UNIVERSITY OF INFORMATION AND TECHNOLOGY

Khoá luận tốt nghiệp 1-2015

hoàn thành bởi

Chương Đặng - 10520010

Duy Nguyễn - 10520011

với sự hướng dẫn của

TS. Nguyễn Anh Tuấn

Đề tài khóa luận
tốt nghiệp - sinh viên khóa MMTT2010

thuộc
Chuyên ngành phát triển ứng dụng Web và di động
Khoa mạng máy tính và truyền thông

Ngày 1 tháng 1 năm 2015

Mục lục

Danh sách hình vẽ

Abbreviations

KB	**K**nowledge **B**ase
KR	**K**nowledge **R**epresentation
DL	**D**escription **L**ogic
MUPS	**M**inimal **U**nsatisfiability **P**reserving **S**ub-TBoxes
HST	**H**itting **S**et **T**ree
HS	**H**itting **S**et

Symbols

$\models$	models of	có nghĩa trong (KB)
$\not\models$	not a model of	không có nghĩa trong KB
$\subseteq$	is a subset of	là tập con của
$\cap$	intersect	giao với
$\neg\ A$	complement of A of	không phải A
$\exists\ R.E$	e.g *has* some E	
$\forall\ R.E$	e.g *has* only E	
$\equiv$	is equivalent to	tương đương với
$\emptyset$	empty set	tập hợp rỗng
$\in$	is member of	thuộc
$\Leftarrow$	preferred for left implication	
$\setminus$	except	ngoại trừ

For/Dedicated to/To my...

Chương 1

Giới thiệu về Semantic Web và Open World Assumption

Trước khi bắt đầu giới thiệu với sâu hơn về Ontology Web Language (OWL), chúng em xin được giới thiệu qua về giả định Thế Giới Mở (Open World Assumption - OWA) được Semantic Web chấp nhận và phân biệt giả định này với giả định Thế Giới Đóng (Closed World Assumption - CWA).

Closed World Assumption Giả định Thế Giới Đóng (CWA) là giả định mà những điều không chắc hoặc không có cơ sở để chứng minh là **đúng** sẽ được chấp nhận là **sai**.

Open World Assumption Giả định Thế Giới Mở (OWA) thì ngược lại, với những điều không chắc hoặc không có cơ sở để chứng minh là **đúng** sẽ được chấp nhận là **chưa biết**.

Ví dụ Xem xét một câu nói sau đây: "A là một công dân của nước Mỹ". Nếu có ai đó hỏi "A có phải là một công dân của Việt Nam hay không ?". Xét theo CWA, câu trả lời là *không*, ngược lại với OWA thì câu trả lời là *chưa biết*.

Chương 2

Các nguyên nhân dẫn đến tính thiếu nhất quán trong ontology

2.1 Các định nghĩa cần lưu ý[3]

Unsatisfiable Class/Concept dùng để chỉ một lớp hay một khái niệm trong một ontology mà ngữ nghĩa xung đột với ngữ nghĩa khác được nêu ra trong ontology hay có thể nói là các phát biểu về lớp hay khái niệm này mâu thuẫn với nhau hoặc mâu thuẫn với những phát biểu khác trong ontology.

Ví dụ `Cow`

```
        SubClassOf: Vegetarian
Vegetarian
     SubClassOf: Animal and eats only Plant
DisjointClasses:
     Plant, Animal
```

Giải thích Trong ví dụ trên thì MadCow chính là một lớp không hợp lý do trong các phát biểu logic của nó mâu thuẫn với nhau Cow là lớp con của Vegeterian mà Vegeterian chỉ ăn Plant (nghĩa là ngoài Plant, Vegeterian không ăn thứ gi khác) trong khi đó khai báo của lớp MadCow là lớp con của Cow và ăn một số Sheep (Sheep là một lớp con của Animal).

Từ đó việc lý luận có thể đưa ra giả định sai là Sheep cũng có khả năng là một phần của Plant . Điểm quan trọng là Plant và Animal là 2 DisjointClasses, nói cách khác không tồn tại một cá thể nào vừa thuộc lớp Plant và vừa thuộc lớp Animal. Như vậy trong tất cả các phát biểu logic ở ví dụ trên đã có 2 phát biểu

gây mâu thuẫn chính là eats only Plant và eats some Sheep, và chúng làm cho lớp MadCow trở nên bất hợp lý (unsatisfiable).

Incoherent Ontology dùng để chỉ một *ontology/model* có ý nghĩa không mạch lạc rõ ràng do nó có chứa ít nhất một *Unsatisfiable Class/Concept* và với điều kiện là trong những *Unsatisfiable Class* này không được chứa bất kì một cá thể (*Individual*) nào.
Giả sử ta có ontology A chứa các các phát biểu trong ví dụ trên ngoại trừ phát biểu cuối cùng `Individual: Dora type: MadCow` thì ta có thể nói ontology A không mạch lạc rõ ràng do nó chứa unsatisfiable class là MadCow. Chúng ta vẫn có sử dụng được ontology A vì nó vẫn còn tính nhất quán (*Consistency*)miễn là không có phần tử nào thuộc lớp MadCow.

Inconsistent Ontology dùng để chỉ một ontology chứa ít nhất một *Unsatisfiable Class* và có ít nhất một cá thể(*Individual*) thuộc một trong những lớp *unsatisfiable* này. Như đã thể hiện trong ví dụ đầu tiên thì cá thể Dora thuộc lớp MadCow (Một lớp unsatisfiable thì không nên phép có bất kì cá thể nào nếu như chúng ta muốn đảm bảo tính consistency cho ontology), như vậy bất kì ontology nào có những phát biểu trên đều được coi là không nhất quán (*inconsistency*), đều này đồng nghĩa là ontology đó không thể sử dụng được nữa.

2.2 Các nguyên nhân phổ biến dẫn đến tính thiếu nhất quán (Inconsistency)[4]

Các nguyên nhân dẫn đến tính thiếu nhất quán trong ontology gây bởi các lỗi được phân loại thành lỗi gây ra bởi phát biểu ở mức độ lớp (Class level - TBox), các lỗi gây ra bởi phát biểu ở mức độ cá thể (Instance/Individual level - ABox) và lỗi gây ra bởi sự kết hợp của cả 2 nguyên nhân vừa nêu trên.

