LỜI CAM ĐOAN

Chúng tôi xin cam đoan rằng ngoại trừ các kết quả tham khảo từ các công trình khác như đã ghi trong luận văn, các kết quả trình bày trong luận văn này là do chính chúng tôi thực hiện và chưa có phần nội dung nào trong luận văn này được nộp để lấy bằng cấp ở bất cứ trường nào.

Nguyễn Ngọc Phước – Nguyễn Duy Nhất.

LỜI CẢM ƠN

Chúng tôi xin gửi lời cám ơn đến thầy Th.S. Nguyễn Thanh Sơn đã nhiệt tình hướng dẫn chúng tôi trong quá trình thực hiện luận văn. Thầy đã gợi ý, giúp đỡ cũng như bù đắp những chỗ kiến thức còn thiếu hụt của chúng tôi, tạo cho chúng tôi một tác phong làm việc rất chuyên nghiệp.

Chúng tôi xin gửi lời cảm ơn tới cha mẹ, những người sinh thành và dưỡng dục chúng tôi, những người luôn ủng hộ chúng tôi cả về vật chất lẫn tinh thần, giúp chúng tôi vượt qua những khó khăn trong suốt thời gian qua.

Chúng tôi cũng gửi lời cám ơn tới những người anh, người chị khoá trước, cùng những người bạn thân thiết đã luôn bên chúng tôi động viên, giúp đỡ chúng tôi rất nhiều trong suốt quá trình thực hiện đề tài luận văn tốt nghiệp.

Chủ để của luận văn là: " Chứng minh tự động logic vị từ bằng phương pháp suy luận tự nhiên".

Công việc chính của đề tài là tạo ra một hệ thống có khả năng tự động sinh ra **chuỗi lập luận** cho một bài toán logic vị từ dựa vào những luật suy luận tự nhiên[1]. Chuỗi lập luận này rất gần với cách suy luận "tự nhiên" của con người và có thể chuyển sang chuỗi lập luận bằng ngôn ngữ tự nhiên.

Trong suy luận tự nhiên, chúng ta có một tập các luật suy luận cơ bản. Chúng cho phép tạo ra một công thức mới từ những công thức đã tồn tại. Theo đó, để giải quyết vấn đề nêu trên, chúng tôi xác định cần có một giải thuật tổng quát có khả năng:

- ✓ Định hướng chuỗi lập luận dựa vào yêu cầu của đề toán
- ✓ Tìm một tập công thức và luật suy luận hợp lý để sinh ra từng bước của chuỗi lập luận theo định hướng trên.

Sau quá trình thực hiện đề tài này, chúng tôi đã đạt được những kết quả sau:

1. Tìm ra được giải thuật ND (Natural Deduction) dựa theo định hướng ban đầu.
2. Hiện thực thành công giải thuật ND.
3. Đưa ra một chương trình tương đối hoàn chỉnh có khả năng giải quyết hầu hết các bài toán logic vị từ.

MỤC LỤC

DANH MỤC HÌNH

DANH MỤC BẢNG

CHƯƠNG I
GIỚI THIỆU VỀ ĐỀ TÀI

Xét một bài toán thực tế:

Mọi đứa trẻ đều thích ông già Noel và những ai thích ông già Noel thì đều thích con tuần lộc. Rudolph là con tuần lộc và nó có cái mũi đỏ. Con vật kì quặc hoặc anh hề đều có mũi đỏ. Con tuần lộc không phải là anh hề. Nam không thích bất kỳ con vật kì quặc nào. Nam có phải là trẻ em hay không???

Tách bài toán thành những câu riêng biệt

1. *Mọi đứa trẻ đều thích ông già Noel.*
2. *Những ai thích ông già Noel thì đều thích con tuần lộc.*
3. *Rudolph là con tuần lộc và Rudolph có cái mũi đỏ.*
4. *Mọi vật có mũi đỏ thì hoặc là con vật kỳ quặc hoặc là anh hề.*
5. *Con tuần lộc không phải là anh hề.*
6. *Nam không thích bất kỳ con vật kì quặc nào.*

→ *Nam có phải là trẻ em hay không???*

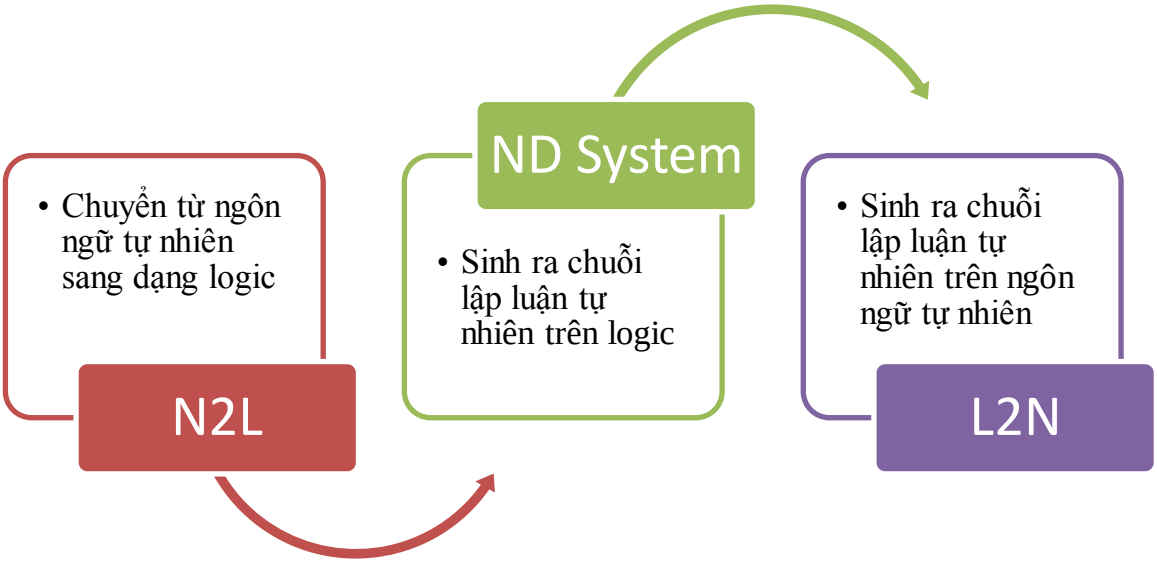
Chuỗi suy luận tự nhiên của bài toán trên là:

- ✓ *Rudolph là con tuần lộc và Rudolph có mũi đỏ (theo (3))*
- ✓ *Rudolph có mũi đỏ nên Rudolph là con vật kì quặc hoặc anh hề (theo (4))*
- ✓ *Rudolph là con tuần lộc theo(5) nên Rudolph không thể là anh hề*
- ✓ *Vậy Rudolph là con vật kì quặc (*)*
- ✓ *Giả sử Nam là trẻ con thì Nam thích ông già Noel (theo (1))*
- ✓ *Nam thích ông già Noel nên Nam thích con tuần lộc (theo (2)) (**)*
- ✓ *Rudolph là con tuần lộc (theo (3)) và từ (**) suy ra Nam thích Rudolph (***)*
- ✓ *Từ (*) và (***) suy ra Nam thích con vật kì quặc.*

  Điều này mâu thuẫn với (6)

  Vậy kết luận *Nam không phải là trẻ em.*

Với máy tính, chúng ta hoàn toàn có thể giải quyết bài toán trên một cách hoàn toàn tự động. Có thể chúng ta xử lý trực tiếp trên ngôn ngữ tự nhiên, và cũng có thể chúng ta chuyển bài toán sang một dạng khác để xử lý. Ở đây chúng tôi giải quyết bằng cách chuyển một bài toán ở ngôn ngữ tự nhiên về ngôn ngữ logic, cụ thể là logic vị từ. Bằng cách áp dụng các luật suy luận tự nhiên sinh ra được chuỗi dẫn xuất suy luận của bài toán ở dạng logic. Bước tiếp theo, chuyển dãy suy luận logic vừa có được về dạng ngôn ngữ tự nhiên dựa vào yêu cầu bài toán.



Hình 1.1: Mô hình suy luận ngôn ngữ tự nhiên

Áp dụng mô hình này vào bài toán trên, các bước sau sẽ được thực hiện

- **N2L**

  Dịch bài toán trên về dạng logic vị từ ta được

  1. $\forall x\ (kid(x) \rightarrow like(x,Santa))$
  2. $\forall x \forall y\ (like(x, Santa) \wedge reindeer\ (y) \rightarrow like\ (x, y))$
  3. $reindeer(Rudolph) \wedge red\_nose(Rudolph)$
  4. $\forall x\ (red\_nose(x) \rightarrow weird\ (x) \vee clown(x))$
  5. $\forall x\ (reindeer(x) \rightarrow \neg clown(x))$
  6. $\forall x(\ weird\ (x) \rightarrow \neg like\ (Nam,x))$

  $\vdash$ $\neg\ kid(Nam)$?

ND System

Sinh ra chuỗi lập luận tự nhiên trên logic vị từ như sau

1. *reindeer(Rudolph) ∧ red_nose(Rudolph) (tiền đề)*
2. *red_nose(Rudolph) ∧e 1*
3. *reindeer(Rudolph) ∧e 1*
4. *∀x (reindeer(x) → ¬clown(x)) (tiền đề)*
5. *reindeer(Rudolph)→ ¬clown(Rudolph) ∀e 4 [Rudolph/x]*
6. *¬clown(Rudolph) →e 3,5*
7. *∀x (red_nose(x) → weird (x) ∨clown(x)) (tiền đề)*
8. *red_nose(Rudolph) → weird (Rudolph) ∀e 3[Rudolph/x]*
9. *weird (Rudolph) ∨clown(Rudolph)→e 4,2*
10. *weird (Rudolph) ∨e 6,9*
11. *∀x(weird (x) → ¬like (Nam,x)) (tiền đề)*
12. *weird (Rudolph) → ¬like (Nam, Rudolph) ∀e 11 [Rudolph/x]*
13. *¬ like(Nam,Rudolph) →e 12,10*
14. *∀x (kid(x) → like(x,Santa)) (tiền đề)*
15. *kid(Nam) → like(Nam, Santa) ∀e 11 [Nam/x]*
16. *∀x∀y (like(x, Santa) ∧ reindeer (y) → like (x, y)) (tiền đề)*
17. *like(Nam, Santa) ∧ reindeer (Rudolph) → like (Nam, Rudolph) ∀e 13*
18. *If kid(Nam)*
19. *like(Nam, Santa) →e 15,12*
20. *like(Nam, Santa) ∧ reindeer (Rudolph) ∧i 16,3*
21. *like (Nam, Rudolph) →e 13,17*
22. *⊥ ¬e 13,21*
23. *Nif ¬kid(Nam)*
24. *¬kid(Nam)*

L2N

Dịch chuỗi lập luận của Inferencer sang ngôn ngữ tự nhiên

1. Rudolph là con tuần lộc và Rudolph có mũi đỏ
2. Vì mọi con tuần lộc không phải là anh hề mà Rudolph là con tuần lộc nên Rudolph không phải là anh hề.
3. Vì Rudolph có mũi đỏ nên hoặc Rudolph là con vật kì quặc hoặc Rudolph là anh hề
4. Theo (2) Rudolph không phải là anh hề nên Rudolph là con vật kì quặc
5. Nam không thích bất kì con vật kì quặc nào mà Rudolph là con vật kì quặc nên Nam không thích Rudolph
6. Giả sử Nam là đứa trẻ nên Nam thích Santa và Nam thích tuần lộc Rudolph
7. (5) và (6) mâu thuẫn nhau.

Do đó kết luận *Nam không phải là đứa trẻ.*

Trên đây là một mô hình của một hệ thống suy luận tự nhiên tự động trên ngôn ngữ tự nhiên. Nhưng do giới hạn về thời gian thực hiện đề tài (chỉ trong một học kỳ) nên ở đây chúng tôi chỉ hiện thực được phần thứ 2, tức là Inferencer. Kết quả của nó có thể hữu ích cho giảng viên cũng như sinh viên trong quá trình giảng dạy, học tập logic vị từ.

CHƯƠNG II
PHÂN TÍCH VÀ THIẾT KẾ HỆ THỐNG

2.1 Phân tích hệ thống ND

Theo yêu cầu của đề tài, phần cần hiện thực là một hệ thống có khả năng sinh ra một chuỗi lập luận tự nhiên từ một danh sách các công thức ở dạng logic vị từ. Mô hình hệ thống được mô tả như sau:

$$A \vdash B$$

- Trong đó hệ thống phải đi tìm những lập luận tự nhiên để dẫn xuất ra công thức B từ tập công thức A
- Tập công thức A có thể trống. Khi đó hệ thống sẽ là:

$$\vdash B$$

2.1.1 Các định nghĩa

2.1.1.1 Dẫn xuất:

Một dẫn xuất F của hệ thống $A \vdash B$ là một công thức có được từ giả thiết hoặc từ các dẫn xuất khác bằng cách áp dụng các luật suy diễn tự nhiên.

Suy luận B từ A là tạo ra một dãy dẫn xuất A0, A1,…,An. Trong đó An chính là B. Như vậy việc sinh ra chuỗi lập luận đồng nghĩa với việc sinh ra một dãy dẫn xuất từ $A \vdash B$.

Hình 2.1: Mô hình ND

2.1.1.2 Sự có mặt

Một công thức bất kỳ F được xem như có mặt trong danh sách L khi và chỉ khi:

- F khác ⊥ và chúng ta tìm được một công thức Pn trong *L* mà tồn tại một phép đồng nhất giữa F và P_n
- F là ⊥ và chúng ta tìm được một cặp công thức P_n , $\neg P_k$ trong L mà tồn tại một phép đồng nhất giữa P_n và P_k

2.1.1.3 Đồng nhất

A,B được gọi là có khả năng đồng nhất khi và chỉ khi tồn tại một thay thế σ mà σ(A) = σ(B).

2.1.1.4 Hệ thống suy luận tự nhiên ND

ND là một hệ thống bao gồm: *list_proof* , *list_goal* , *current_goal* và *tập luật suy luận*.

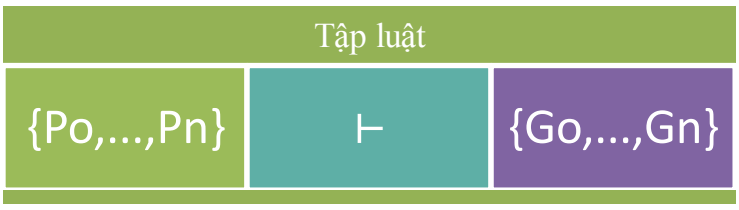
Trong đó:

list_proof: là danh sách {Po,…,Pn} các dẫn xuất có được trong quá trình lập luận.

list_goal: là danh sách {Go,…,Gn} các công thức cần lập luận.

current_goal: là công thức Gn cần lập luận tại thời điểm hiện tại.

tập luật suy luận: là tập các luật suy luật tự nhiên trên logic vị từ (xem phụ đề)



Hình 2.2: Cấu trúc ND

2.1.1.5 Pending – Pending_goal

Công thức F có thuộc tính pending p lớn hơn 0 nghĩa là F được chứng minh khi và chỉ khi p công thức pending_goal của nó được chứng minh.

Pending_goal của một công thức F là tập các công thức cần có để dẫn xuất ra F.

Kí hiệu là:

$F^{(p)}$

Khi p = 0 có thể bỏ qua p trong công thức F.

2.1.1.6 Công thức phụ thuộc

Công thức F được sinh ra từ giả thiết G gọi là công thức phụ thuộc và được kí hiệu như sau:

F[G]

Và G là thuộc tính assumption của F.

2.1.1.7 Rule

Thuộc tính Rule của một công thức F là một luật trong tập luật suy luận tự nhiên đã được áp dụng để dẫn xuất ra F.

2.1.1.8 Hoping_goal

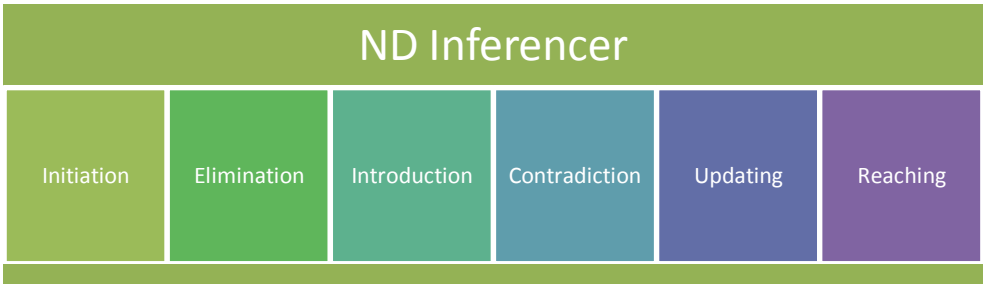
Hoping_goal là một công thức được sinh ra để có thể áp dụng các luật eliminate lên tập list_proof.

- Một công thức F bất kỳ đều có các thuộc tính: *Pending, Pending_goal, Rule* và *Assumption.*

2.1.2 Tổng quát hệ thống suy luận ND

Hệ thống ND là hệ thống GOAL DRIVEN STRAGEDY tức là mọi hành động tại một thời điểm bất kỳ của hệ thống đều phụ thuộc vào current goal.

Hệ thống ND gồm các thành phần sau:



Hình 2.3: ND Inferencer

2.1.2.1 Initiation

Là một module dùng để khởi tạo *list_proof* và *list_goal* từ giả thiết.

Chỉ được gọi một lần duy nhất trong hệ thống

2.1.2.2 Elimination

Là một module dùng để sinh ra dẫn xuất bằng cách áp dụng các luật eliminate của tập luật suy diễn tự nhiên lên tập list_proof của hệ thống ND.

2.1.2.3 Introduction

Là một module dùng để phân rã current_goal thành nhiều sub_goal dựa vào các luật introduction trong tập luật suy diễn tự nhiên.

2.1.2.4 Contradiction

Là một module dùng để tạo ra hoping_goal khi current_goal là $\perp$ và không thể áp dụng elimination lên hệ thống.

2.1.2.5 Updating

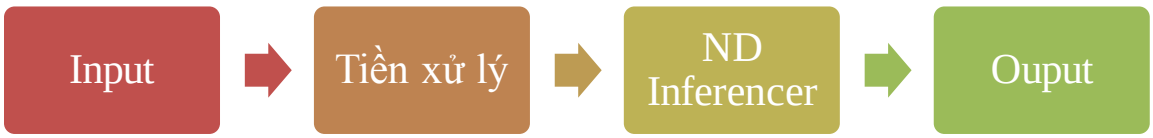
Là một module dùng để cập nhật các thuộc tính cho current_goal nếu current_goal có pending lớn hơn 0.

