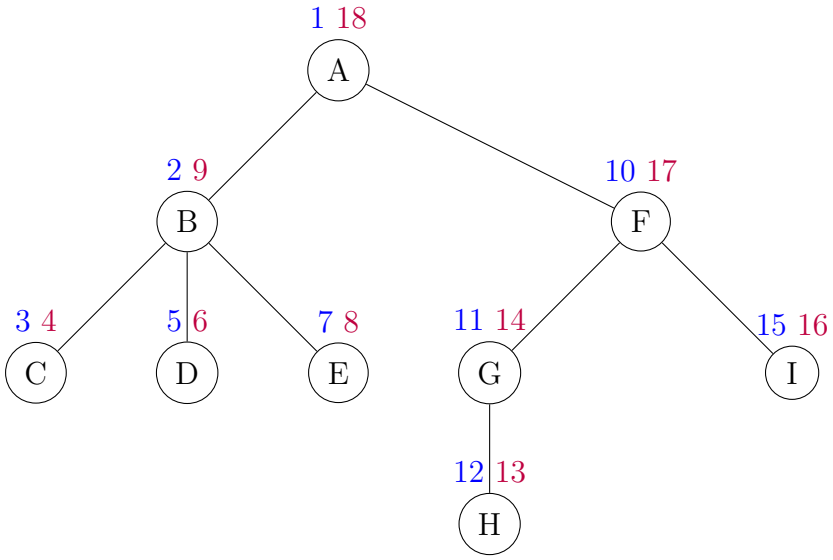
Definitie 4.5. $tiesire : Concepte \to \mathbb{N}$.

$$tiesire(concept) = \begin{cases} tinrare(concept) + 1, \text{daca } fSubconcepte(concept) = \emptyset \\ \max\limits_{subconcept \in fSubconcepte(concept)} tiesire(subconcept) + 1, \text{altfel} \end{cases}$$

Deci pentru a verifica daca o instata a este o generalizare a altei intante b este suficient sa verificam urmatoarea conditie: $tinrare[a] \leq tintrare[b] \leq tiesire[a]$.

Exemplu Sa zicem ca avem conceptele A,B,C,D,E,F,G,H,I cu ierarhia prezentata in imagine si fii unui nod ordonati de la stanga la dreapta incepand cu pozitia 1. Valoarea functiei *tintrare* pentru fiecare nod este reprezentata in culoarea albastra deasupra nodului corespunzator, iar valoare functiei *tiesire* este reprezentata cu in culoarea mov.
O liniarizare a arborelui conform explicatiilor de mai sus este urmatoarea:

A,	B,	C,	C,	D,	D,	E,	E,	B,	F,	G,	H,	H,	G,	I,	I,	F,	A
1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,	16,	17,	18



Pentru nodul A timpul de intrare este $tintrare(A) = 1$, deoarece A este radacina arborelui.
Pentru nodul B care este primul fiu al nodului A avem : $tintrare(B) = tintrare(parinteI(B))+1 = trintrare(A)+1 = 2$, $subconcepte(B) = \{C, D, E\}$, $tiesire(B) = max_{sc\in\{C,D,E\}} tiesire(sc)+1 = max\{4, 6, 8\} + 1 = 9$. Daca ne uitam in liniarizare vedem ca pe pozitia 2 si 9 se afla conceptul B si intre pozitiile 2 si 9 se afla conceptele C, D si E de exact 2 ori.
Pentru nodul E care este al treilea fiu al nodului B avem $tintrare(E) = tiesire(D)+1 = 7$, deoarece $pozitie(D) + 1 = 7 = pozitie(E)$. Fiindca E este frunza in arbore $tiesire(E) = tintrare(E) + 1$.

4.5 Pseudocod

4.5.1 Precalculare necesara pentru a calcula rapid subsumes(ia,ib)

Functia care face liniarizarea arborelui descrisa in sectinea 6.4 si calculeaza cele doua valori corespunzatoare unui nod este descrisa in urmatorul algoritm. Avem o functie dfs care primeste ca argument un concept si face liniarizarea si calcularea celor doua valori(de intrare si de iesire) ale nodurilor din subarborele ce are acel concept ca radacina.
Cele doua valori, desi sunt descrise formal prin doua relatii mai complexe, pot fi calculate usor folosind o variabila globala pe care am notat-o cu *timp* care se incrementeaza de fiecare data cand intram si cand iesim dintr-un nod. *tintrare* ia valoare acestei variabile inainte de a parcurge fiii nodului, iar *tiesire* ia valoare variabilei *timp* dupa ce tot subarborele ce are acel concept ca radacina a fost parcurs.
Din aceasta varianta a descrierii calculului acestor doua valori este mai intuitiv de inteles faptul ca un nod b este succesor al unui nod a ddaca $tintrare[a] < tintrare[b] < tiesire[a]$.
Pentru a calculat *tintrare* si *tiesire* pentru toate conceptele din taxonomie vom apela

aceasta functie dand ca parametru radacina arborelui de concepte, adica cu cel mai general concept existent in taxonomie.

