Relatório de Projeto Final de Programação (INF2102)

Bernardo Alkmim
Orientador Edward Hermann Haeusler
Pontifícia Universidade Católica do Rio de Janeiro

2017.2

1 Resumo

Dedução Natural é um método dedutivo que busca aproximar-se do modo como um ser humano organizaria o raciocínio de demonstrações em lógica. Neste projeto, desenvolvo um Provador Automatizado de Teoremas (ou seja, dado um teorema em logíca, o provador se encarrega de realizar todos os passos de demonstração) que utiliza Dedução Natural como método de demonstração. Além disso, sua estrutura interna se baseia em Grafos, e não em Árvores como se imaginaria a priori. Com isso, busco otimizar o uso de espaço em disco utilizado e lidar melhor com demonstrações de tamanho superpolinomial.

2 Introdução

Neste capítulo darei uma introdução ao domínio do problema a ser solucionado, além de explicação de porque é um problema e sua relevância.

2.1 Contexto

Como a ciência da computação é um campo relativamente novo, principalmente com relação a outras áreas das ciências exatas, ela está em processo constante e acelerado de amadurecimento. Técnicas e métodos padronizados surgem com frequência em áreas relacionadas à engenharia de software devido a uma crescente necessidade de validar os sistemas que estão sendo produzidos, com destaque para sistemas distribuídos e outros cujo comportamento não é facilmente previsível. Para validação verdadeiramente formal de programas assim, o que não é amplamente utilizado em informática, em especial com o advento das metodologias ágeis e a necessidade de entrega rápida e constante de produto, não bastam somente padrões de projeto, testes automatizados ou técnicas similares.

Para poder validar sistemas, utilizamos métodos formais. Métodos formais, como o nome sugere, utilizam fundamentos matemáticos para garantir, a nível

teórico, padrões de comportamento e execução de sistemas. Na prática, claro, sempre pode ocorrer alguma interferência física externa ou algo do gênero a qualquer sistema, o que não é necessariamente validado, mas pode ser previsto via análises estatísticas para controle de qualidade, fugindo ao escopo deste trabalho sendo proposto.

Um exemplo de área que utiliza atualmente métodos formais é a de design de circuitos integrados. Depois de um erro, que acabou ficando conhecido como *bug Pentium FDIV*[1], que envolvia falha na unidade de ponto flutuante de processadores Intel Pentium antigos, o uso de métodos formais passou a ser aplicado para validar que operações aritméticas estejam funcionando corretamente.

2.2 Como aplicar métodos formais

Aplica-se métodos formais para validação de sistemas via uso de Lógica. A validação de um sistema acaba se traduzindo por um ou mais teoremas modelados sobre alguma estrutura lógica. Sentenças lógicas fazem parte do dia-a-dia de cientistas da computação, matemáticos, engenheiros, filósofos, especialmente daqueles que lidam muito com métodos formais e têm necessidade de realizar provas de teoremas os mais diversos.

Como muito ocorre em lógica, as demonstrações podem seguir caminhos praticamente (ou completamente) determinísticos. Com, isso, demonstrar tais teoremas em sentenças lógicas, estando elas em Lógica Clássica, Intuicionista, sendo Proposicional ou Primeira Ordem ou demais categorias, pode se tornar um processo longo, trabalhoso que tende a crescer exponencialmente (o que é bastante limitado pelo tamanho das folhas de papel, infelizmente finito) e basicamente mecânico, uma vez que se divide na aplicação de um conjunto razoavelmente pequeno de regras simples (variando, claro, com o *framework* lógico utilizado).

2.3 Dedução Natural

Uma das propostas da ND é se aproximar do modo como um ser humano escolheria resolver um problema. Introduzida como uma alternativa a sistemas de Hilbert, ela pode ser explicada de modo bastante intuitivo pois lida com conceitos que podem parecer óbvios *a priori* para alguém que passou por algum curso de lógica. Há, claro, as deduções mais complexas que necessitam de maior explicação.

Uma sentença lógica em ND assume a forma de uma *introdução* ou *eliminação* de um operador lógico. A introdução dos operadores se refere a dar uma ou mais premissas e gerar um operador lógico na conclusão da sentença. A eliminação realiza o processo contrário, eliminando um operador lógico das premissas gerando uma conclusão *menor*, por assim dizer. Utilizando vários desses passos, podemos realizar demonstrações de teoremas.

[1] Mais informações em
`https://web.archive.org/web/20060209005434/http://www.byte.com:80/art/9503/sec13/art1.htm`
(acessado em 27/11/2016).

A ND pode ser apresentada principalmente via árvores de Gentzen, que será o modo de representação escolhido para este trabalho, por não haver o problema físico do espaço limitado de uma folha de papel, ou via formato de Fitch[2], que é mais linear e vertical, mas pode ser mais convoluto. Seguem alguns exemplos, seguindo a notação de Gentzen, utilizada em [1] e [2]:

- $\rightarrow$-Eliminação (também conhecido por *Modus Ponens*)

$$\frac{A \quad (A \rightarrow B)}{B} \ (\rightarrow E)$$

  Na $\rightarrow$-Eliminação, temos uma prova de que A é verdade, e temos que $A \rightarrow B$ é verdade. Com isso, fica claro ver que podemos concluir B verdadeiro (eliminando, assim, o operador $\rightarrow$ que ali havia).