2.1.2.6 Reaching

Là một module dùng để kiểm tra *sự có mặt* của current_goal trong list_proof.

2.2 Thiết kế hệ thống

Ở đây, chúng tôi xây dựng một hệ thống đầy đủ Input, Output



Hình 2.4: Mô hình đầy đủ hệ thống chứng minh

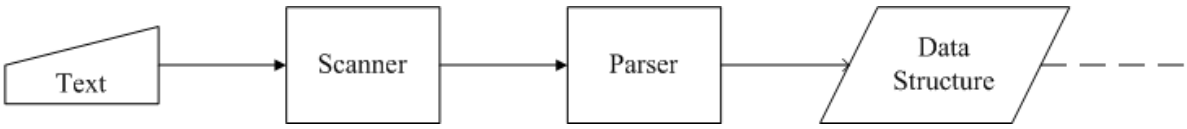
2.2.1 Input

Ở dạng text theo cú pháp BNF của logic vị từ

Sentence	→	AtomicSentence |Sentence Connective Sentence
		| Quantifier Variable, ... Sentence | ¬ Sentence | (Sentence)
AtomicSentence	→	Predicate(Term, …)
Term	→	Function(Term, …) | Constant | Variable
Connective	→	∧ | ∨ | →
Quantifier	→	∀ | ∃
Constant	→	A, B, C, X1 , X2, Jim, Jack
Variable	→	a, b, c, x1 , x2, counter, position
Predicate	→	Adjacent-To, Younger-Than,
Function	→	Father-Of, Square-Position, Sqrt, Cosine

Hình 2.5: BNF Logic vị từ

2.2.2 Tiền xử lý



Hình 2.6: Tiền xử lý

2.2.2.1 Scanner

Nhận dạng chuỗi nhập và chuyển chuỗi nhập thành dạng Token, đồng thời cho biết vị trí của Token đó.

2.2.2.2 Parser

- Nhận kết quả từ bộ SCANNER
- Kiểm tra cú pháp chuỗi nhập có đúng với văn phạm của logic vị từ hay không
    - Nếu đúng cú pháp thì sẽ đưa vào bảng cấu trúc dữ liệu.
    - Nếu sai thì sẽ ngừng chương trình, đồng thời báo lỗi sai và vị trí sai đầu tiên để người dùng có thể dễ dàng sửa chữa.

2.2.3 ND Inferencer

Giải thuật:

1. Gọi module **initiation**
2. Nếu pending > 0

   Gọi **Updating** goto 4
3. Current_goal có mặt trong *list_proof ?*

   a. Nếu có: goto 4

   b. Nếu không thì áp dụng **elimination** vào *list_proof*

      i. Nếu **elimination** thành công: goto 2

      ii. Nếu thất bại: goto c

   c. Áp dụng **introduction** vào goal hiện tại

      i. Nếu **introduction** thành công: goto 2

      ii. Nếu thất bại: goto d

   d. Thực hiện **contradiction** vào *list_proof*

      i. Nếu thành công: goto 2

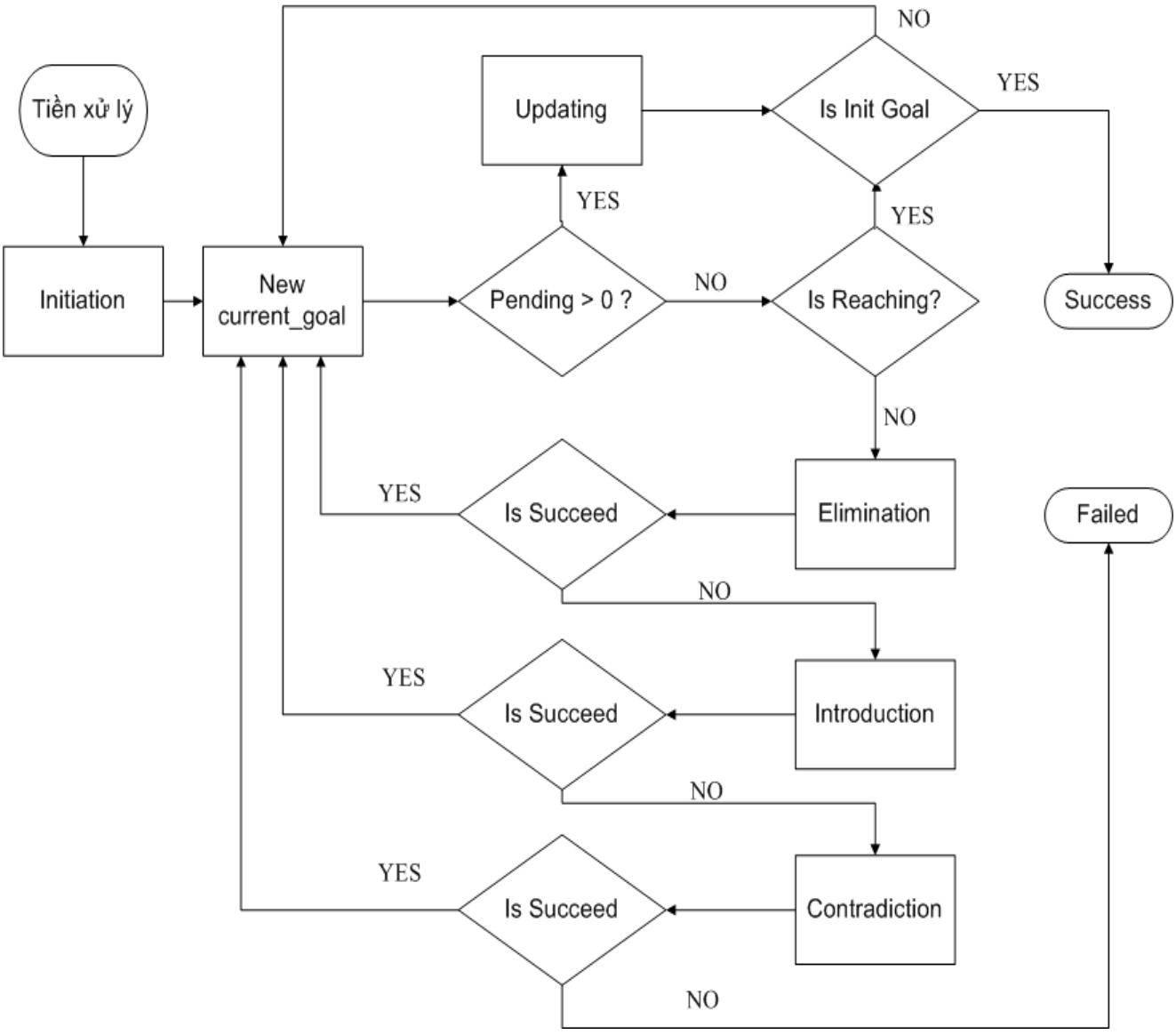
      ii. Nếu thất bại: Failed
4. Lấy goal hiện tại ra khỏi *list_goal*

   e. Nếu *list_goal* trống: Success

   f. Nếu không trống: goto 2

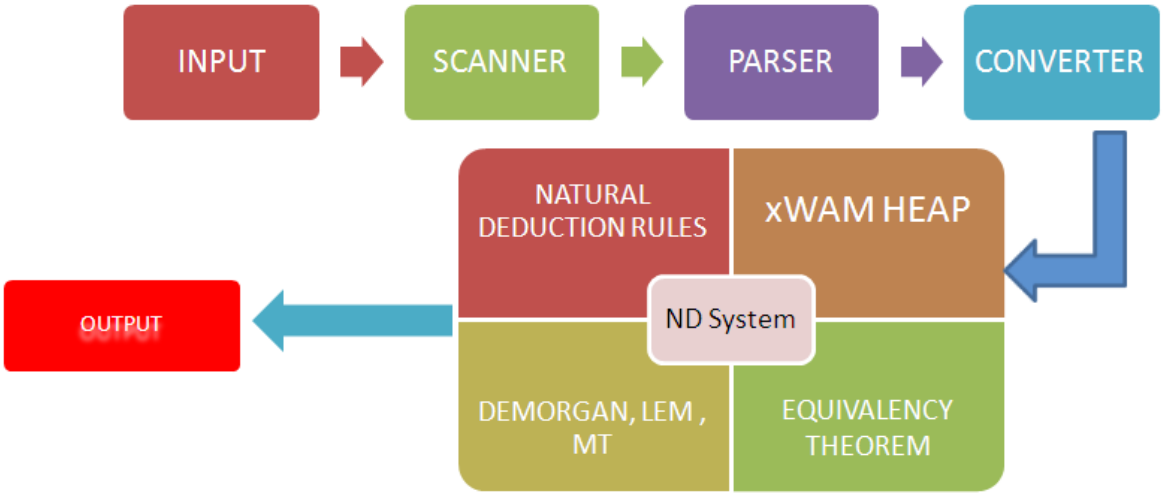
Hình 2.7: Giải thuật chính

FlowChart của giải thuật trên:



Hình 2.8 FlowChart ND Inferencer

CHƯƠNG III
HIỆN THỰC HỆ THỐNG



Hình 3.1: Mô hình hiện thực hệ thống

3.1 INPUT

Như chúng ta đã biết, logic vị từ là một ngôn ngữ hình thức. Do đó, những câu trong ngôn ngữ tự nhiên sẽ được biểu diễn thành những câu trong ngôn ngữ logic với một hình thức khác. Có những kí hiệu đặc biệt trong ngôn ngữ logic. Cụ thể là các toán tử kết nối như: $\wedge$, $\vee$, $\neg$, $\rightarrow$, … Chính vì vậy ta phải thiết kế một bộ kí tự tương đương với những kí tự đặc biệt này.

Bộ kí tự tương đương như sau:

Kí tự đặc biệt trong logic	Kí tự thay thế
∧	&
∨	|
¬	!
→	->
⊢	|-
∀	\-
∃	-]
⊥	_|_

Bảng 3.1: Bảng kí tự tương đương

Input sẽ là một chuỗi (string) gồm nhiều công thức logic vị từ. Các công thức này được ngăn cách nhau bởi dấu phẩy "," . Tiếp theo sau đó là một toán tử kết luận (⊢) và cuối cùng sẽ là một công thức logic (đây là công thức sẽ được dẫn xuất bởi hệ thống).

Input của chương trình là một đề toán logic vị từ gồm có 2 phần: CONDITION và GOAL. Theo cấu trúc:

PREMISES ⊢ GOAL

- PREMISES: gồm một hoặc nhiều công thức trong đó một công thức là một câu theo cấu trúc của BNF đã cho ở phần trên. Các công thức này được ngăn cách nhau bởi dấu phẩy ',' . PREMISES có thể rỗng.
- GOAL: gồm một công thức trong đó một công thức là một câu theo cấu trúc của BNF đã cho ở phần trên

3.2 SCANNER

Nhận dạng chuỗi nhập và chuyển chuỗi nhập thành dạng Token, đồng thời cho biết vị trí của Token đó[5]. Các Token và các lexeme tương ứng được qui ước như sau:

LGC_VAR = ‘x’, ‘y’, ‘z’, …(chuỗi bắt đầu bằng kí tự viết thường)

LGC_INTERSECTION_OP = ‘and’, ‘AND’, ‘ & ’

LGC_UNION_OP = ‘or’, ‘OR’, ‘ ∨ ’

LGC_NEGATION_OP = ‘not’, ‘NOT’, ‘!’, ‘~’

LGC_MAPPING_OP = ‘map’, ‘MAP’, ‘=>’, ‘→’

LGC_EQUIVALENT_OP = ‘<=>’ , ‘<→’

LGC_LEFTPAR = ‘ (‘

LGC_RIGHTPAR = ‘)’

LGC_BOOLEANLITERAL = ‘true’, ‘false’, ‘TRUE’, ‘FALSE’

LGC_CONTRADITION_OP = ‘<>’

LGC_ALL_OP = ‘\-‘ , ‘all’ , ‘ALL’

LGC_EXIST_OP = ‘ ∃’ , ‘∃’ , ‘∃’

LGC_RESULT_OP = ‘=∨’ , ‘∨-‘

LGC_COMMA = ‘,’

LGC_CON = ‘P’, ‘Q’, …(chuỗi bắt đầu bằng kí tự viết hoa)

LGC_NIL = $ (end of file)

LGC_ERROR = others

Ví dụ: Chuỗi nhập là:

p(t) , all x p(x)→ q(x) ∨- q(t)

Các Token tương ứng là:

Token.LGC_VAR Lexeme: p CharStart:2 CharFinish:2

Token.LGC_LEFTPAR Lexeme: (CharStart:3 CharFinish:3

Token.LGC_VAR Lexeme: t CharStart:4 CharFinish:4

Token.LGC_RIGHTPAR Lexeme:) CharStart:5 CharFinish:5

Token.COMMA Lexeme: , CharStart:7 CharFinish:7

Token.LGC_ALL_OP Lexeme: all CharStart:9 CharFinish:11

Token.LGC_VAR Lexeme: x CharStart:13 CharFinish:13

Token.LGC_VAR Lexeme: p CharStart:15 CharFinish:15

Token.LGC_LEFTPAR Lexeme: (CharStart:16 CharFinish:16

Token.LGC_VAR Lexeme: x CharStart:17 CharFinish:17

Token.LGC_RIGHTPAR Lexeme:) CharStart:18 CharFinish:18

Token.LGC_MAPPING_OP Lexeme: → CharStart:20 CharFinish:21

Token.LGC_VAR Lexeme: q CharStart:23 CharFinish:23

Token.LGC_LEFTPAR Lexeme: (CharStart:24 CharFinish:24

Token.LGC_VAR Lexeme: x CharStart:25 CharFinish:25

Token.LGC_RIGHTPAR Lexeme:) CharStart:26 CharFinish:26

Token.LGC_RESULT_OP Lexeme: V- CharStart:28 CharFinish:29

Token.LGC_VAR Lexeme: q CharStart:31 CharFinish:31

Token.LGC_LEFTPAR Lexeme: (CharStart:32 CharFinish:32

Token.LGC_VAR Lexeme: t CharStart:33 CharFinish:33

Token.LGC_RIGHTPAR Lexeme:) CharStart:34 CharFinish:34

Token.LGC_NIL Lexeme: $ CharStart:35 CharFinish:35

3.3 PARSER

- Nhận kết quả từ bộ SCANNER
- Kiểm tra cú pháp chuỗi nhập có đúng với văn phạm của logic vị từ hay không
  - Nếu đúng cú pháp thì sẽ đưa vào bảng cấu trúc dữ liệu (bảng xWAM HEAP).
  - Nếu sai thì sẽ ngừng chương trình, đồng thời báo lỗi sai và vị trí sai đầu tiên để người dùng có thể dễ dàng sửa chữa.

Các qui ước như sau

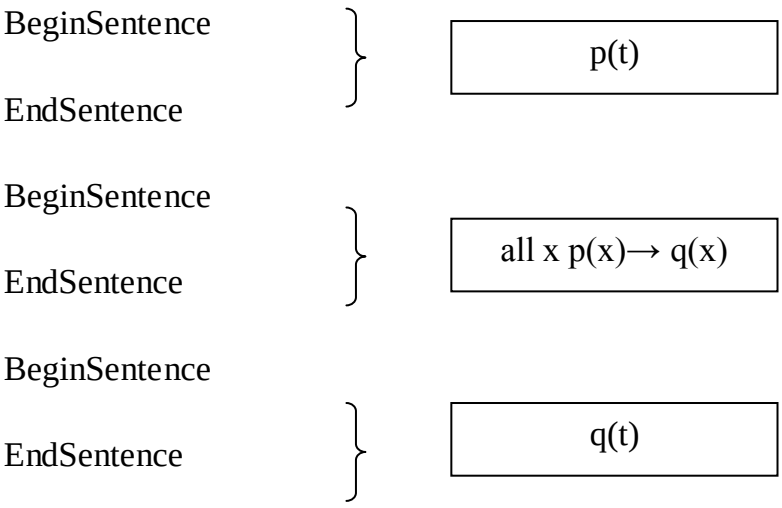
- PREMISES
  - Các tiên đề ngăn cách nhau bởi dấu phẩy và được gọi là một câu (sentence).
  - Một câu sẽ có cú pháp theo cấu trúc văn phạm BNF logic vị từ đã nêu ở trên.
  - PREMISES có thể trống.
- GOAL
  - Là một câu.

Ví dụ 1: Chuỗi nhập là:

p(t) , all x (p(x)→ q(x)) ⊦- q(t)

Kết quả:

Sẽ gồm 3 câu như sau:



Có nghĩa là 3 câu trên sẽ được đưa vào cấu trúc dữ liệu

Ví dụ 2: Chuỗi nhập là:

p(t) , all x p(x → q(x) ∨- q(t) // (thiếu dấu đóng ngoặc tại biến x)

Kết quả:

Unexpected Token.LGC_MAPPING_OP Lexeme: → CharStart:19 CharFinish:20

Chương trình sẽ báo lỗi sai tại Token. LGC_MAPPING_OP tức là dấu : → (vì thiếu đóng ngoặc)

3.4 CONVERTER

Chuyển các toán tử logic thành dạng hàm

Qui ước:

- A ∨ B thành or(A,B)
- A∧ B thành and(A,B)
- A→B thành map(A,B)
- ¬ A thành not(A)

Độ ưu tiên của các toán tử

Theo tứ tự: ( ) ¬ ∧ ∨ →

Ví dụ:

A → ¬B ∧ (¬C ∨ D) thành map (A, and(not(B), or(not(C),D)))

3.5 xWAM HEAP

3.5.1 Mô hình Warren Abstract Machine

3.5.1.1 Lịch sử

Vào năm 1983, David H. D. Warren đã thiết kế ra mô hình Abstract Machine để thực thi các chương trình Prolog được biết đến là WAM [(3)]. Cấu trúc của WAM gồm hai phần:

- ✓ *Kiến trúc bộ nhớ*
- ✓ *Tập lệnh*

Ở đây chúng tôi chỉ quan tâm đến phần *kiến trúc bộ nhớ*. Kiến trúc bộ nhớ trong WAM là WAM HEAP, nó đã trở thành một chuẩn cấu trúc dữ liệu trong chương trình dịch của Prolog trong thực tế.

3.5.1.2 Kiến trúc bộ nhớ trong mô hình WAM

- WAM HEAP sử dụng một array chứa các data cell. Địa chỉ của cell chính là chỉ số index trong array. Chỉ có 2 loại dữ liệu được lưu trong data cell. Đó là variable và structure

- Variable : sử dụng 1 phần tử của WAM heap để lưu trữ tham chiếu đến bảng VAR, được gọi là Variable cell [(4)].

