Trabalho de Formatura Supervisionado
Monografia do Trabalho de Conclusão de Curso -
"Um Provador Automático de Teoremas para a Lógica Modal, Baseado em Anéis Booleanos"

Supervisor: Prof. Dr. Marcelo Finger
Aluno: Fabio Alexandre Campos Tisovec
Tipo do trabalho: Projeto

15 de janeiro de 2007

1 Introdução

Não é difícil notar a enorme capacidade que os computadores tem para realizar cálculos, um dos grandes motivos de admiração dos seres humanos para com estas máquinas. Naturalmente surge o interesse em converter esta capacidade de cálculo para atividades cada vez mais diversas.

Sendo a lógica parte da matemática, é razoável supor que é possível usar o computador para verificar sem erro a validade de teoremas. Prova automática de teoremas é um sub-campo do que se entende por inteligência artificial. Essencialmente, para um provador identificar um seqüente como sendo uma tautologia é necessário demonstrar que este é verdadeiro para qualquer valoração de suas variáveis. É possível realizar transformações neste seqüente e obter uma formula lógica tal que só é necessário encontrar uma valoração que a satisfaça para poder concluir o mesmo resultado.

Este problema é conhecido por SAT e é reconhecidamente NP-difícil, mas uma implementação simplória pode consumir tempo desnecessário em casos que seria possível aplicar simplificações. Note que para analisar um seqüente, o programa precisa trabalhar com um conjunto de axiomas. É a escolha dos axiomas deste conjunto que determina em que lógica o provador opera.

O objetivo principal deste trabalho é o desenvolvimento de um provador automático de teoremas, almejando-se atingir um bom grau de eficiência deste.

Podem-se destacar várias razões para a importância deste produto do trabalho como, por exemplo, a notória quantidade de usos de tal ferramenta. Com relação ao papel que este projeto desempenha na disciplina de trabalho de formatura nota-se como este está relacionado a áreas de grande interesse do aluno, a saber: desenvolvimento de software, inteligência artificial, complexidade de algoritmos e estrutura de dados.

2 Parte I - Aspectos objetivos do trabalho

Inicialmente serão apresentados os aspectos objetivos do trabalho realizado.

2.1 Conceitos e tecnologias estudadas

2.1.1 Estudos existentes nesta área

Não é muito conhecido o uso da teoria de anéis booleanos para apoiar a resolução do problema da satisfatibilidade. Uma referência nesta área é o artigo [2], onde descreve-se um algoritmo para o SAT. Em essência, a expressão trabalhada é subdividida em inúmeras mini-expressões, que então são comparadas duas a duas para verificar se há contradições entre ela. Caso haja, sabe-se que a expressão não é válida, caso contrário ela é aceita.

2.1.2 Caracterizando um provador automático de teoremas

Para caracterizar o contexto em que se insere o projeto começamos por estudar quais as características essenciais de um provador automático de teoremas. Atualmente existem vários tipos de provadores de teoremas, sendo os principais:

- Verificadores de tautologias - dado um seqüente, procuram inferir se este é uma tautologia, verificando a existência de valorações que o tornam inválido;
- Verificadores de provas - o programa lê uma seqüência de fórmulas lógicas e verifica se cada uma pode ser derivada da anterior, dado seu conjunto de axiomas;
- Focados em um único teorema - o programa é projetado para verificar um determinado teorema. Em geral, trata-se de teoremas notoriamente difíceis e busca-se por simplificações que tornem a verificação possível.

O provador criado neste projeto é do primeiro tipo mencionado. Ele recebe uma seqüência de expressões lógicas e verifica a validade de cada uma delas.

Caso a expressão seja uma fórmula, ele verifica se há uma valoração de suas variáveis que a torne verdadeira e imprime (na saída padrão) a primeira solução encontrada ou imprime que não há soluções para a expressão.

Caso a expressão seja um seqüente, o programa verifica se este é uma tautologia ou não, ou seja: se para todas as valorações possíveis das variáveis o seqüente é verdadeiro. Isto é realizado fazendo uma pequena transformação na expressão e então aplicando o mesmo algoritmo de cálculo de satisfatibilidade do primeiro caso. Esta transformação, bem como uma definição dos termos usados será apresentada a seguir.

2.1.3 Uma introdução à lógica modal

Na lógica proposicional estão definidos quatro operadores lógicos. Um unário (negação) e três binários (conjunção, disjunção e implicação). Estes operadores são identificados respectivamente por $\neg$, $\wedge$, $\vee$ e $\rightarrow$. É possível fazer uso de variáveis em uma expressão lógica, cada qual assumindo um valor booleano.