2.2.1 Khởi tạo cá thể cho một Unsatisfiable Class) - (TBox + ABox)

- Khởi tạo cá thể cho một Unsatisfiable Class được xem là nguyên nhân phổ biến nhất gây ra tính thiếu nhất quán trong ontology.
- Ví dụ:

```
Individual: Dora type: MadCow
```

- Chúng ta không quan tâm đâu là nguyên nhân làm cho `MadCow` trở nên mâu thuẫn, chỉ cần biết là một Unsatisfiable Class thì không nên có bất kì cá thể nào trong đó. Rõ ràng là không có bất kì ontology nào mà cá thể Dora có thể đáp ứng các điều kiện như trong ví dụ đầu tiên, nói cách khác không tồn tại model nào có thể thỏa được điều kiện trên. Chúng ta phát biểu đó là một ontology không nhất quán.

2.2.2 Khởi tạo cá thể thuộc 2 class được disjoint với nhau (TBox + ABox)

- Đây là một trường hợp dễ bắt gặp vì nó sai ngay trong phát biểu về logic.
- Ví dụ

```
Individual: Dora
Types: Vegetarian, Carnivore
DisjointClasses: Vegetarian, Carnivore
```

- Lớp A disjoint với B khi và chỉ chi lớp A không có chung bất kì một phần tử/cá thể nào với lớp B. Phát biểu Disjoint Classes(A B C) có nghĩa là mỗi lớp trong đó disjoint với từng lớp còn lại (mutually disjoint). Phát biểu ABox dạng `DisjointClasses(Vegetarian Carnivore)` là sai vì Dora vừa thuộc Vegetarian vừa thuộc Carnivore dựa vào phát biểu `Individual: Dora Types: Vegetarian, Carnivore`.

2.2.3 Các phát biểu ABox xung đột với nhau

- Trường hợp này thì tương tự như nguyên nhân ở trên nhưng khác ở chỗ là lần này sự mâu thuẫn nằm trong các biểu ở cấp độ cá thể (ABox).
- Ví dụ:

```
Individual: Dora
   Types: Vegetarian, not Vegetarian
```

- Dễ thấy được sự mâu thuẫn trong trong phát biểu trên vừa yêu cầu Dora là Vegetarian vừa yêu cầu nó không phải Vegetarian.

2.2.4 Phát biểu xung đột với nghĩa "oneOf" (All TBox)

- Phát biểu bao gồm hoặc một trong (oneOf trong cú pháp của OWL) cho phép sử dụng các cá thể trong khai báo phát biểu ABox, sự kết hợp này có thể dẫn đến sự thiếu nhất quán.

- Lấy ví dụ sau:

```
Class: MyFavouriteCow
   EquivalentTo: {Dora}
Class: AllMyCows
   EquivalentTo: {Dora, Daisy, Patty}
DisjointClasses: MyFavouriteCow, AllMyCows
```

- Phần đầu tiên của các phát biểu trên tất cả các thể thuộc lớp `MyFavouriteCow` phải tương đương với cá thể tên Dora, nói cách khác là `SameIndividual` với `Dora`. Phần thứ hai cũng tương tự tất cả các cá thể thuộc lớp `AllMyCows` buộc phải tương đương với một trong 3 cá thể tên `Dora`, `Daisy` hoặc `Patty`. Do 2 phát biểu trên chúng ta đã nói Dora thuộc cả 2 lớp `MyFavoriteCow` và `AllMyCows` nên mâu thuẫn với phát biểu cuối cùng khi nói 2 lớp này không có chung một cá thể nào. Vì vậy dẫn tới ontology bị thiếu nhất quán (inconsistent).

2.2.5 Không có khả năng khởi tạo bất kì cá thể nào (all TBox)

- Ví dụ:

```
Vegetarian or not Vegetarian
SubClassOf: Cow and not Cow
```

- Đây chỉ là một ví dụ đơn giản để minh họa cho trường hợp này. Thực tế sẽ ít người dùng nào tạo ra một phát biểu ngớ ngẩn như vậy nhưng nó vẫn có khả năng xảy ra khi phát biểu trên là kết quả từ suy luận (reasoning) của những phát biểu lớn và phức tạp hơn.

- Có thể giải thích ví dụ trên như sau. Đầu tiên để đáp ứng ý nghĩa dòng đầu tiên yêu cầu cá thể vừa là `Vegetarian` hoặc không phải `Vegetarian` - bất kỳ phát biểu nào dạng này, "cá thể thuộc hoặc không thuộc một lớp" chính là tất cả cá thể xuất hiện trong ontology. Dòng thứ hai yêu cầu cá thể vừa là Cow vừa không phải là Cow, phát biểu này rơi vào một trong các nguyên nhân vừa nêu ở trên. Tổng hợp lại chúng yêu cầu tất cả cá thể vừa là Cow vừa không phải Cow, điều này gây ra mâu thuẫn trên toàn ontology do phát biểu đầu tiên chỉ tới tất cả các cá thể.

Kết luận Trên đây chúng em đã liệt kê những nguyên nhân phổ biến dẫn đến thiếu nhất quán qua những ví dụ đã được đơn giản hoá để dễ dàng nắm bắt được đâu là căn nguyên gây ra sự mâu thuẫn về logic. Trên thực tế với những ontology có số lượng phát biểu lớn và phức tạp rất khó để người dùng có thể nhận diện được đâu là nguyên nhân

chính xác gây ra mâu thuẫn, do vậy sự ra đời của một công cụ giúp chúng ta phát hiện chính xác nguyên nhân gây lỗi là rất cần thiết. Vì vậy trong nội dung chương 2, chúng em sẽ đề cập tới Ontology Debugging một khía cạnh rất được chú trọng khi số lượng phát biểu của ontology ngày càng tăng.

Chương 3

Giải pháp để sửa chữa inconsistent ontology

- Như đã được đề cập trong chương 1, trong các nguyên nhân dẫn đến tính thiếu nhất quán (*Inconsistency*) trong ontology thì **Unsatisfiable Class** (lớp không thỏa về tính logic) là nguyên nhân nếu có thể được phát hiện sớm để loại bỏ hoặc sửa lại các phát biểu gây mâu thuẫn thì giúp cho ontology tránh bị inconsistent.
- Đã có rất nhiều nghiên cứu thành công trong việc tìm và phát hiện lỗi (các phát biểu mâu thuẫn) trong ontology. Trong đó có một nghiên cứu nổi bật[1], không chỉ có khả năng phát hiện gần như chính xác các nguyên nhân gây lỗi mà còn được đưa ra các giải pháp tối ưu* để sửa lỗi. Nghiên cứu này đã được ứng dụng để đưa ra các giải thích về các lớp không thỏa về nghĩa (unsatisfiable classes) trong bộ thự viện lập trình ontology thông dụng hiện nay là OWL-API[2]. Sau đây chúng em xin được trình bày lại những điểm quan trọng trong nghiên cứu vừa được đề cập**