- Structure: f(x1,x2,…,xn) : dùng n+2 phần tử của WAM heap để lưu trữ [(4)]

Trong đó phần tử thứ nhất dùng để lưu kiểu structure và địa chỉ chứa nội dung của nó. Phần tử thứ 2 không nhất thiết phải liên tiếp với phần tử thứ nhất. Phần tử thứ hai dùng để chứa bậc structure (cụ thể ở đây là n). N phần tử tiếp theo (liên tiếp với phần tử thứ hai) dùng để lưu trữ các thành phần con của structure.

Ví dụ: p (Z, h(X,W), f(W)).

0	STR	1
1	$h/2$	
2	REF	2
3	REF	3
4	STR	5
5	$f/1$	
6	REF	3
7	STR	8
8	$p/3$	
9	REF	2
10	STR	1
11	STR	5

Figure 2.1: Heap representation of $p(Z, h(Z, W), f(W))$.

Hình 3. 2: Cấu trúc dữ liệu WamHeap

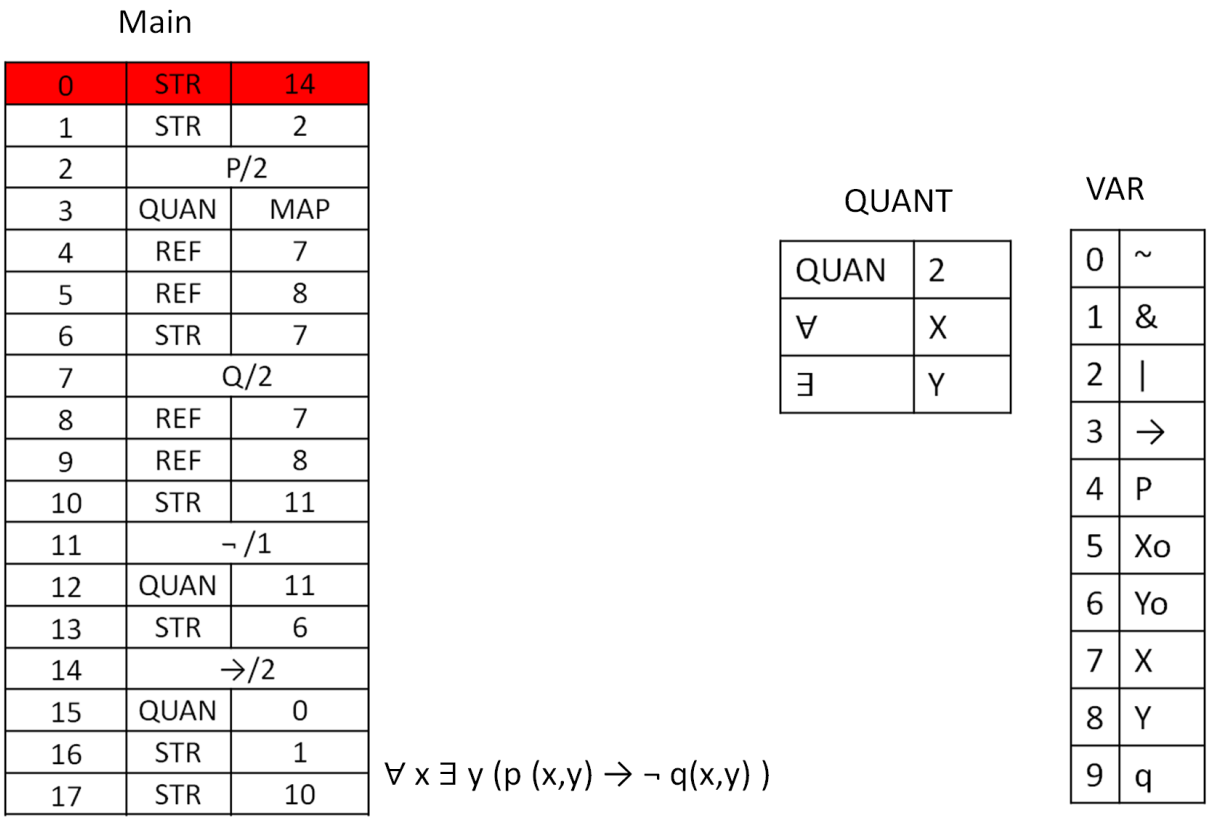
3.5.2 xWAM HEAP

WAM HEAP được thiết kế cho PROLOG mà trong ngôn ngữ PROLOG không có chứa lượng từ hay chính xác hơn mặc định của PROLOG là universal (∀)

Logic vị từ có chứa lượng từ và chương trình có khả năng thể hiện ý nghĩa của lượng từ. Do đó, chúng tôi mở rộng WAM HEAP thành một cấu trúc dữ liệu mới có khả năng giải quyết vấn đề trên. Được đặt tên là xWAM HEAP.

- Đặc điểm của xWAM HEAP
  - Chứa được lượng từ: ∀ ∃
  - VAR cell giống WAM Heap
  - Structure cell thêm phần tử q tham chiếu đến bảng QUANT
  - Bảng QUANT: Chứa các lượng từ

Ví dụ: $\forall x \exists y (p(x,y) \rightarrow \neg q(x,y))$



Hình 3. 3: Mô hình xWAM HEAP

3.6 ND Inferencer

Ý tưởng chính của giải thuật là việc nắm giữ 2 danh sách công thức *list_proof* và *list_goal*. Trong đó *list_proof* là các công thức đã được dẫn xuất, còn *list_goal* nắm giữ các công thức cần phải suy luận. *list_goal* ban đầu chỉ là công thức cần chứng minh, nhưng được biến đổi trong quá trình chứng minh. *list_proof* ban đầu chỉ bao gồm các công thức lấy được từ giả thiết. Trong quá chứng minh ở một thời điểm bất kỳ *list_proof* và *list_goal* có dạng:

$$\{P_0,P_1,P_2..P_n\} \vdash \{G_0,G_1,G_2..G_n\}$$

Ở đây goal có thể là một công thức bất kỳ hoặc mâu thuẫn ($\perp$) . Mọi hoạt động của giải thuật sẽ tác động lên *list_proof* và *list_goal*, đồng thời các hoạt động đó sẽ bị ảnh hưởng bởi current_goal (goal được chọn trong *list_goal*). Tại mỗi bước của giải thuật , current_goal sẽ được kiểm tra nó có mặt trong *list_proof* hay chưa?

Chúng ta sẽ trình bày rõ ràng các module được thực hiện trong quá trình chứng minh:

3.6.1 Initiation

Module này có chức năng dùng để khởi tạo *list_proof* và *list_goal* sau khi xWAM đã được tạo ra.

Module này chỉ được gọi 1 lần vào lúc bắt đầu chứng minh.

3.6.2 Elimination

Module này có chức năng tạo ra những những dẫn xuất mới từ những dẫn xuất có trong *list_proof* . Module này cố gắng áp dụng các luật eliminate vào 1 hay 1 tập công thức trong *list_proof*. Các dẫn xuất mới này sẽ được thêm vào *list_proof*. Việc tìm kiếm các công thức hay cặp công thức sẽ được hoạt động theo breadth-first-search. *list_goal* trong trường hợp này không thay đổi trừ các trường hợp đặc biệt sẽ được trình bày sau.

Thí dụ : Xét một bài toán

$$\{A \wedge B \ , A \rightarrow C, \ \ B \rightarrow \neg \neg E\} \vdash \{Gn\}$$

Áp dụng elimination lên tập *list_proof* của bài toán sẽ được bài toán mới :

list_proof	list_goal
A ∧ B A→C B→ ¬ ¬ E A (∧e A ∧ B) B (∧e A ∧ B)	Gn

list_proof	list_goal
A ∧ B A→C B→ ¬ ¬ E A (∧e A ∧ B) B (∧e A ∧ B) C (→e A → C, A)	Gn

list_proof	list_goal
A ∧ B A→C B→ ¬ ¬ E A (∧e A ∧ B) B (∧e A ∧ B) C (→e A → C, A) ¬ ¬ E (→e B→ ¬ ¬ E , B)	Gn

list_proof	list_goal
A ∧ B A→C B→ ¬ ¬ E A (∧e A ∧ B) B (∧e A ∧ B) C (→e A → C, A) ¬ ¬ E (→e B→ ¬ ¬ E , B) E (¬ ¬ ¬ ¬ E)	Gn

Sau khi áp dụng elimination lên tập *list_proof* ta có được bài toán mới

{A ∧ B , A→C, B→ ¬ ¬ E , A , B, C , ¬ ¬ E , E} ⊢ {Gn}

Lưu ý : các phần tử trong *list_proof* phải được đánh dấu sao cho việc áp dụng luật eliminate không bị lặp lại

Trường hợp đặc biệt: OR, ∃ sẽ được giải thích kỹ ở chương sau.

3.6.3 Introduction

Khi Elimination không thể áp dụng được nữa và current_goal chưa có mặt trong *list_proof*. Chúng ta phải "rã" current_goal ra thành nhiều goal mới, việc rã này có thể ảnh hưởng đến *list_proof*. Giả sử bài toán hiện tại là:

{P0,P1,P2…Pn} ⊢ {G0,G1,G2…Gn}

Gn là current_goal và Gn có một trong các dạng sau:

- Gn = A ∧ B

Để chứng minh A ∧ B ,một cách "tự nhiên", chúng ta phải chứng minh được công thức A và công thức B. Do đó thêm vào *list_goal* 2 goal nữa là A, B đồng thời thiết lập thuộc tính *pending* cho Gn bằng 2 , tức là Gn sẽ xem như được chứng minh nếu 2 goal sau nó Gn+1, Gn+2 được chứng minh. Bài toán thành

$$\{P0,P1,P2\ldots Pn\} \vdash \{G0,G1,G2\ldots A \wedge B^{(2)}, B, A\}$$

- Gn = A → B

Để chứng minh A→B , chúng ta phải chứng minh B với giả thiết A . Do đó việc cập nhật *list_proof* và *list_goal* được thực hiện như sau:

Thêm vào *list_proof* 1 công thức là A , đồng thời đánh dấu đây là giả sử , tức là sau khi chứng minh được A→B, giả sử này phải được loại bỏ đi, đồng thời các dẫn xuất có liên quan đến nó cũng được bỏ đi.

Thêm vào *list_goal* một công thức mới là B và thiết lập thuộc tính *pending* cho A→B là 1.

Bài toán thành

$$\{P0,P1,P2\ldots Pn, A^{(*)}\} \vdash \{G0,G1,G2\ldots, A \rightarrow B^{(1)}, B\}$$

- Gn = A ∨ B

Đây là trường hợp phức tạp hơn so với 2 trường hợp đầu . Để chứng minh được A ∨ B , chúng ta có thể chỉ cần chứng minh được A hoặc B .

Việc đi chứng minh A trước hay B trước được quyết định bởi một heuristic. Nhưng ở đây chúng tôi trình bày theo trình tự A , B.

Nếu A ∨ B rã lần đầu tiên : Để có A ∨ B , chúng ta cần chứng minh A

Thêm công thức A vào *list_goal* , thiết lập thuộc tính *pending* cho A ∨ B là 1 , đồng thời đánh dấu công thức A ∨ B để có thể quay về "rã" A ∨ B lần thứ 2

Bài toán thành

$$\{P0,P1,P2\ldots Pn\} \vdash \{G0,G1,G2\ldots, A \vee B^{(1)(1)}, A\}$$

Nếu A ∨ B rã lần thứ hai:

Thêm công thức B vào *list_goal* , sau đó thiết lập thuộc tính *pending* cho A ∨ B là 1, đồng thời đánh dấu công thức A ∨ B để A ∨ B không thể phân rã thêm nữa.

Bài toán thành

$$\{P0,P1,P2\ldots Pn\}\vdash\{G0,G1,G2\ldots,A \vee B^{(1)(2)} , B\}$$

- Gn = ¬ A

Để chứng minh được ¬ A , chỉ có thể hoặc ¬ A có mặt trong *list_proof* hoặc chứng minh ⊥ với điều kiện giả thiết A. Đây là lối chứng minh phản chứng. Do đó

$$\{P0,P1,P2\ldots Pn, \neg F\} \vdash \{G0,G1,G2\ldots, F, \bot\}$$

list_goal và *list_proof* được cập nhật như sau:

Thêm ⊥ vào *list_goal*, thiết lập thuộc tính pending của ¬ A là 1

Thêm công thức A vào *list_proof*, đánh dấu A là giả sử

Bài toán thành

$$\{P0,P1,P2\ldots Pn,A\} \vdash \{G0,G1,G2\ldots,\neg A, \bot\}$$

- Gn = ∀x f(x)

Để chứng minh ∀x f(x), chúng ta luôn phải tìm một f(x0) với giả thiết x0 bất kỳ. Do đó *list_proof* sẽ được thêm vào 1 công thức x0 , đánh dấu x0 là 1 biến bất kỳ không có ràng buộc, đồng thời x0 là giả sử . *list_goal* sẽ được thêm vào công thức f(x0), và đồng thời thiết lập thuộc tính *pending* của ∀ x f(x) là 1. Sau khi ∀x f(x) được chứng minh, x0 hoặc các công thức có chứa x0 không được xuất hiện trong dãy dẫn xuất nữa.

$$\{P0,P1,P2\ldots Pn, x0\}\vdash\{G0,G1,G2\ldots,\forall xf(x),f(x0)\}$$

- Gn = ∃x g(x)

Để chứng minh ∃x f(x) , chúng ta chỉ cần tìm một công thức f(α) với α là một giá trị bất kỳ. Do đó chỉ có *list_goal* được cập nhật. Thêm f(α) vào *list_goal* với α được thiết lập thuộc tính ANY_VALUE , cập nhật thuộc tính *pending* của ∃x f(x) là 1.

$$\{P0,P1,P2\ldots Pn, x0\} \vdash \{G0,G1,G2\ldots,\exists x\ f(x),f(\alpha)\}$$

- Gn = F

Khi Gn không phải là một trong các dạng trên hoặc Gn ở dạng A ∨ B nhưng đã phân rã 2 lần . Chúng ta phải đi chứng minh mâu thuẫn , tức là thêm vào *list_proof* ¬ F , đánh dấu ¬ F là giả sử đồng thời thêm vào *list_goal* ⊥. Khi đó thuộc tính *pending* của F là 1.

3.6.4 Contradiction

Xét bài toán:

$$\{P0,P1,P2,..\ \mathbf{P_k}\ \ldots\ Pn\} \vdash \{G0,G1,G2..Gn\text{-}1\ ,\ \bot\}$$

Khi không thể áp dụng elimination nữa , và vì ⊥ không thể "rã" , module này sẽ thực thi, nó sẽ tuần tự tìm kiếm các công thức phức tạp trong *list_proof* để tạo hoping_goal. Hoping_goal là goal được mong đợi với sự xuất hiện của nó trong *list_proof* có thể áp dụng được elimination. Giả sử P_k là công thức phức tạp chưa bị tác động của contradiction. Việc cập nhật *list_goal* sẽ phụ thuộc vào dạng của P_k .

P_k = A→B

Nếu chứng minh được công thức A, khi đó A, A→B có thể áp dụng → rule.

Do đó , hoping goal trong trường hợp này là A . Thêm A vào *list_goal* , thiết lập A là hoping_goal, đồng thời đánh dấu A→B đã được contradiction tác động lên.

Bài toán trở thành

$$\{P0,P1,P2,..,\ \mathbf{A\rightarrow B}\ ,\ldots\ Pn\} \vdash \{G0,G1,G2..Gn\text{-}1\ ,\ \bot\ ,\ A\}$$

Pk = ¬ A

Nếu chứng minh được công thức A, khi đó ¬ A, A có thể áp dụng ⊥ rule.

Do đó , hoping _goal trong trường hợp này là A . Thêm A vào *list_goal* , thiết lập A là hoping_goal, đồng thời đánh dấu ¬ A đã được contradiction tác động lên.

Bài toán trở thành

{P0,P1,P2,.., ¬**A** ,… Pn} ⊢{G0,G1,G2..Gn-1 , ⊥ , A}

3.6.5 Reaching

Chúng ta sử dụng unification để tìm tập thay thế của 2 công thức x, y. Việc tìm kiếm tập thay thế σ sẽ được thực hiện theo giải thuật sau[(2)(7)]:

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound
          y, a variable, constant, list, or compound
          θ, the substitution built up so far

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far
  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

3.6.6 Updating

Khi current goal có *pending* lớn hơn 0 hay là current goal là một goal có được bằng cách áp dụng các luật *introduction* vào các dẫn xuất vừa có. Module này được thực thi để cập nhật rules được dùng để sinh ra current goal này.

3.6.7 Trường hợp đặc biệt

3.6.7.1 OR Elimination

$$\frac{A \vee B, A \rightarrow C\ , B \rightarrow C}{C}$$

Một cách tự nhiên, nếu có A →C, B → C thì có được (A ∨ B)→ C. Công thức này do chính chúng tôi đưa ra . Do đó để áp dụng OR elimination vào A ∨ B thì chúng ta tìm A→C, B→C trong *list_proof*. Khả năng tìm được A→C, B→C trong *list_proof* là rất thấp .Vì thế , thay chỉ tác động lên *list_proof* , chúng tôi đi chứng minh công thức A→C, B→C với C là current goal.

Xét bài toán

{P0,P1,P2,.., **A ∨ B** ,… Pn} ⊢{G0,G1,G2..Gn-1 , Gn}

Để áp dụng OR elimination lên A∨B với mục tiêu đi chứng minh Gn. Chúng ta thêm A→Gn, B→Gn vào list_proof. Khi đó ,sau khi có A→Gn và B→Gn, chúng ta có áp dụng luật OR Eliminate vào A∨B, $Gn_{[A]}$.$Gn_{[B]}$ để có được Gn.

$$\{P0,P1,P2,..,\ \mathbf{A \vee B}^{(*)},\ldots\ Pn\} \vdash \{G0,G1,G2..Gn,\ Gn_{[A]},\ Gn_{[B]}\}$$

Dạng trình bày của OR Elimination là:

- dòng m : F ∨ G
- dòng n : if F
- dòng n+p : nif H
- dòng k : if G
- dòng k+q : nif H
- dòng k+q+1 : H

Nếu F sinh ra H và G cũng sinh ra H thì F ∨ G cũng sinh ra H.