Apenas as fórmulas bem formadas são consideradas para efeito de dedução lógica, e são definidas da seguinte maneira:

- Se x é uma variável, então x é uma fórmula bem formada;
- Se f_1 e f_2 são fórmulas bem formadas, então $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ e $f_1 \rightarrow f_2$ são fórmulas bem formadas;
- Apenas fórmulas obtidas dessa maneira são fórmulas bem formadas.

Valoração é uma ênupla de valores booleanos, cada qual associado a uma variável de uma expressão lógica. Sejam $v_1, \ldots, v_n$ as variáveis de uma fórmula f, então $V = \langle b_1, \ldots, b_n \rangle, b_i \in \{T, F\}, \forall i \in \{1, n\}$ é uma valoração que associa o valor b_i à variável $v_i, \forall i \in \{1, n\}$ na fórmula f.

Um sequente s é uma expressão lógica com o formato $p_1, \ldots, p_n \vdash c$, onde c e $p_i, \forall i \in \{1, n\}$ são fórmulas bem formadas. Para saber se s é inválido para alguma valoração v de suas variáveis, basta verificar se a expressão $p_1 \wedge \ldots \wedge p_n \wedge \neg c$ é verdadeira em v.

Para calcular o valor de uma expressão lógica, dada uma valoração usa-se as regras de valoração dos operadores, como se segue na tabela:

f_1	f_2	$\neg f_1$	$f_1 \wedge f_2$	$f_1 \vee f_2$	$f_1 \rightarrow f_2$
T	T	T	T	T	T
T	F	T	F	T	F
F	T	F	F	T	T
F	F	F	F	F	T

A lógica modal é uma extensão da lógica proposicional. Além dos quatro operadores básicos, são definidos dois novos operadores, 'suficiente' e 'necessário'. Como um pode ser derivado do outro, pode-se manter uma representação de apenas um deles e fazer uma transformação na expressão trabalhada sempre que se encontra o outro. Há algumas variações de lógica modal, dependendo de quais axiomas são incluídos no conjunto de axiomas básicos (da lógica proposicional).

Há outros operados lógicos que podem ser derivados dos já definidos. Vamos mencionar apenas mais um, o 'ou-exclusivo'. Apesar de não ter uma notação padrão, é comum representá-lo por $\bigoplus$. Sua regra de cálculo de valor é:

f_1	f_2	$f_1 \bigoplus f_2$
T	T	F
T	F	T
F	T	T
F	F	F

Para uma explicação mais detalhada, recorra às fontes [4] e [5].

2.1.4 Princípios básicos de anéis booleanos

Anéis são entidades matemáticas definidos por uma tripla $\langle C, +, . \rangle$, onde C é o conjunto de valores existentes no anel, $+$ e $.$ são operadores binários, definidos por funções $+ : C \times C \rightarrow C$ e $. : C \times C \rightarrow C$.

Sejam $x, y, z \in C$. Os anéis respeitam as seguintes propriedades:

1. $x.0 = 0$
2. $x.1 = x$
3. $x + 0 = x$
4. $x.y = y.x$
5. $(x.y).z = x.(y.z)$
6. $x + y = y + x$
7. $(x + y) + z = x + (y + z)$
8. $x.(y + z) = x.y + x.z$

Anéis booleanos integram um conjunto restrito de anéis, onde $C = \{0, 1\}$. Assim sendo pode-se provar as seguintes propriedades sobre anéis booleanos (além das demais propriedades apresentadas):

1. $x.x = x$

2. $x + x = 0$

Desta forma, a operação . é análoga à conjunção e a operação + é análoga ao ou-exclusivo, o que permite a tradução de expressões lógicas em expressões aritméticas e, portanto, o uso de anéis booleanos para o cálculo de satisfatibilidade de seqüentes.

Para uma explicação mais detalhada, recorra à fonte [1].

2.1.5 NP-completude, consumo exponencial de tempo e de memória

Uma das áreas da ciência da computação é o estudo de complexidade de algoritmos. Um dos resultados mais importantes para esta área foi provado por Alan Turing, relacionado ao que se conhece por problema da parada. Em resumo, o que foi provado é que há problemas que podem ser descritos inteiramente de forma matemática, mas que nenhum computador é capaz de resolvê-los.