Data: nod
Result:
initialization $timp \leftarrow 0$;
Function `dfs(`*concept*`):`
  $timp \leftarrow timp + 1$;
  $tintrare[concept] \leftarrow timp$;
  foreach $subconcept \in subconcepte(concept)$ **do**
    $dfs(subconcept)$
  end
  $timp \leftarrow timp + 1$;
  $tiesire[concept] \leftarrow timp$;
$timp \leftarrow 0$;
$dfs(radacina)$;

Algorithm 1: calculare timp intrare si iesire ale fiecarui nod

4.5.2 Functia principala (conform ideii de baza)

In functia urmatoare este bucla principala a algoritmului in care se alterneaza parcurgerea de la stanga la dreapta cu parcurgerea de la dreapta la stanga pana cand nu se mai obtin imbunatatiri ale solutiei.
La apelul functiei $findASolution()$ se face parcurgerea de la stanga la drepta conform celor descrise in sectiunea 6.1 si se returneaza compozitia gasita.
Aceasta compozitie este parcursa pe nivele in ordine descrescatoare (de la dreapta la stanga conform sectiunii 6.2) si in ea se pastreaza doar instantele si serviciile utile. Aceasta parcurgere se face in functia $keepOnlyUsefulServsAndInsts$. Se observa ca in repository se pastreaza doar serviciile ce fac parte din compozitie pentru a spori viteza algoritmului. Scorul serviciilor (conform sectiunii 4.3) este calculat si folosit in ambele functii.

Data:
Result: solutie valida cu exec path minim
initialization $prevSol \leftarrow \emptyset$ $solution \leftarrow \emptyset$;
do
  $prevSol \leftarrow solution$;
  $solution \leftarrow findASolution\,()$;
  $repository \leftarrow \cup_{layer \in result} layer$;
  $keepOnlyUsefulServsAndInsts\,(solution)$;
while $prevSol \neq solution$;
return $solution$;

Algorithm 2: functia solve

4.5.3 Parcurgerea de la stanga la dreapta

Functia $findASolution$ construieste compozitia nivel cu nivel pornind de la instantele C_{in} oferite de cel ce face solicitarea pe care in algoritm le-am notat cu $providedInsts$. Constructia are loc atat timp cat mai exista servicii ce se pot apela folosind ca input instantele obtinute pana in acel moment sau pana cand am obtinut toate instantele din C_{out} pe care in algoritm le-am notat cu $wantedInsts$.
Parcurgerea serviciilor ce pot fi adaugate pe nivelul curent se face in ordinea descrescatoare a scorurilor serviciilor pentru a incerca sa avem cat mai putine servicii pe nivel. Un serviciu se adauga pe nivel doar daca acesta ofera outputuri noi. Adaugarea pe nivel se face in functia $addServiceToLayer$.

Data:
Result: compozitie valida
initialization $auxWInsts \leftarrow wantedInsts$;
$result \leftarrow \emptyset$ $servicesReadyToCall \leftarrow \emptyset$;
$ownedInsts \leftarrow \{inst | \exists pI \in providedInsts\, subsumes(inst, pI)\}$;
$mark(ownedInsts, null)$;
while $wantedInsts \neq \emptyset$ *and* $servicesReadyToCall \neq \emptyset$ **do**
  $layer \leftarrow \emptyset$;
  $servicesToCall \leftarrow servicesReadyToCall$;
  $servicesReadyToCall \leftarrow \emptyset$;
  while $servicesToCall \neq \emptyset$ **do**
    $bestService \leftarrow \max\{servicesToCall\}$;
    $servicesToCall \leftarrow servicesToCall \setminus \{bestService\}$;
    if $\neg subsumes(newOuts[bestService], ownedInsts)$ **then**
      $addServiceToLayer(bestService, layer)$;
    end
  end
  if $layer \neq \emptyset$ **then**
    $result \leftarrow result.layer$;
  end
end
$wantedInsts \leftarrow auxWInsts$;
foreach $service \in repository$ **do**
  $neededInps\,[service] \leftarrow inps\,[service]$;
  $newOuts\,[service] \leftarrow outs\,[service]$;
end
return $result$;

Algorithm 3: functia findASolution

Functia $addServiceToLayer$ adauga serviciul $service$ primit ca parametru pe nivelul

curent. Atunci cand adaugam un serviciu la compozitie trebuie sa vedem ce instante output ne ofera acesta pe care nu le avem deja (adica nu sunt in $ownedInsts$): $instsObtained \leftarrow \{gInst | subsumes\,(gInst, inst)\} \setminus ownedInsts$. Deoarece taxonomia poate avea dimensiuni mari este vital ca aceasta operatie sa se faca cat de optim este posibil. In implementarea algoritmului este folosita o structura de date care sa mentina instantele pe care le avem si care sa aiba proprietatea ca putem afla "rapid" daca o instanta se afla sau nu in ea. In acest scop am ales o tabela hash. Pentru a vedea ce instante noi ofera un serviciu la output parcurgem outputurile sale si pentru fiecare instanta output in parte urcam in taxonomie de la nodul corespunzator conceptului parinte al instantei spre radacina si ne oprim atunci cand am dat peste un concept ale carui instante sunt deja in multimea $ownedInsts$. Nu are rost sa continuam urcarea in taxonomie deoarece toate instantele corespunzatoare nodurilor de pe drumul de la nodul in care ne-am oprit la radacina sunt deja in multimea $ownedInsts$ fiindca au fost adaugate cel tarziu cand am adaugat macar o instanta din nodul in care ne-am oprit. Cu aceasta optimizare garantam ca nodurile din taxonomie nu sunt vizitate de mai multe ori in cadrul aceleiasi constructii.