Xét bài toán :

$$\{A \vee B,\ C,\ C \rightarrow D\} \vdash \{D\}$$

Dễ dàng tìm ra lời giải cho bài toán bằng cách áp dụng → e vào C, C→D. Nhưng nếu chúng ta đã áp dụng ∨e vào A∨B, thì rõ ràng D chịu sự ảnh hưởng của A∨B.

Tức là :

1. C → D
2. C
3. A∨B
4. If A
5. Nif D (→e 1,2)
6. If B
7. Nif D (→e 1,2)
8. D

Chúng ta cần loại bỏ sự ảnh hưởng vô nghĩa của A∨B. Để giải quyết vấn đề này chúng ta cần phải lưu lại nguồn gốc của dẫn xuất. Nếu trong trường hợp này D không hề có được dẫn xuất từ A∨B thì sự ảnh hưởng sẽ bị loại bỏ.

Xét bài toán khác:

$\{F \rightarrow H\} \vdash \{ F \vee G \rightarrow H \vee G \vee R\}$

Bài giải được xem là hoàn hảo:

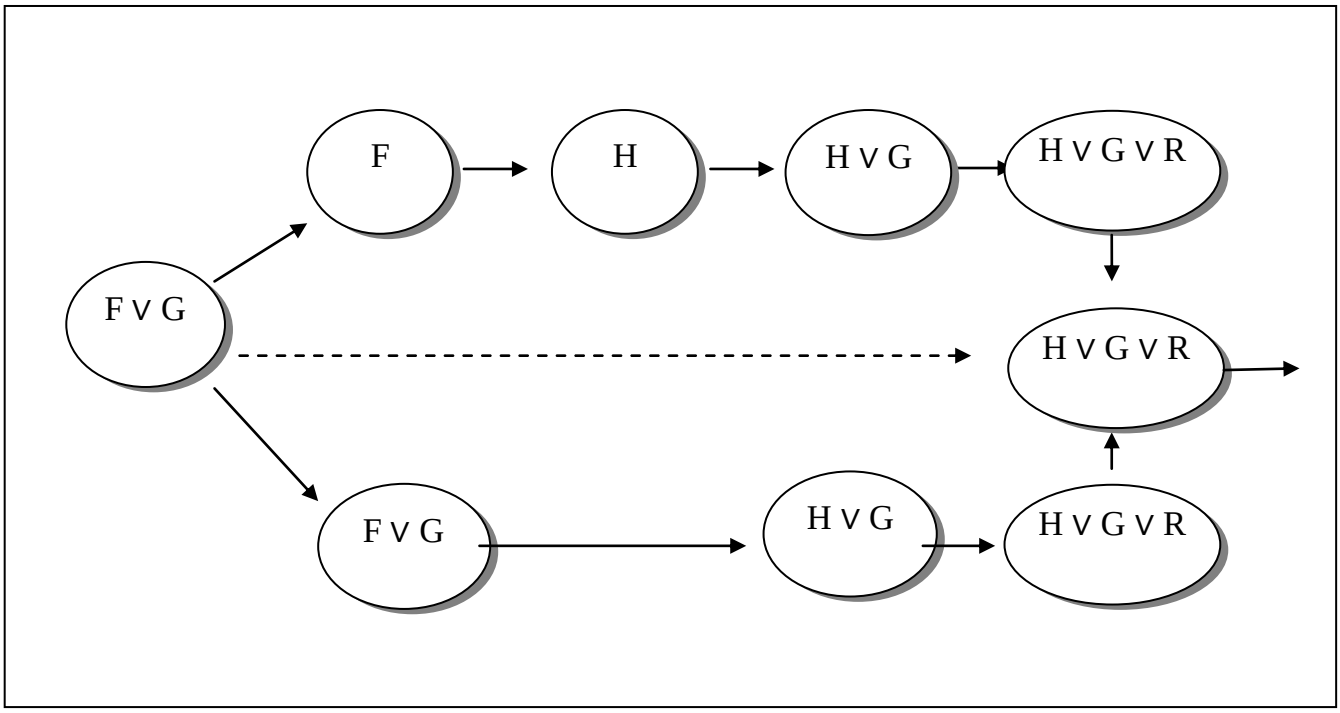
1. $F \rightarrow H$
2. If $F \vee H$
3. If F
4. H ($\rightarrow$e 1,3)
5. Nif $H \vee G$
6. If G
7. Nif $H \vee G$
8. $H \vee G$
9. Nif $H \vee G \vee R$
10. $F \vee G \rightarrow H \vee G \vee R$

Thế nhưng, với giải thuật trên bài toán sẽ được giải là

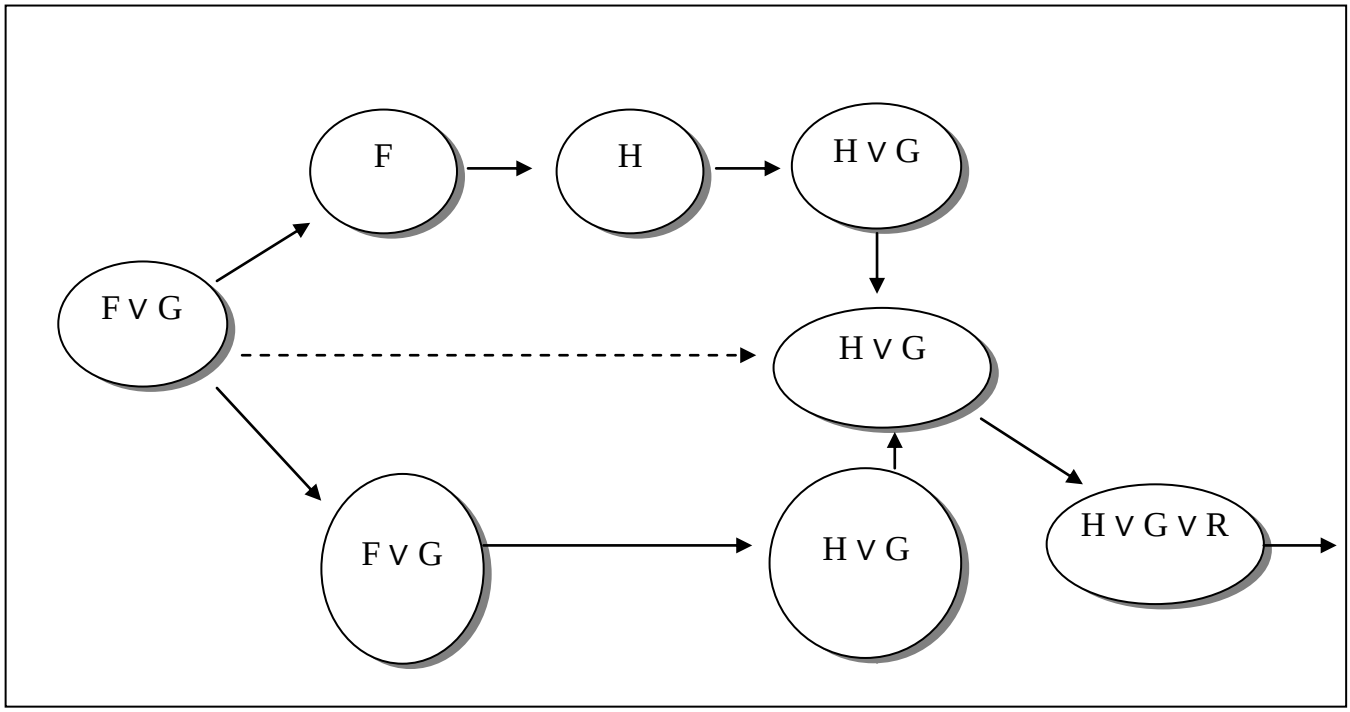
1. $F \rightarrow H$
2. If $F \vee H$
3. If F
4. H ($\rightarrow$e 1,3)
5. $H \vee G$
6. Nif $H \vee G \vee R$
7. If G
8. $H \vee G$
9. Nif $H \vee G \vee R$
10. Nif $H \vee G \vee R$
11. $F \vee G \rightarrow H \vee G \vee R$

Rõ ràng sự ảnh hưởng của FVH chỉ tác động trực tiếp lên HVG còn HVGVR chi chịu ảnh hưởng từ HVG. Do đó chúng ta cần xác định lại phạm vi ảnh hưởng của FVH lên các dẫn xuất

F V H ảnh hưởng lên H V G V R



F V G ảnh hưởng lên H V G



Để loại bỏ sự ảnh hưởng vô nghĩa của F ∨ G lên các dẫn xuất của bài toán dễ dàng được thực hiện dựa vào 2 hình trên.

3.6.7.2 ∃ Eliminate:

Nếu f(x0) với x0 là giá trị bất kỳ thỏa mãn f(x0) là một công thức và với f(x0) , chúng ta dẫn xuất được công thức G. Như thế chúng ta sẽ có ∃x f(x) → G bằng cách áp dụng luật ∃ eliminate.

Mô hình

If x 0 f(x0)
…
Nif G
G

Xét bài toán

..., ∃x f(x) ⊢ Gn

Để áp dụng ∃e vào ∃x f(x), chúng ta phải tìm một giá trị x0 chưa từng xuất hiện trong các dẫn xuất trước đó. Thêm vào công thức f(x0) vào *list_proof* đồng thời xóa sự có mặt của ∃x f(x). Có thể ∃ x f(x) và f(x0) không hoàn toàn tương đương nhau nhưng trong trường hợp này là tương đương vì từ ∃xf(x) chúng ta có được f(x0) và ngược lại.

f(x0) sẽ tồn tại xuyên suốt quá trình chứng minh, vấn đề đặt ra là dẫn xuất nào chịu ảnh hưởng, dẫn xuất nào không bị ảnh hưởng.

Xét bài toán :

{∃x f(x), ∀ x f(x) → G } ⊢ {G∨H}

Bài chứng minh hoàn hảo là:

1. ∀ x f(x) → G
2. {∃x f(x)
3. If x0 , f(x0)
4. f(x0) → G (∀e 1)
5. Nif G (→e 3,4)
6. G (∃e 2)
7. G ∨ H

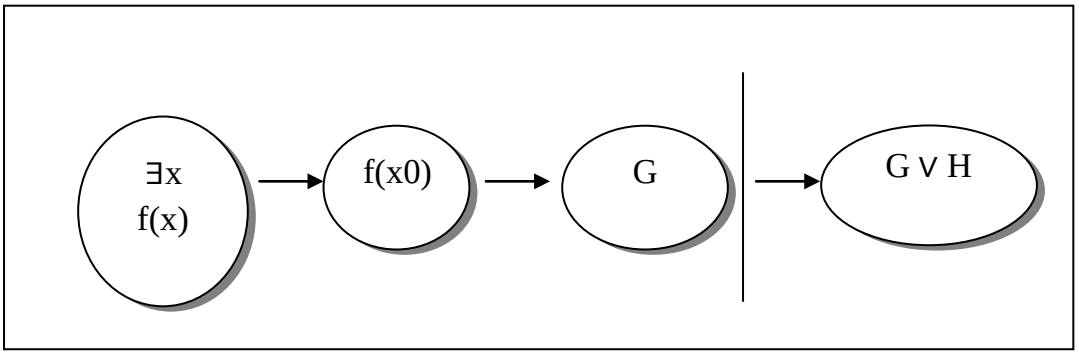
Thế nhưng theo giải thuật trên bài toán sẽ được giải như sau:

1. ∀ x f(x) → G
2. {∃x f(x)
3. If x0 , f(x0)
4. f(x0) → G (∀e 1)
5. G (→e 3,4)
6. Nif G ∨ H
7. G ∨ H (∃e 2)

Nif G chính là điểm dừng sự ảnh hưởng của f(x0) lên tập dẫn xuất.

Rõ ràng với giải thuật trên chưa , vấn đề này giải quyết được.

Để giải quyết vấn đề này chúng ta bắt đầu thì G0, đi dọc theo các goal con nó để chọn điểm cắt theo hình sau



3.6.8 Các thí dụ

3.6.8.1 Thí dụ 1

Ở phần này chúng tôi sẽ trình chi tiết từng bước trong giải thuật trên.

Thí dụ

$$F \rightarrow H \vdash F \vee G \rightarrow H \vee G$$

Ở bước khởi tạo chúng ta sẽ được 2 danh sách list_proof và list_goal như sau

List_proof	List_goal
F→H	Go = {F∨G → H∨G}

Tiếp theo chúng ta thực hiện elimination cho tập *list_proof*. Việc eliminate thất bại vì có F→H nhưng không tìm thấy F.

Do đó trạng thái hiện tại của *list_proof* và *list_goal* vẫn không đổi

List_proof	List_goal
F→H	Go = {F∨G → H∨G}

Theo giải thuật nếu việc eliminate thất bại, chúng ta tiến hành thực thi introduction lên current goal. Current goal chính là G0 và G0 có dạng A→B. Dựa theo module Introduction, giả thiết F∨G sẽ được thêm vào *list_proof* và H∨G sẽ được thêm vào *list_goal*. G0 sẽ được thiết lập thuộc tính *pending* là 1.

List_proof	List_goal
F→H F∨G	Go = {F∨G → H∨G} {F∨G→H∨G, H∨G}

Sau khi introduction thành công , current goal sẽ được thiết lập lại bằng G1 = {H∨G}. Theo giải thuật, chúng ta sẽ quay lại bước 1 tức là sẽ đi kiểm tra H∨G có mặt trong tập *list_proof* hay chưa? Kết quả H∨G chưa có mặt trong *list_proof*.

Chúng ta tiến hành thực thi module elimination lên tập *list_proof*. Lúc này *list_proof* có 2 phần tử F→H và F∨G . Module Elimination thực hiện theo breadth first search nên sẽ duyệt các phần tử từ trái sang phải . F→H không thể eliminate được vì không tìm được F. Công thức còn lại F∨G có thể eliminate theo trường hợp đặc biệt OR. Theo (1) việc tiến hành eliminate OR sẽ thay đổi *list_goal* và *list_proof* như sau: *list_goal* sẽ được thêm vào 2 công thức H∨G [F] và H∨G [G] còn Go = {F∨G → H∨G} được thiết lập thuộc tính *pending* là 2.

List_proof	List_goal
F→H F∨ G assumption	F∨G → H∨G F∨G→H∨G, H∨G F∨G→H∨G , H∨G , H∨G[G] , H∨G[F]

Lúc này chúng ta lại quay về bước 1. Current goal được thiết lập là H∨G[F]. Tức là tìm H∨G với điều kiện F. Thêm giả thiết F vào tập *list_proof*, đánh dấu F là giả sử của H∨G.

List_proof	List_goal
F→H F ∨ G assumption F assumption	F∨G → H∨G F∨G→H∨G, H∨G F∨G→H∨G, H∨G , H∨G[G], H∨G[F] F∨G→H∨G, H∨G , H∨G[G], H∨G

Lúc này tiến hành tìm kiếm sự có mặt của H∨G. Dễ dàng nhận ra H∨G không có mặt trong tập *list_proof*. Tiến hành áp dụng elimination lên *list_proof*. F→H áp dụng thành công elimination nhờ có F. Do đó công thức H sẽ được thêm vào *list_proof*

List_proof	List_goal
F→H F ∨ G assumption F assumption H	F∨G → H∨G F∨G→H∨G, H∨G F∨G→H∨G, H∨G ,H∨G[G], H∨G[F] F∨G→H∨G, H∨G , H∨G[G], H∨G

Việc elimination thành công, chương trình quay về bước 1, tiến hành xác định H∨G có mặt hay chưa? Kết quả là H∨G vẫn chưa có mặt. Tiến hành elimination tiếp tục và thất bại. Do không thể áp dụng eliminate vào *list_proof* nên phải "rã" current goal. Việc rã sẽ được thực hiện theo OR Introduction. Do H∨G rã lần đầu tiên nên *list_goal* sẽ thêm vào công thức thứ nhất là H

List_proof	List_goal
F→H F ∨ G assumption F assumption H →e 1,3 	F∨G → H∨G F∨G→H∨G, H∨G F∨G→H∨G, H∨G , H∨G[G], H∨G[F] F∨G→H∨G, H∨G , H∨G[G], H∨G F∨G→H∨G, H∨G , H∨G[G], H∨G,H

Quay về bước 1, tiến hành tìm kiếm sự có mặt của H trong *list_proof*. H có mặt. Như vậy goal H đã được chứng minh. Lấy goal H ra khỏi *list_goal*. Thiết lập current goal là H∨G. Quay về bước 1.

List_proof		List_goal
F→H		FVG → HVG
F V G	assumption	FVG→HVG, HVG
F	assumption	FVG→HVG, HVG , HVG[G], HVG[F]
		FVG→HVG, HVG , HVG[G], HVG
H	→e 1,3	
		FVG→HVG, HVG , HVG[G], HVG

Current goal là HVG và HVG có *pending* bằng HVG tức là 1 được sinh ra từ 1 goal kế trước nó. Gọi module Updating, ta có được H V G sinh ra từ H theo Vi rules. Đưa H VG vào *list_proof*.

List_proof		List_goal
F→H		FVG → HVG
F V G	assumption	FVG→HVG, HVG
F	assumption	FVG→HVG, HVG , HVG[G], HVG[F]
		FVG→HVG, HVG , HVG[G], HVG , H
H	→e 1,3	
		FVG→HVG, HVG , HVG[G], HVG
H V G	Vi 4	FVG→HVG, HVG , HVG[G]

Loại bỏ HVG ra khỏi *list_goal*. Ở bước này, đồng thời xóa sự có mặt của giả sử F và các dẫn xuất từ F.

List_proof		List_goal
F→H		FVG → HVG
F V G	assumption	FVG→HVG, HVG
~~F~~	assumption	FVG→HVG, HVG , HVG[G], HVG[F]
		FVG→HVG, HVG , HVG[G], HVG , H
~~H~~	→e 1,3	
		FVG→HVG, HVG , HVG[G], HVG
~~H V G~~	Vi 4	FVG→HVG, HVG , HVG[G]

Thiết lập current goal là HVG[G], thêm công thức G vào *list_proof* và đồng thời biến HVG[G] thành HVG.