Dentre os problemas resolvíveis, estuda-se a complexidade computacional de seus algoritmos de resolução. A classe P é o conjunto dos problemas para os quais existem algoritmos que os resolvam, polinomiais no tamanho da entrada.

Um problema D é dito de decisão se sua resposta é $r, r \in \{SIM, NO\}$. Seja D um tal problema e A um algoritmo que resolve D.

Uma *justificativa polinomial* j é um elemento de tamanho polinomial na entrada de A, capaz de justificar a resposta SIM, quando A gerar tal resposta. Caso a verificação de j possa ser feita em tempo polinomial no tamanho da entrada, então diz-se que D pertence à classe NP.

Pode-se fazer algumas observações quanto às implicações desta definição:

- Não se exige uma solução polinomial para os problemas da classe NP;
- $P \subseteq NP$, pois se $D \in P$, então existe um algoritmo polinomial que apresenta a solução de D, e que pode ser utilizado como algoritmo de verificação para uma justificativa. Logo, $D \in NP$;
- P é a subclasse de "menor dificuldade"de NP. A subclasse de "maior dificuldade"de NP é a classe dos problemas NP-completos. Intuitivamente, se uma solução polinomial for encontrada para um problema NP-completo, então todo problema de NP também admite solução polinomial;
- Não se sabe se $P = NP$. Não foi encontrado até o momento nenhum algoritmo polinomial para nenhum dos problemas NP-completos conhecidos, mas também não foi provado que não existe tal algoritmo.

Este estudo é importante ao trabalho realizado, pois foi demonstrado que o $SAT \in NP$ e atualmente só se conhece algoritmos que consomem tempo exponencial no tamanho da entrada para resolvê-lo. Vários destes algoritmos são polinomiais para um subconjunto de instâncias do SAT, mas todos apresentam algum contra-exemplo para o qual rodam em tempo exponencial. Além disso, alguns deles apresentam uma característica indesejável: consomem espaço de memória exponencial no tamanho da entrada.

2.1.6 Técnicas de desenvolvimento empregadas

Atualmente, há várias metodologias de desenvolvimento de software conhecidas e estudadas. Para a implementação do projeto não foi usada uma metodologia específica, mas um apanhado de técnicas presentes em algumas das metodologias mais recentes, conhecidas por metodologias ágeis. Dentre estas, a que mais tem destaque como inspiração para o projeto é a eXtreme Programming.

A seguir, enumeram-se as principais técnicas usadas na elaboração do projeto, bem como uma breve descrição do que significa e quais as possíveis conseqüências de seu uso:

1. `Desenvolvimento incremental` - O software é criado aos poucos, com certas funcionalidades implementadas primeiro e, posteriormente (quando estas já estão funcionando corretamente), são acrescentadas novas funcionalidades.

   Os métodos tradicionais de desenvolvimento almejam projetar o software inteiramente antes de iniciar a sua implementação. Isso possibilita uma verificação completa do sistema, evitando-se assim possíveis falhas. Infelizmente este processo toma uma quantidade considerável de tempo do projeto, o que permite que erros de planejamento passem despercebidos por muito tempo, ou mesmo que haja mudanças nos requisitos do sistema. Aliado a isso, a verificação completa nunca é realizada, quando muito é feita uma verificação parcial.

   Com o uso do desenvolvimento incremental, possíveis erros de projeto podem ser percebidos mais rapidamente, e mudanças nos requisitos do software não afetam todo o projeto. Em contrapartida, adicionar novas funcionalidades pode causar conflito com a parte já implementada do programa.

2. `Testes automáticos a priori` - São criadas funções que verificam a corretude de execução das funções do programa. Estas funções não necessitam de interação com um usuário para serem executadas e são escritas antes que as funções a que se destinam testar sejam implementadas.

   Não realizar testes significa que as falhas do software serão descobertas pelo uso deste por usuários quando o software estiver em produção. Estas falhas podem causar grande transtorno aos usuários (que correm o risco de perder grande parte de seus trabalhos) ou, ainda pior, o uso mal intencionado do sistema pode usar estas falhas como brechas de segurança. Fazer a manutenção de um sistema em produção quase sempre significa interromper seus serviços e, como este não tem testes, corrigir uma falha possivelmente cria novas falhas.

   O uso de testes não-automáticos é possível em programas extremamente pequenos, mas torna-se ineficiente a medida que este cresce. Como é necessária interação com um usuário, rodar os testes toma mais tempo do que rodar uma bateria de testes automáticos, e não pode ser executado em segundo plano ou enquanto o desenvolvedor não está ativo. Além disso, em sistemas grandes os desenvolvedores ficam inibidos a rodar os testes (sabendo que estes tomarão muito tempo de trabalho), excluindo os testes do processo de implementação do sistema e efetivamente recaindo no caso em que não há testes verificando o software.