Em ND para Lógica Proposicional há regras de introdução e eliminação de $\wedge$, $\rightarrow$, $\vee$, $\neg$, além de uma regra para o absurdo ($\perp$). Em especial, a regra do absurdo tem uma versão para Lógica Clássica (*reductio ad absurdum*) e outra para Lógica Intuicionista (*ex falso quodlibet*), evidenciando ainda mais a necessidade de haver diferentes tipos de provadores de teoremas, ou ao menos provadores que tenham comportamentos diferentes para tipos diferentes de lógicas.

Para Lógica de Primeira Ordem, além das regras da lógica proposicional, há também regras de introdução e eliminação de $\forall$ e $\exists$.

Neste projeto, focaremos em Lógica Minimal Implicacional, utilizando apenas o operador $\rightarrow$.

Há outro conceito que também é utilizado em ND: o *descarte de hipótese*. O descarte de hipótese ocorre quando um passo da prova depende de alguma outra hipótese que não as que se encontram diretamente nas premissas (ou seja, ele depende de uma hipótese). Esse mecanismo é melhor explicado via um exemplo:

- $\rightarrow$-Introdução

$$\frac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{(A \rightarrow B)} \ (\rightarrow I)$$

  Na $\rightarrow$-Introdução, se temos alguma prova de A e a partir de alguma derivação dela chegamos a uma prova de B, podemos concluir que A implica em B, ou seja, $A \rightarrow B$, e descartar essa prova de A. Informalmente, podemos dizer que de A *chegamos* a B por algum caminho ou sequência de passos dedutivos.

[2] A notação de Fitch pode ser vista em `http://www.iep.utm.edu/nat-ded/` (acessado em 27/11/2016).

2.3.1 Porque usar Dedução Natural

Dedução Natural se propõe a ser um sistema dedutivo mais natural para o tratamento de lógica, mais próximo do estilo no qual um ser humano comum encara razão lógica, conforme dito anteriormente.

Ao invés de ter uma regra *mágica* aplicada várias vezes, a dedução natural busca ter a demonstração de uma sentença como algo legível e de fácil compreensão. Entendamos *fácil compreensão* aqui para alguém da área de lógica. Em outras palavras, dedução natural não envolve conceitos muito abstratos ou específicos demais, ao mesmo tempo em que busca legibilidade. Isso é algo que outros métodos acabam pegando por um lado ou outro: ou envolvem conceitos muito específicos, ou acabam tendo a demonstração quase ilegível por seres humanos.

Um exemplo é o Cálculo de Sequentes: ele envolve conceitos muito abstratos para produzir um esquema de provas extremamente eficiente computacionalmente, mas não muito legível, por carregar praticamente todo o contexto da prova a cada passo. Ele é, na verdade, um dos métodos mais utilizados para confecção de ATPs, mas suas provas são em geral tratadas como caixas pretas, dado que lê-las requer o uso de papel e caneta para organização dos pensamentos quase que constantemente. Provas em Dedução Natural são localmente autocontidas (com a exceção de descartes de hipóteses, que são sempre indexados para facilitar a busca do termo ou passo de prova a que se referem).

3 Especificação

3.1 Objetivos e Requisitos

Para este projeto, foi feito um sistema que utiliza interface gráfica para interação com usuário, cujo objetivo é receber como entrada alguma sentença lógica do usuário e retornar uma demonstração da veracidade dessa sentença lógica via Dedução Natural. Também é mostrado um grafo que representa a prova na interface do programa.

O sistema foi feito na linguagem Lua (versão 5.2) e a interface gráfica em Löve, um framework gráfico para Lua, e foi desenvolvido utilizando o editor de texto Sublime no sistema operacional Ubuntu 16.04.

Na categoria de requisitos, o necessário para o sistema é:

- Oferecer completo suporte para que o usuário entre com fórmulas em lógica minimal implicacional.
- O sistema deve gerar provas em LaTeX, em dedução natural, para a entrada do usuário.
- O usuário precisa das provas geradas em formato `pdf` e `jpg`.
- De modo a tornar mais explícito o funcionamento do sistema ao usuário, deve ser mostrado um grafo que representa a estrutura interna corrente da prova. Conforme ela é alterada, esse grafo também deve ser.

- Deve haver sugestões de fórmulas pré-escritas para que o usuário entre com elas bastando apretar um número sequencial, para facilitar a interação.
- O sistema deve ter uam parte genérica de grafos e uam parte em dedução natural que o estende. Assim, ele se torna extensível e adaptável para diferentes táticas dedutivas (como, por exemplo, cálculo de sequentes).
- O sistema deve apresentar as provas geradas também em DOT (requisito novo, ainda em desenvolvimento).

3.2 Casos de Uso

O usuário imaginado aqui é sempre o mesmo: um pesquisador ou estudante buscando gerar uma prova para um teorema em lógica minimal implicacional que ele tem em mente (ou então verificar se é realmente um teorema). Como existem dois modos de entrar com uma fórmula e duas possíveis saídas do programa, há 4 casos básicos de uso (com um caso extra de o usuário não entrar com a fórmula corretamente):

1. Usuário escreve uma fórmula que é um teorema.

   O usuário abre o programa e imediatamente começa a escrever a fórmula (automaticamente transferida para o campo de texto de entrada). Ao terminar, ele aperta a tecla `Enter` de seu teclado.

   Pode-se ver que a visualização do grafo foi alterada de acordo. Ele, então, clica em `Expand All`. Esperando geralmente menos de um segundo, ele verá o grafo alterado, já com a prova gerada. Ao clicar em `Print Proof`, abrir-se-á em seu navegador de internet padrão um arquivo html contendo uma imagem com a prova gerada.