List_proof		List_goal
F→H		FVG → HVG
F V G	assumption	FVG→HVG, HVG
~~F~~	assumption	FVG→HVG, HVG , HVG[G], HVG[F]
		FVG→HVG, HVG , HVG[G], HVG , H
~~H~~	→e 1,3	
		FVG→HVG, HVG , HVG[G], HVG
~~H V G~~	Vi 4	FVG→HVG, HVG , HVG[G]
		FVG→HVG, HVG , HVG
G	assumption	

Ở bước tiếp theo ta đi tìm sự có mặt H V G trong *list_proof*, dễ dàng nhận ra H V G chưa có mặt trong *list_proof*. Theo tuần tự của giải thuật, ở bước này module elimination sẽ được gọi. Và kết quả elimination thất bại. Do đó, công thức H V G sẽ được introduction tác động lên. H V G ở dạng OR nên sẽ được "rã" theo OR introduction. Đây là lần "rã" đầu tiên nên việc "rã" diễn ra như sau:

- Thêm công thức H vào *list_goal*.
- Đánh dấu H V G đã được rã 1 lần

List_proof		List_goal
F→H		FVG → HVG
F V G	assumption	FVG→HVG, HVG
~~F~~	assumption	FVG→HVG, HVG , HVG[G], HVG[F]
		FVG→HVG, HVG , HVG[G], HVG , H
~~H~~	→e 1,3	
		FVG→HVG, HVG , HVG[G], HVG
~~H V G~~	Vi 4	FVG→HVG, HVG , HVG[G]
		FVG→HVG, HVG , HVG
G	assumption	FVG→HVG, HVG , HVG , H

Thiết lập current goal là H. Theo giải thuật, sẽ đi tìm sự có mặt của H có trong *list_proof* hay chưa? Kết quả H chưa có mặt trong *list_proof.* Tiến hành áp dụng elimination lên tập *list_proof.* Kết quả thất bại do đó công thức H sẽ được introduction. Do H không phải là các dạng công thức phức tạp nên việc introduction sẽ phản chứng. Tức là thêm ⊥ vào *list_goal* và ¬ H vào *list_proof.* Thiết lập current goal là ⊥. Bước tiếp theo, đi tìm sự có mặt của ⊥ . Theo định nghĩa sự có mặt, chúng ta phải đi tìm 1 cặp ¬A và A nhưng trong trường hợp này *list_proof* không thỏa mãn điều kiện này. Theo giải thuật, contradiction sẽ được gọi trong bước này. Dựa vào tập *list_proof* F → H sẽ được contradiction. Theo module contradiction, F sẽ được thêm vào *list_goal.*

List_proof		List_goal
F→H		FVG → HVG
F V G	assumption	FVG→HVG, HVG
~~F~~	assumption	FVG→HVG, HVG , HVG[G], HVG[F]
		FVG→HVG, HVG , HVG[G], HVG , H
~~H~~	→e 1,3	
		FVG→HVG, HVG , HVG[G], HVG
~~H V G~~	Vi 4	FVG→HVG, HVG , HVG[G]
		FVG→HVG, HVG , HVG
G	assumption	FVG→HVG, HVG , HVG , H
		FVG→HVG, HVG , HVG , H , ⊥
¬ H	assumption	FVG→HVG, HVG , HVG , H , ⊥ , F

Thiết lập current goal là F, bước tiếp theo sẽ là tìm sự có mặt của F trong *list_proof*. Kết quả là thất bại. Tiến hành introduction F, tức là thêm ⊥ vào *list_goal*, thêm giả sử ¬F vào *list_proof*.

List_proof		List_goal
F→H		FVG → HVG
F V G	assumption	FVG→HVG, HVG
~~F~~	assumption	FVG→HVG, HVG , HVG[G], HVG[F]
		FVG→HVG, HVG , HVG[G], HVG , H
~~H~~	→e 1,3	
		FVG→HVG, HVG , HVG[G], HVG
~~H V G~~	Vi 4	FVG→HVG, HVG , HVG[G]
		FVG→HVG, HVG , HVG
G	assumption	FVG→HVG, HVG , HVG , H
		FVG→HVG, HVG , HVG , H , ⊥
¬ H	assumption	FVG→HVG, HVG , HVG , H , ⊥ , F
		FVG→HVG, HVG , HVG , H , ⊥ , F , ⊥
¬ F	assumption	

Thiết lập current goal là ⊥. Theo giải thuật, đi tìm sự có mặt của ⊥ trong *list_proof*. Dễ dàng nhận ra trong *list_proof* không chứa 1 cặp công thức ¬A và A. Bước tiếp theo sẽ là contradiction tập *list_proof*. Kết quả việc áp dụng contradiction là thất bại nên sẽ quay về bước "rã" OR gần nhất. Theo bài toán, bước "rã" OR gần nhất là bước "rã" H ∨ G. Xóa hết những dẫn xuất hay công thức đã được thêm vào *list_goal* hay *list_proof* trong quá trình chứng minh của lần "rã" thứ nhất. Áp dụng lần "rã" thứ 2 cho H ∨ G ta được:

List_proof		List_goal
F→H		F∨G → H∨G
F ∨ G	assumption	F∨G→H∨G, H∨G
~~F~~	assumption	F∨G→H∨G, H∨G , H∨G[G], H∨G[F]
		F∨G→H∨G, H∨G , H∨G[G], H∨G , H
~~H~~	→e 1,3	
		F∨G→H∨G, H∨G , H∨G[G], H∨G
~~H ∨ G~~	∨i 4	F∨G→H∨G, H∨G , H∨G[G]
		F∨G→H∨G, H∨G , H∨G
G	assumption	F∨G→H∨G, H∨G , H∨G , G

Thiết lập current goal là G. Dễ dàng nhận ra G có mặt trong *list_proof*. Xóa G ra khỏi *list_goal*. Thiết lập current goal là H ∨ G. H ∨ G có *pending* bằng 1. Tức là H ∨ G được chứng minh khi có G. Áp dụng module updating ta có bảng sau:

List_proof		List_goal
F→H		FvG → HvG
F v G	assumption	FvG→HvG, HvG
~~F~~	assumption	FvG→HvG, HvG , HvG[G], HvG[F]
		FvG→HvG, HvG , HvG[G], HvG , H
~~H~~	→e 1,3	
		FvG→HvG, HvG , HvG[G], HvG
~~H v G~~	vi 4	FvG→HvG, HvG , HvG[G]
		FvG→HvG, HvG , HvG
G	assumption	FvG→HvG, HvG , HvG , G
		FvG→HvG, HvG
H v G	vi 6	

Thiết lập current goal vào H v G. H v G có *pending* là 2. Gọi updating ta được:

List_proof		List_goal
F→H		FvG → HvG
F v G	assumption	FvG→HvG, HvG
~~F~~	assumption	FvG→HvG, HvG , HvG[G], HvG[F]
		FvG→HvG, HvG , HvG[G], HvG , H
~~H~~	→e 1,3	
		FvG→HvG, HvG , HvG[G], HvG
~~H v G~~	vi 4	FvG→HvG, HvG , HvG[G]
		FvG→HvG, HvG , HvG
~~G~~	assumption	FvG→HvG, HvG , HvG , G
		FvG→HvG, HvG
~~H v G~~	vi 6	FvG→HvG
H v G	ve 2,3,6	

Thiết lập current goal là FVG→HVG. FVG→HVG có *pending* bằng 2. Gọi updating lên FVG→HVG ta có được

List_proof		List_goal
F→H		FVG → HVG
F V G	assumption	FVG→HVG, HVG
~~F~~	assumption	FVG→HVG, HVG , HVG[G], HVG[F]
		FVG→HVG, HVG , HVG[G], HVG , H
~~H~~	→e 1,3	
		FVG→HVG, HVG , HVG[G], HVG
~~H V G~~	Vi 4	FVG→HVG, HVG , HVG[G]
		FVG→HVG, HVG , HVG
~~G~~	assumption	FVG→HVG, HVG , HVG , G
		FVG→HVG, HVG
~~H V G~~	Vi 6	FVG→HVG
H V G	Ve 2,3,6	
FVG→HVG	→i 2,8	

Vì FVG→HVG là goal ban đầu nên giải thuật kết thúc.

Phần trình bày trên chưa sử dụng heuristic ở bước trước OR Introduction. Phần heuristic sẽ được giới thiệu ở phục lục

3.6.8.2 Thí dụ 2

Xét bài toán

$$\forall x (f(x) \wedge g(x)) \vdash \forall x f(x) \wedge \forall x g(x)$$

Bước đầu tiên module initiation được thực thi để khởi tạo *list_proof* và *list_goal* như sau:

List_proof	List_goal
$\forall x (f(x) \wedge g(x))$	$\forall x f(x) \wedge \forall x g(x)$

Thiết lập current goal là $\forall x f(x) \wedge \forall x g(x)$. Kiểm tra sự có mặt của current goal trong *list_proof*. Kết quả thất bại. Bước tiếp theo áp dụng elimination lên *list_proof*. Do không có biến, mệnh đề hay giá trị hằng nào khác x nên không thể elimination $\forall x (f(x) \wedge g(x))$. Tiến hành introduction current goal ta được

List_proof	List_goal
$\forall x (f(x) \wedge g(x))$	$\forall x f(x) \wedge \forall x g(x)$ $\forall x f(x) \wedge \forall x g(x)$, $\forall x g(x)$, $\forall x f(x)$

Current goal mới là $\forall x f(x)$. Vì nó không có mặt trong *list_proof* và *list_proof* không elimination được nên introduction current goal. Theo đặc tả introduction $\forall x f(x)$ sẽ được introduction như sau:

Thêm vào *list_proof* một giá trị xo chưa xuất hiện trong dẫn xuất với xo không có bất kỳ ràng buộc nào (+ $\forall$).

Thêm vào *list_goal* f(xo)

Thiết lập pending của $\forall x f(x)$ là 1.

List_proof	List_goal
∀x (f(x) ∧ g(x))	∀x f(x) ∧ ∀x g(x)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x)
xo assumption + ∀	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x), f(xo)

Thiết lập current goal mới là f(xo). f(xo) chưa có mặt trong *list_proof* . Tiến hành elimination *list_proof* được

List_proof	List_goal
∀x (f(x) ∧ g(x))	∀x f(x) ∧ ∀x g(x)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x)
xo assumption + ∀	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x), f(xo)
f(xo) ∧ g(xo) ∀e 1	
f(xo) ∧e 3	
g(xo) ∧e 3	

Lúc này f(xo) có mặt trong *list_proof.* f(xo) được chứng minh nên current goal mới là ∀x f(x) có pending là 1 nên nó được dẫn xuất nhờ updating. Ở bước này, xo và các dẫn xuất chứa nó được xóa khỏi *list_proof.*

List_proof	List_goal
∀x (f(x) ∧ g(x))	∀x f(x) ∧ ∀x g(x)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x)
~~xo~~ assumption + ∀	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x), f(xo)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x)
~~f(xo) ∧ g(xo)~~ ∀e 1	
~~f(xo)~~ ∧e 3	
~~g(xo)~~ ∧e 3	
∀x f(x) ∀i 2,4	

Bước chứng minh ∀x g(x) tương tự ∀x f(x). Các bước chứng minh được minh họa bằng các bảng sau:

List_proof	List_goal
∀x (f(x) ∧ g(x))	∀x f(x) ∧ ∀x g(x)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x)
~~xo~~ assumption + ∀	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x), f(xo)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x)
~~f(xo) ∧ g(xo)~~ ∀e 1	∀x f(x) ∧ ∀x g(x) , ∀x g(x), g(x1)
~~f(xo)~~ ∧e 3	
~~g(xo)~~ ∧e 3	
∀x f(x) ∀i 2,4	
~~x1~~ assumption + ∀	

List_proof	List_goal
∀x (f(x) ∧ g(x))	∀x f(x) ∧ ∀x g(x)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x)
~~xo~~ assumption + ∀	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x), f(xo)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x)
~~f(xo) ∧ g(xo)~~ ∀e 1	∀x f(x) ∧ ∀x g(x) , ∀x g(x), g(x1)
~~f(xo)~~ ∧e 3	
~~g(xo)~~ ∧e 3	
∀x f(x) ∀i 2,4	
x1 assumption + ∀	

List_proof	List_goal
∀x (f(x) ∧ g(x))	∀x f(x) ∧ ∀x g(x)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x)
~~xo~~ assumption + ∀	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x), f(xo)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x)
~~f(xo) ∧ g(xo)~~ ∀e 1	∀x f(x) ∧ ∀x g(x) , ∀x g(x), g(x1)
~~f(xo)~~ ∧e 3	
~~g(xo)~~ ∧e 3	
∀x f(x) ∀i 2,4	
x1 assumption + ∀	
f(x1) ∧ g(x1) ∀e 1	
f(x1) ∧e 8	
g(x1) ∧e 8	

List_proof	List_goal
∀x (f(x) ∧ g(x))	∀x f(x) ∧ ∀x g(x)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x)
~~xo~~ assumption + ∀	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x), f(xo)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x)
~~f(xo) ∧ g(xo)~~ ∀e 1	∀x f(x) ∧ ∀x g(x) , ∀x g(x), g(x1)
	∀x f(x) ∧ ∀x g(x)
~~f(xo)~~ ∧e 3	
~~g(xo)~~ ∧e 3	
∀x f(x) ∀i 2,4	
~~x1~~ assumption + ∀	
~~f(x1) ∧ g(x1)~~ ∀e 1	
~~f(x1)~~ ∧e 8	
~~g(x1)~~ ∧e 8	
∀x g(x) ∀i 7, 10	

Lúc này current goal là ∀x f(x) ∧ ∀x g(x) có pending là 2 nên được chứng minh dễ dàng

List_proof	List_goal
∀x (f(x) ∧ g(x))	∀x f(x) ∧ ∀x g(x)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x)
~~xo~~ assumption + ∀	∀x f(x) ∧ ∀x g(x) , ∀x g(x), ∀x f(x), f(xo)
	∀x f(x) ∧ ∀x g(x) , ∀x g(x)
~~f(xo) ∧ g(xo)~~ ∀e 1	∀x f(x) ∧ ∀x g(x) , ∀x g(x), g(x1)
	∀x f(x) ∧ ∀x g(x)
~~f(xo)~~ ∧e 3	
~~g(xo)~~ ∧e 3	
∀x f(x) ∀i 2,4	
~~x1~~ assumption + ∀	
~~f(x1) ∧ g(x1)~~ ∀e 1	
~~f(x1)~~ ∧e 8	
~~g(x1)~~ ∧e 8	
∀x g(x) ∀i 7, 10	
∀x f(x) ∧ ∀x g(x) ∧i 6,11	

Bài toán được chứng minh.

CHƯƠNG IV
ĐÁNH GIÁ VÀ KẾT LUẬN

4.1 Những kết quả đạt được trong luận văn

Hệ thống ND được xây dựng mới từ quá trình nghiên cứu, tìm hiểu logic.

- ✓ Hệ thống giải quyết được hầu hết các bài toán logic vị từ.
- ✓ Giao diện được thiết kế đơn giản, thân thiện và dễ sử dụng đối với user.
- ✓ Đối với bản thân: Tăng khả năng phân tích hệ thống, khả năng lập trình và làm việc nhóm.

4.2 Điểm hạn chế của luận văn

Bên cạnh những điểm đã được, do hệ thống được hiện thực trong luận văn một giai đoạn (tức một học kỳ tại trường) nên không tránh khỏi một số hạn chế như sau:

- ✓ Hệ thống chưa có khả năng "học" lại những bài toán đã được giải.
- ✓ ???

4.3 Hướng phát triển của hệ thống

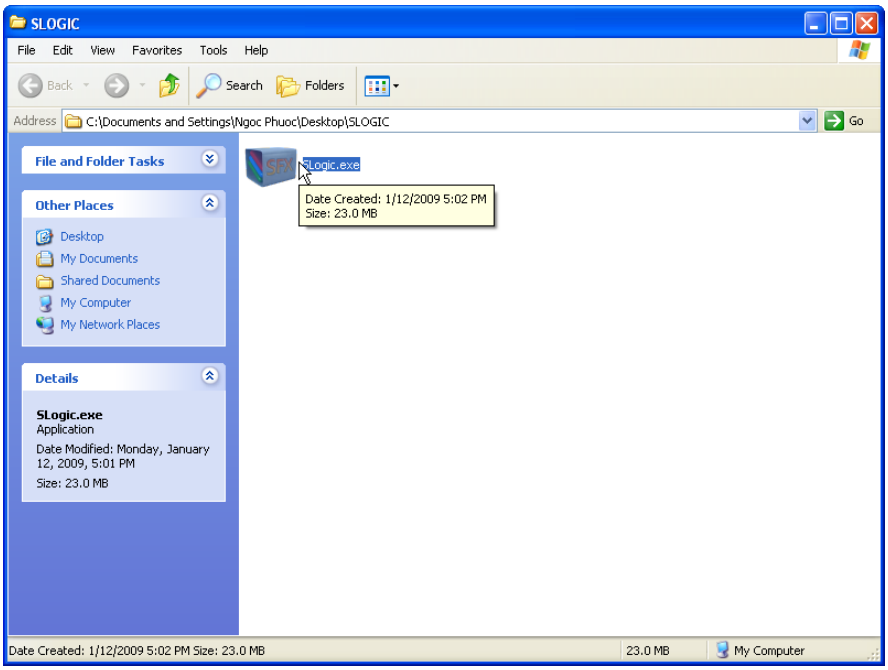
Một số hướng phát triển của hệ thống như sau:

- ✓ Xây dựng một hệ thống suy luận tự nhiên trên ngôn ngữ tự nhiên như đã giới thiệu ở phần đầu.
- ✓ Cải thiện các nhược điểm của hệ thống được nêu ở trên.
- ✓ Có thể đưa thêm phân giải hoặc những phương pháp chứng minh khác vào để hệ thống trở thành "decidable system".

1.Hướng dẫn cài đặt

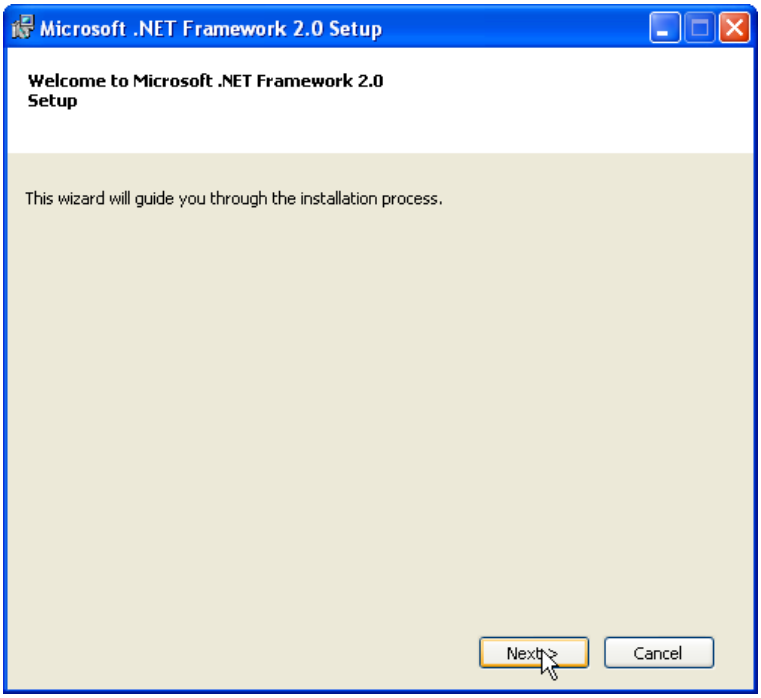
Yêu cầu hệ thống cài đặt hệ điều hành Windows.