Result: adauga serviciul service la nivelul layerLaTeX2e

initialization;

$layer \leftarrow layer \cup \{service\}$;

foreach $inst \in newOuts[service]$ **do**

  $instsObtained \leftarrow \{gInst | subsumes\,(gInst, inst)\} \setminus ownedInsts$;

  $ownedInsts \leftarrow ownedInsts \cup instsObtained$;

  $mark(instsObtained, service)$

end

Algorithm 4: functia addServiceToLayer care ia ca argumente service, layer

Pentru a sti ce servicii pot fi apelate la un moment dat in constructia compozitiei, pentru fiecare serviciu se retin instantele input care sunt necesare pentru apelarea sa si pe care nu le detinem inca ($neededInps$ in algoritm). Atunci cand numarul acestora ajunge la 0, putem apela serviciul, deci il vom adauga la urmatorul nivel.

Pentru a eficientiza calcularea acestei informatii retinem o structura de date care retine pentru fiecare instanta ce servicii au respectiva instanta la input. Cu aceasta optimizare impreuna cu faptul ca fiecare instanta este adaugata maxim o data la $ownedInsts$ garantam faptul ca fiecare input al unui serviciu va fi parcurs maxim o data pe constructie.

Pentru a mentine scorul serviciilor, pentru fiecare instanta noua pe care o obtinem parcurgem toate serviciile $serv$ ce au acea instanta ca output (folosind o structura de date similara cu cea descrisa anterior) si scoatem instanta din multimea de outputuri noi ($newOuts$) pe care le are serviciul $serv$ (evident: pentru toate serviciile $serv$ diferite de cel care ne-a furnizat primul acea instanta: $exceptService$). Astfel, analog cu parcurgerea anterioara se garanteaza ca fiecare output al unui serviciu va fi parcurs maxim o data pe constructie.

```
foreach service ∈ repository do
    if neededIns[service] ∩ instsObtained ≠ ∅ then
        neededIns[service] ← neededIns[service] \ instsObtained;
        if neededIns[service] = ∅ then
            servicesReadyToCall ← servicesReadyToCall ∪ {service};
        end
    end
    if service ≠ exceptService then
        newOuts[service] ← newOuts[service] \ instsObtained;
    end
end
```

Algorithm 5: functia mark care ia instancesObtained, exceptService si marcheaza toate serviciile ce au inputuri sau outputuri din multimea instancesObtained

4.5.4 Parcurgerea de la dreapta la stanga

In aceasta functie parcurgem nivelele de la ultimul la primul pentru a vedea ce este util si ce este inutil pe ficare nivel. Incepem prin a considera utile instantele cerute de cel ce face solicitarea, adica cele din C_{out} pe care in algoritm le-am notat cu $wantedInsts$.
Vom mentine instantele utile intr-o tabela hash $usefulInsts$. Pe masura ce parcurgem noi nivele adaugam instantele utile pentru ultimul nivel parcurs ($usefulForLevel$) la toate instantele utile ($usefulInsts$).
Vom mai mentine inca o tabela hash $reqInsts$ care va fi initializata tot cu $wantedInsts$ si va reprezenta multimea instantelor (input sau wanted) $inst$ de pe nivelele urmatoare pentru care nu am gasit inca o instanta output care sa poata fi data ca parametru in locul instantei $inst$.
Consideram o instanta output out ca este utila daca este o particularizare a altei instante $inst$ din multimea $reqInsts$ si daca nu exista o instanta pr in multimea de instante din C_{in} (adica din $provided$) care sa fie o particularizare a lui out (deoarece nu are rost sa obtinem instante mai generale decat ceea ce avem in orice compozitie inca de la inceputul ei: instantele din $provided$).
Daca un serviciu nu ofera la output nici o instanta utila, atunci el va fi considerat inutil si va fi scos din compozitie. In cazul in care serviciul este considerat util, atunci toate instantele sale input care nu sunt mai generale decat macar o instanta din cele $provided$ vor fi adaugate la multimiea $usefulForLevel$. Pentru un serviciu util mai trebuie sa identificam instantele din multimea $reqInsts$ care sunt mai generale decat macar un output dat de serviciu si sa scoatem instantele identificate din multime, deoarece daca serviciul este apelat, atunci putem folosi outputurile lui ca inputuri in locul instantelor identificate.
La finalul parcurgerii fiecarui nivel scoatem din compozitie serviciile inutile de pe nivelul respectiv, adaugam la $reqInsts$ si la $usefulInsts$ instantele input identificate a fi utile

(si putem spune chiar necesare) pentru acest nivel $usefulForLevel$.