2. Usuário usa um dos atalhos para uma fórmula que é um teorema.

   O usuário abre o programa e imediatamente começa a escrever um dos números que representam uma das fórmulas que ele vê em sua tela. Ao terminar, ele aperta a tecla `Enter` de seu teclado.

   Pode-se ver que a visualização do grafo foi alterada de acordo. Ele, então, clica em `Expand All`. Esperando geralmente menos de um segundo, ele verá o grafo alterado, já com a prova gerada. Ao clicar em `Print Proof`, abrir-se-á em seu navegador de internet padrão um arquivo html contendo uma imagem com a prova gerada.

3. Usuário escreve uma fórmula que não é um teorema.

   O usuário abre o programa e imediatamente começa a escrever a fórmula (automaticamente transferida para o campo de texto de entrada). Ao terminar, ele aperta a tecla `Enter` de seu teclado.

   Pode-se ver que a visualização do grafo foi alterada de acordo. Ele, então, clica em `Expand All`. Esperando geralmente menos de um segundo, ele verá o grafo alterado, já com a prova gerada. Ao clicar em `Print Proof`,

abrir-se-á em seu navegador de internet padrão um arquivo html contendo uma imagem com a prova gerada. Nesta prova, haverá uma fórmula em vermelho, explicitando o ramo da prova que não pode ser completamente expandido, o que por sua vez indica que não foi possível gerar uma prova para o teorema inserido.

4. Usuário usa um dos atalhos para uma fórmula que não é um teorema.

   O usuário abre o programa e imediatamente começa a escrever um dos números que representam uma das fórmulas que ele vê em sua tela. Ao terminar, ele aperta a tecla `Enter` de seu teclado.

   Pode-se ver que a visualização do grafo foi alterada de acordo. Ele, então, clica em `Expand All`. Esperando geralmente menos de um segundo, ele verá o grafo alterado, já com a prova gerada. Ao clicar em `Print Proof`, abrir-se-á em seu navegador de internet padrão um arquivo html contendo uma imagem com a prova gerada. Nesta prova, haverá uma fórmula em vermelho, explicitando o ramo da prova que não pode ser completamente expandido, o que por sua vez indica que não foi possível gerar uma prova para o teorema inserido.

5. Falha na escrita da entrada.

   Entrada do usuário: alguma fórmula escrita incorretamente (ver seção 6.1). O sistema avisa do erro de parsing e nada amis acontece. Basta que o usuátio clique em `Input Formula` para limpar o buffer de entrada e entrar com a fórmula corretamente, caindo em algum dos 4 primeiros casos.

4 Projeto de Programa

Neste capítulo serão mostrados a arquitetura e os módulos do projeto, bem como suas características mais marcantes e como se dá sua interação com o usuário.

4.1 Módulos e arquivos

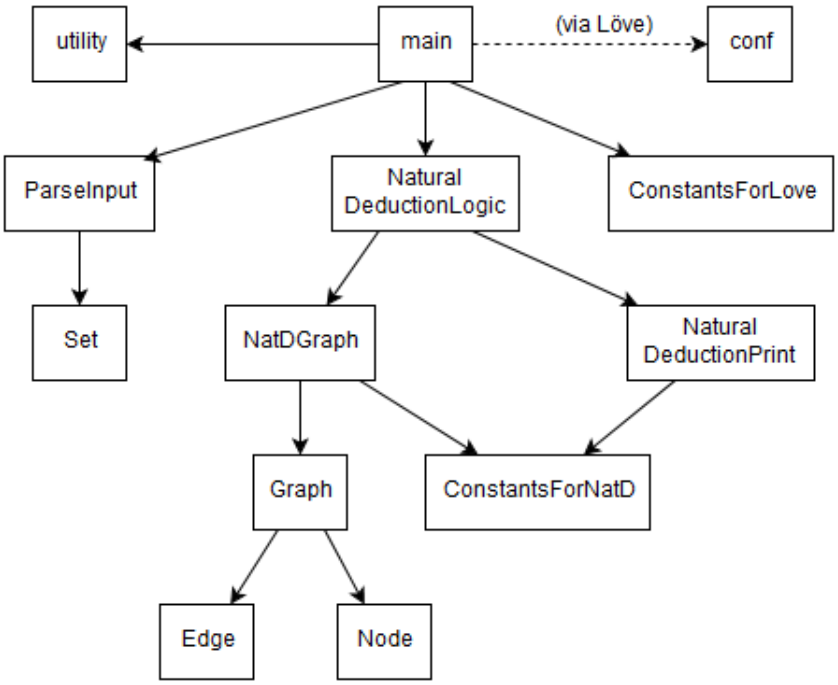

Figura 1: Arquitetura do projeto

O programa está dividido em 13 (com um 14º em desenvolvimento) módulos, divididos hierarquicamente conforme o diagrama acima. As setas no diagrama representam comandos `require` de Lua, e representam dependência de módulos, à exceção do módulo `conf`, cuja relação com o módulo `main` vem internamente via o framework Löve. Veja a estrutura hierárquica dos módulos na figura 1.

Os módulos estão divididos de acordo com funcionalidade principal, e a hierarquia deles tem o objetivo de seguir as lições de Engenharia de Software aprendidas ao longo do curso de graduação.