* *Tối ưu có nghĩa là hạn chế tối đa các thay đổi về ý nghĩa mà việc xóa hoặc thay đổi phát biểu mâu thuẫn có thể gây ra cho các phát biểu khác (other axioms) trong ontology.*

** *Mọi quan điểm và ý tưởng trình bày ở phần sau của Chương 2 đều thuộc sở hữu của các tác giả bài báo[1] và [9]. Chúng em chỉ trình bày lại sau khi đã đọc và nắm được ý tưởng chính yếu của bài báo.*

3.1 Mục tiêu của việc debuging ontology

Mục tiêu chính của việc debuging ontology gồm hai phần quan trọng. Thứ nhất, với một ontology có số lượng lớn các lớp unsatisfiable, cần tìm và nhận dạng được nguyên nhân gây ra mâu thuẫn và các lớp bị ảnh hưởng bởi sự mâu thuẫn đó trong ontology. Thứ hai, cho biết trước một Unsatisfiable Class cụ thể, trích xuất và trình bày cho người sử dụng ontology(*modeler*) một tập hợp tối tiểu các phát biểu (*minimal set of axioms*) từ ontology hay nguyên nhân chính xác chịu trách nghiệm trong việc gây ra sự mâu thuẫn về logic.

3.2 Khái niệm và các kỹ thuật cần biết

Các hệ thống Description Logic thường cung cấp một tập hợp các tác vụ suy luận đã được chuẩn hóa như phân loại các khái niệm (*concept classification*), kiểm tra tính đáp ứng về logic (*concept satisfiability*) và kiểm tra tính nhất quán của knowledge base (KB). Hầu hết các reasoner thông dụng hiện nay đều buộc phải cung cấp đủ 3 tác vụ nêu trên, nhưng tất cả chúng đều không thân thiện với người dùng. Do tất cả những gì chúng ta biết được đều là kết quả (hay output) từ sự suy luận(reasoning) của reasoner.

Để giúp cho các tác vụ suy luận (reasoning) trở nên thân thiện với người dùng hơn, một hệ thống DL-based Knowledge Representation (KR) phải mở rộng thêm các lựa chọn về các tác vụ không nằm trong tiêu chuẩn của DL. Một ví dụ cụ thể là việc tạo ra các giải thích tại sao một lớp lại bị reasoner đánh giá là unsatisfiable. Thêm một tình huống mà người dùng cần được giải thích là tại sao reasoner đánh giá một lớp là lớp con của một lớp khác - đâu là lý do. Việc ra đời tác vụ giải thích nguyên nhân và kết quả là thật sự cần thiết trong bối cảnh sự phát triển nhanh của Semantic Web và cộng đồng người dùng/nhà phát triển Ontology ngày càng tăng nhanh.

3.2.1 Dịch vụ Axiom Pinpointing

Axiom Pinpointing service chính là dịch vụ có khả năng thực hiện tác vụ giải thích vừa được đề cập, với một KB và bất kì kết quả suy luận nào từ KB, dịch vụ này sẽ trả về tập các chứng minh/giải thích cho suy luận đó bằng những phát biểu đã được khai báo trong KB.

Có thể giải thích ngắn gọn như sau, cho một phát biểu kết quả họ SHOIN α được suy ra từ một knowledge base K, một kiểm chứng (justification) cho α trong K là một phần tối tiểu $K' \subseteq K$ chịu trách nhiệm cho α xảy ra. Kiểm chứng K' là tối tiểu với điều

kiện α là một kết quả logic được suy ra từ K', hay nói cách khác K' tối tiểu khi và chỉ khi bất kì tập con nào của K' đều không suy ra được α. Nói chung có thể tồn tại nhiều giải thích/chứng minh cho α trong K.
Sau đây là một ví dụ cho ý tưởng vừa nêu. Cho KB K với các phát biểu như sau:

1. $A \subseteq B \cap C$
2. $B \subseteq \neg\ E$
3. $A \subseteq D \cap \exists\ R.E$
4. $D \subseteq C \cap \forall\ R.B$

Trong KB trên, A, B, C, D, E là atomic concepts và R là atomic role. Chúng ta sẽ dùng số thứ tự của từng câu phát biểu trên thay vì lặp lại nguyên văn.

Từ các phát biểu trên ta có $K \models (A \subseteq C)$. Tuy nhiên, điều kiện cần và đủ để suy ra được một kết quả tương tự từ 2 phần nhỏ hơn của KB K là $K_1 = 1$ và $K_2 =$3,4. Chúng ta nói K_1 và K_2 là các kiểm chứng cho kết luận nói C là tập con của A - $A \subseteq C$.

KB trong ví dụ vừa nêu được xem là khá nhỏ, qua đó dễ dàng nhận ra lợi ích đáng kể khi số lượng phát biểu trong KB tăng lên vài trăm hay vài ngàn phát biểu. Bằng cách nhận dạng chính xác các tập tối tiểu chứa các phát biểu khẳng định (asserted) là những giả thiết cho kết quả được suy ra, dịch vụ này có thể được dùng để cô lập, đánh dấu và giải thích nguyên nhân hoặc cơ sở của các kết quả suy luận. Điều này cực kì quang trọng trên khía cạnh debugging, lấy ví dụ trường hợp cần giải thích là một Unsatisfiable Class/Concept, dịch vụ này sẽ khám phá tất cả và chỉ những phát biểu là nguyên nhân gây lỗi. Trong trường hợp vừa nêu, tìm kiếm tất cả các kiểm chứng là rất cần thiết vì để sửa lại unsatisfiable class cần loại bỏ ít nhất một phát biểu trong tập các phát biểu tối tiểu nguyên nhân gây lỗi MUPS - sẽ được đề cập trong mục bên dưới.

Tuy nhiên, dịch vụ axiom pinpointing chúng ta đề cập có một giới hạn là nó chỉ làm việc ở mức độ giữa các phát biểu với nhau, chúng vẫn chưa phân biệt được phần cụ thể nào của phát biểu mới là nguyên nhân cần và đủ để giải thích cho kết quả suy luận. Lấy lại ví dụ vừa nếu trên KB K, lớp B trong giao của $B \cap C$ trong phát biểu 1, không phải là giải thiết cần để suy ra $A \subseteq C$. Tương tự, $\exists\ R.E$ và $\forall\ R.B$ trong phát biểu 3 và 4 không phải điều kiện cần để suy ra được $A \subseteq C$.
Do vậy, việc quan tâm xem phần nào của phát biểu mới chính là giả thiết/nguyên nhân của kết quả suy luận rất quan trọng trong nhiều trường hợp, đặc biệt khi sửa chữa một phát biểu gây lỗi thì việc sửa lại một phần của phát biểu sẽ hạn chế sự mất mát về ý nghĩa của ontology hơn là xóa nó đi.