   Outro ponto a se considerar: há alguns motivos para os testes automáticos serem escritos antes das funções propriamente ditas. Em primeiro lugar, é pre-

ciso conscientizar-se de que não é possível criar testes que cubram todos os casos. O que realmente se faz necessário é o uso de testes que cubram os casos cruciais: os com maior probabilidade de ocasionarem falhas. Escrever os testes, após as funções terem sido implementadas, faz com que estes sejam dirigidos, testando apenas os casos em que evidentemente as funções são executadas como esperado. Além disso, se os testes são escritos antes das funções, isto permite que o desenvolvedor tome maior conhecimento do problema que necessita resolver, sendo capaz de elaborar soluções mais completas quando efetivamente irá implementá-las.

Por fim, é importante ressaltar que testes não são infalíveis. O erro pode estar no próprio teste, mas é necessário bom senso para identificar quando o teste realmente está errado (e, portanto, precisa ser alterado) e quando a função está errada e deseja-se alterar o teste apenas para que ele 'passe'.

3. `Refatoração -` É o processo de modificar sistematicamente o código existente, sem alterar seu comportamento externo.

   Incluir novas funcionalidades em um software sem antes passar por uma fase de refatoração pode criar falhas ou simplesmente diminuir o grau de compreensibilidade do código. O uso de refatoração pode ser feito visando uma introdução mais fácil de uma nova funcionalidade, aumentar a clareza do código ou, quando o sistema já está em produção, diminuir a deterioração do código.

4. `Simplicidade de design -` Trata-se em manter a estrutura do software e os algoritmos implementados em cada método o mais simples possível.

   Essa prática visa manter uma boa flexibilidade e clareza do código, bem como aumentar a velocidade com que o projeto é desenvolvido e testado. Observe que isto vai contra o objetivo de eficiência do programa, mas em geral é mais vantajoso ter o software desenvolvido primeiro e então aperfeiçoar os pontos do sistema que realmente são necessários, do que tentar implementar os algoritmos mais eficientes para todo o código e realizar apenas uma fração das funcionalidades que o programa poderia ter, caso fosse usada a outra abordagem.

Como é comumente mencionado em XP, o uso conjunto das técnicas é várias vezes mais eficiente do que seu uso em separado, pois os pontos fracos de uma são compensados pelos pontos fortes das outras.

2.2 Atividades realizadas

Como mencionado anteriormente, não foi feita opção por nenhuma metodologia de desenvolvimento específica, mas por um conjunto de técnicas conhecidas. As atividades realizadas não seguiram totalmente o cronograma planejado:

- `Final de agosto:` término da primeira versão funcional do provador para a lógica proposicional, embora não demonstrasse um bom desempenho;
- `Final de outubro:` término da implementação do algoritmo melhorado, que faz uso de uma tabela de fórmulas;
- `Inicio de novembro:` preparação da apresentação do trabalho realizado;

- `Meados de novembro:` ampliação do provador desenvolvido para que este opere com a lógica modal;
- `Inicio de dezembro:` conclusão da monografia sobre o projeto.

2.3 Resultados e produtos obtidos

Como há a preocupação com eficiência, foi decidido pelo uso da linguagem C++ para a implementação do programa deste projeto. Também foi necessário um cuidado com a organização e compartimentação do código, fatores que influenciam na eficiência do provador. Faz-se a seguir uma exposição do resultado obtido pelo projeto.

2.3.1 Estrutura do programa

O programa apresenta uma estrutura ligeiramente diferente do que foi planejado na proposta de trabalho. Não vemos motivo de preocupação quanto a isso, pois flexibilidade do projeto faz parte da metodologia em que se baseia a confecção do programa.

Neste momento, o programa encontra-se organizado da seguinte maneira:

- `Modelo de representação:` módulo responsável por manter uma representação interna dos objetos sendo analisados. Corresponde aos arquivos de código fonte presentes nos subdiretórios \`formula` e \`map`;
- `Analisador léxico:` converte a entrada do programa (em formato texto) para objetos do modelo. Corresponde aos arquivos de código fonte presentes no subdiretório \`parser`;
- `Coordenador:` integra os outros módulos e é responsável pela saída do programa. Corresponde aos arquivos de código fonte presentes no subdiretório \`main`;
- `Calculador de fórmulas:` sub-módulo responsável por verificar a validade de cada expressão lógica processada pelo programa. Corresponde aos arquivos de código fonte\`main`\`formulaCalc.cpp` e \`main`\`formulaCalc.h`;
- `Testes:` módulo auxiliar para garantir a corretude dos demais módulos. Sua estrutura básica encontra-se no arquivos de código fonte do subdiretório \`test`. Seus demais arquivos encontram-se inseridos nos respectivos módulos a que se prestam verificar.