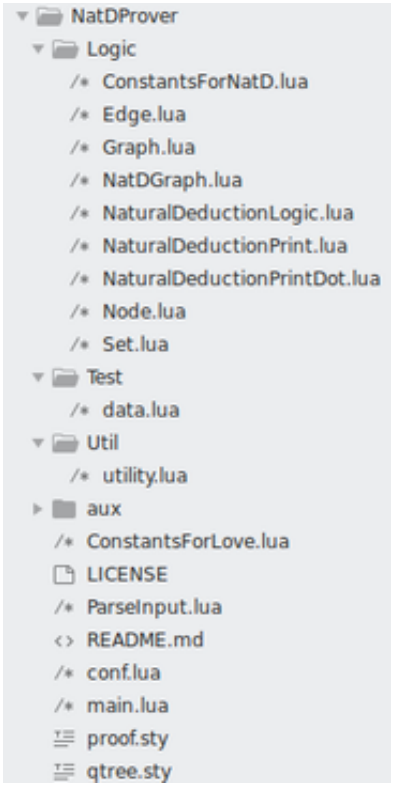

Figura 2: Estrutura em árvore dos diretórios do projeto

Na figura 2 podemos ver a estrutura dos arquivos do projeto. Seguem suas funcionalidades:

- main.lua

  Módulo principal da aplicação. Ele contém as funções de lidar com a interface de Löve e nele se ligam os outros módulos.

- conf.lua

  Módulo de Löve. Contém as configurações da janela para funcionamento do programa no framework.

- ParseInput.lua

  Módulo que gera uma árvore de sintaxe abstrata a partir da entrada (em formato string) do usuário. Utiliza-se aqui `lpeg` para a gramática do parser.

- ConstantsForLove.lua

  Contém constantes para o módulo principal, a fim de evitar a ocorrência de excessivos números "mágicos"e strings constantes pelo código.

- Util/utility.lua

  Módulo auxiliar para manipular strings e lidar com debug do código.

- Logic/Edge.lua, Logic/Graph.lua, Logic/Node.lua

  Módulos para criação de grafos genéricos.

- Logic/Set.lua

  Módulo para criação de estruturas que simulem conjuntos (de Teoria dos Conjuntos).

- Logic/NatDGraph.lua

  Módulo para criação do grafo específico para dedução natural.

- Logic/ConstantsForNatD.lua

  Contém strings para serem utilizadas por todo o código.

- Logic/NaturalDeductionPrint.lua

  Módulo cujo objetivo é imprimir o grafo atual (quer ele represente a demonstração completa ou não) em LaTeX de modo recursivo, utilizando o pacote **proof**.

- Logic/NaturalDeductionPrintDot.lua

  Módulo cujo objetivo é imprimir o grafo atual em formato DOT. Está atualmente em desenvolvimento.

- Logic/NaturalDeductionLogic.lua

  Módulo com a maior parte da funcionalidade da aplicação. Aqui estão funções para as regras de eliminação e introdução da implicação, além da estrutura que guarda as descargas de hipótese e os mecanismos para lidar com elas (e manipular o grafo da demonstração de acordo).

- Test/data.lua

  Este arquivo não é um módulo da aplicação, ele é simplesmente carregado para gerar as sugestões de fórmula na tela para o usuário.

4.2 Características marcantes

O destaque deste ATP é o fato de ele ser baseado em Dedução Natural e ter como estrutura interna um grafo, e não uma árvore, como outros ATPs baseados em Dedução Natural. Os nós de predicados lógicos atômicos não são repetidos. Assim, ele minimiza a quantidade de espaço utilizada, otimizando o uso de memória.

Outro ponto bom é que por ser feito em Lua temos acesso à biblioteca gráfica Löve, que permite que façamos uma interface bastante potente e configurável, e com uma curva de aprendizado pequena.

4.3 Estrutura do grafo da demonstração

O grafo que representa uma demonstração será composta por sequências de predicados lógicos e passos dedutivos alternadamente. Como não repetimos nós que representam predicados atômicos, é comum a ocorrência de ciclos no grafo. isso não é um problema pois tanto as arestas quando os nós apresentam *labels*.

Neste grafo os passos dedutivos são Eliminação da Implicação e Introdução da Implicação. Seguem exeplos de como esses passos são representados no grafo, utilizando os exemplos vistos na seção 2.3:

- Eliminação da Implicação

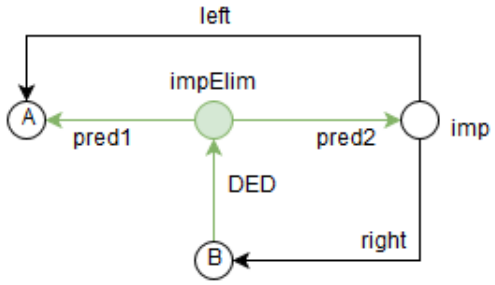

Figura 3: Exemplo de um nó de →-Elim no grafo.

Aqui, os nós e arestas na cor verde são os que são criados ao aplicarmos a →-Elim sobre o nó de *label* B.

- Introdução da Implicação

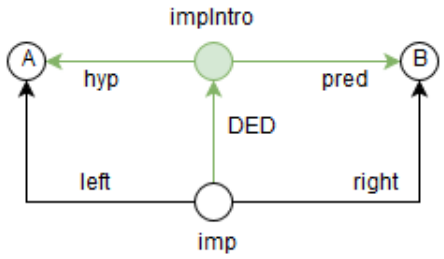

Figura 4: Exemplo de um nó de →-Intro no grafo.

Aqui, os nós e arestas na cor verde são os que são criados ao aplicarmos a →-Intro sobre o nó de *label imp*, que representa a implicação de A para B. Vale notar que a →-Intro por definição só pode ser aplicada em nós que sejam de Implicação.

Aplicando seguidamente esses passos sobre os nós de predicados lógicos obtemos a demonstração geral do teorema inicial que foi digitado pelo usuário.

