UNIVERSIDADE ABERTA

Programação orientada a objectos na determinação das bases dum sistema de fecho

Mestrado em Estatística Matemática e Computação

Aluno: Edgar Augusto de Figueiredo Vigário

Orientador: Professor Doutor João Jorge Ribeiro Soares Gonçalves de Araújo

2010

Resumo

O objectivo desta dissertação foi a criação de uma ferramenta informática capaz de determinar as bases de um sistema de fecho.

Para tal foram criadas diferentes implementações de três algoritmos, o de *Tietze*, o de *Tietze-SA* e o *SA*, utilizando o paradigma da programação orientada a objectos e escolhida como implementação para a construção da ferramenta a que resultava num menor número de chamadas a um oráculo.

O oráculo utilizado foi a aplicação *Prover9/Mace4* da autoria de *William McCune* composto pelo demonstrador automático de teoremas *Prover9* e o construtor de modelos finitos *Mace4*.

As aplicações resultantes executam nos sistemas operativos *Windows XP*, *Vista* e *7* (*32* e *64 bits*).

Abstract

The aim of this thesis is to provide a software tool capable of determining the basis of a closure system.

To this end, we created different implementations of three algorithms, the algorithm Tietze, the algorithm Tietze-SA and the SA algorithm using the paradigm of object-oriented programming, and selected the one that could reach the goal in a minimal number of calls to an oracle.

The oracle was Prover9/Mace4 authored by William McCune consisted by the automated deduction tool Prover9 and the finite model builder Mace4.

The resulting applications are executable under Windows XP, Vista and 7 (32 and 64 bits), determining the bases with an exponential time complexity.

Esta tese é dedicada aos meus pais.

Agradecimentos

Queria em primeiro lugar expressar o meu agradecimento ao Professor Doutor João Araújo pela sua paciência infinita, interesse e apoio sempre dado ao longo da realização deste trabalho.

Também gostaria de expressar o meu agradecimento aos demais professores da Universidade Aberta com os quais tive mais contacto e que também sempre me apoiaram ao longo do percurso que realizei e culminou na realização desta tese, Prof. Doutora Teresa Oliveira, Prof. Doutor Amílcar Oliveira, Prof. Doutor Mário Edmundo, Prof. Doutor José Coelho, Prof. Doutor Vitor Rocio, Prof. Doutora Maria do Rosário Ramos, Prof. Doutor Luís Cavique e Prof. Doutor Jorge Valadares.

Por último gostaria também de expressar o meu agradecimento a Vera Filipe pela paciência também infinita e ajuda na revisão do trabalho.

Índice

Índice de Figuras[1]

[1] Os diagramas de classe e as listagens de pseudocódigo foram realizados com o auxílio de software proprietário (ver [1] e [32]).

Índice de Quadros

1. Introdução

1.1. Plano de Dissertação

Seja (X, ‹.›) um sistema finito de fecho, isto é, X é um conjunto finito e ‹.› : $2^X \rightarrow 2^X$, uma operação em que:

1. $X \subseteq \langle X \rangle$;
2. $X \subseteq Y \Rightarrow \langle X \rangle \subseteq \langle Y \rangle$;
3. $\langle X \rangle = \langle\langle X \rangle\rangle$.

Um subconjunto $A \subseteq X$, diz-se independente se $\forall x \in A$, $x \notin \langle A \setminus x \rangle$. Um conjunto independente $A \subseteq X$ diz-se uma base de X se $\langle A \rangle = \langle X \rangle$.

O objectivo da dissertação é a criação de uma ferramenta informática capaz de identificar as bases de A, ou seja, todos os conjuntos $Y \subseteq X$ tais que Y é independente e $\langle Y \rangle = \langle X \rangle$ sendo (X, ‹.›) um sistema finito de fecho com $A \subseteq X$.

Para a construção da ferramenta foram utilizados e testados três algoritmos, o algoritmo de *Tietze*, o algoritmo de *SA* e o algoritmo de *Tietze-SA* que serão explicados em detalhe ao longo da dissertação nos subcapítulos 2.4, 2.5 e 2.6, sendo o primeiro da autoria de *Hienrich Tietze* (ver [35]), e os segundo e terceiro da autoria de João Araújo.

A cada um dos algoritmos foram adicionados dois aceleradores, definindo-se acelerador, como alterações no código ou na estrutura da implementação original do algoritmo com o objectivo de eliminar as derivações redundantes de conjuntos axiomáticos, denominados acelerador *(+)* e o acelerador *(V)*, sendo o primeiro da autoria de *João Araújo* e o segundo do autor da dissertação. Uma explicação detalhada será fornecida nos subcapítulos 2.7 e 2.8.

Foram assim construídas numa primeira fase nove implementações, definindo-se implementação como a formalização na linguagem orientada a objectos C# dos algoritmos ou dos algoritmos acrescidos de cada um dos dois aceleradores atrás referidos. Posteriormente foram construídas mais duas implementações motivadas pela análise dos resultados obtidos pelas primeiras.

No que respeita à sua estrutura, a dissertação é composta por cinco capítulos que reflectem a ordem cronológica da construção da ferramenta informática, existindo ainda um anexo contendo os manuais de utilização das duas aplicações finais criadas, definindo-se aplicação como um programa "*standalone*" auto instalável e executável nos sistemas operativos *Windows XP*, *Vista* e *7* (32 e 64 bits).

Assim, além deste capítulo introdutório, no segundo capítulo será discutida a implementação dos três algoritmos e dois aceleradores acima referidos, e também, as primeiras aplicações criadas. No terceiro capítulo serão apresentados os resultados dos testes efectuados às implementações referidas no capítulo precedente e feita uma análise estatística destes resultados. No quarto capítulo serão discutidas duas novas implementações construídas após a análise dos resultados obtidos pelas implementações anteriores, e por último o quinto capítulo consistirá numa conclusão onde serão tecidas considerações e apresentados possíveis desenvolvimentos futuros.

1.2. Notas prévias

As definições semânticas dos termos utilizados serão feitas ao longo do texto num modo semelhante ao efectuado no subcapítulo anterior para aceleradores, implementações e aplicações, referindo-se aqui:

- Os conjuntos de axiomas serão designados como nós, iniciais se forem o ponto de partida para a execução da aplicação e derivados se resultarem da execução dos diferentes algoritmos, destacando-se nestes últimos os finais, ou seja, os conjuntos de axiomas resultantes da execução da implementação;
- Quando nos referimos a bases, pressupomos uma execução da aplicação sem erros, caso contrário, é mais correcto designar os nós finais resultantes como subconjuntos de geradores. Esta diferença deve-se à não satisfação da condição de independência para o último caso.

Outro aspecto que convém aqui referir relaciona-se com a validade das ferramentas informáticas apresentadas. De facto, tal como será adiante descrito todas as implementações utilizam um demonstrador automático de teoremas cuja validade das provas por ele encontradas nunca foi verificada com qualquer tipo de verificador de provas tal como o programa IVY (ver [26]) como é sugerido em artigos que abordam a mesma temática (ver por exemplo [6]).

2. Algoritmos - Implementação

2.1. Introdução

Para a realização deste trabalho foram construídos nove aplicações na perspectiva da programação orientada a objectos, três que resultam da implementação directa dos três algoritmos estudados, o de *Tietze*, o de *Tietze-SA* e o *SA*, e seis que resultam da implementação de cada um destes algoritmos acrescida de cada um dos dois aceleradores, tendo sido escolhida a linguagem C# e utilizada a interface de programação *Visual Studio 2008 PE* para as elaborar.

Todas estas aplicações, embora com variantes nas suas construções e concepções, seguem em linhas gerais um modo de execução similar, ou seja, é lido um ficheiro de texto, contendo um conjunto de axiomas representado por um conjunto de cadeias de caracteres (“*strings”)*, e a partir dele, construídas as estruturas de dados e os ficheiros necessárias à execução do algoritmo e do oráculo a que todos recorrem, sendo posteriormente os resultados salvaguardados em vários ficheiros de textos.

Como oráculo foi escolhida a aplicação *Prover9-Mace4* (versão *LADR-Dec-2007)* executável no sistema operativo *Windows* da autoria de *William McCune* (ver [23] e [24]), sendo esta aplicação constituída por dois programas que são executados por intermédio de dois processos distintos, o demonstrador automático de teoremas *Prover9* executado pelo processo *Prover.exe* e o motor de busca de modelos finitos *Mace4* executada pelo processo *Mace4.exe*.

A conjugação dos resultados obtidos pela execução em simultâneo destes dois processos determina a resposta do oráculo. Esta resposta condicionará a derivação de novos conjuntos axiomáticos a partir dos conjuntos axiomáticos actualmente presentes de acordo com o tipo de algoritmo utilizado. De facto, é a utilização do oráculo que permite a implementação destes mesmos algoritmos constituindo-se como um factor fundamental na construção da ferramenta informática que permite a identificação das bases de um sistema finito de fecho.

Assim, nas páginas seguintes, será feita uma descrição detalhada da ideia subjacente à construção das diferentes implementações, dos diferentes algoritmos e aceleradores atrás referidos.

Na descrição dos algoritmos e aceleradores tentar-se-á sempre destacar o papel que neles é desempenhado pelo oráculo, iniciando-se no primeiro caso com o seu *modus operandis* seguindo-se o modo como foi implementado. No caso dos aceleradores, será dada mais ênfase às alterações efectuadas nas implementações originais, não deixando contudo de abordar o objectivo da sua utilização.

Existem também dois subcapítulos, um exclusivamente dedicado à classe *Oraculo* dada a sua importância na execução das diferentes aplicações e na interface que estabelece entre elas e os processos *Prover9.exe* e *Mace4.exe* que constituem o já referido oráculo, e outro que conclui este capítulo abordando aspectos relacionados com a utilização das primeiras aplicações criadas.

2.2. Ideia Geral

Consideremos o sistema de fecho (X, ‹.›). Consideremos também $A \subseteq X$. Queremos determinar todas as bases de A, isto é, queremos encontrar todos os conjuntos $B \subseteq A$ tais que B é independente e ‹B› = ‹A›.

O algoritmo mais básico consiste em considerar o conjunto $2^A = \{Y: Y \subseteq A\}$e depois utilizar um oráculo que responde a duas perguntas para cada $Y \in 2^A$:

1. Y é independente?
2. ‹Y› = ‹A›?

Se a resposta às duas perguntas for positiva, então o conjunto Y é adicionado ao conjunto dos resultados. Quando tivermos esgotado todos os $Y \in 2^A$, temos a resposta ao problema inicial.

Em geral poderíamos ter considerado diversos sistemas de fecho (do tipo "álgebra gerada"), mas optamos pelo sistema de fecho de inferência lógica (em predicados de 1ª ordem). Assim o nosso espaço X será composto por um conjunto de formulas em alguma linguagem e alem disso, para $x \in X$ e $A \subseteq X$, dizemos que $x \in$ ‹A› se e só se $A \Longrightarrow x$.

Para trabalhar neste sistema de fecho necessitamos de um oráculo especial denominado Prover9/Mace4.

Assim, perante a pergunta "$A \Longrightarrow x$?" três coisas podem acontecer:

1. O Prover9 encontra uma prova e portanto $A \Longrightarrow x$;
2. O Mace4 encontrou um modelo e portanto $\neg(A \Longrightarrow x)$;
3. Nenhuma das duas anteriores.

Se o oráculo fosse composto apenas pelo Prover9, ele trabalharia indefinidamente nas situações 2 e 3. O mesmo se diga caso o oráculo fosse composto apenas pelo Mace4.

Assim, cada pergunta é feita em simultâneo ao Prover9 e Mace4 e o primeiro a terminar acaba com o outro processo. Para resolver a terceira situação estabelece-se um tempo limite para cada pergunta.

2.3. Classe *Oraculo*

A classe *Oraculo*, cujo diagrama é apresentado na figura 2-1, é a interface que medeia a troca de informação entre as diversas implementações e os processos *Prover9*.exe e *Mace4.exe* tendo como particularidade as suas instâncias serem sempre criadas aos pares e executadas em duas "*threads*" distintas.