2.3.2 Modelo de representação

O conceito fundamental que determinou como este módulo foi implementado é o polimorfismo. Todas as entidades extraídas das expressões lógicas analisadas são armazenadas em classes, cada qual derivada da classe `Formula`. Dessa forma, todas as operações nessas instâncias como, por exemplo, verificação de igualdade (`equals()`) ou valor associado (`getValue(assignment)`), são tratadas de uma forma padrão e cabe à estrutura de classes decidir qual o método que será chamado.

É ainda o polimorfismo que permite alternar dinamicamente entre os algoritmos de cálculo lógico. O algoritmo trivial organiza as instâncias em uma árvore de sub-fórmulas. O segundo algoritmo desenvolvido mantém uma tabela de fórmulas já existentes, onde as sub-fórmulas são organizadas em um grafo dirigido acíclico.

Para realizar o mapeamento é usada a classe `FormulaStamp` como um invólucro das estruturas originais, indicando em que lugar da tabela de fórmulas encontra-se a sub-fórmula desejada. Dessa forma, o polimorfismo determinará que os métodos da instância de `FormulaStamp` sejam executados. Estes apenas delegam a chamada para a instância apropriada.

O modelo também contém a classe `Map`, que associa cada variável (identificada pela string de seu nome) usada em uma expressão lógica a um ID usado na representação interna dessa expressão. O modelo também contém a classe `Assignment`, que mantém a associação de um valor booleano a cada ID em uso.

2.3.3 Analisador léxico

A principal classe desse módulo é a classe `Parser`. Ela é responsável por extrair Tokens da expressão lógica a ser analisada (uma string), e criar as instâncias do modelo de acordo com os Tokens extraídos. É a implementação desta classe que define qual a linguagem reconhecida pelo programa.

Também neste módulo utiliza-se polimorfismo para alternar entre os métodos de cálculo. Uma derivação da classe `Parser` modifica a forma como as instâncias da classe `Formula` são criadas, passando a usar a tabela de fórmulas.

2.3.4 Testes

A estrutura básica define o funcionamento geral dos testes individuais, bem como o de um grupo de testes. Como um grupo de testes também pode ser visto como um teste, é possível inserir grupos de testes em grupos maiores de testes.

É desta forma que estão organizados os testes do programa. Cada módulo tem seu próprio conjunto de testes, e todos estes conjuntos estão inseridos em um teste completo (a classe `FullTest`).

2.3.5 Principais estruturas de dados utilizadas

O programa não faz uso de nenhuma estrutura de dados muito elaborada.

Como estruturas-base utiliza-se as classes `std::vector` e `std::string`. Estas são usadas em atributos dentro das classes implementadas. Dentre estas, cabe mencionar as classes:

- `Stack`: usa uma instância de `std::vector<Formula>` para gerenciar uma pilha de fórmulas;
- `IDmap`: usa uma instância de `std::vector<std::string>` para associar IDs (números naturais) a strings;
- `FormulaAnd`, `FormulaXor` e `Sequent`: usam cada qual uma instância de `std::vector<Formula>` para armazenar uma lista de seus elementos;

- `FormulaTable`: usa uma instância de `std::vector<Formula>` e cinco instâncias de `std::vector<unsigned>` para associar IDs (números naturais) a fórmulas. A primeira lista é a tabela geral de fórmulas e as demais são tabelas parciais, cada qual indicando a posição de todas as fórmulas de um determinado tipo na tabela geral.

2.3.6 Formato da entrada do programa

O programa lê expressões lógicas tanto da linha de comando quanto de arquivos de entrada. O modo de uso interativo não foi inteiramente implementado, principalmente por não ser prático seu uso durante o desenvolvimento. Em seguida encontra-se a descrição da linguagem reconhecida nas expressões lógicas:

- Seqüente : premissas |- conclusão;
- Premissas : fórmula | premissas, fórmula;
- Fórmula : identificador | (fórmula)And conjunção | Not(fórmula) | (fórmula)Or disjunção | (fórmula)->(fórmula) | Box(fórmula);
- Conjunção : (fórmula) | (fórmula)And conjunção;
- Disjunção : (fórmula) | (fórmula)Or disjunção;
- Identificador : seqüência de letras e/ou dígitos (o primeiro caracter deve ser uma letra). Atenção ao usar 'And', 'Or' ou 'Not' como identificadores (embora isto seja possível, não é recomendado).