4.4 Algumas outras decisões de projeto

Quando uma prova é válida, vemos que todos os ramos possíveis "fecham"com descartes de hipótese, o que não ocorre em provas consideradas inválidas (ou, em tom menos presunçoso, provas para as quais o ATP não conseguiu verificar a validade). Nelas, os ramos ficam abertos. Nesses casos, os ramos abertos são coloridos na prova em LaTeX da cor vermelha. A seguir vemos um exemplo disso:

$$\dfrac{\dfrac{\overset{A}{}}{(B \to A)} \to intro_2}{((A \to B) \to (B \to A))} \to intro_1$$

Acima temos uma demonstração que não pôde ser finalizada (ramo aberto e tentativas de expansão todas esgotadas sem sucesso). Conforme esperado, quando A implica em B, não necessariamente o contrário ocorre. Portanto, o teorema não é necessariamente válido.

O grafo gerado contém apenas o essencial à prova, com estruturas de dados auxiliares sendo mantidas fora dele.

4.5 Heurísticas e estruturas auxiliares utilizadas

Para realizar a aplicação sucessiva e automática das regras de eliminação e introdução, é necessário utilizar heurísticas para saber qual dos dois possíveis passos de prova aplicar a cada momento. Para isso, a ríncipal fonte de inspitação foi [3].

Para tal, foi criada uma lista com os ramos abertos da árvore da prova, que é atualizado a cada aplicação de passo de dedução. Com essa estrutura, ficamos ao final com 3 estruturas principais da aplicação: o grafo com a demonstração em si, a lista de hipóteses a descartar (que é também utilizada como lista de *Goals*, ou objetivos da demonstração) e a lista de ramos abertos da árvore.

A função responsável por realizar a a expansão da demonstração é a função principal `LogicModule.expandAll` do módulo `NaturalDeductionLogic`. Para lembrar como funciona cada passo da demonstração, reveja a seção 2.3. A demonstração em si é realizada em duas etapas:

1. Aplicar sucessivamente passos de →-Intro partindo da entrada do usuário até chegar a uma fórmula atômica, ou seja, até que não se possa mais aplicar →-Intro, pois ela requer uma implicação presente na conclusão. Vale notar que a cada →-Intro adiciona-se uma hipótese nova à lista de hipóteses a descartar.

   Essa etapa tem dois principais objetivos: evitar crescimento desnecessário da prova, dado que a →-Intro não cria novos ramos abertos, apenas substitui um ramo antigo por um novo, e gerar uma base grande de hipóteses a descartar, para poder utilizar posteriormente na prova.

   Outro ponto bom dessa parte da demonstração é a facilidade de implementação, pois aplica-se →-Intro para todos os casos.

2. Decidir entre →-Elim e →-Intro para aplicar aos ramos abertos da demonstração. Essa parte é a que contém toda a parte de tomada de decisão da função.

   Aqui, verificamos se, partindo do nó corrente a cada iteração, se ele é *alcançável* (i.e. caso seja possível deduzi-lo após uma iteração, o que verificamos com a função `isGoalReachable`) pelo nó objetivo, que sempre é extraído da lista de hipóteses a descartar. Caso o seja, verificamos se é um nó de implicação, pois, caso não o seja (i.e. um nó atômico) a única opção é aplicar a →-Elim. Caso contrário, damos preferência à →-Intro por esta não aumentar a ramificação da prova.

   Caso não seja alcançável, tentamos na iteração seguinte com alguma subfórmula do nó objetivo. Se após repetidas tentativas de encontrar subfórmulas chegarmos a um predicado atômico e não conseguirmos encontrar relação entre os nós, passamos a tentar uma relação com o objetivo seguinte da lista de hipóteses a descartar.

   Se chegarmos ao fim da lista de hipóteses a descartar e não conseguirmos de modo algum encontrar alguma relação com o nó corrente, marcamos que não foi impossível encontrar solução e encerramos a demonstração, marcando o ramo como `Invalid`. Para tal, é necessário que passemos, para um mesmo nó, até 2^i vezes, onde i é o número de fórmulas atômicas (esse é o limite teórico de expansões para um nó, visto em [4]) e não tenhamos conseguido expandir. Esse nó que não pôde ser expandido aparecerá em vermelho na saída em LaTeX, para evidenciar que não foi possível validar o teorema devido àquele nó.

Sempre, a cada passo aplicado, verifica-se se os nós superiores gerados estão presentes na lista de hipóteses a descartar, pois assim já se fecha um ramo da prova, encurtando seu tamanho. Ao aplicar cada passo, verifica-se se há ainda algum ramo aberto. Caso não exista mais ramos abertos, ou caso exista ramo aberto mas que não pôde ser fechado, a prova é terminada. No primeiro caso, temos uma demonstração da validade do teorema inserido pelo usuário, e, no segundo, que não foi possível encontrar tal demonstração para o teorema.

4.6 Diagrama de Classes (módulos)

O diagrama de classes/módulos do sistema se encontra em `Diagrams/`.

5 Testes

Aqui será apresentada a versão inicial dos testes (e as modificações necessárias no sistema que foram realizadas de acordo) e em seguida a adaptação para testes automatizados.

5.1 Planejamento e execução de testes funcionais

Primeiramente, todos os testes foram realizados com algumas sentenças lógicas simples deixadas como atalho, para as quais era necessário o usuário escrever o número correspondente, sem precisar escrevê-la por completo, e com a sentença mais simples $a \quad imp \quad (b)$. Alguns dos exemplos utilizados não são teoremas válidos, e para eles o comportamento esperado é não gerar uma prova final correta.