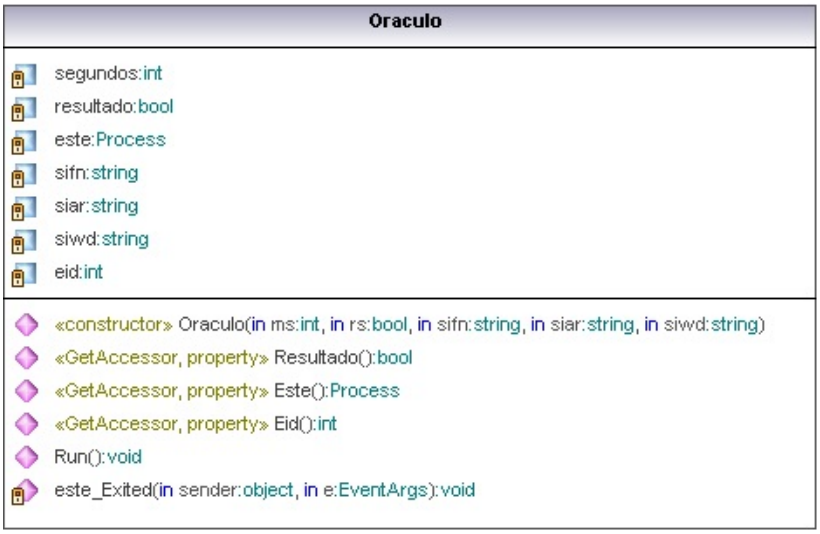

Figura 2-1 Diagrama da classe *Oraculo*

Esta exigência na execução das instâncias da classe em "*threads*" separadas resulta na possibilidade de abortar automaticamente a "*thread*" e o processo que nela é executado quando a sua "*thread*" gémea terminar devido ao término do processo a ela associado, podendo assim os dois processos serem executados em simultâneo ou abortados quando já não for necessária a sua execução.

De modo a clarificar o parágrafo anterior convém referir dois factores, um relacionado com as execuções e resultados fornecidos pelos processos que constituem o oráculo, e outro, com a utilização destes mesmos resultados pelas diferentes implementações dos algoritmos.

No que se refere ao primeiro dos dois factores, o processo Prover9.exe termina a sua execução de um modo normal quando é por ele encontrada uma prova dedutiva e o processo Mace4.exe quando é por ele encontrado um contra-exemplo, ou seja, dado um conjunto de axiomas como premissa e um outro axioma como objectivo, o processo Prover9.exe terminará se do primeiro conjunto se poder deduzir o axioma tido como objectivo, exibindo o processo Mace4.exe um comportamento complementar terminando quando for encontrado um contra-exemplo

indicando que os axiomas do primeiro conjunto não conseguem deduzir o axioma tido como objectivo.

As diferentes implementações utilizam esta informação, terminando as "*threads*" associadas às instâncias da classe *Oraculo* quando um dos dois processos terminar ou quando o tempo máximo atribuído a ambos os processos se esgotar (ver figuras 2-2 e 2-3), criando ou não novos conjuntos de axiomas consoante os valores contidos na propriedade *Oraculo.resultado* de ambas as instâncias desta classe.

Voltando ao diagrama da figura 2-1, as propriedades nele listadas contêm a informação necessária para a construção e execução do processo que a classe parametriza, seja ele o *Prover9.exe* ou o *Mace4.exe*. Assim:

- *Oraculo.segundos,* contém o tempo máximo atribuído para a execução do processo;
- *Oraculo.resultado*, contém o resultado booleano por ele obtido, verdadeiro se foi encontrada uma prova dedutiva no caso do processo *Prover9* ou um contra-exemplo no caso do processo Mace4, tal como acima foi previamente referido;
- *Oraculo.este*, contém uma referência para o processo em execução;
- *Oraculo.sifn*, contém o caminho do ficheiro executável associado ao processo;
- *Oraculo.siar*, contém os argumentos necessários à execução do processo;
- *Oraculo.siwd*, contém o caminho do directório de trabalho do processo;
- *Oraculo.eid*, contém o número de identificação que o sistema operativo atribuiu ao processo.

O seu método constituinte *Oraculo.Run*, com o pseudocódigo listado na figura 2-4, assegura a leitura e tratamento dos dados contidos nos ficheiros de saída sequenciais ("*streams*") dos processos associados, sendo estes "*streams"*, lidos linha a linha até ser satisfeita uma das três seguintes condições:

- O tempo atribuído ao processo foi excedido;
- O resultado foi encontrado;
- A marca de fim de ficheiro for atingida (do "*stream*").

Tal como já foi referido, quando o método termina, termina tanto a "*thread"* que lhe está associada como a que está a ser executada em simultâneo. Pode assim surgir uma situação de erro originada por erros nos processos de chamada, na qual, ambas as "*threads"* são abortadas

sem retornar um resultado válido (um resultado é válido quando se verifica uma das três condições acima listadas).

Para prevenir este tipo de erros é utilizada a instrução *Thread.Sleep* que retém a execução do método durante o período de tempo especificado no seu argumento, possibilitando o término da "*thread*" associada caso exista um erro num dos processos (ver figura 2-4).

Na figura 2-2 é apresentado o fluxograma do método *Oraculo.Run*, e na figura 2-3 os fluxos de informação que existem entre as "*threads*" criadas pelas duas instâncias da classe e entre elas e a "*thread*" de chamada.

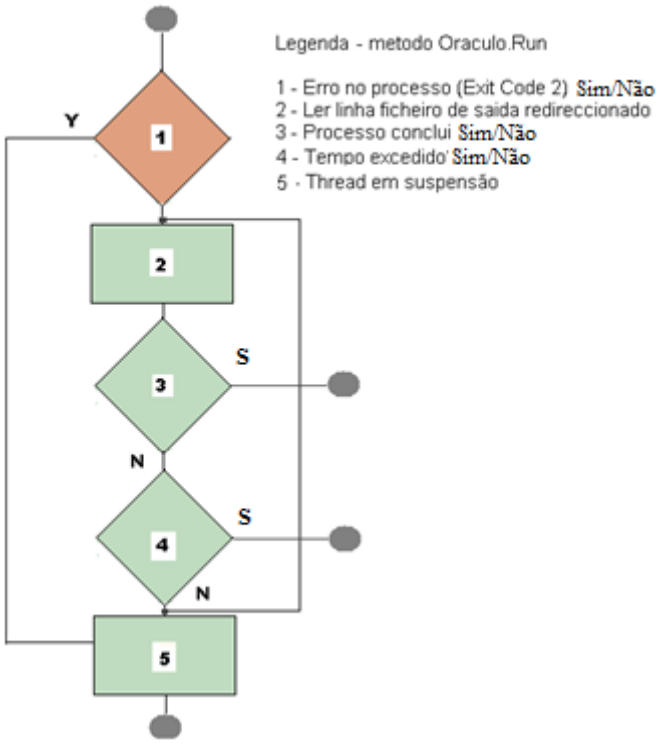

Figura 2-2 – Fluxograma do método *Oraculo.Run*

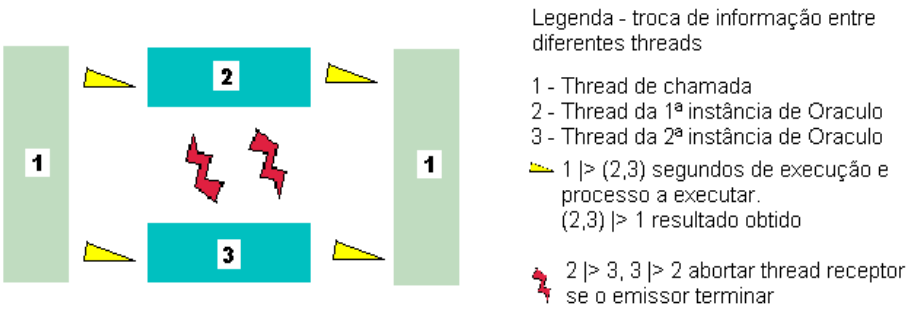

Figura 2-3 – Fluxos de informação existentes entre as duas "*threads*".

Esta classe, utilizada em todas as implementações, sofre uma alteração na sua estrutura quando é utilizado o acelerador *(+)* que consiste no acréscimo da propriedade *Oraculo.lstProver*, constituída por um contentor de "*strings*" no qual são salvaguardados os axiomas utilizados nas diferentes regras dedutivas encontradas durante a execução do algoritmo.

Assim durante a leitura do "*stream*" de saída do processo associado o contentor será preenchido com as "*strings*" que contenham a palavra "*assumption[numero_de_premissa]*". Estas "*strings*" constituem uma indicação fornecida pelo processo *Prover9.exe* de que o axioma referido pelo seu numero em "*[numero_de_premissa]*" foi utilizado na construção de uma prova.

Posteriormente a informação recolhida e salvaguardada no contentor será utilizada pelas diferentes classes que constituem as implementações dos algoritmos, como será descrito no subcapítulo 2.7, quando se proceder à análise das alterações necessárias ao código e estrutura dos programas para a sua implementação, apresenta-se no entanto desde já a alteração efectuada ao método *Oraculo.Run* quando ele é acrescido (linhas 11 e 12 da figura 2.4).

```
1:  this · este = newProcess
2:  parametriza argumentos do processo
3:  redirecciona o stream de saida do processo e inicia-o
4:  while ¬this · este · StandardOutput · EndOfStream do
5:    string l ← this · este · StandardOutput · Readline
6:    if l · Contains(endofproof) ∨ l · Contains(endofmodel) then
7:      this.resultado← true
8:      termina a execução
9:    else if o tempo de processamento excedeu o tempo maximo atribuido
      then
10:     termina a execução
11:   else if l · Contains([assumption]) then
12:     adicciona l ao contentor de strings Oraculo · lstprover
13:   end if
14: end while
15: Thread · Sleep(this · segundos ∗ 1000)
```

Figura 2-4 – Pseudocódigo do método *Oraculo.Run*

2.4. Algoritmo de *Tietze*

O algoritmo de *Tietze* para determinar os subconjuntos independentes num sistema finito de fecho utiliza uma estratégia de busca cega que conduz à construção de uma estrutura em árvore.

Neste algoritmo o conjunto inicial V é subdivido em n subconjuntos $V_n \subseteq V$ (um para cada axioma), sendo para cada um deles questionado se o axioma excluído é dedutível dos restantes, ou seja se $V \setminus a_n \rightarrow a_n$. Se sim, é criado um novo conjunto inicial $V_{an} = V \setminus a_n$ ao qual será de novo aplicado o processo atrás referido, se não, é testado o axioma seguinte. Se de todos os subconjuntos de V não for derivado nenhum outro conjunto inicial V_{an} então o subconjunto é independente constituindo uma base do sistema de fecho.

A figura 2-5 esquematiza o modo de actuação do algoritmo, considerando um conjunto inicial de quatro axiomas e a existência de cinco regras de derivação, sendo nela visível quer os novos conjuntos de axiomas criados, quer os conjuntos finais de axiomas (assinalados a cinzento).

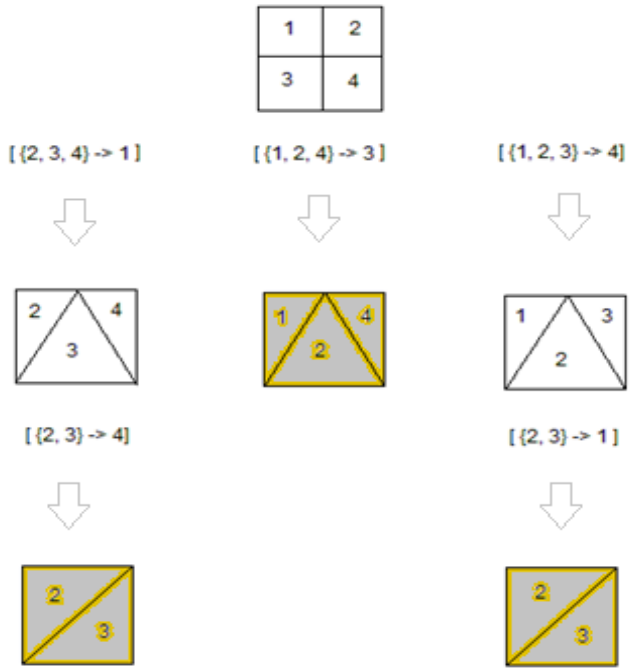

Figura 2-5 – Conjuntos de axiomas criados pelo algoritmo de *Tietze* durante a sua execução.

Este algoritmo foi implementado com o recurso às classes listadas no diagrama da figura 2-6, tendo no entanto, sido nela excluídas as relacionadas com a interface gráfica criadas pela interface de programação e também a classe *Oraculo* já referida no subcapítulo anterior.