Observação: brancos não alteram a identificação da fórmula, a menos em um identificador de variável.

Para arquivos de entrada, a linguagem reconhecida é a mesma, mas há algumas restrições no formato para permitir múltiplas expressões por arquivo:

- Cada expressão pode se estender por mais de uma linha;
- Cada expressão deve ter os caracteres "\>"no início de sua primeira linha;
- Todas as linhas antes da primeira expressão são descartadas;
- Não é possível colocar comentários entre cada expressão.

2.3.7 Comparação com métodos de outros provadores de teoremas

Alguns dos provadores de teoremas mais conhecidos atualmente são o OTTER, o VAMPIRE e o zChaff.

O OTTER é voltado para a lógica de primeira ordem. A idéia geral usada para verificar um dado teorema é separá-lo em cláusulas e então começa a derivar novas cláusulas e simplificar as já existentes até chegar a uma contradição (e descartar o teorema) ou até não ser possível derivar nenhuma cláusula 'útil' (sendo aceito o teorema). São tomados certos cuidados para tornar o processo mais eficiente como, por exemplo, evitando derivar uma mesma cláusula múltiplas vezes.

O VAMPIRE também é voltado para a lógica de primeira ordem. A idéia geral usada para verificar um dado teorema é passá-lo para a CNF e aplicar resolução binária em suas cláusulas para simplificá-las. Caso consiga-se obter a cláusula vazia o teorema é aceito, caso contrário é descartado.

O zChaff opera na lógica proposicional. Ele usa um algoritmo semelhante ao VAMPIRE, uma variação do DPLL.

2.3.8 Análise analítica

Pode-se dividir a análise em duas partes: o pré-processamento da expressão lógica e o cálculo de satisfatibilidade. Não é necessário analisar a inicialização do programa, pois no momento não é realizada nenhuma tarefa considerável nesta parte.

2.3.9 Pré-processamento

Consideremos primeiramente o que há de comum nos dois algoritmos implementados.

Considerando o analisador léxico, o número de Tokens presentes na entrada é proporcional ao tamanho da entrada. Seja n_T o número de Tokens e n_c o número de caracteres da entrada, então $n_T = O(n_c)$.

Há dois fatores que dificultam a análise do número de operações realizadas dependendo de n_T: a atribuição de IDs às variáveis e as chamadas à função `rollBackLastToken()`.

Sejam n_v o número de variáveis distintas na entrada, $maxt_{nome}$ o número de caracteres no nome da variável de maior nome e n_o o número de ocorrências de variáveis na entrada. A classe `Map` usa um algoritmo linear para busca e atribuição de IDs a strings, portanto o número de operações realizadas para obter todos os IDs da entrada é de $op_{IDs} = O(maxt_{nome}.n_v.n_o)$.

Como $maxt_{nome}.n_o = O(n_c)$, e $n_v < n_T$, então $op_{IDs} = O(n_T^2)$.

Observe que a função `rollBackLastToken()` nunca é chamada duas vezes sem que ao menos um Token seja inteiramente avaliado. Portanto $op_{analex} = O(n_T) + op_{IDs} = O(n_T^2)$.

As operações exclusivas ao algoritmo trivial são todas constantes por Token, portanto a complexidade do pré-processamento não é alterada.

As operações exclusivas ao algoritmo com o uso da tabela de fórmulas dependem do número de sub-fórmulas já inseridas. Para fórmulas zeroárias as operações são constantes. Para fórmulas unárias é feita uma busca linear. Para fórmulas enárias é feita uma ordenação quadrática de seus elementos e então é feita uma busca linear.

Visivelmente é o último caso que domina a ordem de complexidade desse pré-processamento, sendo de $O(n_T^3)$.

Não é feita nenhuma simplificação na estrutura obtida pelo analisador léxico, portanto o número de operações realizadas no pré-processamento é de $O(n_T^2)$ no algoritmo trivial e de $O(n_T^3)$ no algoritmo com a tabela de fórmulas. Como $n_T = O(n_c)$, ambos são polinomiais no tamanho da entrada.