DoubleClick vào file cài đặt SLogic.exe.

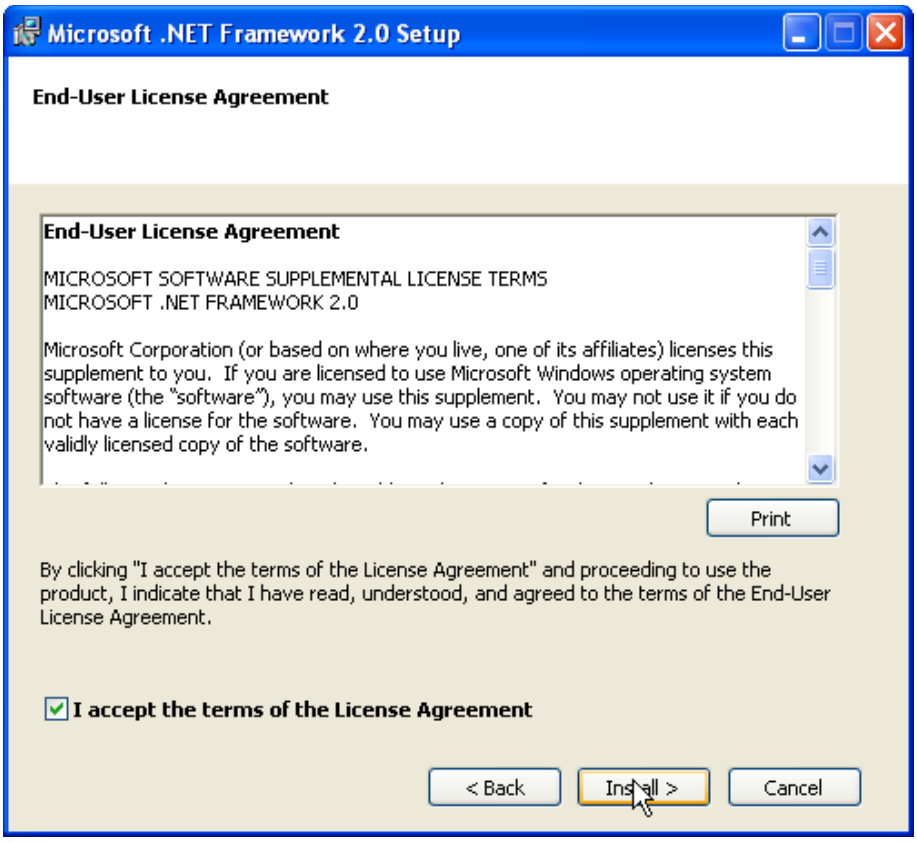


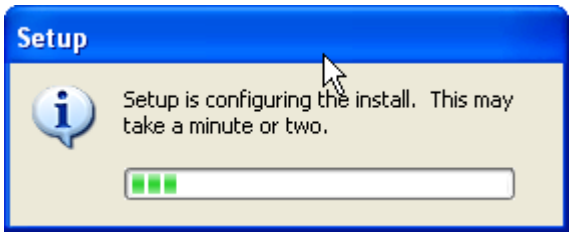
Nếu hệ thống của bạn không có .Net Framework 2.0. Chương trình sẽ tự động cài đặt.

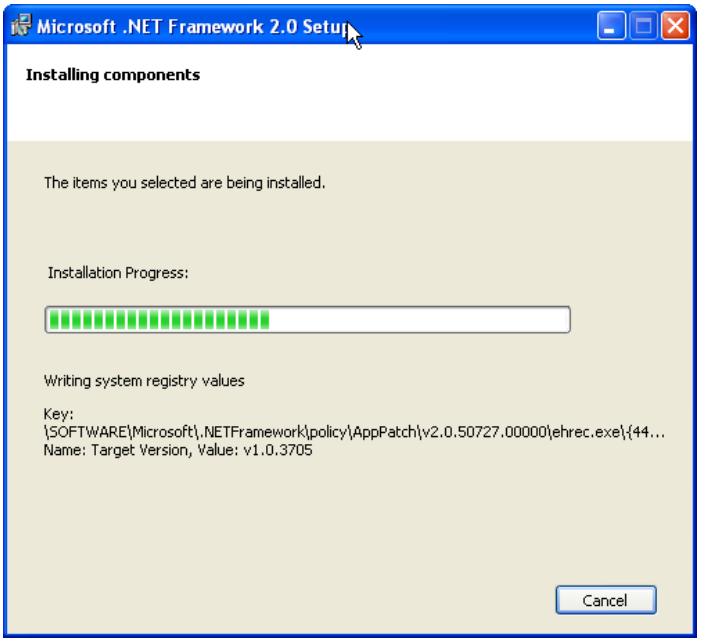
Chọn Next



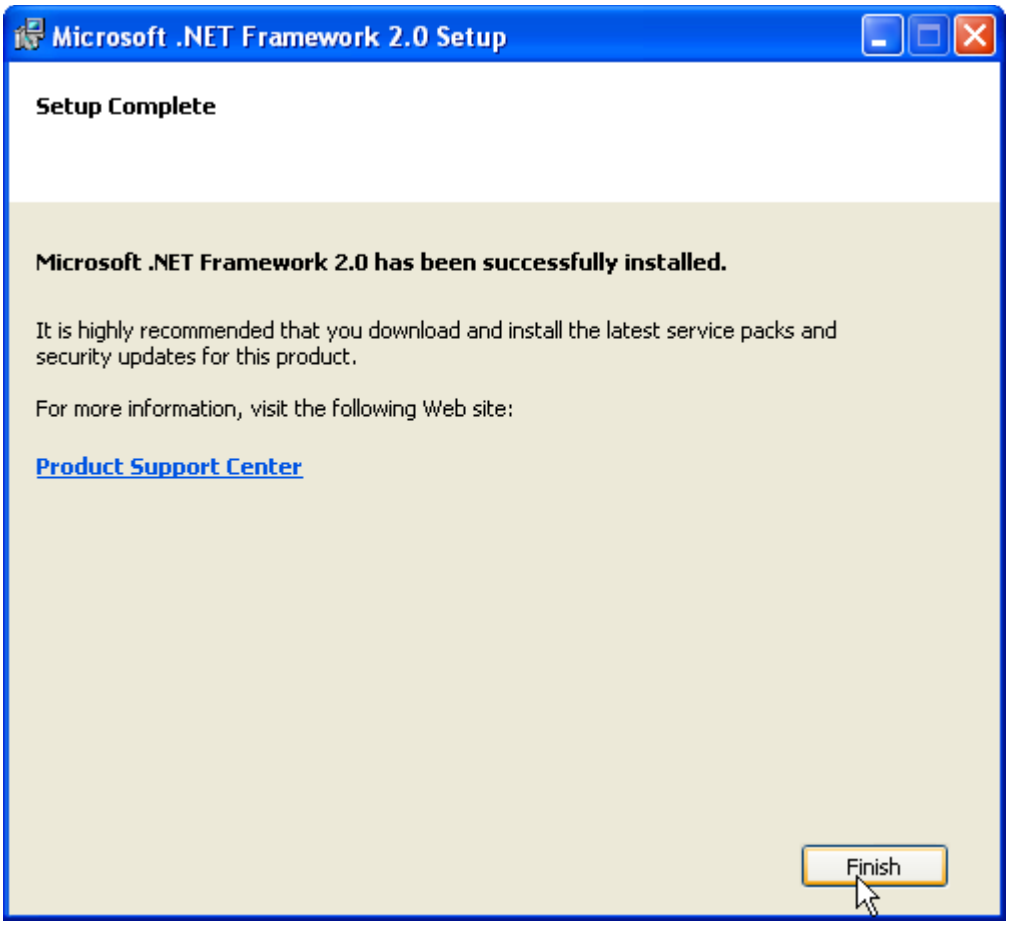
Check vào *I accept the terms of the Licence Agreement* và Click Install.





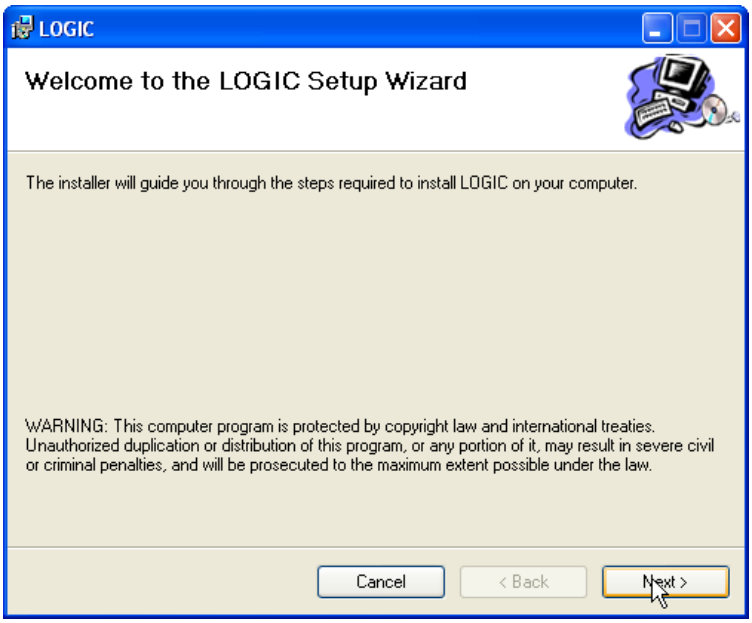


Click Finish để hoàn tất cài đặt .Net Framework 2.0

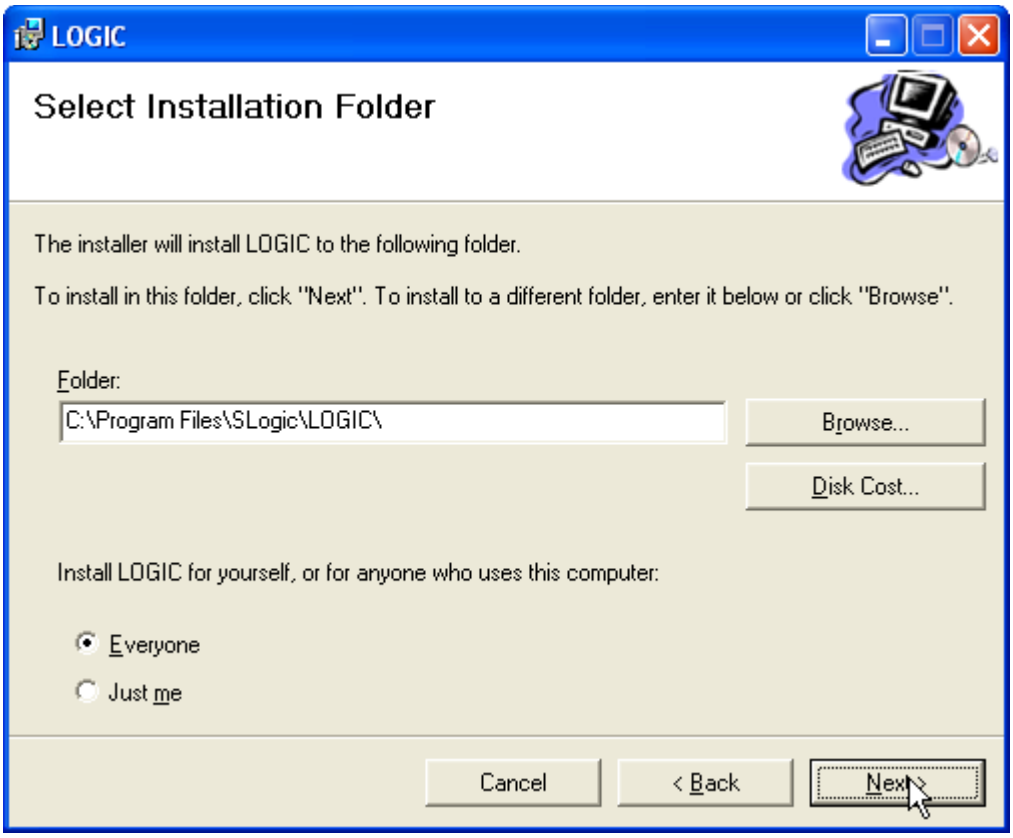


Sau khi cài xong .Net Framework 2.0 hoặc hệ thống đã có sẵn .Net Framework 2.0 thì chương trình sẽ bắt đầu cài SLogic.