Để đáp ứng yêu cầu này, họ để định nghĩa một *hàm chia nhỏ KB*. Ý tưởng là viết lại một phát biểu bất kì trong KB thành những dạng tập gồm các phát biểu nhỏ và

đơn giản hơn với ý nghĩa tương đương. Sau đó, sử dụng *Axiom Pinpointing Service* lên những tập những phát biểu trong KB K_s đã được viết lại từ K để tìm kiếm nguyên nhân hay giải thích cho kết quả suy luận.
Lấy phát biểu 1 trong ví dụ trên:

$$A \subseteq B \cap C \text{ (1) được viết lại thành } A \subseteq B,\ A \subseteq C\ (1^*)$$

Dễ dàng thấy phần $A \subseteq C$ trong 1^* chính là điều phải chứng minh cho $K \models (A \subseteq C)$, những phần còn lại không cần thiết. Tương tự, ta viết lại (3) và (4) như sau:

$$A \subseteq D \cap \exists\ R.E \Leftrightarrow A \subseteq D,\ A \subseteq \exists\ R.E$$
$$D \subseteq C \cap \forall\ R.B \Leftrightarrow D \subseteq C,\ D \subseteq \exists\ R.B$$

Bây giờ, điều kiện cần và đủ để chứng minh $K \models (A \subseteq C)$ là $A \subseteq D$ và $D \subseteq C$. Tuy nhiên, trong một vài trường hợp "hàm chia nhỏ KB" này đòi hỏi phải giới thiệu ra một tên lớp mới, viết giới thiệu tên lớp mới này chỉ phục vụ cho mục đích viết lại phát biểu. Ví dụ:

$$A \subseteq \exists\ R.\ (C \cap D) \text{ không tương đương với } A \subseteq \exists\ R.C,\ A \subseteq \exists\ R.D$$

Để chia nhỏ phát biểu trên chúng ta sẽ giới thiệu một tên lớp mới, gọi là E. Như vậy ta có:

$$A \subseteq \exists\ R.\ (C \cap D) \Leftrightarrow A \subseteq \exists\ R.E,\ E \subseteq C,\ E \subseteq D,\ C \cap D \subseteq E$$

Để thực hiện được cái gọi là *"hàm chia nhỏ KB"* các tác giả bài báo [1] và [8] đã đề xuất các giải thuật với tiêu chí xác định các phát biểu chứng minh một cách đầy đủ và chính xác. Các giải thuật này có thể được chia thành 2 nhóm:

1. *Reasoner Dependent(or Glass-box) Algorithm* Đây là nhóm các giải thuật xây dựng trên quy trình đưa ra quyết định Tableau dành cho Description Logic. Tuy nhiên, để áp dụng các giải thuật loại này trong thực tế đòi hỏi phải có những chỉnh sửa đáng kể bên trong quy trình suy luận những DL reasoner hiện nay.

2. *Reasoner Independent(or Black-box) Algorithm* Nhóm này chỉ sử dụng các DL reasoner cho những tác vụ kiểm tra lại kết quả suy luận khi đã viết lại KB K thành K', chúng không đòi hỏi phải chỉnh sửa lại các cách hoạt động của reasoner. Reasoner lúc này có chức năng như một "chiếc hộp đen" chấp nhận các input là lớp/các phát biểu đã được viết lại hoặc một KB K' viết lại từ KB K, sau đó trả

về output là một câu trả lời xác nhận hay phủ định rằng các lớp và các phát biểu này có là tập tối tiểu để chứng minh cho kết quả suy luận hay không. Ví dụ trong trường hợp:

$$A \subseteq B \cap C \Leftrightarrow A \subseteq B,\ A \subseteq C$$

Các inputs của reasoner sẽ lần lượt sau mỗi vòng là

(a) $A \subseteq B,\ A \subseteq C$

(b) $A \subseteq B$

(c) $A \subseteq C$

Giải thuật sẽ lần lượt loại bỏ từng phát biểu một (sau mỗi vòng) để xem những phát biểu còn lại có đủ chứng minh $K \models (A \subseteq C)$. Đến khi nào giải thuật không tồn tại tập phát biểu nào đủ để chứng minh $K \models (A \subseteq C)$ thì sẽ dừng vòng lặp.

Kết luận: trên đây chỉ là những bước hoạt động cơ bản nhất của Axiom Pinpointing Service và Blackbox Algorithm xin đọc thêm [8]. Các giải thuật và dịch vụ này cũng đã được áp dụng trong package com.clarkparsia.owlapi.explanation của [2].

3.2.2 Minimal Unsatisfiability Preserving Sub-TBoxes (MUPS)

Khái niệm MUPS lần đầu được giới thiệu trong[6].Như đã được để cập trong phần đầu của mục này, một MUPS thật ra chính là một phần nhỏ nhất của KB K mà trong đó lý giải tại sao một lớp lại unsatisfiable, nói cách khác một MUPS là một tập tối tiểu các phát biểu mà trong đó các phát biểu này giải thích chính xác nguyên nhân gây ra mâu thuẫn về logic(unsatisfiable). Một lớp unsatisfiable có thể có nhiều MUPS trong KB K (hay cụ thể là trong ontologies). Ví dụ có KB K_α với những phát biểu như sau:

1. $S \equiv A \cap \exists\ R.B$
2. $S \subseteq \exists\ R.(C \cap D)$
3. $(C \cap D) = \emptyset$

Dựa vào các phát biểu trên ta thấy S unsatisfiable. MUPS của S từ K_α là:

$$S \subseteq \exists\ R.(C \cap D)\ (2)$$
$$(C \cap D) = \emptyset\ (3)$$

Để sửa lại một lớp không đáp ứng(*unsatisfiable class*) chúng ta cần loại bỏ tối thiểu ít nhất một phát biểu từ từng tập các phát biểu tối tiểu MUPS lý giải cho unsatisfiable class đó. Trong ví dụ vừa rồi do chỉ có 1 MUPS, ta bỏ tất cả các phát biểu trong MUPS xuất hiện trong KB K_α thì S sẽ lại *satisfiable*.

3.3 Các bước sửa chữa các phát biểu bị lỗi

3.3.1 Tìm tất cả các MUPS của một Unsatisfiable Class

Như vừa nói ở trên MUPS thật ra chính là một phần nhỏ nhất trong KB khiến cho một lớp unsatisfiable. Do vậy tìm và xác định MUPS chính là tìm và xác định các tập tối tiểu các phát biểu cho một lớp được suy luận là unsatisfiable. Chúng ta sẽ sử dụng *Axiom Pinpointing Service*[8] để tìm MUPS với các bước tương tự đã được mô tả chi tiết trong mục trên. Nhiệm vụ tìm kiếm *precise* MUPS của lớp không đáp ứng trong KB K được đơn giản hóa thành vấn đề tìm MUPS trong những phiên bản đã được tách nhỏ trong KB K_s.