Passada a primeira parte, passei a lidar com as funcionalidades do código em si. O código que havia sido feito lidava especificamente com Cálculo de Sequentes, que tem uma estrutura mais rígida que Dedução Natural, i.e. todos os passos da prova têm um formato parecido. Dedução Natural tem passos que ramificam para um, dois ou até três outros nós do grafo, ou que precisam acessar uma parte mais distante da demonstração. Foi necessário, portanto, generalizar o código que ali havia. Para tal, tive de estudar como funciona o Cálculo de Sequentes para poder então remover as partes a ele específicas, como as noções de *implicação à esquerda*, *implicação à direita*, *focus*, e a noção de sequentes em si. As mudanças aqui realizadas foram testadas ao checar como o programa gera o grafo a partir da entrada do usuário. Ao fim, o grafo gerado é mais genérico e adaptável à Dedução Natural.

Com o grafo adaptado, foi necessário criar as funções de aplicação dos passos de demonstração: *Introdução da implicação* e *Eliminação da implicação*. Para testar, foram criados botões na tela que aplicam esses passos sobre o nó do grafo selecionado pelo usuário. Após essa parte, temos, teoricamente, um assistente de provas, pois o usuário fica no controle das ações. Como o projeto é de um ATP, para a versão final esses botões foram removidos, deixando apenas a parte de geração automatizada da demonstração.

Um adendo: também, para verificação da atualização da lista de hipóteses a serem descartadas (ver o conceito de *descarte de hipótese* em 2.3) foi adicionada na tela uma representação textual das hipóteses. A cada passo da prova feito pelo usuário (ou seja, ao apertar um dos botões de *Introdução da implicação* ou *Eliminação da implicação*) o modo como essa lista era atualizado era verificado.

Com os passos individuais de demonstração funcionando corretamente, o foco passou para a parte realmente automatizada do processo: a geração da demonstração pelo programa.

5.2 Automatização dos testes

Para passar para a automatização dos testes, foi preciso descobrir um bom framework de testes unitários e automatizados. Na busca, foi encontrado o `luaunit`, que apresenta uma interface simples de usar e facilidade de instalar, especialmente com `luarocks` em mãos.

O objetivo da automatização dos testes, nesse caso, é realizar uma grande variedade de provas diferentes para checar diferentes modos de desenvolver a prova, em especial focando em provas de tamanho superpolinomial (em especial, as famílias de testes chamadas `Fibonacci` e `HermannTest`), que apresentam

repetições de seções da prova.

Esse caráter repetitivo de algumas provas é justamente o atrativo que quero explicitar do demonstrador: como sua estrutura de grafos sem repetição gera essas provas sem necessitar de uma explosão de uso de memória.

5.2.1 Roteiro

Para gerar os casos de teste, focamos na parte interna de geração de prova. Primeiro geramos os grafos com entradas *mock* simulando entradas de usuários, geramos o grafo a partir delas e então geramos a prova em cima desses grafos.

Com isso, naturalmente aparecem três partes dos casos de teste: uma que primeiro apenas parseia a entrada, uma que gera os grafos de entrada (em que verifico se são condizentes com o grafo que se esperaria ser gerado), e outra que gera as provas deles.

5.2.2 Descrição dos casos de teste

No total, há $12 + 8 + 7 = 27$ casos de teste, separados em três categorias: parsing, geração de grafo e geração de prova. Aqui serão listados os casos de teste utilizados:

1. Testes de parsing
   (a) Falha: `"a imp ("`.
   Nos testes que o parsing falha, nenhuma AST é gerada. Lança-se uma mensagem de erro.
   (b) Falha: `"a imp b"`.
   (c) Falha: `"a im (b)"`.
   (d) Falha: `"a ("`.
   (e) Falha: `"( a"`.
   (f) Sucesso: `"a"`.
   Com os casos que geram a AST, verificamos se seus campos estão preenchidos corretamente.
   (g) Sucesso: `"a imp (b)"`.
   (h) Sucesso: `"a imp (b imp (a))"`.
   (i) Sucesso: `"(a imp (b imp (c))) imp (b imp (a imp (c)))"`
   (j) Sucesso: `"(A1 imp (A2)) imp (((A1 imp (A2 imp (A3)))) imp (((A2 imp (A3 imp (A4)))) imp (((A3 imp (A4 imp (A5)))) imp (((A4 imp (A5 imp (A6)))) imp (((A5 imp (A6 imp (A7)))) imp (A1 imp (A7)))))))"`.
   Essa é uma das provas da família batizada de *Fibonacci*. É a de nível 7.
   (k) Sucesso: `"(((A imp (B)) imp (B)) imp (A))"` (fórmula de Peirce).

(l) Sucesso: `"((((A imp (B)) imp (A)) imp (A)) imp (B)) imp (B)"`.
Essa é uma das provas da família batizada de *HermannTest*.

2. Testes de geração de grafo

Esperamos que os grafos gerados tenham $p + i + 1$ nós e $2 * i + 1$ arestas, onde p é o número de fórmulas atômicas e i é o número de implicações. Somamos um ao final pois há sempre um nó raiz, do qual sai uma aresta.

(a) (string vazia).
Caso padrão da aplicação, em que ela é aberta e o usuário ainda não inseriu nenhuma entrada. Deve gerar apenas um grafo com o nó raiz.

(b) `"a"`.

(c) `"a imp (b)"`.

(d) `"a imp (b imp (a))"`.
Esse caso e os demais que repetem fórmulas explicitam como a estrutura de grafos não repete estruturas, apenas mantém referências, otimizando uso de espaço.