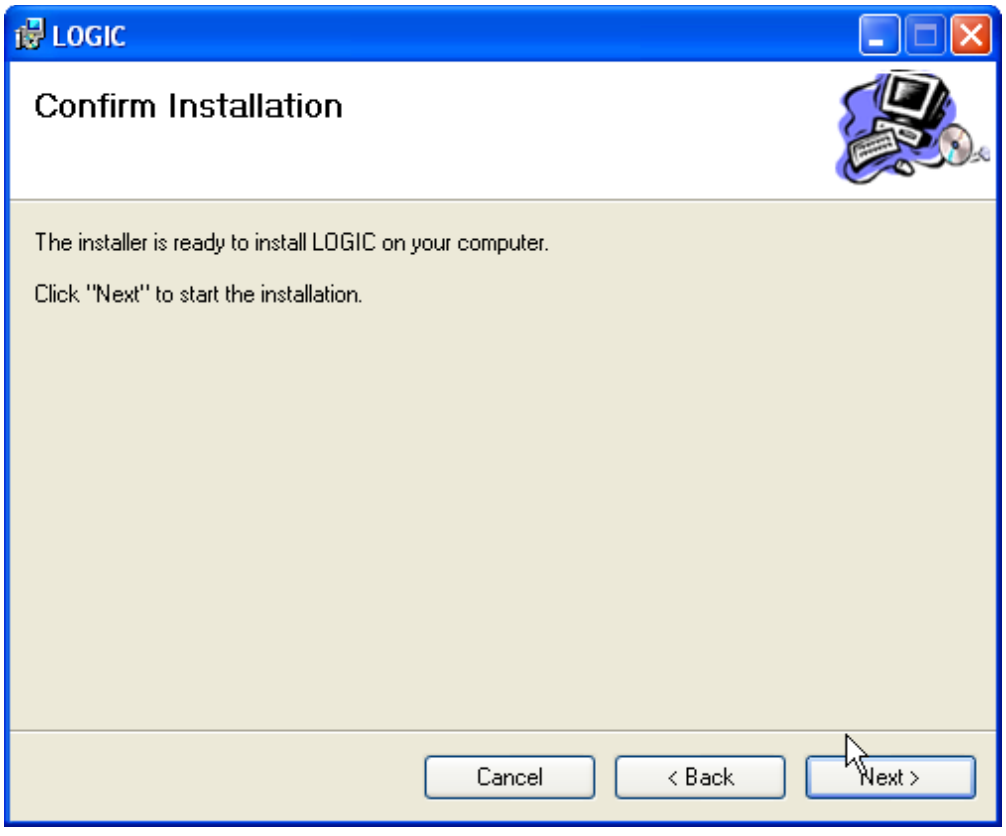
Chọn Next

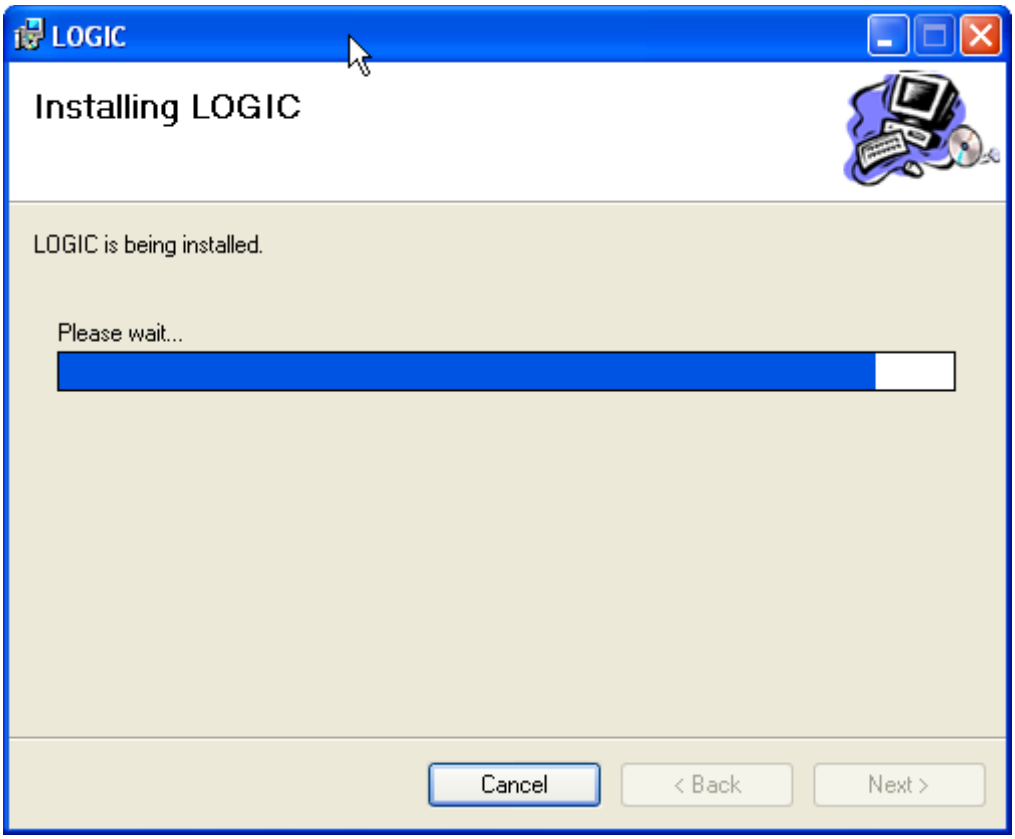


Tiếp tục chọn Next

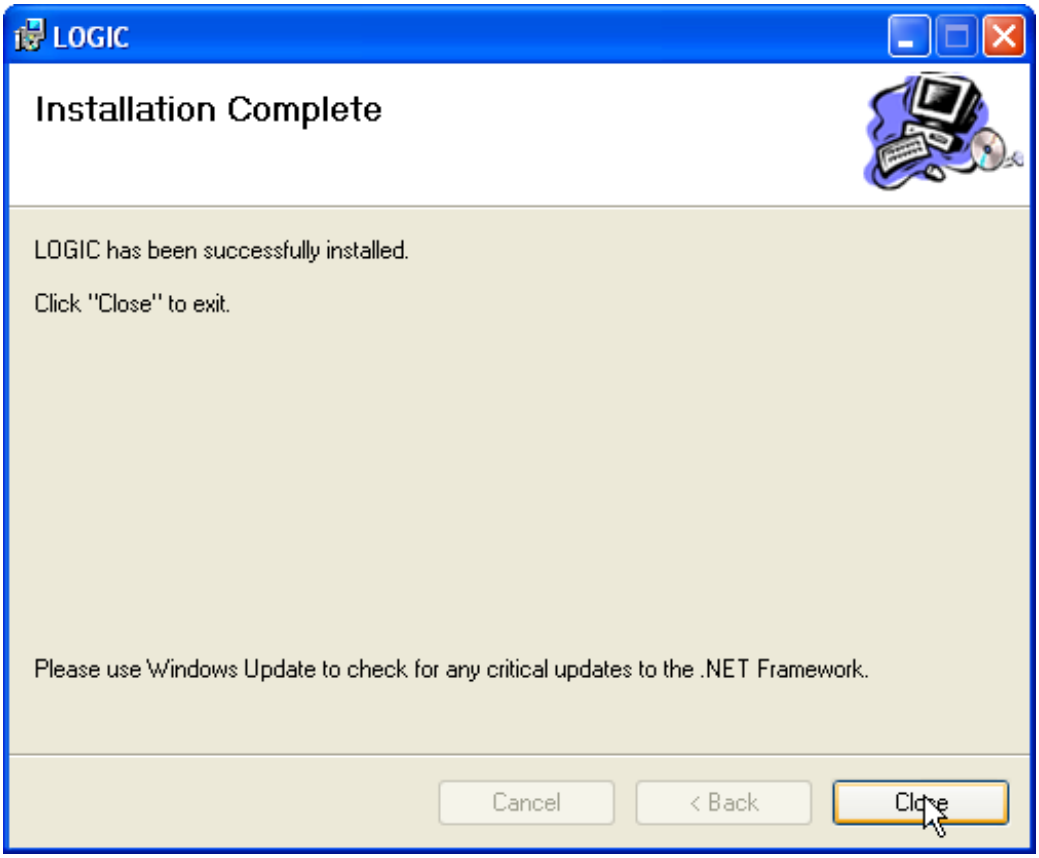


Chọn Next để install



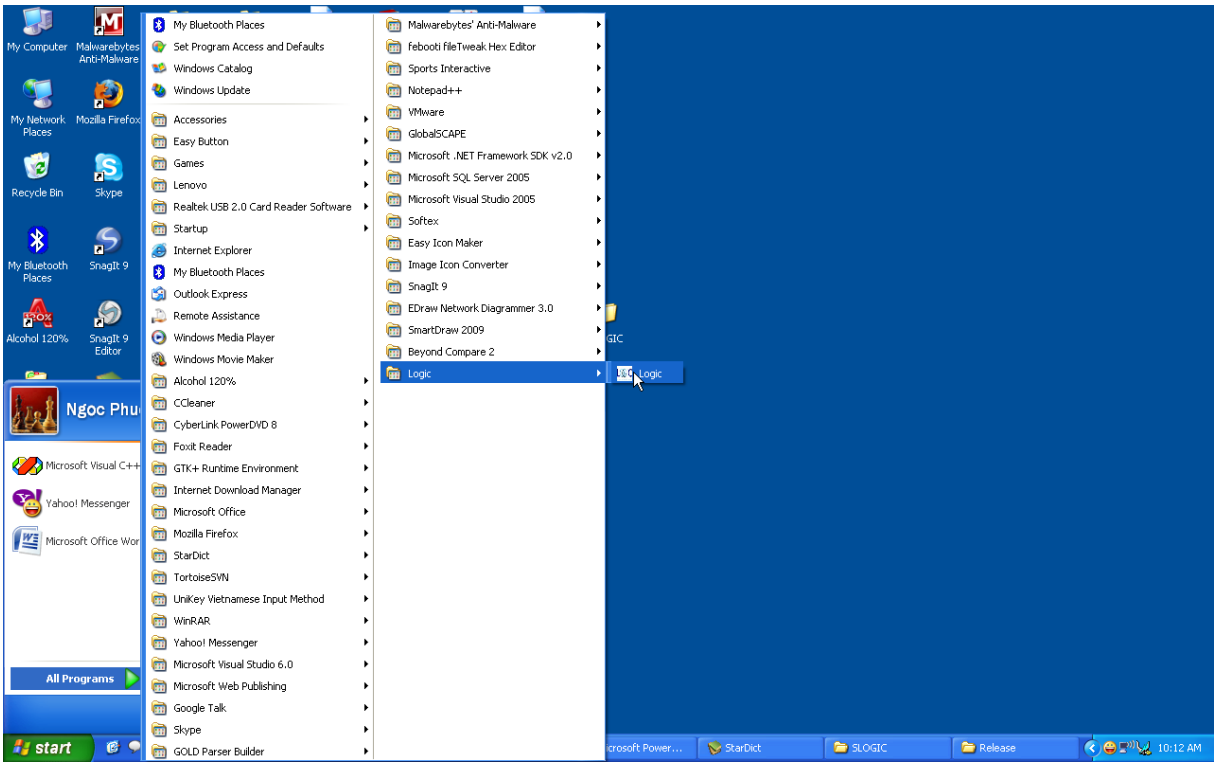


Chọn Close để hoàn tất cài đặt

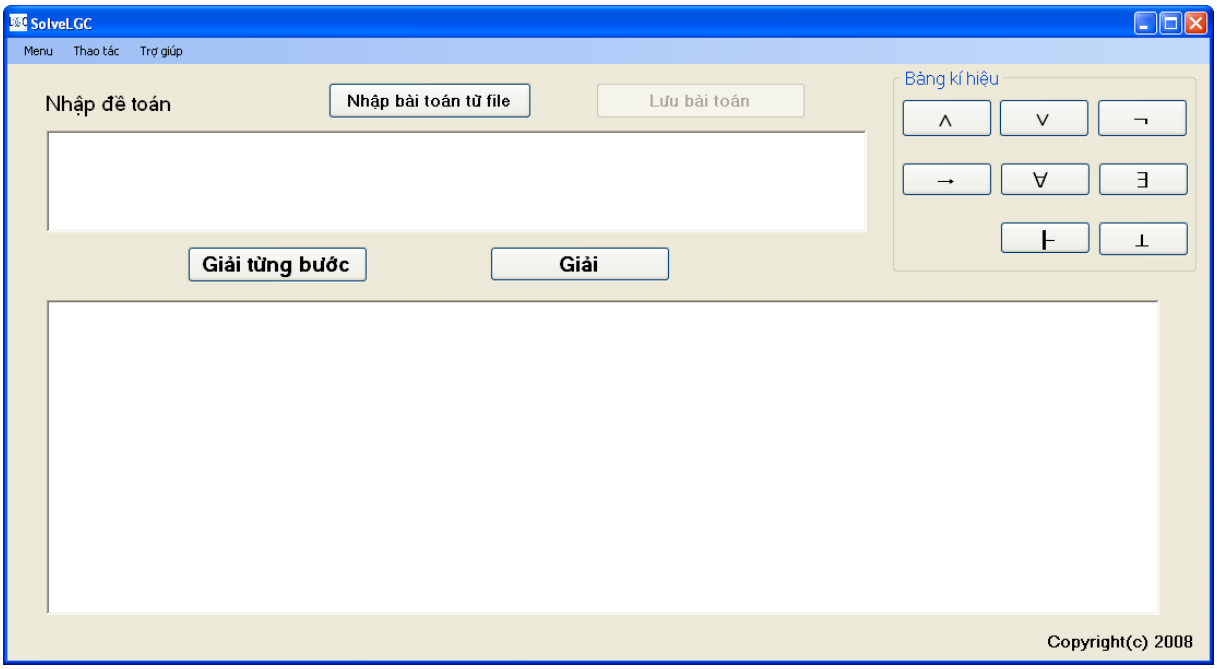


2.Hướng dẫn sử dụng

Vào Start → AllProgram → Logic → chọn Logic.



Giao diện chương trình như sau:



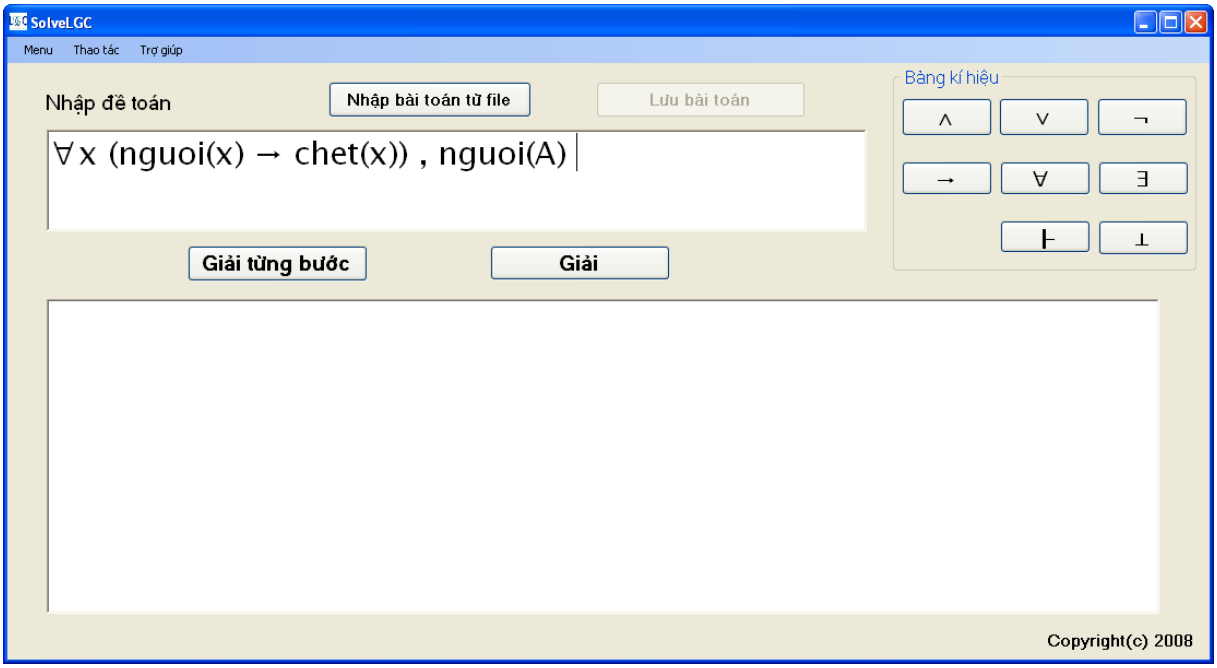
Gõ đề toán vào Nhập đề toán. Các kí tự tương đương trên bàn phím như sau:

Kí hiệu trong logic	Kí tự bàn phím thay thế
∧	&
∨	|
¬	!
→	- >
⊢	|-
∀	\-
∃	-]
⊥	_|_

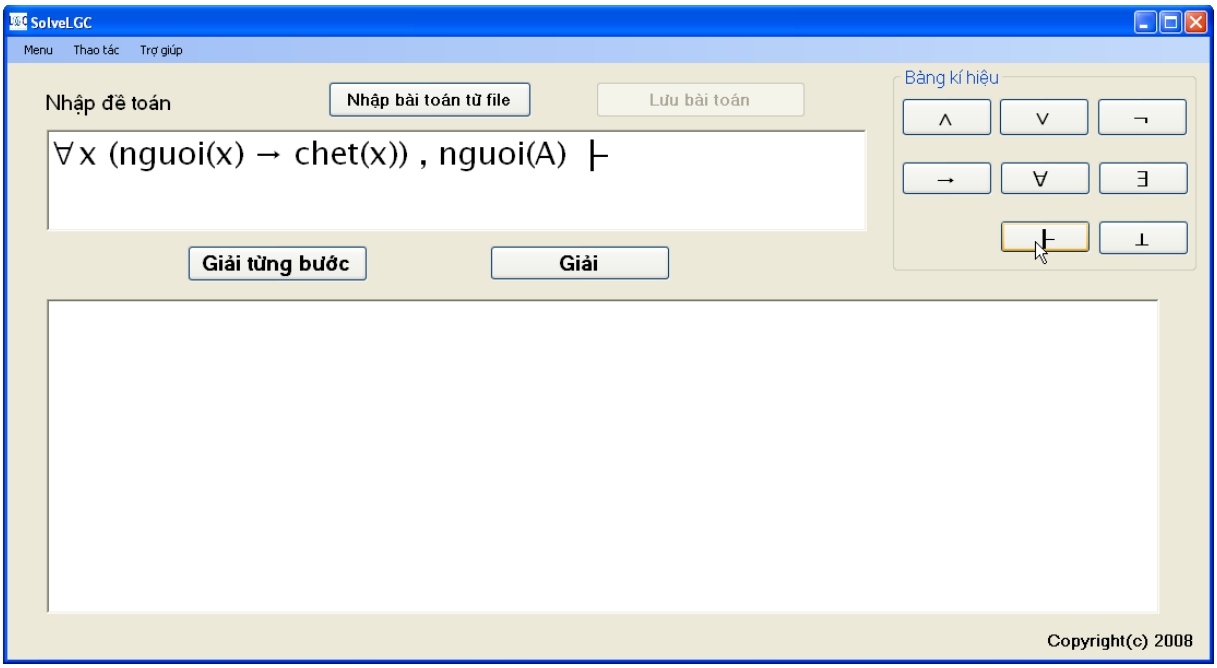
Ví dụ nhập bài toán:

∀ x (nguoi(x) → chet(x)) , nguoi(Socrate) ⊢ chet(Socrate)

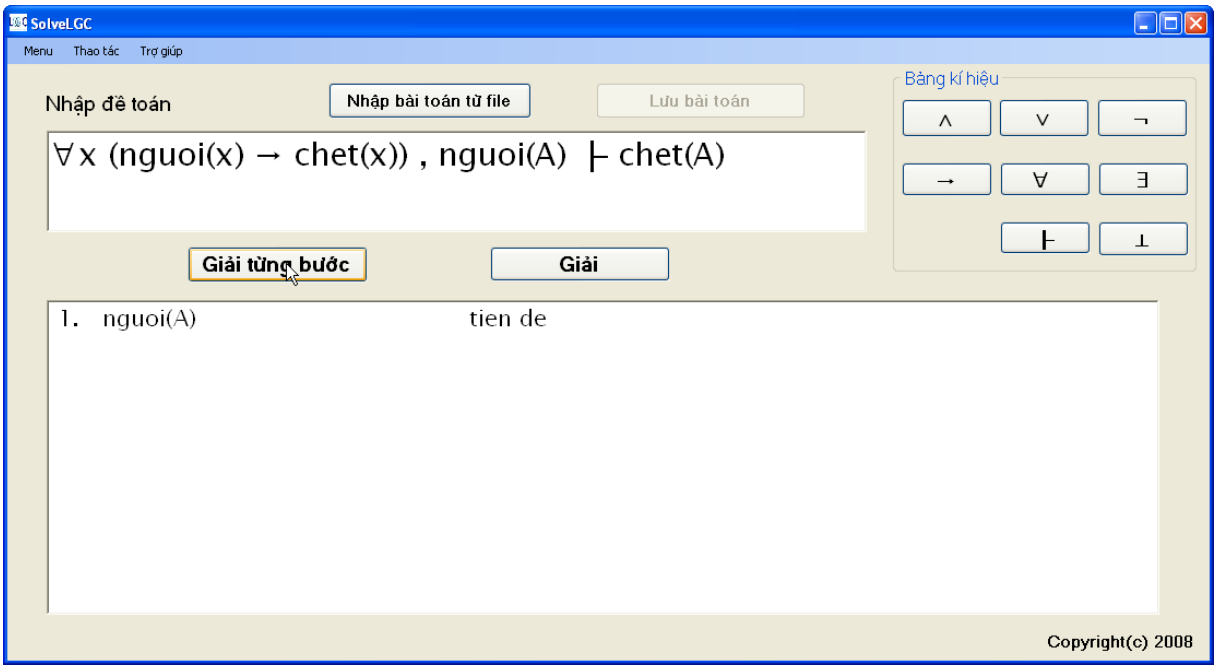
Gõ bài toán vào Nhập đề toán



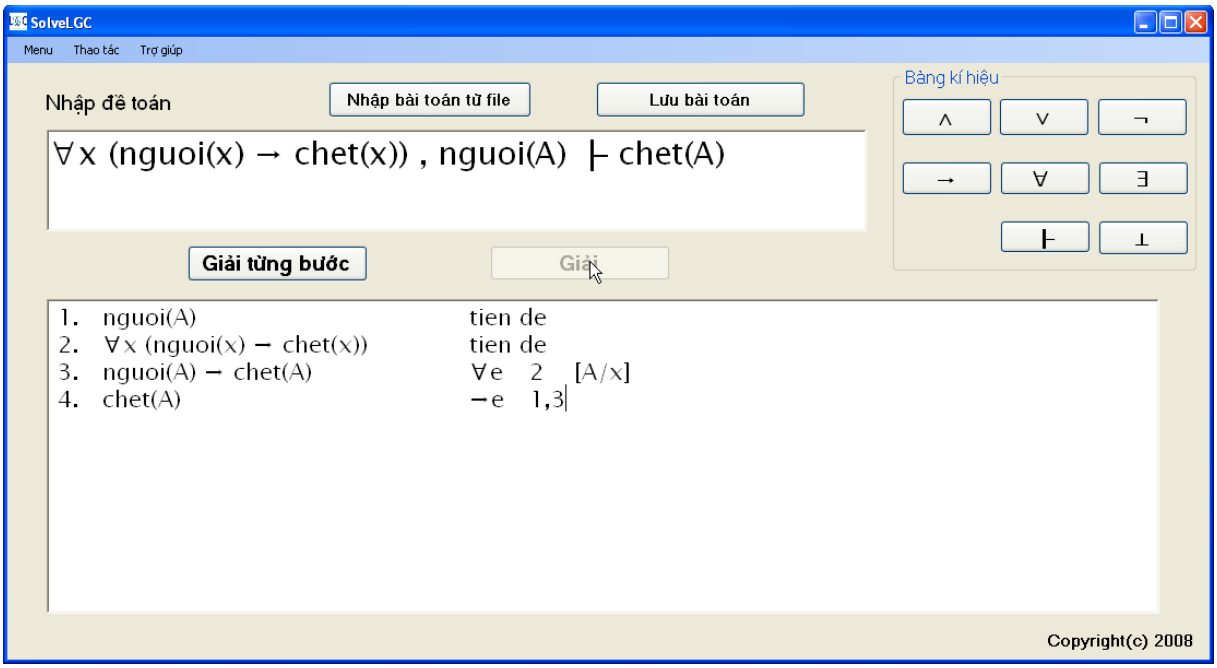
Hoặc có thể click vào các kí tự có sẵn



Sau đó Click Giải từng bước để xem từng bước giải



Hoặc Click Giải để xem toàn bộ bài giải



3.Các chức năng khác

Nhập bài toán từ file: cho phép người dùng nhập đề toán từ một file có sẵn. File có định dạng .lff (logic file format) hoặc .txt (text).