3.3.2 Chiến thuật xếp hạng các phát biểu (*Axioms*)

Đây là một giai đoạn khá quan trọng trong quá trình chỉnh sửa lại các phát biểu gây lỗi, quyết định xem nên loại bỏ phát biểu nào từ các MUPS để lớp/khái niệm được satisfiable.

Với mục tiêu này, một nhân tố đáng quan tâm là các phát biểu trong MUPS có thể được *xếp hạng* dựa theo mức độ quan trọng của chúng. Việc sửa chữa các nguyên nhân gây lỗi được trở thành một vấn đề cần được tối ưu để đáp ứng các tiêu chí vừa phải loại bỏ tất cả các lỗi gây ra tính thiếu nhất quán trong ontology, trong khi vẫn chắc chắn rằng những phát biểu có thứ hạng cao, nói cách khác là có giá trị quan trọng về nghĩa sẽ được ưu tiên giữ lại và các phát biểu có thứ hạng thấp nhất sẽ bị loại bỏ.

Tiêu chí đơn giản nhất để xếp hạng các phát biểu là đếm số lần chúng xuất hiện trong MUPS từ những lớp unsatisfiable xuất hiện trong một ontology. Nếu một phát biểu xuất hiện trong n MUPS khác nhau (trong từng tập phát biểu của MUPS), bỏ đi phát biểu đó sẽ đảm bảo rằng n lớp/khái niệm được satisfiable. Số lần phát biểu xuất hiện càng nhiều, thứ hạng của nó càng thấp.

Ngoài tần suất xuất hiện của phát biểu trong MUPS, chúng ta cũng có thể quan tâm đến những yếu tố sau để đưa vào tiêu chí xếp hạng:

- Tác động lên ontology khi loại bỏ phát biểu hoặc thay đổi nội dung phát biểu - cần phải nhận diện được những tác động tối tiểu (*minimal impact*) gây ra thay đổi.
- Tự xây dựng những test cases cụ thể để xếp hạng các phát biểu dựa theo tiêu chí của người dùng tự đề ra.
- Dựa trên những metadata của phát biểu như tác giả, độ tin cậy của nguồn tài liệu, timestamp, etc.
- Sự liên quan tới ontology ở khía cạnh phát biểu được sử dùng vào mục đích gì và sử dụng như thế nào.

Lưu ý: Chi tiết về cách áp dụng từng tiêu chí xếp hạng trên xin đọc [1].

3.3.3 Tạo ra các giải pháp sửa lỗi

Qua các phần trên, chúng ta đã biết được làm thế nào để tìm MUPS cho một lợp unsatisfiable bằng Axiom Pinpointing Service trong một OWL-DL ontology và thấy được một loạt các tiêu chí để xếp hạng phát biểu trong MUPS. Bước tiếp theo là tạo ra một kế hoạch sửa lỗi (hay một loạt các thay đổi trong ontology) để sửa các lỗi trong một tập các lớp/khái niệm bị unsatisfiable, với các dữ kiện đã có qua các bước trên như các MUPS tìm được và thứ hạng các phát biểu.

Điều chỉnh giải thuật Reiter Giải thuật Hitting Set của Reiter[7], đưa ra nhằm để xác định căn nguyên(*root cause*) của một vấn đề từ một bộ(*collection*) gồm nhiều tập hợp đụng độ chứa các nguyên nhân dẫn tới vấn đề, giải thuật này sẽ tạo ra những tập tối tiểu (*minimal hitting set*) chứa các nguyên nhân gây ra vấn đề. Một tập hợp đụng độ (*hitting set*) trong một bộ **C** các tập hợp là tập hợp giao (có chung phần tử) với từng tập hợp trong **C**. Một tập hợp đụng độ là tối tiểu nếu không có bất kì tập con nào của nó lại là một tập đụng độ cho **C**. Trong trường hợp của chúng ta, bộ **C** chứa các HST chính là các MUPS tìm được trong ontology.

Ý tưởng là áp dụng giải thuật Reiter để tìm ra tập tối tiểu các phát biểu gây lỗi từ các MUPS đã tìm được, rồi loại bỏ tất cả các phát biểu trong tập đụng độ tối tiểu từ đó giúp loại bỏ từng phát biểu gây lỗi xuất hiện trong từng tập phát biểu từng MUPS và cuối cùng giúp cho sửa chữa được cho lớp/khái niệm được satisfiable. Nguyên lý tương tự cũng được áp dụng cho việc giải pháp sửa lỗi ngoại trừ cần phải điều chỉnh lại giải thuật HS để nó có thể hoạt động dựa trên thứ hạng của các phát biểu.

Cho một bộ C gồm những tập đụng độ, giải thuật Reiter giới thiệu một khái niệm về hitting set tree (HST), là một cấu trúc cây có số cạnh nhỏ nhất và số node nhỏ nhất,

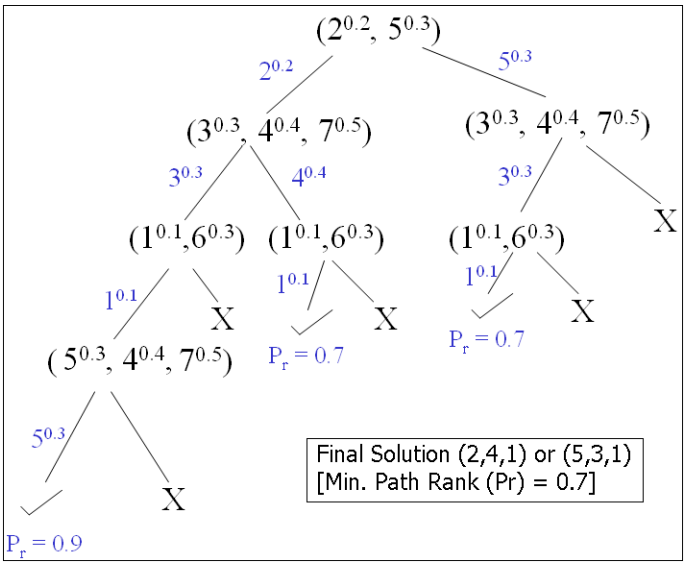
với cạnh và node đều được dán nhãn (labeled). Một node n trong HST được dán nhãn bởi dấu tick (✓) nếu C rỗng, ngược lại node này sẽ được dán nhãn bởi bất kì tập hợp $s \in C$. Với mỗi node n, ta có $H(n)$ là tập gồm các nhãn của cạnh (edge labels) trên đường đi từ gốc cây tới n (root to n); và nhãn cho n là bất kì tập $s \in C$, thỏa điều kiện $s \cap H(n) \Leftarrow \emptyset$, nếu có một tập hợp nào như vậy tồn tại. Nếu n được dán nhãn bởi một tập s, thì với từng $\sigma \in s$, n có một node kế cận là n_σ nối với n bởi một cạnh được dán nhãn bằng σ. Với bất kì node nào được dán nhãn bằng ✓, tập chứa các nhãn mô tả đường đi (theo cạnh) của node này tới gốc cây là một tập đụng độ(hitting set) của C. Khi tạo ra HST từ gốc, nếu trong quá trình tìm kiếm phát hiện được giải một giải pháp tối ưu hiện thời, thì quá trình sẽ được kết thúc sớm hơn, đánh dấu bằng một bằng dấu chéo (✗) trên nhãn của node.