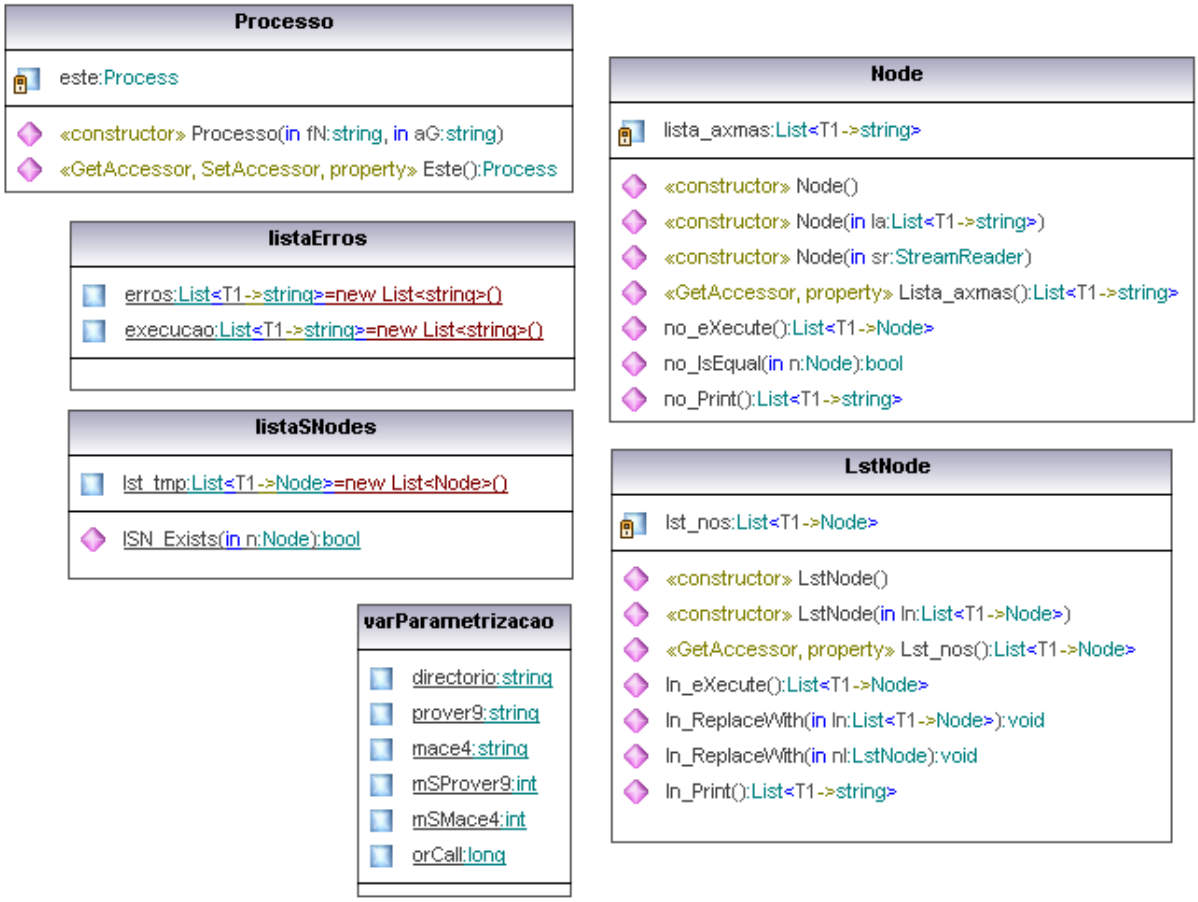

Figura 2-6 – Diagrama de classes na implementação *Tietze*.

De um modo geral estas classes podem ser subdivididas em dois tipos, as que são constituídas exclusivamente por propriedades estáticas e as que não, compondo o primeiro grupo as classes *listaErros*, *ListaSnodes* e *varParametrizacao* e o segundo as restantes três.

No primeiro grupo encontram-se as classes que parametrizam aspectos fundamentais na execução da aplicação, como é o caso de *varParametrização*, ou que vão recolhendo informação resultante da execução de modo a posteriormente a poderem disponibilizar ao utilizador sob a forma de um ficheiro de texto quando a execução terminar, casos de *listaSNodes* e *listaErros*.

No segundo grupo, constituído pelas classes *Node*, *LstNode* e *Oraculo* encontram-se as que directamente implementam o algoritmo, assegurando a chamada ao oráculo e a recolha e tratamento da informação que este provida.

Destaca-se no primeiro grupo, a classe *varParametrizacao* que parametriza a execução dos processos que constituem o oráculo contendo as propriedades:

- *varParametrizacao.directorio*, que identifica o directório de trabalho dos processos;
- *varParametrizacao.prover9*, que identifica o caminho do processo *Prover9.exe*;
- *varParametrizacao.mace4*, que identifica o caminho do processo *Mace4.exe;*
- *varParametrizacao*.mSProver9, que parametriza o tempo máximo de execução do processo *Prover9.exe;*
- *varParametrizacao*.mSMace4, que parametriza o tempo máximo de execução do processo *Mace4.exe*.

As outras classes do primeiro grupo, *listaErros* e *listaSNodes,* contêm respectivamente as propriedades *listaErros.execucao* e *listaErros.erros* e *listaSNodes.lst_tmp* onde são salvaguardadas as mensagens de erro e os conjuntos independentes obtidos.

Convém aqui fazer uma pausa na descrição das diferentes classes e abordar o tipo de erros que surgem devido à importância que adquirem no resultado final. Os erros podem ser de dois tipos:

- De execução - resultam de erros na execução dos processos *Prover9.exe* e *Mace4.exe*, já discutidos na descrição da classe *Oraculo* e cujas mensagens de erro são salvaguardadas no contentor *listaErros.execucao*;
- De tempo - ocorrem quando o oráculo não consegue obter uma resposta conclusiva no tempo estabelecido pelo utilizador e cujas mensagens de erro são salvaguardadas no contentor *listaErros.erros*.

Destes dois tipos destacam-se os erros de tempo que conduzem frequentemente a um resultado final erróneo tal como é exemplificado no quadro 2-1, onde as listas de nós terminais de uma execução sem erros e outra onde eles existiram divergem significativamente.

Este comportamento deve-se ao modo como o algoritmo é implementado, ou seja, dado ambos os processos não conseguirem encontrar uma prova ou um contra-exemplo, o resultado contido em ambas as propriedades das classes *Oraculo.Resultado* terá o valor lógico falso e consequentemente não existirá a criação de um novo nó inicial (ver figuras 2-5 e 2-8), terminando-se a derivação de um ramo da árvore de busca e constituindo-se o nó como uma

folha onde poderiam ser derivados novos ramos, caso o processo *Prover9.exe* tivesse tido tempo suficiente para encontrar uma prova.

Como solução para este tipo de erros, sugere-se o incremento do tempo máximo de processamento de um ou de ambos os processos, pois eles devem-se, tal como já foi dito, a não ter sido atribuído ao processo *Prover9.exe* ou ao processo *Mace4.exe* ou a ambos em simultâneo, o tempo de processamento necessário para a sua correcta execução.

Ficheiro de entrada	Resultado com erro	Resultado sem erro
(x * y) * z = x * (y * z). (x * g(x)) * x = x. (g(x) * x) * g(x) = g(x). (x * g(x)) * (g(y) * y) = (g(y) * y) * (x * g(x)). g(x * y) = g(y) * g(x). (x * y) * z = x * (y * g(g(z))).	[node 1] (x * y) * z = x * (y * z). (x * g(x)) * x = x. (g(x) * x) * g(x) = g(x). (x * g(x)) * (g(y) * y) = (g(y) * y) * (x * g(x)). g(x * y) = g(y) * g(x). [node 2] (x * g(x)) * x = x. (x * g(x)) * (g(y) * y) = (g(y) * y) * (x * g(x)). g(x * y) = g(y) * g(x). (x * y) * z = x * (y * g(g(z))).	[node 1] (x * y) * z = x * (y * z). (x * g(x)) * x = x. (g(x) * x) * g(x) = g(x). (x * g(x)) * (g(y) * y) = (g(y) * y) * (x * g(x)). [node 2] (x * g(x)) * x = x. (x * g(x)) * (g(y) * y) = (g(y) * y) * (x * g(x)). (x * y) * z = x * (y * g(g(z))).

Quadro 2-1 – Resultados finais com e sem a existência de erros.

No que respeita aos erros ditos de execução, estes são capturados por intermédio da função delegada *Oraculo.este_Exited*, que testa o seu código de finalização do processo e é executada quando o evento do sistema operativo fim de processo é disparado. Se este código for 2 (indicador de que existiu um erro relacionado com os ficheiros de entrada ou de saída nos processos *Prover9.exe* e/ou *Mace4.exe*) é adicionada a mensagem de erro fornecida pelo sistema operativo ao contentor *listaErros.execucao*.

O segundo grupo como já foi referido é constituído pelas classes *Node*, *LstNode* e *Oraculo*, sendo estas classes as responsáveis pela construção dos nós sejam eles iniciais, finais ou intermédios. O modo como são construídos os nós é o resultado da formalização do algoritmo por intermédio das classes *Node* e *LstNode* e da execução dos seus métodos *Node.no_eXecute*, *LstNode.ln_eXecute* e *LstNode.ln_ReplaceWith*, o que será abordado no final deste subcapítulo procedendo-se em seguida à descrição destas três ultimas classes.

A classe *Node* tem como propriedade o contentor de "*strings*" *Node.lista_axmas*, e podem-se criar instâncias desta classe de três modos distintos:

- Através de um construtor nulo, criando-se uma instância da classe contendo uma lista de "*strings"* vazia. Este construtor é dos três o que assume uma maior importância devido a todas as implementações serem iniciadas com a criação de uma lista de axiomas pelo seu intermédio;
- Através de um construtor que tem como argumento uma lista de "*strings";*
- Através de um construtor que tem como argumento um ficheiro de texto.

Contém também três métodos:

- *Node.no_eXecute*, que formaliza e executa grande parte do algoritmo, construindo dois ficheiros de texto com a extensão "*.in"* e criando duas instâncias da classe *Oraculo* e dois *threats* de execução (ver figura 2-8);
- *Node.no_IsEqual*, que testa a igualdade entre duas listas de axiomas, resumindo-se no caso deste algoritmo a um teste de igualdade entre duas listas de "*strings";*
- *Node.no_Print*, que converte o conteúdo do contentor *Node.lista_axmas* num formato que facilite a sua visualização.

A última classe a ser descrita neste subcapítulo é a *LstNode* cuja única propriedade *LstNode.lst_nos* é um contentor de instâncias da classe *Node*, o que se pode considerar como sendo a formalização de um nível na árvore de derivação (ver figura 2-7). Esta classe contém dois construtores:

- O construtor nulo que cria instâncias com o contentor *LstNode.lst_nos* vazio;
- O construtor cujo argumento é um contentor de instâncias da classe *Node* que cria uma instância da classe com o contentor *LstNode.lst_nos*, contendo as instâncias fornecidas no argumento.

E três métodos:

- *LstNode.ln_Print*, sem argumentos e que retorna um contentor de "*strings"*;
- *LstNode.ln_ReplaceWith* que actualiza o contentor *LstNode.lst_nos* com o conjunto de instâncias da classe *Node* que lhe é fornecido no argumento;
- *LstNode.ln_eXecute,* também sem argumentos e que retorna uma instância da classe *List<Node>* (ver figura 2-9).

Tendo sido descritas as classes envolvidas na implementação, vamos abordar a execução do algoritmo, primeiro de um modo geral procedendo-se de seguida à análise dos métodos que mais interferem na sua implementação.

Iniciando com um exemplo prático listado na figura 2-7, é neste visível, que existindo as cinco regras de derivação $R_1 = [\{2, 3, 4\} \rightarrow 1]$, $R_2 = [\{1, 2, 4\} \rightarrow 3]$, $R_3 = [\{1, 2, 4\} \rightarrow 1]$, $R_4 = [\{2, 3\} \rightarrow 1]$ e $R_5 = [\{2, 3\} \rightarrow 4]$, na derivação do nó inicial serão construídos três novos nós, um para cada conjunto de axiomas constituinte da regra de derivação utilizada, construindo-se assim o primeiro nível da arvore de derivação.

Aplicando de novo o algoritmo serão criados dois novos nós contento ambos o conjunto de axiomas {2, 3}, dado as únicas regras dedutivas aplicáveis neste nível serem $R_4 = [\{2, 3\} \rightarrow 1]$ e $R_5 = [\{2, 3\} \rightarrow 4]$, construindo-se assim o segundo nível. Não existindo mais regras dedutivas aplicáveis o algoritmo termina, tendo como resultado final os nós {1, 2, 4} e {2, 3}.