Data:
Result: solutie valida cu exec path minim
initialization $usefulInsts \leftarrow wantedInsts$, $toFindInsts \leftarrow wantedInsts$;
$reqInsts \leftarrow reqInsts \setminus \{inst \in reqInsts | \exists pr \in providedInsts, subsumes(inst, pr)\}$;
for $i \leftarrow |solution| - 1$ **to** 0 **do**
  $nivel \leftarrow solution[i]$;
  $servsToRemove \leftarrow \emptyset$;
  $usefulForLevel \leftarrow \emptyset$;
  foreach $serv \in nivel$ **do**
    $instsToRemove \leftarrow \{inst | \exists out \in outs[serv], subsumes(inst, out)\}$;
    if $instsToRemove = \emptyset$ **then**
      $servsToRemove \leftarrow servsToRemove \cup \{serv\}$;
    end
    else
      $usefulForLevel \leftarrow usefulForLevel \cup \{inp \in inps[serv] | \nexists pr \in providedInsts, subsumes(inp, pr)\}$;
    end
    $reqInsts \leftarrow reqInsts \setminus instsToRemove$;
  end
  $solution[i] \leftarrow nivel \setminus servicesToRemove$;
  $reqInsts \leftarrow reqInsts \cup usefulForLevel$;
  $usefulInsts \leftarrow usefulInsts \cup usefulForLevel$;
end
$insts \leftarrow usefulInsts$;

Algorithm 6: keepOnlyUsefulServsAndInsts

4.6 Paralelizare

Algoritmul prezentat are avantajul ca o parte importanta si costisitoare ca timp din el poate fi executata pe mai multe fire de executie. Pe testele pe care algoritmul a fost testat, aceasta implementare nu a imbunatatit semnificativ rezultatele, deoarece algoritmul initial este prea rapid pe teste pentru a fi avantajoasa investitia de timp pentru a crea fire noi de executie. Pe teste mai mari insa, paralelizarea algoritmului s-ar putea sa fie mai avantajoasa.

Cea mai costisitoare parte a algoritmului din punctul de vedere al timpului de executie este functia $findASolution$, adica parcurgerea de la stanga la dreapta.

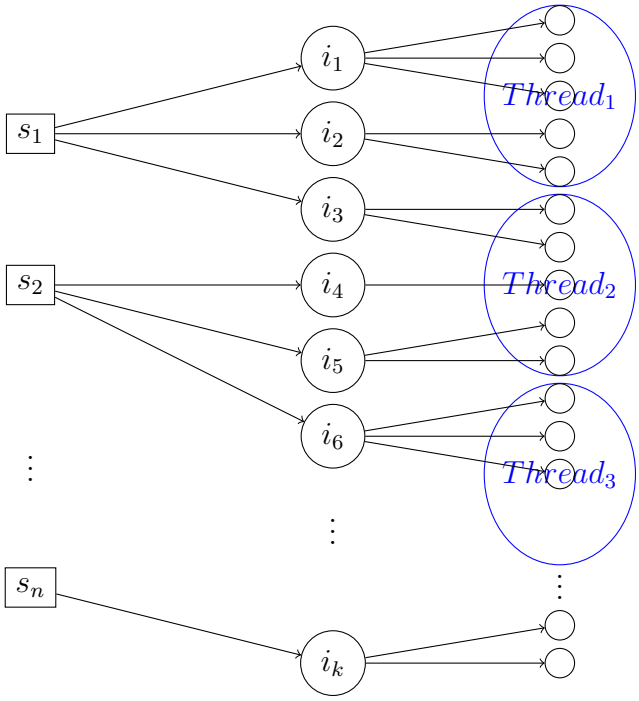
In tabelul urmator am prezentat cat timp ia pe fiecare test fiecare din cele doua parcurgeri si care este timpul total de rulare al algoritmului. Pentru fiecare parcurgere am facut suma toatala a timpilor de executie al fiecarui apel al functiei, deoarece fiecare functie

poate fi apelata de mai multe ori, pana cand ele nu mai aduc imbunatatiri solutiei.
Coloanele a doua si a treia corespund pargurgerii de la stanga la dreapta respectiv de la dreapta la stanga, iar a patra coloana reprezinta timpul total de rulare al algoritmului (fara citirea fisierelor corespunzatoare taxonomiei si repository-ului). Toti timpii sunt masurati in milisecunde.