(e) `"(a imp (b imp (c))) imp (b imp (a imp (c)))"`.

(f) `"(A1 imp (A2)) imp (((A1 imp (A2 imp (A3)))) imp (((A2 imp (A3 imp (A4)))) imp (((A3 imp (A4 imp (A5)))) imp (((A4 imp (A5 imp (A6)))) imp (((A5 imp (A6 imp (A7)))) imp (A1 imp (A7)))))))"`.
Essa é uma das provas da família batizada de *Fibonacci*. É a de nível 7.

(g) `"(((A imp (B)) imp (B)) imp (A))"` (fórmula de Peirce).

(h) `"((((A imp (B)) imp (A)) imp (A)) imp (B)) imp (B)"`.
Essa é uma das provas da família batizada de *HermannTest*.

3. Testes de geração de prova

(a) Prova não gerada: `"a"`.

(b) Prova incompleta: `"a imp (b)"`.
Os casos de falha geram tudo corretamente como os casos de sucesso, mas claramente o que é gerado não é uma prova pois não está completa. Espera-se que nos casos de falha ocorra algum nó com label vermelha, indicando que não pôde ser expandido.

(c) Sucesso: `"a imp (b imp (a))"`.

(d) Sucesso: `"(a imp (b imp (c))) imp (b imp (a imp (c)))"`.

(e) Sucesso: `"(A1 imp (A2)) imp (((A1 imp (A2 imp (A3)))) imp (((A2 imp (A3 imp (A4)))) imp (((A3 imp (A4 imp (A5)))) imp (((A4 imp (A5 imp (A6)))) imp (((A5 imp (A6 imp (A7)))) imp (A1 imp (A7)))))))"`.
Essa é uma das provas da família batizada de *Fibonacci*. É a de nível 7.

(f) Prova incompleta: `"(((A imp (B)) imp (B)) imp (A))"` (fórmula de Peirce).

Essa prova só pode ser gerada em Lógica Clássica, e não em Lógica Minimal Implicacional (que utilizamos.

(g) Sucesso: `"((((A imp (B)) imp (A)) imp (A)) imp (B)) imp (B)"`.
Essa é uma das provas da família batizada de *HermannTest*.

5.2.3 Scripts e Logs de teste

Os scripts e logs de teste encontram-se em `Test/`. Ao executar os scripts (informações na seção 6), os logs vão sendo gerados. No nome dos arquivos de log se encontra a data de sua geração, e cada entrada deles apresenta data e hora. No diretório há um servindo de exemplo.

Observação: pela natureza da procura por arquivos de `luaunit`, os arquivos da aplicação utilizados nos testes foram copiados para o diretório `Test/`, junto dos arquivos de teste.

Os scripts de teste são basicamente:

- `testMain.lua`

  A suíte de testes. A partir desse script são chamados os demais. Aqui é definido o logger.

- `testParsing.lua`

  Contém os testes de parsing da entrada do usuário.

- `testGraph.lua`

  Contém os testes de geração de grafos a partir do resultado do parsing.

- `testProofs.lua`

  Contém os testes de geração de provas a aprtir de grafos.

6 Como Utilizar

Para utilizar este sistema é necessário ter os seguintes pacotes (com as versões atualizadas para a entrega deste projeto):

- `lua` (5.2)

  Como o programa é em Lua é necessário ter o interpretador associado.

- `love` (0.10.1)

  Pacote do framework Löve.

- `luarocks` (2.2.0)

  Gerenciador de pacotes de Lua.

- `luasocket` (3.0rc1-1)
- `lpeg` (0.12-1)
- `lualogging` (1.3.0-1)
- `luaunit` (3.2.1-1)

Para executar, basta ir ao diretório onde se encontra o arquivo `main.lua` e executar via linha de comando:

```
$ love .
```

Com isso, que deve funcionar independente de sistema operacional, abrirá a janela com o título `NatDProver`.

Para executar os casos de teste, vá ao diretório `Test/` e, execute no terminal:

```
$ lua testMain.lua
```

Observação: a execução foi realizada em uma máquina com sistema operacional baseado em Linux (Ubuntu 16.04).

6.1 Exemplo de interação com usuário

Ao utilizar o sistema, o usuário deve escrever uma sentença lógica que consista em símbolos atômicos e quantificadores lógicos. Como o provador oferece suporte a lógica minimal, ele traduz a entrada do usuário para implicações apenas. As implicações devem ser feitas na forma $a \quad imp \quad (b)$, com parênteses obrigatórios do lado direito do operador. Os operadores podem ser encadeados, assim como parênteses. Um exemplo se encontra na figura 5.