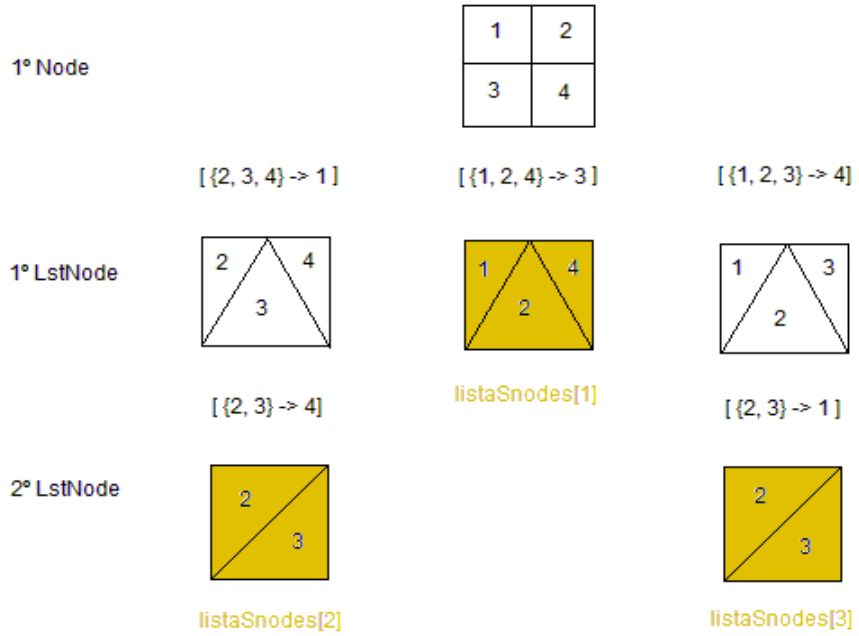

Figura 2-7 – Esquema da criação de nós durante a execução da implementação *Tietze*.

O modo como o algoritmo é implementado e os nós construídos é quase exclusivamente o resultado da sua formalização por intermédio das classes *Node* e *LstNode* e da execução dos métodos *Node.no_eXecute* e *LstNode.ln_eXecute*.

Este último método, cujo pseudocódigo se encontra listado na figura 2-8, aplica a cada instância da classe *Node* contida no contentor *LstNode.lst_Nos* o método *Node.no_eXecute* sendo as instâncias por este retornadas salvaguardadas num contentor temporário, até o método *Node.no_eXecute* devolver um contentor vazio indicando que o nó sobre o qual foi executado o método é terminal, constituindo uma folha na arvore de derivação.

Quanto ao método *Node.no_eXecute* com o pseudocódigo listado na figura 2-9 inicia a sua execução criando um contentor de nós temporário que no final da sua execução retornará ao método de chamada, e em seguida percorrerá todos os axiomas contidos na sua lista de axiomas criando os ficheiros de entrada para os processos *Prover9.exe* e *Mace4.exe* e executando-os em duas "*threads"* distintas, mantendo-se a "*thread*" onde o método é executado num estado de espera forçado devido a um ciclo infinito (ver estrutura "*loop … end loop*" na figura 2-9).

```
1:  List⟨Node⟩listanosrepositorio ← newList⟨Node⟩();
2:  for all Node n ∈ this · lst_nos do
3:     List⟨Node⟩listanostemporarios ← newList⟨Node⟩ · n.no_eXecute;
4:     if listanostemporarios · Count ≠ 0 then
5:        adiciona nós a listanosrepositorio
6:     else if ¬listaSNodes · lSN_Exists(n) then
7:        adiciona nó n a listaSNodes · lst_tmp
8:     end if
9:  end for
10: retorna o contentor listanosrepositorio
```

Figura 2-8 – Pseudocódigo do método *LstNode.ln_eXecute.*

Quando um dos processos termina por um dos três motivos já previamente apontados na página 6 (aquando da discussão da classe *Oraculo),* as "*threats"* onde os processos são executados são eliminadas, uma pelo sistema operativo pelo término do seu método *Thread.Run* e outra pelo código inserido na estrutura "*loop … end loop*", retornando o controlo da execução à "*thread*" de chamada, testando de seguida o valor das propriedades *Oraculo.resultado* de ambas as instâncias.

Se, o resultado da instância que executa o processo *Prover9,exe* for verdadeiro então é criado um novo nó sem o axioma dedutível ou seja, uma nova lista de axiomas com a exclusão do axioma dedutível, tal como é exemplificado nas linhas 20: a 28: da figura 2-9 e este, adicionado ao contentor de retorno.

Tendo-se derivado um nó e se dele não tiverem sido criadas novas instâncias, conclui-se que é terminal sendo por sua vez adicionado ao contentor estático *listaSNodes.lst_tmp* da classe *listaSNodes* caso ainda não exista uma instância igual no referido contentor, tal como também é exemplificado nas linhas 6: a 8: da figura 2-8.

Para terminar este subcapítulo é listado e descrito o código da figura 2-10, que a par dos métodos e propriedades das classes que utiliza implementa o algoritmo de *Tietze*.

```
1:  List⟨Node⟩nln ← newList⟨Node⟩();
2:  for all string s ∈ this · lista_axmas do
3:    cria StreamWriter p9 e m4 e associa-os aos ficheiros "ap · in" e "am · in"
4:    escreve string s em p9 e m4
5:    for all string r ∈ this · lista_axmas do
6:      if r ≠ s then
7:        escreve string r em p9 e m4
8:      end if
9:    end for
10:   cria duas instâncias da classe Oraculo prover e mace4 e as threads de ex-
      ecução proverThread e mace4Thread associadas aos métodos prover.Run
      e mace4.Run iniciando-as através dos métodos proverThread.Start() e
      mace4Thread.Start()
11:   loop
12:     if ¬ proverThread.IsAlive then
13:       aborta mace4Thread e sai do ciclo
14:     else if ¬ mace4Thread.IsAlive then
15:       aborta proverThread e sai do ciclo
16:     end if
17:   end loop
18:   if prover.Resultado = false ∧ mace4.Resultado = false then
19:     adiciona mensagem de erro a listaErros.erros
20:   else if prover.Resultado = true then
21:     Node nl ← new Node()
22:     for all string v ∈ this · lista_axmas do
23:       if v ≠ s then
24:         adiciona v a nl · lista_axmas
25:       end if
26:     end for
27:     adiciona nl ao contentor nln
28:   end if
29: end for
30: retorna o contentor nln contendo o conjunto de nós do proximo nivel de
    derivação
```

Figura 2-9 – Pseudocódigo do método *Node.no_eXecute.*

Assim, é criada uma instância da classe *Node* por intermédio do seu construtor que tem como argumento um "*stream*" de leitura de um ficheiro de texto e duas instâncias da classe *LstNode*, uma recorrendo ao construtor nulo e outra utilizando o construtor que admite como argumento um contentor de nós, sendo o conteúdo do argumento o resultado da aplicação do método *Node.no_eXecute* ao nó inicial.

Em seguida é executado um ciclo em que o contentor de uma instância vai sendo actualizado com o resultado da aplicação do método anterior à outra instância, terminando o ciclo quando ambas as instâncias forem nulas tendo os nós terminais entretanto criados sido salvaguardados

no contentor *listaSnodes.lst_tmp*, sendo no final o seu conteúdo impresso num ficheiro de texto.

```
1: cria stream de leitura para um ficheiro em disco
2: Node n ← new Node(stream criado na linha anterior)
3: List ⟨Node⟩listanostemporarios1 ← newList⟨Node⟩n · no_eXecute
4: List ⟨Node⟩listanostemporarios2 ← newList⟨Node⟩
5: loop
6:    listanostemporarios2 · ln_ReplaceWith(listanostemporarios1 ·
      ln_eXecute)
7:    if listanostemporarios2 · Lst_nos.Count ≠ 0 then
8:       listanostemporarios1 · ln_ReplaceWith(listanostemporarios2)
9:    else
10:      deixa o ciclo
11:   end if
12: end loop
13: if listaSNodes · lst_tmp · Count ≠ 0 then
14:   for all Node n ∈ listaSNodes · lst_tmp do
15:      escreve conteudo ficheiro de saída
16:   end for
17: end if
```

Figura 2-10 – Pseudocódigo do método que implementa o algoritmo de *Tietze*.

2.5. Algoritmo *SA*

O algoritmo SA baseia-se na subdivisão dos diferentes conjuntos de axiomas em três subconjuntos disjuntos. Assim, o conjunto inicial de axiomas V é subdividido em dois subconjuntos de axiomas $V_I \subseteq V$ e $V_V \subseteq V$, em que $V_I \cap V_V = \emptyset$, com V_V contendo os axiomas a_n tais que $V \setminus a_n \rightarrow a_n$, e V_I os restantes. O subconjunto V_V é por sua vez subdividido em dois subconjuntos, V_R e $V_V \setminus V_R$ sendo V_R constituído pelos axiomas cuja dedução pode ser efectuada utilizando exclusivamente axiomas contidos em V_I, obtendo-se três subconjuntos V_I, $V_V \setminus V_R$ e V_R, designados respectivamente por subconjunto de invariantes, de variantes e de redundantes.

O algoritmo prossegue com a exclusão dos axiomas redundantes e a constituição de $|V_v \setminus V_R|$ subconjuntos tal como exemplificado na figura 2-11. Aos subconjuntos assim obtidos é de novo aplicado o processo supracitado até que $|V_V| = 0$.

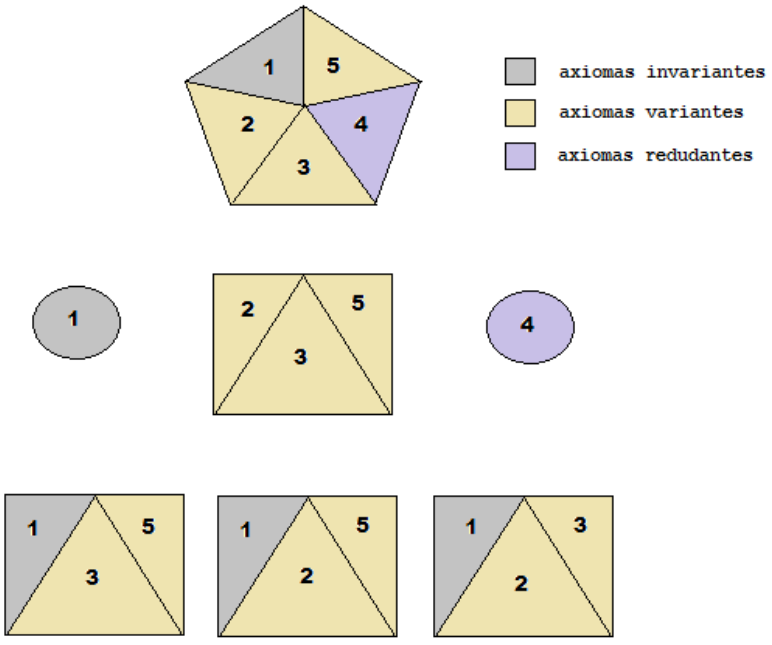

Figura 2-11 – Representação em diagrama do algoritmo *SA*.

A implementação deste algoritmo utiliza o mesmo conjunto de classes que o anterior, existindo no entanto alterações significativas nas classes *Node* e *LstNode*, na primeira delas no seu número de propriedades e construtores e número e código dos métodos que a constituem, e na segunda ao código do método *LstNode.ln_eXecute*.

No lado direito da figura 2-12 á apresentado o diagrama da classe *Node*. Esta classe contém agora duas propriedades, ambas contentores de "*strings"*, *Node.lista_invts* e *Node.lista_vrtes* onde são salvaguardados os axiomas invariantes e variantes.