2.3.10 Cálculo de satisfatibilidade

O algoritmo para o cálculo de satisfatibilidade é o mesmo para ambos os algoritmos e não apresenta dificuldades de análise.

Para cada valoração das variáveis, pode-se verificar o valor da expressão lógica em tempo proporcional a n_T. Como há 2^{n_v} valorações possíveis, o número de operações realizadas no cálculo de satisfatibilidade é de $O(n_T.2^{n_v})$, ou seja, é exponencial no tamanho da entrada. Este resultado já era esperado, caso contrário implicaria que $P = NP$, pois o SAT é um problema NP-completo.

Nota-se que esta complexidade largamente domina a de pré-processamento, tornando questionável investir tempo na redução da complexidade das operações realizadas nesta fase (por exemplo: aprimorando a busca linear para a obtenção dos IDs de variáveis, ou reescrever o algoritmo quadrático de ordenação dos elementos de fórmulas enárias).

2.3.11 Análise empírica

Foram rodados testes com duas famílias de seqüentes, ambas bastante conhecidas e estudadas na lógica proposicional [3] .

A família Gamma é definida por:

$$a_1 \vee b_1, \Gamma_n \vdash a_{n+1} \vee b_{n+1} \tag{1}$$

onde

$$\Gamma_n = \cup_{i=1}^{n} \{a_i \rightarrow (a_{i+1} \vee b_{i+1}), b_i \rightarrow (a_{i+1} \vee b_{i+1})\} \tag{2}$$

A família H é definida por

$$\vdash H_n \tag{3}$$

onde

$$\begin{aligned} H_1 &= p_1 \vee \neg p_1 \\ H_2 &= (p_1 \wedge p_2) \vee (\neg p_1 \wedge p_2) \vee (p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge \neg p_2) \\ &\vdots \end{aligned} \tag{4}$$

Como era esperado pela analise analítica, o programa rodou em um tempo exponencial em n, o índice de cada equação lógica testada.

Realizando o 'profiling' do programa (medir o tempo gasto em cada método do programa) pudemos observar que a maior parte do tempo foi gasta em funções usadas do arcabouço da linguagem, em especial nas funções internas da classe `std::vector`. Esse consumo de tempo correspondeu a cerca de 85% do tempo no teste da família H e 75% do tempo no teste da família Gamma.

Das funções implementadas no programa, as que mais consumiram tempo foram:

função	família H	família Gamma
(FormulaAnd::getValue(assignment))	3,17%	3,03%
(FormulaAnd::equals(formula))	2,38%	$\sim 0\%$
(FormulaXor::getValue(assignment))	1,98%	9,51%
(FormulaXor::equals(formula))	1,98%	$\sim 0\%$

Tempo consumido nos testes:

índice	família H	família Gamma
1	33,385	116,131
2	100,505	303,851
3	333,63	667,5
4	762,86	2223,34
5	2944,1	5469,306
6	6411,9	27021,8
7	22340	102141,2
8	70304	416625
9	264690	1738801
10	1096150	6810110

3 Parte II - Aspectos subjetivos do trabalho

Nesta parte serão apresentados os aspectos subjetivos do trabalho realizado.

3.1 Desafios e frustrações encontrados

3.1.1 Desafios

Em primeiro lugar, pode-se citar o objetivo do trabalho como um desafio. Quando estava procurando um supervisor para o trabalho de formatura não tinha uma idéia clara do que gostaria de fazer, apenas de que gostaria que fosse, preferencialmente, sobre a área de inteligência artificial. Entre as várias sugestões de trabalho propostas pelo Prof. Dr. Marcelo Finger estava o tema deste trabalho, que me chamou a atenção por parecer interessante e desafiador.

Interessante por ser da área que eu pretendia estudar, envolver um problema de grande peso na teoria da ciência da computação - a classe dos problemas NP-completos - e por envolver aspectos práticos da computação - a implementação do projeto proposto. Desafiador, pois não me parecia trivial implementar individualmente um provador de teoremas. Certamente este foi meu maior trabalho individual até o momento.

Outro desafio está relacionado a certas técnicas escolhidas para a elaboração do trabalho. É necessária muita disciplina para aplicar técnicas como testes a priori ou simplicidade de design quando se está individualmente implementando código. Com a experiência que tive em certas disciplinas do curso posso afirmar que estas técnicas são mais fáceis de serem seguidas quando se trabalha em uma equipe, mesmo que pequena. No âmbito do projeto, por vezes elas não foram aplicadas no início dos trabalhos, mas à medida que o tempo passava tornou-se mais e mais natural o seu uso.