Nr. test	Stg→ Drp timp(ms)	Drp → Stg timp(ms)	Tot algoritmul timp(ms)
1.	10	2	22
2.	11	6	27
3.	38	7	76
4.	15	8	34
5.	55	8	87
6.	68	18	132
7.	47	14	95
8.	43	29	109

Se poate vedea ca timpul de executie al parcurgerii de la stanga la drepta reprezinta cam jumatate din timpul total de executie al agloritmului in toate cele opt teste. Deci o optimizare a acestei functii ar putea micsora semnificativ timpul total de executie al algoritmului.
Deoarece compozitia se calculeaza nivel cu nivel, iar fiecare nivel nou este calculat in functie de nivelele anterioare, paralelizarea se poate face doar in cadrul anumitor calcule de pe un nivel. Parcurgerea serviciilor de pe un nivel se face in ordinea scorurilor serviciilor si decizia de a elimina sau nu un serviciu de pe nivel se face in functie de serviciile anterior parcurse pe acest nivel, deci aceasta parte va ramane secventiala.
Cand decidem sa pastram un serviciu pe nivelul curent, atunci trebuie sa parcurgem toate outputurile sale si sa vedem ce instante noi obtinem cu ajutorul lor si daca avem servicii ce pot fi acum apelate pentru prima oara folosind instantele vechi impreuna cu instanele noi obtinute, le vom adauga pe urmatorul nivel. Aceast calcul se face in functiile *addServiceToLayer* (in care iteram prin outputuri si vedem ce instante noi pe care nu le aveam ne ofera fiecare, parcurgand un drum in taxonomie de la un nod spre radacina pana cand ajungem la un concept pe care-l detinem deja) si *mark*(in care se parcurg toate serviciile ce au o anumita instanta la input si verificam daca ele pot fi acum apelate) descrise in pseudocod. Observam insa ca pasii descrisi anterior sunt independenti de la serviciu la serviciu, deci pot fi executati in paralel.
Am ales sa grupez mai multe instante pentru care se apeleaza functia *mark* si sa creez un fir de executie pentru fiecare grup. Dimensiunea potrivita a grupului de instante pentru un fir de executie ar urma sa fie determinata prin experimente pe teste mai mari.
Singurele obiecte care se modifica in cadrul functiei *mark* sunt seriviciile din repository (se modifica numarul de inputuri de care mai are nevoie un serviciu si numarul de outpu-

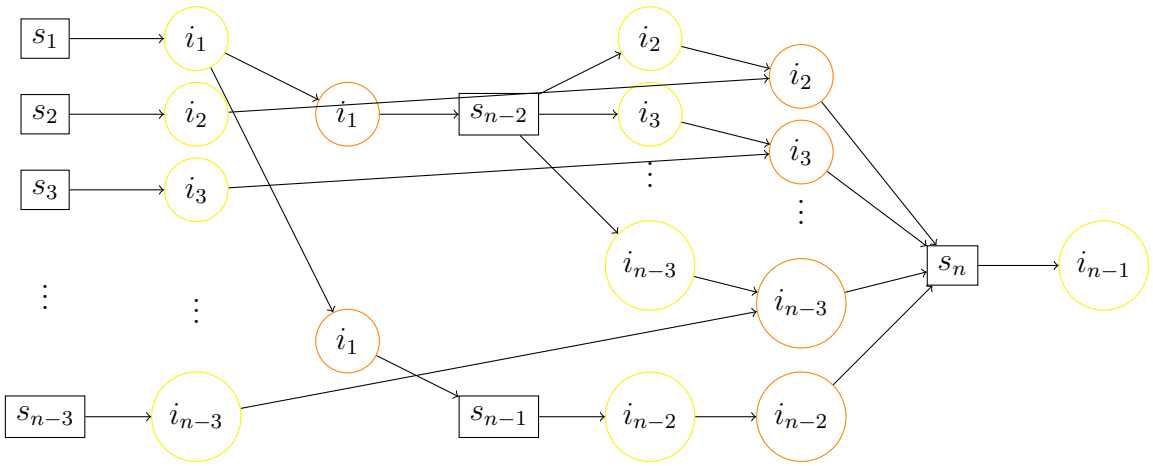
turi noi) si lista care ne tine serviciile ce sunt acum gata de a fi apelate si vor fi puse pe nivelul urmator. Astfel, pe portinunile de cod in care se acceseaza aceste obiecte trebuie sa avem grija ca accesul firelor de executie la aceste obiecte sa fie sa se faca neconcurent. In plus trebuie sa asteptam sa se termine executia tuturor firelor de executie in afara de firul prinicpal atunci cand vrem sa trecem la urmatorul nivel.



4.7 Limita de eroare

In cazul in care consideram ca un serviciu poate avea oricate instante input si oricate instante output (chiar de ordinul dimensiunii respository-ului) putem construi un exemplu pe care algoritmul prezentat sa returneze o compozitie care este cu $O(n)$ servicii mai mare decat compozitia optima, unde n este numarul de servicii din repository.
In imaginea urmatoare am reprezentat grafic un exemplu ce corespunde urmatorului input: $(C_{in}, C_{out}, repository, taxonomy)$, unde:

- $C_{in} = \emptyset$
- $C_{out} = \{i_{n+2}\}$
- $repository = \{s_1 = (\emptyset, \{i_1\}), s_2 = (\emptyset, \{i_2\}), s_3 = (\emptyset, \{i_3\}), ..., s_{n-3} = (\emptyset, \{i_{n-3}\}),$ $s_{n-2} = (\{i_1\}, \{i_2, i_3, ..., i_n\}), s_{n-1} = (\{i_1\}, \{i_{n-2}\}), s_n = (\{i_2, i_3, ..., i_{n-2}\}, \{i_{n-1}\})\}$
- taxonomia contine doar un concept de baza si toate instantele fac parte din el.