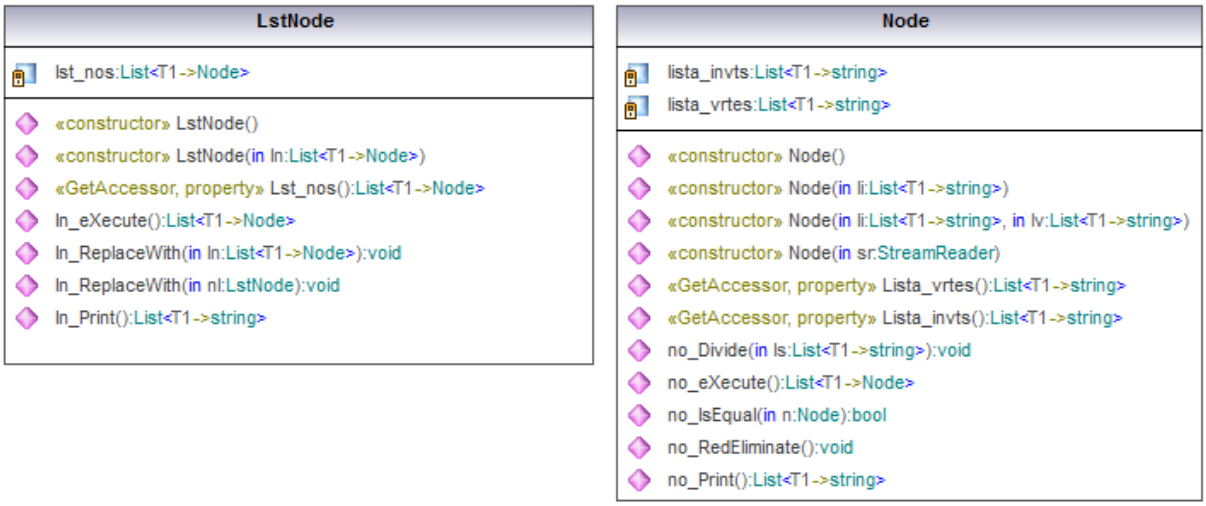

Figura 2-12 – Classes *LstNode* e *Node* na implementação *SA.*

Existem agora para esta classe quatro tipos de construtores:

- O construtor sem argumentos que cria instâncias nulas da classe, ou seja, cujos contentores de "*strings*" estão vazios;
- O construtor com dois contentores de "*strings"* como argumentos;
- O construtor com um contentor de "*strings"* como argumento;
- O construtor com um ficheiro de texto como argumento que tal como na implementação do algoritmo de *Tietze*, é o responsável pela criação do conjunto inicial de axiomas.

Destaca-se também a completa reformulação do código do método *Node.no_eXecute* que adiante será examinada e a inclusão de dois novos métodos o *Node.no_Divide* e o *Node.no_RedEliminate* utilizados, o primeiro deles na divisão do conjunto inicial de axiomas nos subconjuntos de axiomas invariantes e variantes e o segundo na eliminação dos axiomas redundantes do conjunto de variantes. Estes dois últimos métodos são unicamente utilizados na construção do nó inicial sendo para os restantes nós, utilizado o método *Node.no_eXecute* que por intermédio de dois ciclos consecutivos assegura grande parte do algoritmo.

Quanto à classe *LstNode* e ao seu método *LstNode.ln_eXecute* com o pseudocódigo listado na figura 2-13, comparativamente ao do algoritmo de *Tietze* listado na figura 2-9, sobressai que, a inclusão dos nós intermédios em futuras interacções ou a sua classificação em nós finais não

é mais decidida pelo número de axiomas contidos no contentor de retorno do método *Node.no_eXecute*, mas sim pelo número de axiomas variantes que cada um dos nós contém.

```
1: List⟨Node⟩listanosrepositorio ← newList⟨Node⟩
2: for all Node n ∈ this · lst_nos do
3:    List⟨Node⟩listanostemporarios ← newList⟨Node⟩ · n.no_eXecute
4:    for all Node d ∈ listanostemporarios do
5:       if d · Lista_vrtes· > 0 then
6:          adiciona nó a listanosrepositorio
7:       else if ¬listaSNodes · lSN_Exists(d) then
8:          adiciona nó n a listaSNodes · lst_tmp
9:       end if
10:   end for
11: end for
12: retorna o contentor listanosrepositorio
```

Figura 2-13 – Pseudocódigo do método *LstNode.ln_eXecute*.

Voltando à classe *Node* e ao seu método *Node.no_eXecute*, por sua vez listado na figura 2-14, sendo nela visíveis os dois ciclos acima referidos. No primeiro é efectuada a primeira divisão construindo-se dois subconjuntos de variantes e invariantes. Se não persistirem axiomas variantes é adicionado ao contentor de retorno um nó contendo exclusivamente axiomas invariantes, se pelo contrário persistirem, no segundo ciclo é efectuada a eliminação do subconjunto de redundantes.

Após a eliminação dos redundantes se já não existirem axiomas variantes é também criado e adicionado ao contentor de retorno um nó contendo exclusivamente axiomas invariantes, senão é adicionado a este mesmo contentor um novo nó com os axiomas contidos nos contentores de "*strings*" temporários *listastringsinvariantes* e *listastringsvariantes2*.

Nesta implementação mantém-se válido, o que foi referido na do algoritmo de *Tietze* quanto ao controle de erros, em especial no que respeita à não fiabilidade do resultado final quando existem erros relacionados com o limite de tempo de execução dos processos, sendo o controlo de erros efectuado na divisão dos subconjuntos de axiomas e na eliminação de axiomas redundante dando origem a duas mensagens distintas.

O código que implementa este algoritmo é praticamente igual ao listado na figura 2-10, construindo-se também neste caso dois contentores de nós, um com o resultado da interacção anterior e outro com a da interacção corrente, terminando-se o processo quando o último estiver vazio, residindo a diferença entre eles na eliminação dos redundantes após a leitura da "*stream*" associado ficheiro de entrada.

```
1: List⟨Node⟩listanosrepositorio ← newList⟨Node⟩
2: List⟨string⟩listatemporariainvariantes ← newList⟨string⟩
3: List⟨string⟩listatemporariavariantes ← newList⟨string⟩
4: for all string s ∈ this · lista_vrtes do
5:    cria StreamWriter p9 e m4 e associa-os aos ficheiros "ap · in" e "am · in"
6:    escreve informação necessária à execução dos processos em p9 e m4
7:    cria duas instâncias da classe Oraculo prover e mace4 e as threads de ex-
      ecução proverThread e mace4Thread associadas aos métodos prover.Run
      e mace4.Run iniciando-as através dos métodos proverThread.Start e
      mace4Thread.Start
8:    loop
9:       testa o término das Threads e abandona o ciclo se uma delas terminar
10:   end loop
11:   if prover.Resultado = false ∧ mace4.Resultado = false then
12:      adiciona mensagem de erro a listaErros.erros
13:      adiciona string s listatemporariainvariantes
14:   else if prover.Resultado = true then
15:      adiciona string s listatemporariavariantes
16:   else if mace4.Resultado = true then
17:      adiciona string s listatemporariainvariantes
18:   end if
19: end for
20: if listatemporariavariantes · Count > 0 then
21:   List⟨string⟩listatemporariavariantes2 ← newList⟨string⟩
22:   for all string s1 ∈ this · listatemporariavariantes do
23:      cria StreamWriter p9 e m4 e associa-os aos ficheiros "ap·in" e "am·in"
24:      escreve informação necessária à execução dos processos em p9 e m4
25:      cria duas instâncias da classe Oraculo prover e mace4 e as threads
         de execução proverThread e mace4Thread associadas aos métodos
         prover.Run e mace4.Run iniciando-as através dos métodos prover-
         Thread.Start e mace4Thread.Start
26:      loop
27:         testa o término das Threads e abandona o ciclo se uma delas terminar
28:      end loop
29:      if prover.Resultado = false ∧ mace4.Resultado = false then
30:         adiciona mensagem de erro a listaErros.erros
31:      else if mace4.Resultado = true then
32:         adiciona string s listatemporariavariantes2
33:      end if
34:      if listatemporariavariantes2 · Count ≠ 0 then
35:         cria um nó com a listatemporariainvariantes como argumento e adi-
            ciona ao contentor listanosrepositorio
36:      else
37:         cria um nó com a listatemporariainvariantes e listatemporariavari-
            antes como argumentos e adiciona ao contentor listanosrepositorio
38:      end if
39:   end for
40: else if listatemporariavariantes · Count = 0 then
41:   cria um nó com a listatemporariainvariantes como argumento e adiciona
      ao contentor listanosrepositorio
42: end if
43: retorna a listanosrepositorio
```

Figura 2-14 – Pseudocódigo do método *Node.no_eXecute*.

A terminar este subcapítulo são apresentadas as figuras 2-15 e 2-16 contendo respectivamente os pseudocódigos dos métodos *Node.no_Divide* e *Node.no_RedEliminate*.

```
1: for all string s ∈ parametrolistadestrings do
2:    cria StreamWriter p9 e m4 e associa-os aos ficheiros "ap · in" e "am · in"
3:    escreve informação necessária à execução dos processos em p9 e m4
4:    cria duas instâncias da classe Oraculo prover e mace4 e as threads de ex-
      ecução proverThread e mace4Thread associadas aos métodos prover.Run
      e mace4.Run iniciando-as através dos métodos proverThread.Start e
      mace4Thread.Start
5:    loop
6:       testa o término das Threads e abandona o ciclo se uma delas terminar
7:    end loop
8:    if prover.Resultado = false ∧ mace4.Resultado = false then
9:       adiciona mensagem de erro a listaErros.erros
10:      adiciona string s this · lista_invts
11:   else if prover.Resultado = true then
12:      adiciona string s this · lista_vrtes
13:   else if mace4.Resultado = true then
14:      adiciona string s this · lista_invts
15:   end if
16: end for
```

Figura 2-15 – Pseudocódigo do método *Node.no_Divide*.

```
1: List⟨string⟩listatemporariavariantes ← newList⟨string⟩
2: for all string s ∈ this · lista_vrtes do
3:    cria StreamWriter p9 e m4 e associa-os aos ficheiros "ap · in" e "am · in"
4:    escreve informação necessária à execução dos processos em p9 e m4
5:    cria duas instâncias da classe Oraculo prover e mace4 e as threads de ex-
      ecução proverThread e mace4Thread associadas aos métodos prover.Run
      e mace4.Run iniciando-as através dos métodos proverThread.Start e
      mace4Thread.Start
6:    loop
7:       testa o término das Threads e abandona o ciclo se uma delas terminar
8:    end loop
9:    if prover.Resultado = false ∧ mace4.Resultado = false then
10:      adiciona mensagem de erro a listaErros.erros
11:      adiciona string s listatemporariavariantes
12:   else if mace4.Resultado = true then
13:      adiciona string s listatemporariavariantes
14:   end if
15: end for
16: this · lista_vrtes ← listatemporariavariantes
```

Figura 2-16 – Pseudocódigo do método *Node.no_ReadEliminate*.

2.6. Algoritmo *Tietze-SA*

O algoritmo de *Tietze-SA* é um híbrido dos dois anteriormente referidos, sendo ao conjunto de axiomas apresentado, numa primeira fase aplicada a subdivisão em três conjuntos de axiomas disjuntos tal como no algoritmo *SA* e numa segunda fase o algoritmo de *Tietze* a $(V_V \setminus V_R) \cup V_I$, com a diferença de neste algoritmo existir a herança do conjunto V_I, como é exemplificado na figura 2-17.

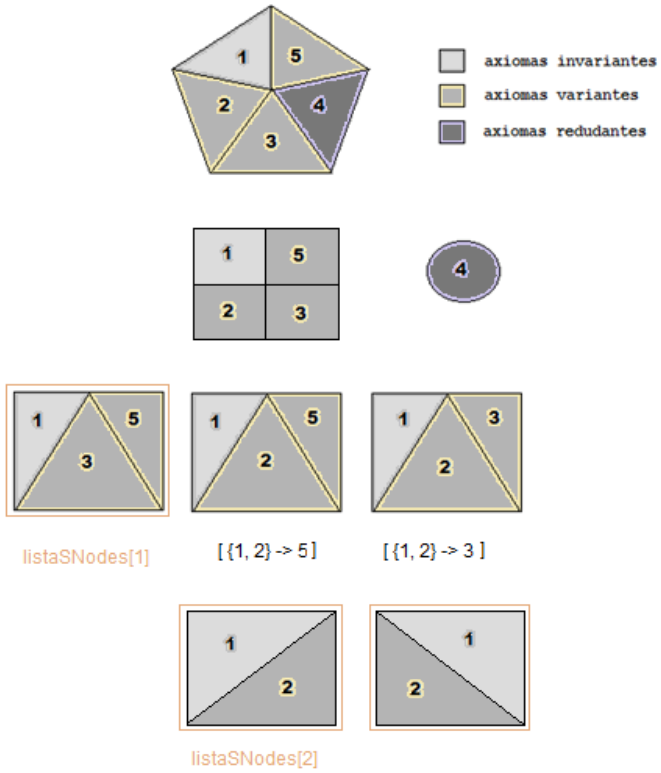

Figura 2-17 – Diagrama da criação de nós na implementação *Tietze-SA*.