3.1.2 Frustrações

Há algumas frustrações encontradas durante o projeto que vale a pena enumerar aqui.

Em primeiro lugar, uma certa dificuldade em poder usar um bom ambiente de desenvolvimento (algo essencial para alcançar uma boa produtividade). O projeto foi desenvolvido ora em um sistema Unix, ora em um sistema Windows. No sistema Unix usou-se o ambiente Eclipse com um pacote para desenvolvimento em C++. No sistema Windows este pacote não pôde ser instalado, sendo então usado um outro ambiente, o Dev-C++. Infelizmente a transição de um ambiente para outro precisava ser feita manualmente, tendo sido gastas muitas horas de desenvolvimento reescrevendo arquivos Makefile e coisas do gênero, algo que poderia ter sido evitado com o uso de um ambiente mais amigável.

Outra frustração foi com os arcabouços de testes automáticos existentes para a linguagem usada. Muitos são exclusivos de certos ambientes de desenvolvimento (software privado), outros não são de fácil inserção no projeto. Para resolver este problema foi implementado um mini-arcabouço de testes automáticos no projeto. Apesar de não ser sofisticado, este mini-arcabouço foi em geral suficiente para o projeto.

Uma outra frustração que ocorreu no projeto foi uma falha no código que atrasou o desenvolvimento em quase um mês no início do segundo semenstre. Como não há coletor automático de lixo em C++, todas as instâncias devem ser manualmente apagadas. A falha encontrava-se nas funções `Formula::clean()` e `Formula::erase()`, não cobertas pelos testes automáticos. Durante a execução, uma mesma instância era apagada duas vezes, causando um erro no gerenciador de memória e, como conseqüência, o término anormal do programa.

3.2 Lista das disciplinas cursadas no BCC mais relevantes para o trabalho

- MAC0110 - Introdução à Computação e MAC0122 - Princípios de Desenvolvimento de Algoritmos: Indispensáveis no aprendizado da ciência da computação.
- MAC0211 - Laboratório de Programação I, MAC0323 - Estruturas de Dados, MAC0242 - Laboratório de Programação II, MAC0332 - Engenharia de Software, MAC0340 - Laboratório de Engenharia de Software e MAC0342 - Laboratório de Programação eXtrema: Todas estas disciplinas foram relevantes para proporcionar experiência em desenvolvimento de software.
- MAC0239 - Métodos Formais em Programação e MAC0425 - Inteligência Artificial: Nestas disciplinas adquiri conhecimento de lógica e inteligência artificial, ambas necessárias para o projeto.
- MAT0213 - Álgebra II: Nesta disciplina foram estudados os anéis matemáticos e suas propriedades.

3.3 Interação com o supervisor

Procurei aproveitar a disponibilidade de um supervisor como uma orientação para o trabalho realizado. Estive atento para que sempre obtivesse alguns conselhos para seguir, mas que nunca ficasse dependente de uma constante supervisão de todas as decisões tomadas. Para tanto, foram realizados encontros com intervalos de um a um mês e meio. Em suma, procurei o supervisor como um guia que me apontasse a direção a seguir, reservando para mim a atividade de trilhar na direção dada.

4 Conclusão

O objetivo do trabalho não era, e não poderia ser, esgotar as possibilidades da área, mas sim verificar a praticidade de se implementar uma nova técnica para tratar o *SAT*, baseada em uma abordagem pouco conhecida. Neste ponto, o trabalho cumpriu seu objetivo.

Há a possibilidade de se incluir mais funcionalidades no código criado como, por exemplo, o uso de novas estruturas de dados, de heurísticas e de novas manipulações das expressões avaliadas. As bases já estão prontas.

O trabalho também foi proveitoso para criar um maior amadurecimento no aluno, tanto em aspectos práticos como teóricos da ciência da computação. Certamente isto está condizente com os objetivos da disciplina de trabalho de formatura supervisionado.

Referências

[1] S. Burris e H. Sankappanavar, *A course in universal algebra*, 1981.

[2] Nachum Dershowitz, Jieh Hsiang, Guan-Shieng Huang e Daher Kaiss, *Boolean ring satisfiability.*, SAT, 2004.

[3] Wagner Dias, *Implementações de tableaux para raciocínio por aproximações*, Master's thesis, IME-USP, 2002.

[4] Michael Huth, *Logic in computer science: tool-based modeling and reasoning about systems.*

[5] Johan van Benthem, *The range of modal logic.*