Áp dụng vào trường hợp của chúng ta, MUPS của các lớp unsatisfiable tương đương với các tập hợp đụng độ. Tuy nhiên, trong giải thuật HST bình thường được tối ưu theo tiêu chí đường đi ngắn nhất, thay vì đường đi ngắn nhất chúng ta sẽ sử dụng thứ hạng nhỏ nhất (minimal path rank), nói cách khác tổng thứ hạng của các phát biểu trong $H(n)$ sẽ phải nhỏ nhất. Thêm nữa, là trong giải thuật HST cơ bản, không tồn tại khái niệm lựa chọn một phát biểu trong những phát biểu khác trong khi xây dựng cạnh của HST, trong khi chúng ta có thể sử dụng thứ hạng của các phát biểu trong lúc quyết định lựa chọn để thu hẹp không gian tìm kiếm, hay nói dễ hiểu là trong mỗi giai đoạn xây cạnh chúng ta sẽ chọn phát biểu có thứ hạng thấp nhất.

Hình 2.1 Thể hiện một HST của một collection C chứ các phát biểu từ 1 - 7 C = 2,5, 3,4,7, 1,6, 4, 5, 7, 1, 2, 3 với thứ hạng của các phát biểu từ 1 - 7 như sau: $r(1) = 0.1, r(2) = 0.2, r(3) = 0.3, r(4) = 0.4, r(5) = 0.3, r(6) = 0.3, r(7) = 0.5$, trong đó $r(x)$ là hạng của phát biểu x. Thứ hạng này được tính ra dựa trên những yếu tố được đề cập ở phần 2.3.2 như tần suất xuất hiện, tác động ngữ nghĩa, etc. mỗi tiêu chí được đánh gía riêng biệt, nếu cần chúng ta có thể quy ước một hệ số để đánh giá tất cả cùng một lúc. Số mũ trên từng phát biểu chính biểu diễn hạng của phát biểu đó, và P_r là *path rank* được tính bằng tổng hạng của các phát biểu nằm trên đường đi (theo cạnh) từ gốc tới một node. Ví dụ, cạnh cận trái nhất có *path rank*: $P_r = 0.2 + 0.3 + 0.1 + 0.3 = 0.9$.

Như được thể hiện trong hình, bằng cách chọn phát biểu có hạng thấp nhất trong từng tập trong khi xây cạnh của HST, giải thuật chỉ tạo ra 3 hitting sets, 2 trong số đó tối tiểu, trong khi hạn chế được một số lượng lớn số lần kiểm tra đường đi, (thể hiện bằng ✗). Giải pháp sửa lỗi được tìm ra trong tập có P_r nhỏ nhất là 2,4,1 hoặc 5,3,1.

Tuy vậy, có một hạn chế khi sử dụng quy trình vừa nêu trên để tạo ra kế hoạch sửa lỗi, như phân tích tác động ngữ nghĩa của phát biểu chỉ được thực hiện ở cấp độ là một phát biểu đơn lẻ, trong khi một loạt tác động khác chưa được tính tới mỗi lần một HS được tìm thấy. Điều này có thể dẫn tới một giải pháp kém tối ưu. Ví dụ:



HÌNH 3.1: Giải thuật HST được chỉnh sửa dựa theo thứ hạng của phát biểu

1. DisjointClasses(Car Plane Ship) EquivalentClass(FlyingCar (Car and Plane))
2.

Trong ví dụ trên, bỏ `Plane` ra khỏi phát biểu (1) sẽ hạn chế mất mát về nghĩa hơn là xóa hết cả phát biểu (1), vì có thể disjoint giữa Car và Ship có thể được sử dụng đâu đó trong ontology mà chưa được tính đến.

Để khắc phục hạn chế này, một chỉnh sửa khác được đưa ra là cứ mỗi lần tìm ra hitting-set(HS), chúng ta sẽ tính lại thứ hạng của đường đi (path-rank) cho HS dựa trên một loạt tác động của các phát biểu trong hitting-set. Giải thuật bây giờ sẽ tìm được giải pháp tối thiểu được path-ranks mới.
Trên đây là ý tưởng cơ bản của giải thuật HST, ngoài ta còn những mục về cải thiện các giải pháp sửa lỗi và gợi ý các phát biểu sửa lỗi xin đọc thêm ở [1].

3.4 Ứng dụng HST để xác định tất cả kiểm chứng cho kết quả suy luận[9]

Ngoài ứng dụng vừa được đề cập ở trên, Hitting Set Tree còn được áp dụng để tìm tất cả các giải thích cho một kết quả suy luận, giống như một trong chức năng chính của Axiom Pinpointing Service được đề cập lúc nãy. Tuy nhiên, không giống như khái niệm HST của Reiter vừa được giới thiệu ở trên, ở đây chúng ta sẽ có HST với quy ước như sau.

Với ontology $\beta \models \eta$, một cây hitting set (HST) cho η trong β là một cây hữu hạn, bao gồm node được dán nhãn bằng các kiểm chứng(justifications) hay các phát biểu

chứng minh $\beta \models \eta$ và cạnh được đánh dấu với phát biểu trong β. Từng non-leaf(không phải lá) node v nối với một node kế cận v' qua một cạnh được dán nhãn với một phát biểu α với α nằm trong nhãn của v, nhưng không nằm trong nhãn của v'. Nhãn của v' có thể là một tập hợp rỗng, trong trường hợp đó v' phải là node lá (leaf node). Ngoài ra, với bất kì node v'', tập chứa các phát biểu dán nhãn cho đường đi từ v'' tới gốc cây (tree root) *không* giao với kiếm chứng(hay các phát biểu chứng minh) dán nhãn v''.