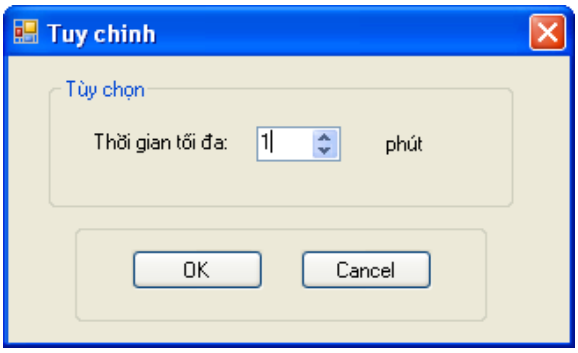
Lưu bài toán: Lưu đề và bài giải vào một file có định dạng .lff (logic file format) hoặc .txt (text).

Menu:

- *Tạo mới:* Xóa hết đề toán và bài giải để có thể nhập mới một bài toán khác.
- *Nhập bài toán từ file:* cho phép người dùng nhập đề toán từ một file có sẵn. File có định dạng .lff (logic file format) hoặc .txt (text).
- *Lưu bài toán:* Lưu đề và bài giải vào một file có định dạng .lff (logic file format) hoặc .txt (text).
- *Lưu đề bài:* Lưu đề vào một file có định dạng .lff (logic file format) hoặc .txt (text).
- *Thoát:* Thoát chương trình.

Thao tác:

- *Cắt:* Cut đoạn text được chọn ở đề toán vào clipboard.
- *Sao chép:* Sao chép đoạn text được chọn ở đề toán vào clipboard.
- *Dán:* Dán đoạn text có trên clipboard vào *Nhập đề toán.*
- *Chọn hết:* Chọn hết *Nhập đề toán.*
- *Tùy chọn:* Tùy chỉnh thời gian tối đa cho phép để chương trình giải một bài toán.



Trợ giúp:

- *Hướng dẫn sử dụng:* Hướng dẫn sử dụng chương trình.
- *Phím tắt:* Các kí tự bàn phím tương đương với các toán tử trong logic vị từ.
- *Tác giả:* Giới thiệu về tác giả.

TÀI LIỆU THAM KHẢO

1. *Luận lý toán học* – Nguyễn Thanh Sơn - Khoa KHMT&CN ĐH Bách Khoa TpHCM
2. *Artificial Intelligence: A Modern Approach* – Stuart Russell and Peter Norvig – Prentice Hall
3. *Warren's Abstract Machine: A Tutorial reconstruction* - HASSAN A¨ IT-KACI
4. *The WAM - Definition and Compiler Correctness* - Egon Borger and Dean Rosenzweig
5. *Principles of Compiler Design* – ULLMAN, JEFFREY D.
6. *Logic in Computer Science : Modelling and Reasoning about Systems* . Michael Huth & Mark Ryan
7. *An Efficient Unification Algorithm by ALBERTO MARTELLI*
8. *Automated Reasoning IJCAR 2004*
9. *Automated Reasoning IJCAR 2008*
10. *A First Course in Logic* SHAWN HEDMAN
11. *www.wikipedia.com*

1. Logic[11]

Logic học là khoa học xuất hiện rất sớm trong lịch sử. Nó xuất hiện vào thế kỷ IV trước công nguyên, khi sự phát triển của khoa học nói riêng và tư duy nói chung đã đòi hỏi phải trả lời câu hỏi: làm thế nào để đảm bảo suy ra được kết luận đúng đắn từ các tiền đề chân thực

Logic hay luận lý học, từ tiếng Hy Lạp cổ điển λόγος (logos), nghĩa nguyên thủy là *từ ngữ*, hoặc *điều đã được nói*, (nhưng trong nhiều ngôn ngữ châu Âu đã trở thành có ý nghĩa là *suy nghĩ* hoặc *lập luận* hay *lý trí*). *Logic* thường được nhắc đến như là một ngành nghiên cứu về tiêu chí đánh giá các luận cứ, mặc dù định nghĩa chính xác của logic vẫn là vấn đề còn đang được bàn cãi giữa các triết gia. Tuy nhiên khi môn học được xác định, nhiệm vụ của nhà logic học vẫn như cũ: làm đẩy mạnh tiến bộ của việc phân tích các suy luận có hiệu lực và suy luận ngụy biện để người ta có thể phân biệt được luận cứ nào là hợp lý và luận cứ nào có chỗ không hợp lý.

Theo truyền thống, logic được nghiên cứu như là một nhánh của triết học. Kể từ giữa thế kỉ 19 logic đã thường được nghiên cứu trong toán học và luật. Gần đây nhất logic được áp dụng vào khoa học máy tính và trí tuệ nhân tạo. Là một ngành khoa học hình thức, logic nghiên cứu và phân loại cấu trúc của các khẳng định và các lý lẽ, cả hai đều thông qua việc nghiên cứu các hệ thống hình thức của việc suy luận và qua sự nghiên cứu lý lẽ trong ngôn ngữ tự nhiên. Tầm bao quát của logic do vậy là rất rộng, đi từ các đề tài cốt lõi như là nghiên cứu các lý lẽ ngụy biện và nghịch lý, đến những phân tích chuyên gia về lập luận, chẳng hạn lập luận có xác suất đúng và các lý lẽ có liên quan đến quan hệ nhân quả. Ngày nay, logic còn được sử dụng phổ biến trong lý thuyết lý luận.

Qua suốt quá trình lịch sử, đã có nhiều sự quan tâm trong việc phân biệt lập luận tốt và lập luận không tốt, và do đó logic đã được nghiên cứu trong một số dạng ít nhiều là quen thuộc đối với chúng ta. Logic Aristotle chủ yếu quan tâm đến việc dạy lý luận thế nào cho tốt, và ngày nay vẫn được dạy với mục đích đó, trong khi trong logic toán học và triết học phân tích (*analytical philosophy*) người ta nhấn mạnh vào logic

như là một đối tượng nghiên cứu riêng, và do vậy logic được nghiên cứu ở một mức độ trừu tượng hơn.

Các quan tâm về các loại logic khác nhau giải thích rằng logic không phải là được nghiên cứu trong chân không. Trong khi logic thường có vẻ tự cung cấp sự thúc đẩy chính nó, môn học này phát triển tốt nhất khi lý do mà chúng ta quan tâm đến logic được đặt ra một cách rõ ràng.

2. Logic vị từ

2.1 Giới thiệu logic vị từ [11]

Môn Logic như được nghiên cứu ngày nay rất khác với môn học đã được nghiên cứu trước đây, và sự khác biệt chính là sự phát minh của logic vị từ (predicated logic or first-order logic). Trong khi logic tam đoạn luận của Aristote định ra những dạng thức cho những phần có liên quan với nhau trong mỗi phán đoán, logic vị từ cho phép các câu được phân tích thành chủ đề và các luận cứ theo nhiều cách khác nhau, do vậy cho phép logic vị từ giải quyết được vấn đề tổng quát hóa nhiều lần - vấn đề đã làm bối rối các nhà logic học thời trung cổ. Với logic vị từ, lần đầu tiên, các nhà logic học đã có khả năng đưa ra các phép lượng hóa (*quantifiers*) đủ tổng quát để diễn tả mọi luận cứ có mặt trong ngôn ngữ tự nhiên.

Sự khám phá ra logic vị từ thường được coi là công của Gottlob Frege, người cũng được xem là một trong những sáng lập viên của ngành triết học phân tích, nhưng dạng phát biểu có hệ thống thông dụng nhất ngày nay của logic vị từ là logic bậc nhất (*first-order logic*) được trình bày trong cuốn sách Các nguyên lý về logic lý thuyết (*Grundzüge der theoretischen Logik*) của David Hilbert và Wilhelm Ackermann vào năm 1928. Tính tổng quát có tính phân tích của logic vị từ cho phép hình thức hóa toán học và đẩy mạnh nghiên cứu về lý thuyết tập hợp, cho phép sự phát triển của cách tiếp cận của Alfred Tarski đối với lý thuyết mô hình; và không quá lời khi nói rằng nó là nền tảng của logic toán học hiện đại.

Hệ thống nguyên thủy của Frege về logic vị từ không phải là bậc nhất mà là bậc hai. Logic bậc hai được bảo vệ mạnh mẽ nhất bởi George Boolos và Stewart Shapiro (trước các phê phán của Willard Van Orman Quine và những người khác).

2.2 Cấu trúc logic vị từ (1)

Bảng ký tự :Tập hợp hữu hạn các ký tự.

Thí dụ :

a, b, c, d, …, z

- **Ký hiệu** : Chuỗi hữu hạn ký tự được dùng để đặt tên cho các khái niệm trong FOL (first-oder-logic).

Thí dụ :

tên biến: x, y, …

tên hàm: cong, nhan, chia, …

Miền đối tượng D : là một tập hợp.

Tập hợp các ký hiệu biến : các biến lấy giá trị trên D.

Lượng từ có 2 loại :

- ✓ Phổ dụng ∀ (universal quantifier)
- ✓ Hiện hữu ∃ (existential quantifier).

Hình thức sử dụng :

(∀x), (∃x) : với x là biến.

Hàm là ánh xạ từ $D^n \rightarrow D$, $n \in N$ N.

Thí dụ :

nhân, cộng : $D \times D \rightarrow D$

Ảnh của hàm được gọi là biểu thức hàm.

Thí dụ :

nhân(2, 3), cộng(x, 6)

cộng(nhân(y, z), 5)

Trường hợp đặc biệt : Hàm từ D0 → D được gọi là hằng.

Trường hợp đặc biệt : Hàm được gọi là vị từ nếu thỏa thêm các điều kiện :

− Có miền ảnh là tập {1, 0}.

− Chỉ kết hợp với nhau qua các toán tử logic : ¬, ∧, ∨, →

− Khi sử dụng không được làm thông số của hàm khác.

- Vị từ

  Thí dụ :

  cha, mẹ, bạn : D × D → {1, 0}

  Ảnh của vị từ được gọi là biểu thức vị từ.

  Thí dụ :

  cha(Minh, Vũ)

  mẹ(x, y)

  bạn(y, z)

2.3 BNF logic vị từ

Sentence → AtomicSentence |Sentence Connective Sentence

| Quantifier Variable, ... Sentence |¬ Sentence |(Sentence)

AtomicSentence → Predicate(Term, …) |Term = Term

Term → Function(Term, …)| Constant | Variable

Connective → ∧ | ∨ | →

Quantifier → ∀ | ∃

Constant → A, B, C, X1 , X2, Jim, Jack

Variable → a, b, c, x1 , x2, counter, position

Predicate → Adjacent-To, Younger-Than,

Function → Father-Of, Square-Position, Sqrt, Cosine

3.Suy luận tự nhiên

3.1 Giới thiệu về suy luận tự nhiên [10]

Trong triết học logic, *suy luận tự nhiên* (natural deduction) là một cách tiếp cận lý thuyết chứng minh gần giống với cách suy luận của con người nhất so với những cách suy luận khác. Suy luận tự nhiên cố gắng cung cấp một hệ thống luận lý là mô hình hình thức của logic luận lý làm cho những lý luận này diễn ra theo một cách *tự nhiên*. Suy luận tự nhiên giữ lại cấu trúc của các công thức theo dạng tự nhiên.

Suy luận tự nhiên là nền tảng cơ bản nhất để tìm hiểu và giảng dạy logic. Suy luận tự nhiên dường như không được sử dụng để áp dụng vào lĩnh vực chứng minh tự động. Thay vào đó, người ta dùng các phương pháp như phân giải (resolution), tableau, dpll…

Mô hình của suy luận tự nhiên là:

$$A \vdash B$$

Trong đó hệ thống phải đi tìm những lập luận tự nhiên để dẫn xuất ra công thức B từ tập công thức A

Tập công thức A có thể trống. Khi đó hệ thống sẽ là:

$$\vdash B$$

và lúc này suy luận B đồng nghĩa với suy luận định lý B.

3.2 Tập luật trong suy luận tự nhiên [6]

Suy luận tự nhiên gồm 2 nhóm luật:

Eliminate rules: gồm những luật dùng để phân rã công thức thành những công thức mới

Introduction rules: gồm những luật dùng để tạo ra công thức mới từ những công thức thành phần.

Danh sách tập luật

Eliminate rules	Introduction rules
$\wedge e_1$ $\frac{A \wedge B}{A}$	$\wedge i$ $\frac{A \ , \ B}{A \wedge B}$
$\wedge e_2$ $\frac{A \wedge B}{B}$	$\vee i_1$ $\frac{A}{A \vee B}$
$\rightarrow e$ $\frac{A, A \rightarrow B}{B}$	$\vee i_2$ $\frac{B}{A \vee B}$
$\vee e$ $\frac{A \vee B, B \rightarrow C \ , A \rightarrow C}{C}$	$\rightarrow i$ $\frac{[A], B}{A \rightarrow B}$
$\neg\neg$ $\frac{\neg\neg A}{A}$	$\neg i$ $\frac{\neg A, \bot}{A}$
$\neg e$ $\frac{A \ , \neg A}{\bot}$	$\forall i$ $\frac{f[\frac{\beta}{x}]}{\forall x \ f(x)}$
$\forall e$ $\frac{\forall x \ f(x)}{f[\frac{\alpha}{x}]}$	$\exists i$ $\frac{f[\frac{\alpha}{x}]}{\exists x \ f(x)}$
$\exists e$ $\frac{\exists x \ f(x)}{f[\frac{\beta}{x}]}$	

Chú thích:

α : là giá trị bất kỳ

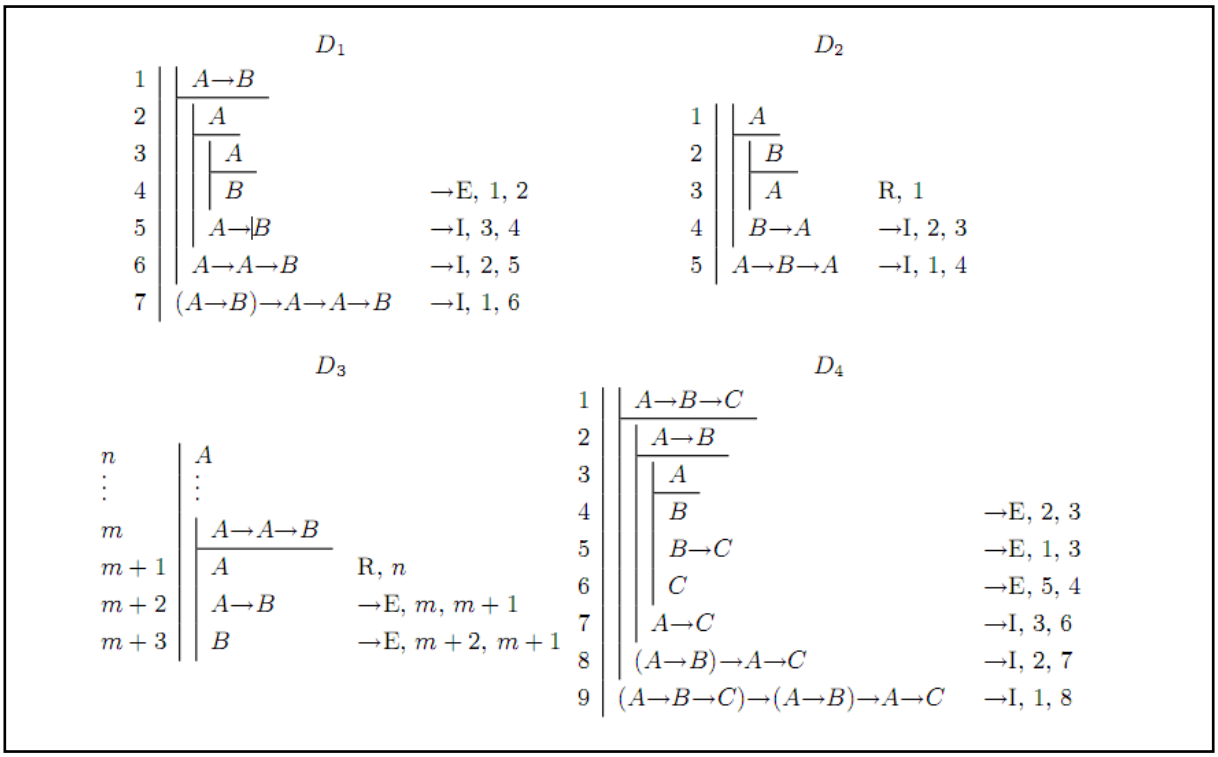
β : là giá trị có ràng buộc

[A] : giả thiết được thêm vào

3.3 Định dạng output của suy luận tự nhiên

Suy luận tự nhiên có hai cách biểu diễn kết quả cổ điển: G-style (do Gerhard Gentzen đề xuất) và J-style (do Stanaslaw Jáskowski đề xuất).

Đối với yêu cầu của đề tài, Ouput của chương trình sẽ theo định dạng J-style, chính xác hơn là F-style [7] (được cải tiến từ J-style bởi nhà logic học người Mỹ Frederic Brenton Fitch).



Định dạng F-style:

Ví dụ: $A \to B \vdash (A \vee H) \to (B \vee H)$

1.	$A \rightarrow B$	tien de	
2.	if $A \vee H$		
3.	if A		
4.	B	$\rightarrow e$	3,1
5.	nif $B \vee H$	$\vee i1$	4
6.	if H		
7.	nif $B \vee H$	$\vee i2$	6
8.	nif $B \vee H$	$\vee e$	2,3,6
9.	$(A \vee H) \rightarrow (B \vee H)$	$\rightarrow i$	2,8