Na implementação deste algoritmo é utilizado o mesmo conjunto de classes que nos dois precedentes, tendo como diferenças em relação à do algoritmo *SA* a inexistência do construtor com um conjunto de contentores como único argumento e no código dos métodos *Node.no_eXecute* e *LstNode.ln_eXecute*.

Como diferenças em relação à implementação do algoritmo de *Tietze*, existem as mesmas nos métodos, construtores e propriedades da classe *Node* que entre a implementação de *Tietze* e a de SA, não existindo contudo diferenças no código do método *LstNode.Ln_eXecute* continuando deste modo a ser feito o controlo do termino do algoritmo tal como na implementação de *Tietze*, pelo numero de nós criados pelo método *Node.no_eXecute*.

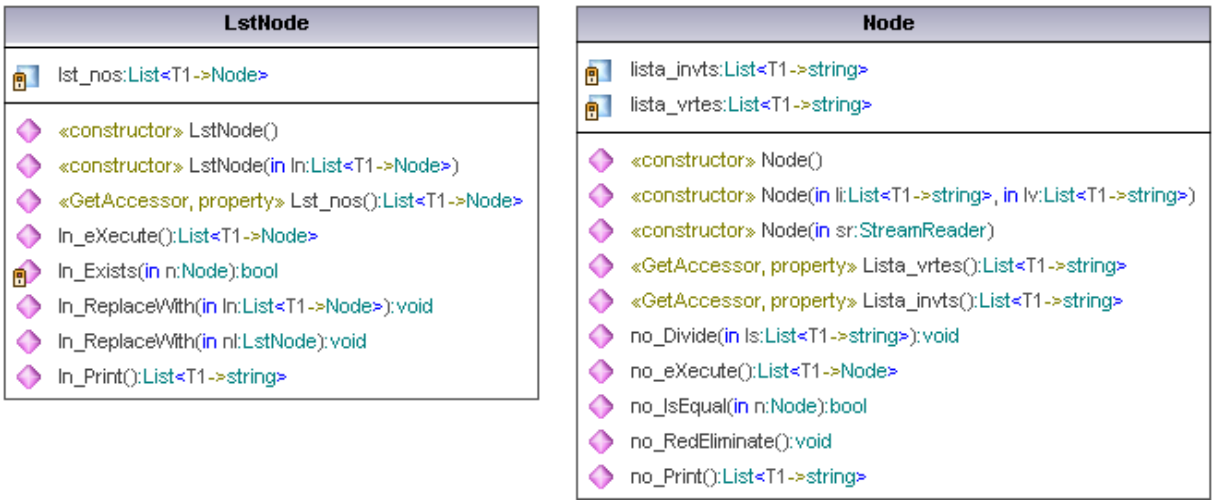

Figura 2-18 – Diagramas das classes *LstNode* e *Node* na implementação *Tietze-SA*.

Na figura 2-18 encontram-se listados os diagramas das classes *LstNode* e *Node* e nas figuras 2-19 e 2-20 são listados os pseudocódigos dos métodos *Node.no_eXecute* e *LstNode.Ln_eXecute*.

1: $List\langle Node\rangle listanosrepositorio \leftarrow newList\langle Node\rangle$
2: **for all** $string\ s \in this \cdot lista\_vrtes$ **do**
3: cria StreamWriter p9 e m4 e associa-os aos ficheiros "$ap \cdot in$" e "$am \cdot in$"
4: escreve informação necessária à execução dos processos em p9 e m4
5: cria duas instâncias da classe Oraculo prover e mace4 e as threads de execução proverThread e mace4Thread associadas aos métodos prover.Run e mace4.Run iniciando-as através dos métodos proverThread.Start e mace4Thread.Start
6: **loop**
7: testa o término das Threads e abandona o ciclo se uma delas terminar
8: **end loop**
9: **if** prover.Resultado = false $\wedge$ mace4.Resultado = false **then**
10: adiciona mensagem de erro a listaErros.erros
11: **else if** prover.Resultado = true **then**
12: $List\langle string\rangle listatemporariainvariantes \leftarrow newList\langle string\rangle(this \cdot lista\_invts)$
13: $List\langle string\rangle listatemporariavariantes \leftarrow newList\langle string\rangle$
14: **for all** $strings2 \in this \cdot lista\_vrtes$ **do**
15: **if** $s2 \neq s$ **then**
16: adiciona string s2 a listatemporariavariantes
17: **end if**
18: **end for**
19: cria um nó com a listatemporariainvariantes e listatemporariavariantes como argumentos e adiciona ao contentor listanosrepositorio
20: **end if**
21: **end for**
22: retorna a listanosrepositorio

Figura 2-19 – Pseudocódigo do método *Node.no_eXecute*.

```
1: List⟨Node⟩listanosrepositorio ← newList⟨Node⟩
2: for all Node n ∈ this · lst_nos do
3:    List⟨Node⟩listanostemporarios ← newList⟨string⟩(n · no_eXecute)
4:    if listanostemporarios ≠ s then
5:       adiciona nó n a listanostemporarios
6:    else if ¬(n ∈ listaSNodes · lst_tmp) then
7:       adiciona o nó n a listaSNodes · lst_tmp
8:    end if
9: end for
10: retorna listanosrepositorio
```

Figura 2-20 – Pseudocódigo do método *LstNode.Ln_eXecute*.

2.7. Acelerador *(+)*

O acelerador *(+)* tem como princípio a criação de um historial de dedutibilidades para cada um dos axiomas envolvidos pela salvaguarda dos diferentes conjuntos de axiomas que participam na sua dedução, de modo a evitar o questionamento do oráculo para deduções que com base no referido historial de dedutibilidades se possa assegurar a validade.

Recorrendo a um exemplo, suponhamos que dado o conjunto de axiomas V = {1, 2, 4, 6, 7}, se contivermos previamente salvaguardada a regra dedutiva R_1 = [{1, 2} → 6], eliminamos automaticamente a necessidade de questionar o oráculo sobre a validade das regras R_2 = [{1, 2, 4} → 6], R_3 = [{1, 2, 7} → 6], R_4 = [{1, 2, 4, 7} → 6] e R_1 = [{1, 2} → 6].

Quando se utiliza este acelerador existe assim a necessidade de acrescentar às implementações atrás estudadas uma estrutura de dados capaz de armazenar as regras dedutivas encontradas e um mecanismo que possibilite a sua gestão. A adição deste mecanismo implica alterações na estrutura de classes com a criação de três novas classes e também alterações nas classes previamente estudadas, *Oraculo* e *Node*.

No texto que se segue, serão abordadas em primeiro lugar os métodos e propriedades destas novas classes denominadas *listaGeradores*, *Geradores* e *Axioma*, e em seguida as alterações requeridas nos diferentes métodos das outras classes.

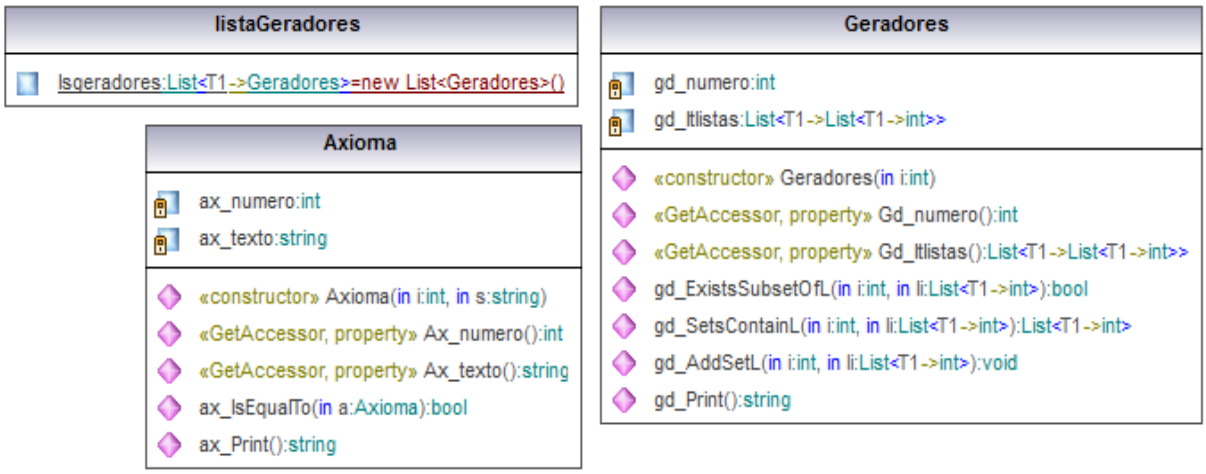

Figura 2-21 – Diagrama de classes de *listaGeradores*, *Geradores* e *Axioma*.

Com os diagramas de classes listados na figura 2-21, a classe *listaGeradores* é constituída exclusivamente por um contentor estático de instâncias da classe *Geradores*, o

listaGeradores.lsgeradores, constituindo-se como um ficheiro virtual que ao longo da implementação arquiva as regras de dedutibilidade dos diferentes axiomas.

A classe *Geradores* apresenta um único construtor que tem como argumento um inteiro sendo utilizado para criar as diferentes instâncias contidas em *listaGeradores.lsGeradores* por intermédio de um dos construtores da classe *Node*, podendo-se considerar estas instâncias como um ficheiro de índices entre os diferentes axiomas e os conjuntos de axiomas que os deduzem. Contém duas propriedades:

- *Geradores.gd_numero* – armazena uma identificação do axioma, igual ao valor da propriedade *Axioma.ax_numero* na instância da classe *Axioma;*
- *Geradores.gd_listas* – armazena como listas de inteiros o conjunto dos conjuntos de axiomas que deduzem o axioma indexado por *Geradores.gd_numero*. Retomando o exemplo da página anterior, teríamos para uma instância desta classe com o valor de *Geradores.gd_numero* igual a 6 contido neste contentor o conjunto de inteiros {1, 2}.

E como métodos:

- *Geradores.gd_ExistsSubsetOfL* - método cujos argumentos são um inteiro e uma lista de inteiros, retorna um valor booleano indicando se no contentor de conjuntos de axiomas do axioma indexado pelo primeiro argumento existe algum que esteja contido na lista de inteiros que lhe é passado como argumento. Se tal suceder o valor de retorno será verdadeiro indicando que já existe para o axioma indicado uma regra de dedutibilidade cujo conjunto de premissas é menor ou igual ao do segundo argumento.
- *Geradores.gd_SetsContainL* - método que retorna uma lista de inteiros indicando quais os conjuntos de axiomas que estão contidos no conjunto que lhe é passado como argumento sendo utilizado pelo método *Geradores.gd_AddSetL* desta mesma classe e este por sua vez utilizado pelo método *Node.no_eXecute* da classe *Node* quando são criados novos conjuntos de axiomas, representados por listas de inteiros, a inserir numa das instâncias do contentor *listaGeradores.lsgeradores*.
- *Geradores.gd_AddSetL* – método que adiciona um nova prova dedutiva ao índice do axioma no contentor *listaGeradores.lsgeradores*, actualizando-o e permitindo que futuras derivações contemplem as diferentes provas encontradas durante a execução.
- *Geradores.gd_Print* – método que transforma a informação contida nas suas propriedades numa "*string*" sendo utilizado na criação do ficheiro de saída.

A classe *Axioma* contém duas propriedades, *Axioma.ax_numero* e *Axioma.ax_texto*, e dois métodos:

- *Axioma.ax_IsEqualTo* – recebe como argumento uma instância da classe *Axioma* e indica se é igual a instância proprietária sendo frequentemente utilizado pelos métodos *Node.no_NmLstWithoutX* e *Node.no_StLstWithoutX* na construção e consulta das listas de inteiros que representam o conjunto de axiomas da regra de dedutibilidade;
- Axioma.ax_Print - formata a informação contida nas propriedades numa "*string*", sendo utilizado na criação do ficheiro de saída.

Quanto às alterações necessárias aos códigos das demais classes referem-se somente os efectuados nos métodos da classes *Node*, devido às efectuadas na classe *Oraculo* já terem sido previamente referidas no subcapítulo 2.3.