Compozitia algoritmului Algoritmul descris alege urmatoarea solutie:
$nivel_1 = \{s_1, s_2, .., s_{n-3}\}$ deoarece toate aceste servicii cer 0 inputuri pentru a fi apelate. Pe nivelul urmator nu va fi pus serviciul s_{n-2} deoarece acesta nu ofera nici un output nou, toate outputurile sale fiind incluse in reuniunea outputurilor serviciilor de pe primul nivel. $nivel_2 = \{s_{n-1}\}$.
Pe cel de-al treilea (si ultimul nivel) va fi ales serviciul s_n care in urma apelarii returneaza la output exact instanta ceruta in C_{out}. Aceasta compozitie are dimensiune $n - 1$.
Faptul ca adaugam la un nivel toate seriviile ce pot fi apelate si ne produc outputuri noi, poate fi o slabiciune pentru cazuri similare cu cel prezentat, deoarece e posibil ca pe un nivel ulterior sa existe servicii care sa ofere la output mai multe din acele instante, dar sa nu le consideram in compozitie, deoarece deja avem instantele.

Compozitia optima Compozitia optima este urmatoarea: $nivel_1 = \{s_{n-3}\}$. Nu punem pe acest nivel celalalte servicii care pot fi apelate deoarece vedem ca exista serviciul s_{n-2} care ne ofera toate outputurile pe care serviciile $s_1, s_2, ..., s_{n-2}$. $nivel_2 = \{s_{n-1}\}$. Cel de-al treilea nivel coincide cu cel din compozitia returnata de algoritm: $nivel_3 = \{s_n\}$.
Compozitia optima are dimensiunea 3.

Concluzie Deci eroarea algoritmului poate fi de ordinul dimensiunii repository-ului. Totusi ca sa obtinem un asemenea exemplu am considerat ca un serviciu poate avea oricate instante input si output ceea ce ofera un rezultat pur teoretic, in practica fiind exagerat sa consideram un asemenea caz.
Ceea ce ar fi util de calculat este o limita de eroare a algoritmului parametrizata de numarul maxim de inputuri si outputuri ale unui serviciu si de numarul de servicii din repository, deoarece in practica serviciile nu au foarte multe instante de input si output.

5 Rezultate pe teste

Pentru a evalua calitatea algoritmului implementat pentru compozitie de servicii web semantice am testat algoritmul pe un set de teste publice care au fost folosite in competitia WSC 2008. Voi prezenta rezultate algoritmului meu in comparatie cu rezultatele castigatorilor competitiei si in comparatie cu un algoritm eficient care obtine solutia optima.

5.1 WSC 08 rezultate

Din analiza rezultatelor se poate observa ca algoritmul prezentat ar fi castigat competitia cu un scor de 48 de puncte, fata de 46 de puncte cat a obtinut locul I din competitie.

Sistemele de compozitie ale concurentilor au fost testate pe 3 seturi de teste. Fiecare sistem poate obtine pana la 18 puncte pe fiecare set de teste. Limita de timp pentru fiecare sistem pentru a rezolva toate testele a fost de 15 minute. Regulile de oferire a punctelor pentru fiecare set de teste au fost urmatoarele:

- +6 Puncte pentru gasirea compozitiei cu numar minim de servicii care rezolva testul.
- +6 Puncte pentru gasirea compozitiei cu execution path minim care rezolva testul.
- Puncte suplimentare pentru:
    1. +6 Puncte pentru sistemul de compozitie care gaseste solutia cu numar minim de servicii sau cu adancime minima in cel mai scurt timp de executie.
    2. +4 Puncte pentru sistemul de compozitie care rezolva testul in al 2-lea cel mai scurt timp de executie.
    3. +2 Puncte pentru sistemul de compozitie care rezolva testul in al 3-lea cel mai scurt timp de executie.

Masurarea timpului de executie incepe dupa ce fisierele ce contin serviciile din repository si taxonomia au fost incarcate si se termina cand algoritmul ofera solutia [2].

La competitie au participat opt echipe din intreaga lume. Rezultatele primelor trei echipe clasate sunt prezentate in tabelul ce urmeaza [2] impreuna cu rezultate obtinute de algortimul prezentat pe testele folosite pentru jurizare. Punctele suplimentare acordate pentru viteza de executie a algoritmului le-am acordat comparand timpii de rulare ai algoritmului cu timpii prezentati de [2] ai celor trei sisteme castigatoare, dar nu am modificat acordarea bonusului echipelor care au participat in concurs. In mod normal pentru fiecare set de teste trebuie sa existe exact un singur punctaj suplimentar cu fiecare valoare din multimea {2, 4, 6}.