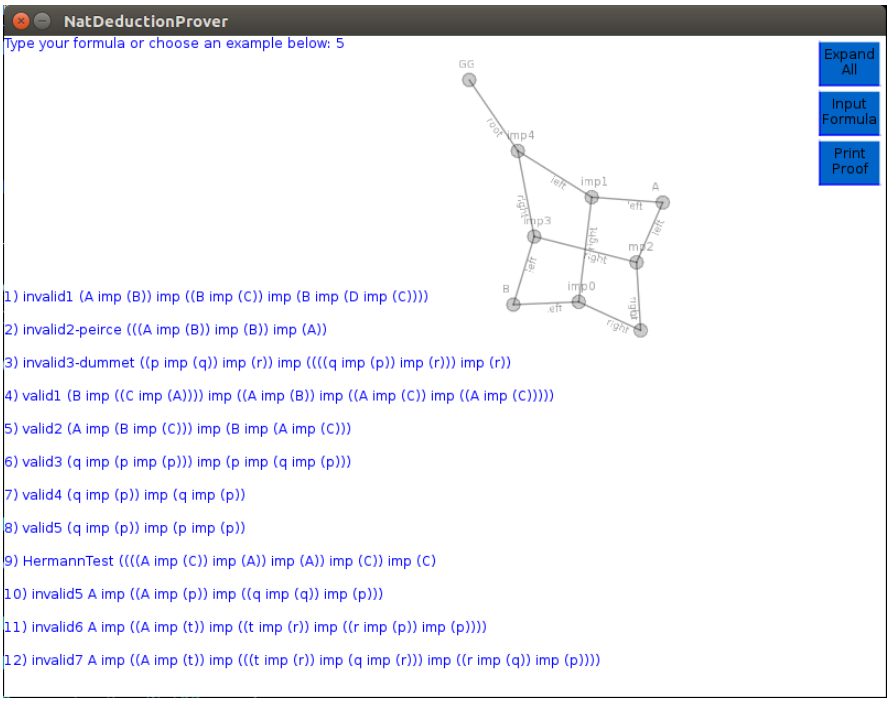

Figura 5: Exemplo de interação em que usuário escolheu a sentença já completa número 5: $(A \to (B \to C)) \to (B \to (A \to C))$.

Tendo inserido a prova, o usuário pode clicar no botão "Expand All"para criar a demonstração. Ao fim do processo interno, o grafo será alterado i.e. a ele serão acrescidos novos nós e arestas referentes aos passos de demonstração realizados. Um exemplo se encontra na figura 6.

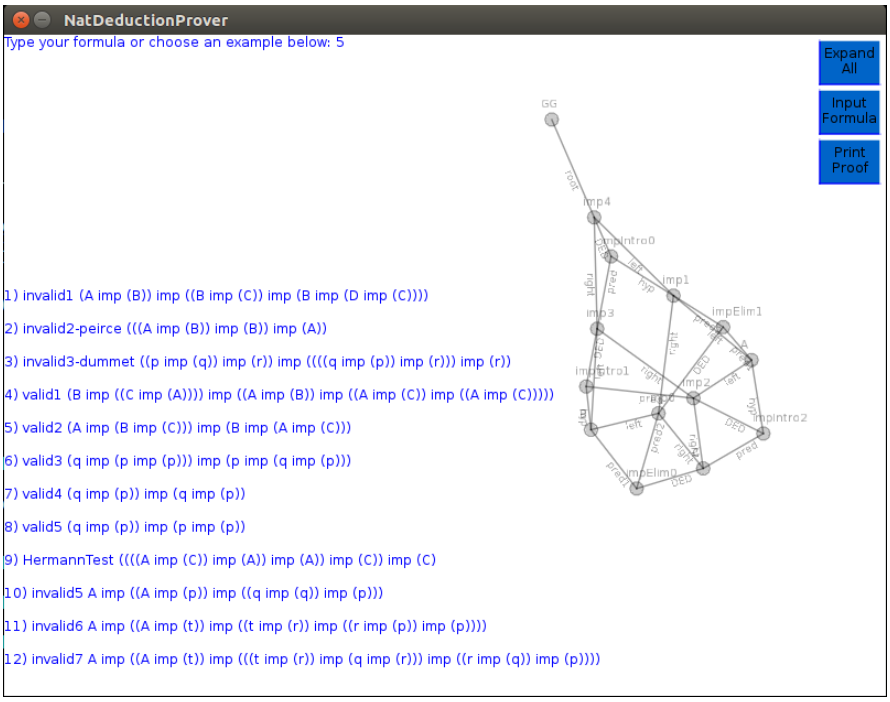

Figura 6: Exemplo de interação em que usuário expandiu a sentença já completa número 5: $(A \to (B \to C)) \to (B \to (A \to C))$.

Tendo a prova concluída, o usuário pode clicar no botão "Print Proof"para imprimi-la em LaTeX. Ao clicar nesse botão, é gerado também um arquivo HTML a partir da proxa em TEX gerada, que é imediatamente aberto no navegador web padrão. Os arquivos de saída têm o formato `prooftreeX.tex`, `prooftreeX.html`, `prooftreeX.css` e assim por diante, onde `X` é um número sequencial. Um exemplo se encontra na figura 7.

$$\dfrac{\dfrac{[B]_2 \qquad \dfrac{[A]_3 \quad [(A \to (B \to C))]_1}{(B \to C)} \to \text{elim}_2}{C} \to \text{elim}_1}{\dfrac{(A \to C)}{(B \to (A \to C))} \to \text{intro}_3} \to \text{intro}_2}{((A \to (B \to C)) \to (B \to (A \to C)))} \to \text{intro}_1$$

Figura 7: Exemplo de resultado da saída do botão "Print Proof"para a sentença exemplo 5: $(A \to (B \to C)) \to (B \to (A \to C))$.

Referências

[1] D. Prawitz. *Natural deduction: a proof-theoretical study*. PhD thesis, Almqvist & Wiksell, 1965.

[2] J. Seldin. Manipulating proofs. Unpublished manuscript. URL: `http://people.uleth.ca/%7Ejonathan.seldin/MPr.pdf`, 1998.

[3] Favio E. Miranda-Perea, P. Selene Linares-Arévalo, and Atocha Aliseda. How to prove it in natural deduction: A tactical approach, 2015.

[4] Edward Hermann Haeusler. How many times do we need an assumption to prove a tautology in minimal logic? examples on the compression power of classical reasoning. *CoRR*, 2014.