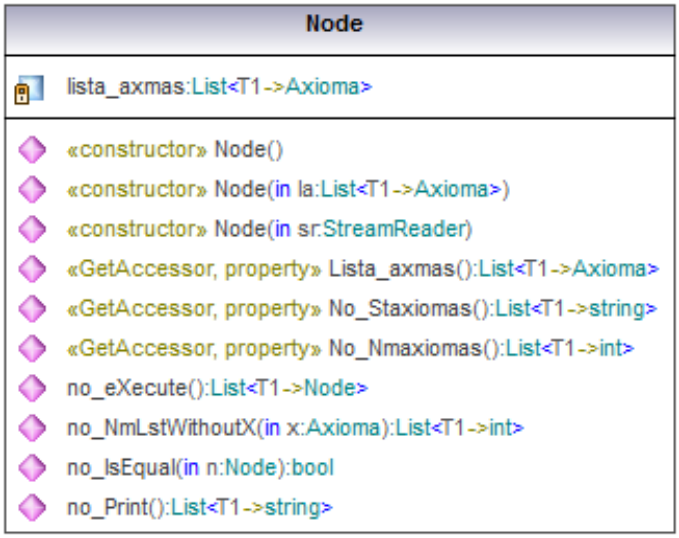

Figura 2-22 – Diagrama da classe *Node* para a implementação *Tietze (+).*

Tomando como exemplo uma implementação que tenha como base o algoritmo de *Tietze*, temos, tal como é visível no pseudocódigo listado na figura 2-23, no construtor da classe *Node* que admite um ficheiro de texto como argumento, a criação a par do nó inicial das instâncias da classe *Geradores* que constituem os elementos do contentor estático da classe *listaGeradores*.

Analisando este pseudocódigo verifica-se, que é definida e inicializada um variável que será utilizada na atribuição de um número sequencial a cada instância da classes *Axioma* e *Geradores* construindo-se assim uma indexação entre ambas.

```
1: this · lista_axmas ← newList⟨Axioma⟩
2: contador ← 1
3: listaGeradores · lsgeradores ← ∅
4: while ¬(fim de streampassadocomoparametro) do
5:   linha ← streampassadocomoparametro · ReadLine
6:   if linha não está vazia ou é comentário then
7:     cria instância da classe Axioma com o construtor
       Axioma(contador, linha) e adiciona-a a this · lista_axmas
8:     cria instância da classe Geradores construtor Gerador(contador) e
       adiciona-a a this · listaGeradores · lsgeradores
9:     contador = contador + 1
10:  end if
11: end while
```

Figura 2-23 – Pseudocódigo do construtor da classe *Node*.

```
1: List⟨Node⟩listanosrepositorio ← newList⟨Node⟩
2: for all Axioma v ∈ this · lista_axmas do
3:   if não existir na listaGeradores · lsgeradores[indicev] um subconjunto
     da regra deductiva actual then
4:     List⟨int⟩listanumeros ← newList⟨int⟩
5:     List⟨Axioma⟩listaaxiomatemporarios ← newList⟨Axioma⟩
6:     cria StreamWriter p9 e m4 e associa-os aos ficheiros "ap·in" e "am·in"
7:     escreve informação necessária à execução dos processos em p9 e m4
8:     cria duas instâncias da classe Oraculo prover e mace4 e as threads
       de execução proverThread e mace4Thread associadas aos métodos
       prover.Run e mace4.Run iniciando-as através dos métodos prover-
       Thread.Start e mace4Thread.Start
9:     loop
10:      testa o término das Threads e abandona o ciclo se uma delas terminar
11:    end loop
12:    if prover.Resultado = false ∧ mace4.Resultado = false then
13:      adiciona mensagem de erro a listaErros.erros
14:    else if prover.Resultado = true then
15:      for all Axioma z ∈ this · lista_axmas do
16:        if Axioma z ≠ v then
17:          adiciona Axioma z a listaaxiomatemporarios
18:        end if
19:        cria um nó com a listaaxiomatemporarios como argumento
20:        for all string s ∈ prover · lista_Lstprover do
21:          adiciona o indice do axioma a listanumeros
22:        end for
23:        adiciona listanumeros a listaGeradores · lsgeradores[indicev]
24:        adiciona o nó criado a listanosrepositorio
25:      end for
26:    end if
27:  end if
28: end for
29: retorna a listanosrepositorio
```

Figura 2-24 – Pseudocódigo do método *Node.no_eXecute*.

Este ficheiro de índices será utilizado no método *Node.no_eXecute*, onde, para cada axioma de um nó, só será questionado o oráculo se não existir na instância do contentor *listaGeradores.lsgeradores* indexada pela propriedade *Axioma.ax_numero* um subconjunto que esteja contido no conjunto resultante do método *Node.no_NmLstWithoutX* (ver pseudocódigo da figura 2-24).

Se o oráculo for questionado e uma prova encontrada pelo processo *Prover9.exe,* esta será adicionada ao *listaGeradores.lsgeradores* pelo método *Geradores.gd_AddSetL.*

2.8. Acelerador *(V)*

Sendo baseado no modo em como a implementação é concebida a inserção deste acelerador contrasta com a do anterior pela sua simplicidade e tem como única alteração à implementação original a modificação do método *LstNode.ln_eXecute*.

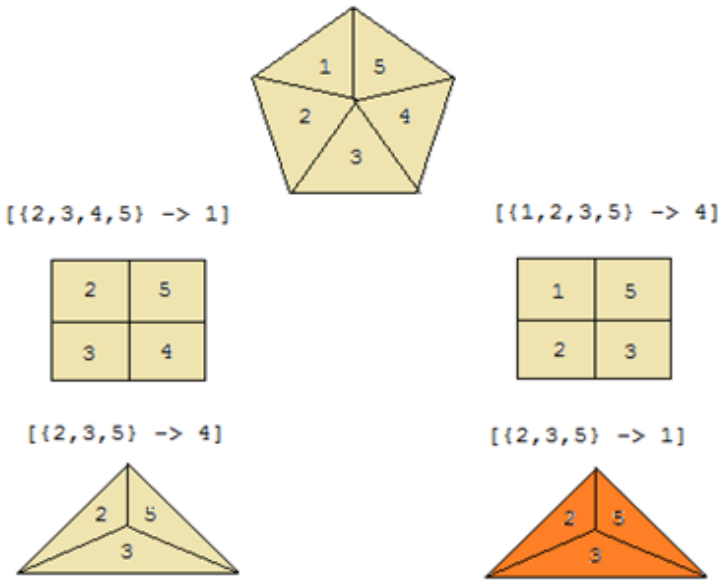

Figura 2-25 – Diagrama da execução do acelerador *(V)* na implementação *Tietze (V)*.

O diagrama da figura 2-25 apresenta graficamente o funcionamento do acelerador no caso da implementação do algoritmo de *Tietze*.

Assim, iniciando-se com o conjunto de axiomas V = {1,2,3,4,5} e existindo as regras dedutivas R_1 = [{2,3,4,5} → 1] e R_2 = [{1,2,3,5} → 4], aquando da execução do método *Node.no_eXecute* serão criados e adicionados ao contentor *LstNode.lst_nos* os nós V_1 = {2,3,4,5} e V_2 = {1,2,3,5}. Executando de novo o método sobre estes dois nós e existindo as regras dedutivas R_3 = [{2,3,5} → 4] e R_4 = [{2,3,5} → 1], pelo algoritmo de Tietze teriam de ser criados, o nó V_{13} = {2,3,5} e igualmente V_{24} = {2,3,5} (representado a vermelho na figura). Estando presente o acelerador ele bloqueará a criação deste último nó, impedindo assim que o algoritmo seja posteriormente aplicado a nós que contenham o mesmo conjunto de axiomas, ou seja, a nós iguais.

Na figura 2-26 são comparados, para o conjunto de axiomas V = {1,2,3,4}, as chamadas ao oráculo e os conjuntos de axiomas guardados no contentor *LstNode.lst_nos*, entre as implementações *Tietze* e *Tietze (V)* considerando-se que todos os axiomas se deduzem

reciprocamente, sendo bem patente a clara redução no número de nós criados e de perguntas efectuadas ao oráculo na implementação onde o é acrescido acelerador.

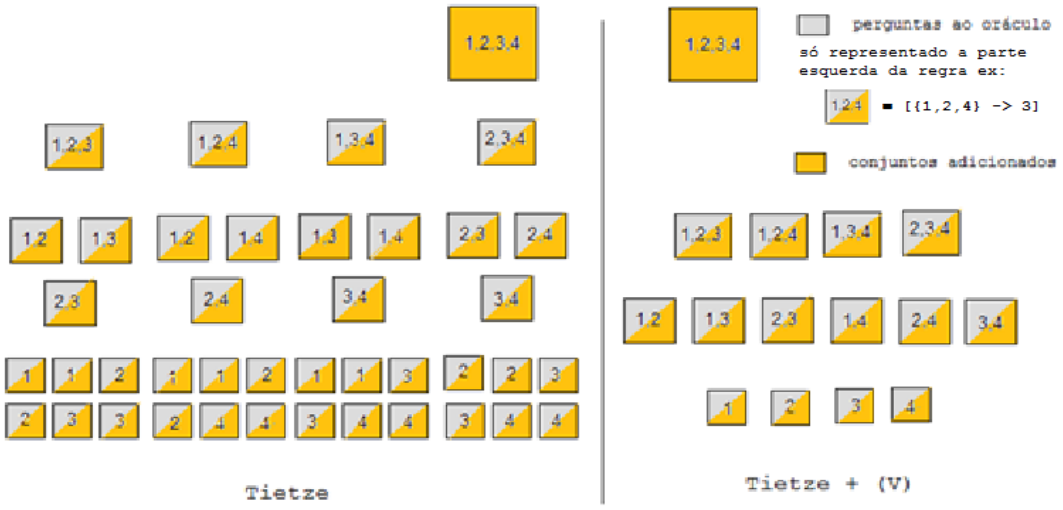

Figura 2-26 – Representação gráfica das implementações *Tietze* e *Tietze (V).*

Termina-se este subcapítulo com a figura 2-27 onde está listado o pseudocódigo do método *LstNode.ln_eXecute* responsável por estas alterações.

```
1: List⟨Node⟩listanosrepositorio ← newList⟨Node⟩()
2: List⟨Node⟩listanostemporarios ← newList⟨Node⟩()
3: for all Node n ∈ this · lst_nos do
4:   if ¬(n ∈ listanosrepositorio) then
5:     List⟨Node⟩listanostemporarios2 ← newList⟨Node⟩ · n.no_eXecute
6:     if listanostemporarios2 · Count ≠ 0 then
7:       for all Node d ∈ listanostemporarios2 do
8:         if ¬(d ∈ listanosrepositorio) then
9:           adiciona o nó listanosrepositorio
10:        end if
11:        if ¬(d ∈ listanostemporarios) then
12:          adiciona o nó listanostemporarios
13:        end if
14:      end for
15:    else if ¬listaSNodes · lSN_Exists(n) then
16:      adiciona nó n a listaSNodes · lst_tmp
17:    end if
18:  end if
19: end for
20: retorna o contentor listanosrepositorio
```

Figura 2-27 – Pseudocódigo do método *LstNode.ln_eXecute*.

2.9. Modo de Utilização

Neste subcapítulo são abordados aspectos relacionados com o modo de utilização das primeiras aplicações criadas sendo também apresentadas algumas das suas interfaces gráficas numa perspectiva de introduzir a aplicabilidade prática da tese de dissertação.

Tal como já foi referido, as diferentes aplicações foram construídas com o auxílio da interface de programação *Visual Studio 2008 PE*, e apresentam uma interface gráfica semelhante à da figura 2-28, onde estão assinalados seis grupos de componentes consoante a sua função.

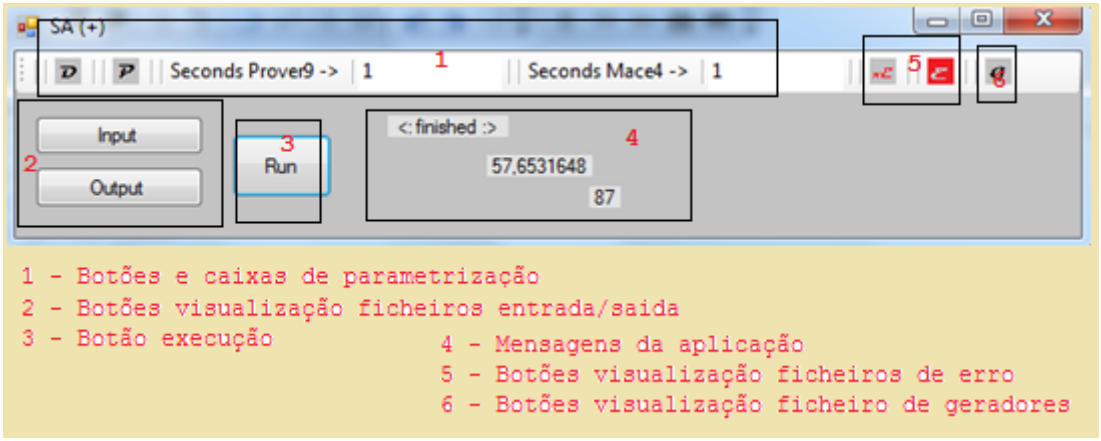

Figura 2-28 – Interface da aplicação *SA (+)* com diferentes grupos de componentes assinalados.