Quá trình xây dựng HST có thể được thực hiện bằng các giải thuật *breadth first* hay *depth first*. Dù được sử dụng theo cách nào, các nguyên lý và các luật để tạo và dán nhãn cạnh, node trên cây đều như nhau. Khi mở rộng cây từ node v đến một node mới v' quy trình cơ bản đều diễn ra như sau:

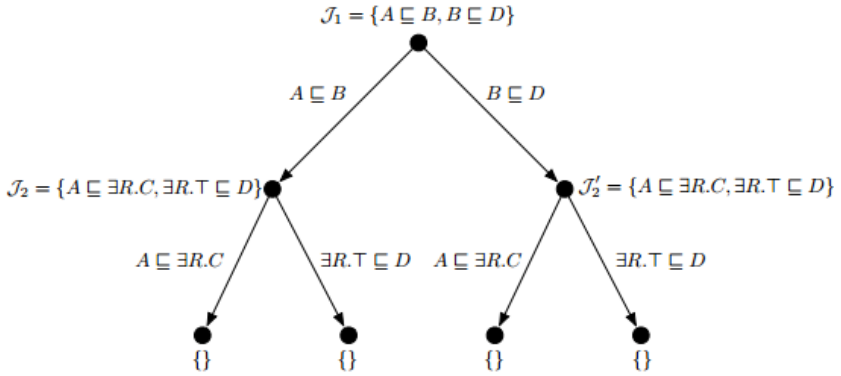
1. Chọn một phát biểu α nằm trong nhãn của v nhưng không dán nhãn một cạnh nối v tới bất kì node kế cận nào.
2. Gọi S là hội của (union of) α và tập những phát biểu nằm trên cạnh, tạo thành đường đi từ v tới root node. Loại bỏ S khỏi β ta được β'.
3. Nếu η thỏa $\beta' \models \eta$ thì tìm một kiểm chứng (justification) J cho η trong β'. Nếu $\beta' \not\models \eta$ thì gán $J = \emptyset$.
4. Tạo một node mới v' và dán nhãn cho v' bằng tập J ở bước trên.
5. Tiếp tục mở rộng HST theo một hướng bằng cạnh $e = (v, v')$ tương tự như bước 1 cho tới khi không tìm được tập $J = \emptyset$ như ở bước 2.
6. Đưa các phát biểu trong S trở lại β.

Ví dụ sau mô tả lại quy trình trên, cho ontology β với các phát biểu sau:

1. $A \subseteq B$
2. $B \subseteq D$
3. $A \subseteq \exists\, R.C$
4. $\exists\, R.\top \subseteq D$

Trong đó $\eta = A \subseteq D$. HST cho $\beta \models A$ *subseteq* D được thể hiện ở hình 2.2. Bắt đầu di chuyển tại root node, node được dán nhãn bởi ${J_1}^*$, mở rộng HST về phía bên trái bằng cách chọn phát biểu $A \subseteq B$ trong J_1 với điều kiện $A \subseteq B$ chưa dán nhãn bất kì cạnh nào nối với root node sau đó loại bỏ phát biểu $A \subseteq B$ khỏi β và tính toán lại kiếm chứng (justifications) thỏa $\langle \beta \setminus \{A \subseteq B\} \rangle \models A \subseteq D$. Trong trường hợp này, chúng ta tìm được kiểm chứng J_2 trong $\beta \setminus \{A \subseteq B\}$, giải thích được tại sao $A \subseteq D$. Tiếp tục đi

về phía bên trái của node J_2, chọn $A \subseteq \exists\ R.C$ trong J_2 tương tự cách chọn ở bước 1. Sau đó tìm kiếm các kiểm chứng trong $\beta' \equiv \beta \setminus \{A \subseteq \exists\ R.C,\ A \subseteq B\}$, kết quả là chúng ta không tìm được kiểm chứng trong β' giải thích cho $A \subseteq D$ hay có thể nói là $\beta' \not\models (A \subseteq D)$, do vậy node kế được dán nhãn bằng $\emptyset$ vì lúc này $J = \emptyset$ (bước 3). Mỗi lần như vậy ta tìm ra leaf-node, chúng ta sẽ đưa S trở lại β và bắt đầu lại quá trình tìm kiếm.



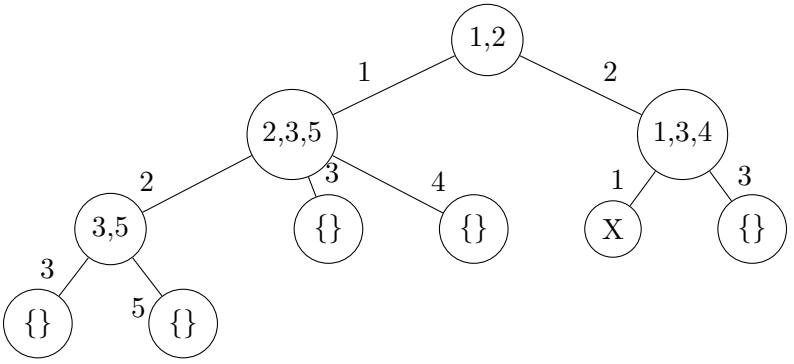
Hình 3.2: Hitting Set Tree dùng để tìm kiếm kiểm chứng

Quá trình lặp lại cho đến khi chúng ta không còn tạo được node mới nào, tại thời điểm này thì việc xây dựng HST cũng hoàn tất. *Tất cả* các kiểm chứng để giả thích cho $\beta \models \eta$ chính là nhãn của các node trong HST. Thêm một điều nữa, là tất cả các đặc điểm đơn giản nhất để chứng minh $\beta \models \eta$ nằm trên các cạnh từ leaf-node tới root-node của cây. Ví dụ vừa nếu trên chỉ biểu diễn những bước cơ bản nhất để xây dựng một HST nhưng quan tâm đến bất kì khả năng tối ưu hóa nào cho giải thuật. Để đạt được một hiệu năng chấp nhận được khi ứng dụng trong thực tế thì 2 giải pháp tối ưu sẽ được nêu ra sau đây.