	Tsinghua University		University of Groningen		Pennsylvania State University		UAIC	
	Rez	Pct	Rez	Pct	Rez	Pct	Rez	Pct
Set teste 1								
Nr. min. servicii	10	+6	10	+6	10	+6	10	+6
Adancime min.	5	+6	5	+6	5	+6	5	+6
Timp (ms)	312	+4	219	+6	28078		34	+6
Set teste 2								
Nr. min. servicii	20	+6	20	+6	20	+6	20	+6
Adancime min.	8	+6	10		8	+6	8	+6
Timp (ms)	250	+6	14734	+4	726078	+6	86	+6
Set teste 3								
Nr. min. servicii	46		37	+6	Fara rezultat		45	
Adancime min.	7	+6	17				7	+6
Timp (ms)	406	+6	241672	+4			132	+6
Punctaj final	46 Puncte		38 Puncte		24 Puncte		48 Puncte	

Se poate observa ca sistemul implementat ar fi obtinut valoarea maxima ce s-ar putea acorda ca puncte suplimentare pentru viteza algoritmului si de asemenea ar fi castigat competitia.

5.2 Comparatie cu un algoritm ce obtine solutie optima

Pentru a putea analiza mai bine rezultatele algoritmului prezentat pe setul public de teste am considerat necesara o comparatie cu un algoritm [3] care garanteaza ca obtine o solutie optima (in sensul definit anterior) pentru fiecare test. In tabelul urmator sunt prezentate comparativ rezultatele pe teste obtinute de algoritmul prezentat si de [3]. Acesta din urma este un algoritm euristic de compozitie automata a serviciilor web prezentat la conferinta 2011 IEEE International Conference on Web Service care abordeaza problema ca o problema de cautare pe grafuri, implementand o metoda euristica bazata pe algoritmul A* care gaseste o compozitie optima. Desi sunt folosite optimizari pentru a reduce dimensiunea grafului si pentru a reduce dinamic posibilele drumuri de explorare in timpul cautarii, filtrand compozitii echivalente, se poate observa ca pe testele de dimensiune mai mare, algoritmul merge de peste 50 de ori mai incet decat algoritmul prezentat in aceasta lucrare.

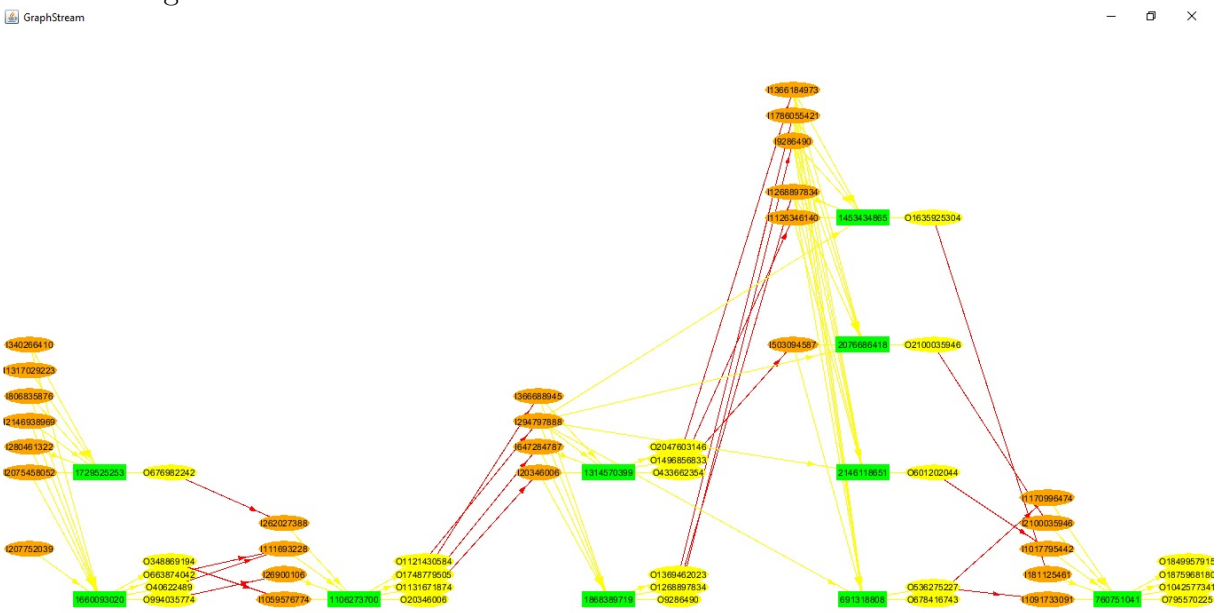
Nr. test	Algoritmul meu			Algoritmul		
	Nr. serv	Exec. path	Timp(ms)	Nr. serv	Exec. path	Timp(ms)
1.	10	3	22	10	3	91
2.	5	3	27	5	3	123
3.	40	23	76	40	23	1929
4.	10	5	34	10	5	314
5.	20	8	87	20	8	6356
6.	45	7	132	42	7	777
7.	20	12	95	20	12	9835
8.	30	20	109	30	20	6398

Se poate observa ca pe 7 din 8 teste algoritmul meu obtine solutie optima, atat din punct de vedere al numarului de servicii, cat si din punct de vedere al execution path. Pe un singur test algoritmul meu obtine 45 de servicii, cand solutia optima are 42 de servicii. Execution path obtinuta este minima pe toate testele, deoarece algoritmul obtine mereu aceasta valoare minima.

Concluzie In concluzie putem vedea ca algoritmul a obtinut rezultate foarte bune pe teste, reusind pentru toate testele sa obtina execution path minim, dupa cum era de asteptat si pe 7 din 8 teste a obtinut si numar minim de servicii. In ceea ce priveste timpul de executie, se observa pe testele mari ca este eficient, producand o solutie buna chiar si de mai mult decat 50 de ori mai rapid decat un algoritm euristic ce obtine solutia optima. Desi se pot construi exemple pe care algortimul sa greseasca oricat de mult, acesta se comporta foarte bine pe teste. Aceste teste au fost generate astfel incat sa fie foarte similare cu situatiile reale din arhitecturi orientate pe servicii din industrie [2].

6 Vizualizare

Pentru o analiza mai buna a compozitiei obtinute, o reprezentare grafica a acesteia sub forma unui graf este afisata odata cu crearea fisierului cu solutia finala.



Serviciile sunt reprezentate prin noduri in forma de dreptunghiuri in culoarea verde. Acestea sunt asezate pe nivele, conform cu solutia obtinuta in urma rularii algoritmului.
Instantele sunt reprezentate prin noduri in forma de elipse. Instantele input sunt colorate cu portocaliu, iar cele output sunt colorate cu galben.
Pentru un serviciu au fost reprezentate doar instantele output ce sunt utile in compozitie, adica sunt folosite pentru a apela un serviciu de pe un nivel urmator.
Semnificatia arelor este urmatoare:

- De la o instanta i la un serviciu s :instanta i este input al serviciului s.
- De la un serviciu s la o instanta i: instanta i este output al serviciului s
- De la o instanta i_1 la o instanta i_2 : i_1 este instanta output a unui serviciu s_1 sau este din C_{in}, i_2 este instanta input a unui serviciu s_2 sau este din C_out, s_1 este pe un nivel mai mic ca $s2$ si $subsumes(i_2, i_1)$ e adevarat, adica instanta i_1 poate fi folosita ca input in locul instaintei i_2 si i_1 poate fi returnata in locul lui i_2 pentru instantele cerute (cele din C_{out})

Arcele dintre instante sunt trasate doar de la stanga la dreapta, deoarece prin aceasta reprezentare dorim sa vedem cat de "util" este un serviciu in aceasta compozitie, care instate output sunt cat mai folosite pentru a apela alte servicii. Reprezentarea arcelor de la dreapta la stanga nu ar face decat sa ingreuneze desenul, fara sa dea vreo informatie relevanta despre serviciile din solutie.
Pentru aceasta reprezentare am folosit biblioteca Java GraphStream.

7 Concluzii

Algoritmul polinomial bazat pe grafuri pentru compozitia serviciilor web semantice rezolva o problema ce a devenit de actualitate odata cu cresterea numarului de servicii web, algoritmul gasind intr-un timp foarte scurt o compozitie care are numar minim de pasi de executie (execution path) si un numar mic de servicii web.

Algoritmul se comporta foarte bine pe teste ce au fost folosite in competitii si au fost generate astfel incat sa fie foarte similare cu situatiile reale din arhitecturi orientate pe servicii din industrie. Viteza algoritmului se datoreaza si anumitor optimizari, cum ar fi liniarizarea arborelui de concepte si aflarea valorii functiei $subsumes(a, b)$ in timp constant, folosirea unor structuri de date care sa mentina legatura dintre instante si servicii, garantarea parcurgerii conceptelor din taxonomie maxim o data per gasirea unei compozitii s.a.

Directii de viitor: Se doreste a se incerca extinderea relatiilor dintre concepte, astfel incat sa nu poata fi reprezentata doar relatia de particularizare. Se vor depune eforturi in gasirea unei limite maxime de eroare a algoritmului in functie de numarul de servicii din repository si de numarul maxim de instante input si output al unui serviciu.

8 Bibliografie

Bibliografie

[1] P. Diac, *Engineering Polynomial-Time Solution for Automatic Web Service Composition.*

[2] K. G. Steffen Bleul, Thomas Weise, *Proceedings of the Workshop on Service-Oriented Computing.* KIVS, 2009.

[3] M. L. Pablo Rodriguez-Mier, Manuel Mucientes, *Automatic web service composition with heuristic-based search algorithm.* IEEE International Conference on Web Services, 2011.