Early Path Termination - Trong phiên bản chưa tối ưu ở trên, một node n bất kì khi tạo cạnh mới với phát biểu thuộc tập $H(n)$, với $H(n)$ là tập phát biểu dán nhãn n, phát biểu trên cạnh mới này không được nằm trên bất kì cạnh nào nối n với một node kế cận (successor nodes). Chúng ta gọi các nodes có khả năng mở rộng (tạo được cạnh mới) là *open* nodes, ngược lại các leaf-nodes không có khả năng mở rộng là *closed* nodes,. Để thực hiện tối ưu hóa, sẽ có trường hợp mà những nodes không phải leaf-nodes có thể được dán nhãn bởi những tập khác $\emptyset$ những vẫn sẽ được đánh dấu là *closed* nodes. Trong tình huống này, đường đi từ *closed* node tới root được chỉ định là *early termination* - kết thúc sớm. Để phát hiện được *early termination* chúng ta sẽ làm theo quy trình sau: Nếu một HST T chứa một open node v_1, có đường đi P_1 tới root node, có thêm một open node v_2 cũng trong T, có đường tới root node là P_2. Nếu tập dán nhãn cho P_1 bằng với tập dán nhãn cho P_2 thì v_2 sẽ được đánh dấu là *closed* node và không cần thiết phải mở

* J_1,J_2,J_2' *được tính ra nhờ giải thuật Black-box hoặc Glass-box được đề cập ở trên.*

rộng thêm nữa. Ví dụ chúng ta có ontology O chứa các phát biểu sau $O = \{1, 2, 3, 4, 5\}$ và $O \models \alpha$ Chúng ta bắt đầu di chuyển root node với tập $\{1, 2\}$ là các phát biểu đầu tiên



Hình 3.3: Early Termination trong HST Explanation

tìm được trong O chứng minh được $O \models \alpha$, thực hiện tương tự các bước đã được miêu tả ở trên ta thu được node $\{3, 5\}$ là các phát biểu giải thích cho $O \models \alpha$, tới lúc này ta có thể thấy tập chứ đường đi theo cạnh từ node $\{3, 5\}$ tới root node là $P_1 = \{2, 1\}$. Nhìn về phía bên phải ta phát hiện khi mở rộng cạnh từ node $\{1, 3, 4\}$ ta thu được đường đi về root node là $P_2 = \{1, 2\}$, ta thấy $P_1 \equiv P_2$ do vậy nên khi dán nhãn cho node kế tiếp (được đánh dâu ✗cho *closed* node) chúng ta sẽ bỏ $\{1, 2\}$ khỏi O để được $O^{'} = \{3, 4, 5\}$, sẽ có một node giống y như node $\{3, 5\}$ sẽ xuất hiện lần nữa ở node kế tiếp này nên việc kết thúc ở đây là cần thiết vì chúng ta sẽ tiếp kiệm được việc kiểm tra lại $\{3, 5\}$ như ở bên trái.

Justification Reuse - Cách quan trọng thứ hai để tối ưu là sử dụng lại các kiểm chứng. Trong phiên bản không tối ưu sử dụng ở ví dụ ontology β ở trên, kiểm chứng được tìm ra nhờ các giải thuật Blackbox hay Glassbox cho từng node v được thêm vào cây. Kiểm chứng hay tập các phát biểu được sử dụng để dán nhãn v, được tính toán dựa trên $O \backslash S$, với S là tập các nhãn trên đường đi từ v về root node. Thay vì dùng Glassbox hay Blackbox để tính J trong $O \backslash S$, chúng ta có thể làm theo cách sau: nếu HST chứa vài node khác $v^{'}$ mà được dán nhãn với kiểm chứng J, và S không giao (có phần tử chung) với J thì J có thể được sử dụng làm nhãn cho v. Lý do là vì khi $J \subseteq O$ và $S \cap J = \emptyset$ thì sẽ tồn tại trường hợp $J \subseteq O \backslash S$, từ đó J được tính như một kiểm chứng cho để có thể dán nhãn v. Sử dụng lại các phát biểu chứng minh (hay kiểm chứng) sẽ giúp tiết kiệm nhiều lời gọi hàm không cần thiết tới Blackbox hoặc Glassbox từ đó tăng được hiệu năng.

Chương 4

Kết luận

Trong quá trình nghiên cứu về các nguyên nhân gây inconsistency trong ontology, chúng em đã tìm hiểu được nhiều giải pháp đã được nghiên cứu và ứng dụng. Chúng em cũng nắm được nguyên lý hoạt động những giải thuật và quy trình này như HST Explaination[9] ,Blackbox Algorithm[8] từ các tác giả của các bài báo [1],[8] và [9]. Hiện nay, những kỹ thuật trên đã được áp dụng trong nhiều ứng dụng thực tế mà cụ thể là thư viện OWL-API[2] để tạo ra các giải thích cho một kết quả suy luận(entailment). Ngoài ra, chúng em cũng biết thêm được về khả năng sửa chữa lại các phát biểu unsatisfiable trong ontology là khả thi căn cứ vào bài báo [1]. Cuối cùng, chúng em nghĩ rằng việc tìm hiểu được các ứng dụng của các giải thuật Hitting Set Tree trong việc tìm ra phát biểu giải thích chính xác và ngắn gọn nhất hay tìm ra một tối tiểu trong các MUPS của ontology, sẽ rất hữu ích để ứng dụng vào việc đề xuất tập hợp các câu hỏi ngắn gọn và chính xác nhất từ tập hợp lớn các SWRL Rules trong ontology.

Chương 5

Tài liệu tham khảo

1. Aditya Kalyanpur, Bijan Parsia, Evren Sirin , Bernardo Cuenca-Grau.Repairing Unsatisfiable Concepts in OWL Ontologies, 2007. Available online at http://www.cs.ox.ac.uk/people/bernardo.cuencagrau/publications/repair.pdf

2. The OWL API. The OWL API is a Java API for creating, parsing, manipulating and serialising OWL Ontologies. Available online at https://github.com/owlcs/owlapi

3. Pellet - OWL 2 Reasoner for Java.Pellet is an OWL 2 reasoner. Pellet provides standard and cutting-edge reasoning services for OWL ontologies. Available online at http://clarkparsia.com/pellet/

4. Robert Stevens. Ontogenesis, (I can't get no) satisfiability Available online at http://ontogenesis.knowledgeblog.org/1329

5. Samantha Bail.Ontogenesis, Common reasons for ontology inconsistency.Available online at http://ontogenesis.knowledgeblog.org/1343

6. S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In Proceedings of IJCAI, 2003.

7. R. Reiter. A theory of diagnosis from first principles. 1987. Artificial Intelligence 32:57-95.

8. A. Kalyanpur, B. Parsia, B. Cuenca-Grau, and E. Sirin. Axiom pinpointing: Finding (precise) justifications for arbitrary entailments in SHOIN (owl-dl). Technical report, UMIACS, 2005-66, 2006. Technical Report

9. Matthew Horridge, the University of Manchester. JUSTIFICATION BASED EXPLANATION IN ONTOLOGIES. Available online at: http://www.bcs.org/upload/pdf/dd-matthew-horridge.pdf