Não sendo o objectivo deste subcapítulo, o de apresentar um manual de utilização das aplicações criadas (ver Anexo A), mas sim o modo como pode ser utilizado, são apresentados dois exemplos com a pretensão de melhor se apreender o funcionamento das aplicações.

No primeiro exemplo é utilizado o conjunto de axiomas para a álgebra booleana retirado da página *Web* da Universidade de *Chapman* (ver [10]) e determinados os seus subconjuntos geradores e no segundo, utilizando o conjunto de axiomas dos semigrupos de *Clifford* extraídos deste mesmo *site* e tendo a ele adicionado consequências geradas pelo processo *Prover9.exe*, determinar novos conjuntos de axiomas de cardinal menor que o inicialmente apresentado.

Na figura 2-29, encontra-se listado o conjunto de axiomas utilizado no primeiro exemplo, a interface gráfica da aplicação *Tietze (V)* e alguns dos conjuntos geradores resultantes, onde é visível a drástica redução no cardinal dos conjuntos iniciais após um curto período de processamento.

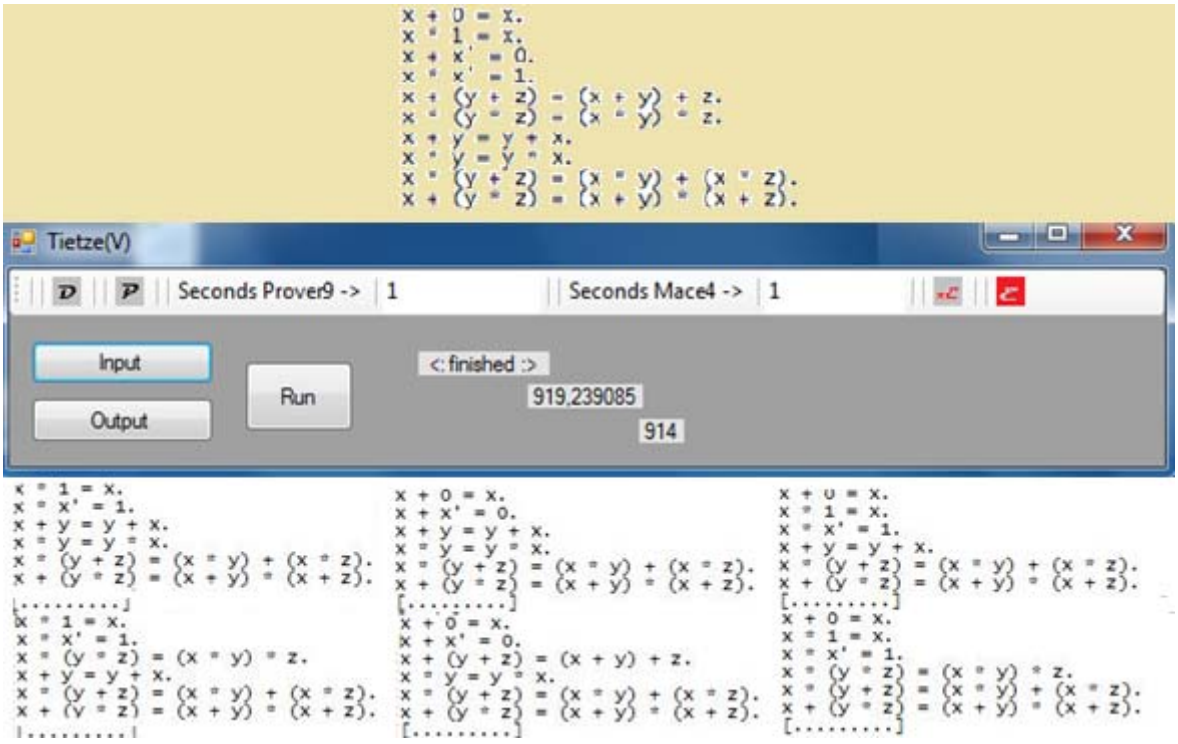

Figura 2-29 – Conjunto axiomas álgebra booleana, interface gráfica e conjuntos geradores obtidos.

Para o segundo exemplo apresentado na figura 2-30 foi utilizada a aplicação *SA (+)* e o conjunto de axiomas para os semigrupos de *Clifford*. Também nesta figura se encontram listados os conjuntos de axiomas resultantes, sendo nela visível dois novos conjuntos de cardinal inferior ao conjunto inicial.

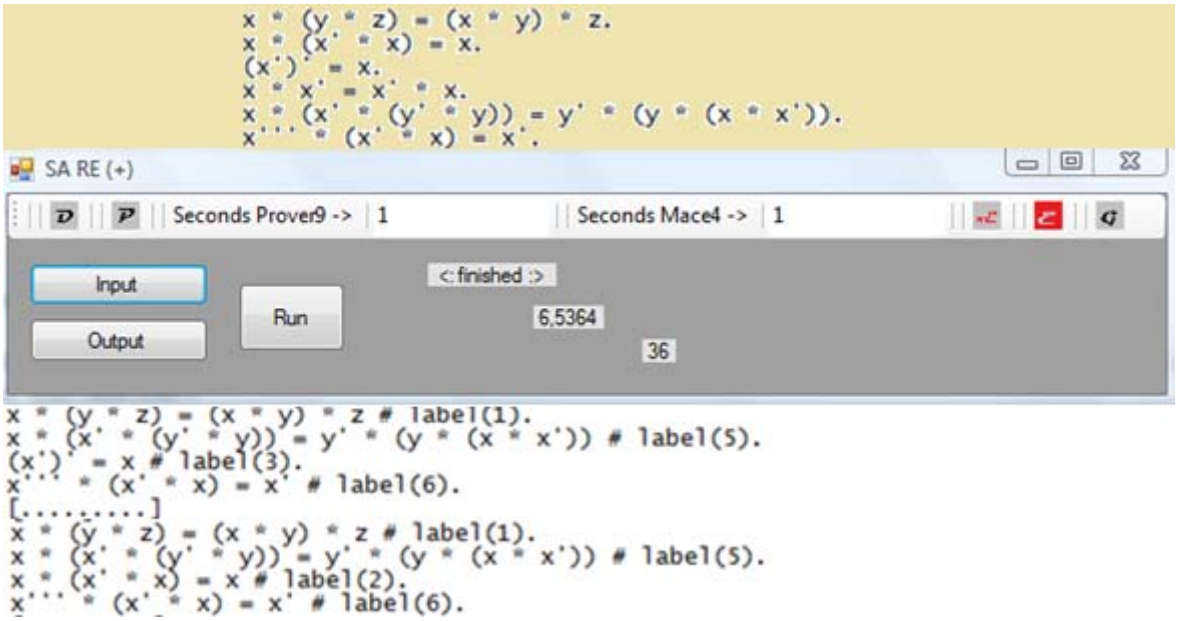

Figura 2-30 – Conjunto axiomas semigrupos de *Clifford*, interface gráfica e bases obtidas.

Na figura 2-31 encontra-se ainda listado um ficheiro contendo os diferentes conjuntos de regras de dedução encontradas indexadas pelo axioma que deduzem, sendo este conjunto vazio para os axiomas invariantes.

```
1 ::
2 :: [ 1, 5, 3, 6]
3 :: [ 1, 5, 2, 6]
4 :: [ 1, 5, 3, 6][ 1, 5, 2, 6]
5 ::
6 :: [ 1, 5, 2, 3, 4]
```

Figura 2-31 – Conjunto de regras de dedução para os axiomas do segundo exemplo.

3. Algoritmos - Resultados

3.1. Introdução

As diferentes implementações foram testadas com o auxílio das quinze estruturas algébricas listadas no quadro 3-1.

A escolha destas estruturas em detrimento de outras deveu-se por um lado à sua importância na actual investigação matemática e por outro à possibilidade do processo *Prover9.exe* conseguir gerar consequências a partir dos conjuntos de axiomas que as definem tendo sido a negação desta ultima condição o que originou a omissão de estruturas tais como os "*loops"* e monoides.

Referência	Designação	Tipo de Classe	Teoria de 1ª Ordem	Teoria Equacional
18	Álgebras BCK	Quasivariedade	Indecidível	-
26	Grupos booleanos	Variedade	Decidível	Decidível P
40	Monoides comutativos cancelativos	Quasivariedade	Indecidível	-
47	Semigrupos de Clifford	Variedade	-	-
52	Semigrupos inversos comutativos	Variedade	-	-
67	Anéis abelianos	Variedade	Indecidível	Decidível
93	Directóides	Variedade	-	-
113	Corpos	1ª Ordem	-	-
128	Grupos	Variedade	Indecidível	Decidível P
-	Semigrupos inversos (B. Schein)	Variedade	-	-
158	Reticulados	Variedade	Indecidível	Decidível P
187	Álgebras MV	Variedade	-	Decidível
253	Anéis	Variedade	Indecidível	Decídível
261	Semi-reticulados	Variedade	Indecidível	Decidível P
264	Semi-anéis	Variedade	Indecidível	Decidível

Quadro 3-1 – Designações e referências das estruturas algébricas estudadas.

Os conjuntos de axiomas que definem as estruturas acima listadas, à excepção dos semigrupos inversos, foram retirados da página *Web* da Universidade de *Chapman* (ver [10]) tendo para os semigrupos sido utilizada a base-5 proposta por *B. Schein* (ver [6]).

Assim, para cada uma destas estruturas algébricas foram construídos cinquenta conjuntos, cada um deles constituído por um conjunto equivalente ao conjunto de axiomas que define a estrutura algébrica, em união, a um conjunto de três das suas consequências geradas pelo

processo *Prover9.exe*, tal como é apresentado no quadro 3-2 para a estrutura algébrica reticulados.

Axiomas	Consequências
(x + y) + z = x + (y + z). (x * y) * z = x * (y * z). x + y = y + x. x * y = y * x. (x + y) * x = x. (x * y) + x = x.	x + ((y * (x * z)) + u) = x + u. (x * y) + (z * (x * (y * u))) = x * y. (x * y) + (z + x) = z + x.

Quadro 3-2 – Conjuntos de axiomas e de três consequências para a estrutura algébrica reticulados.

Foram também criadas nove aplicações, três por cada algoritmo, uma para cada algoritmo e acelerador, com o objectivo de obter os números de chamadas ao oráculo efectuadas. A escolha desta variável como indicador da eficiência do algoritmo em detrimento de uma outra tal como o tempo de execução, deveu-se à podermos considerar para limites bastante alargados como uma constante independente do computador ou sistema operativo utilizado, e também, ao conhecimento empírico de serem as chamadas ao oráculo o factor que mais influi no tempo total de execução das diferentes implementações.

Além do número de chamadas ao oráculo também se salvaguardaram os conjuntos geradores de cardinal inferior ao utilizado na representação inicial da estrutura algébrica, tendo-se para cada um dos conjuntos testados recolhido informação semelhante à apresentada no quadro 3-3, onde é listada a obtida pela execução da aplicação ao conjunto da união dos axiomas e consequências do quadro 3-2.

Chamadas	Conjuntos geradores	
522	[node] (x + y) * x = x. x + ((y * (x * z)) + u) = x + u. (x * y) + (z * (x * (y * u))) = x * y. (x * y) + (z + x) = z + x. [node] (x + y) + z = x + (y + z). (x + y) * x = x. (x * y) + (z * (x * (y * u))) = x * y. (x * y) + (z + x) = z + x.	[node] (x + y) + z = x + (y + z). x + y = y + x. (x + y) * x = x. (x * y) + (z * (x * (y * u))) = x * y.

Quadro 3-3 – Número de chamadas ao oráculo e diferentes conjuntos geradores.

Ao longo deste capítulo serão apresentados quadros em muito semelhantes ao lado direito do quadro 3-3 mas cujo conteúdo e titulo variará consoante a existência ou não durante a execução dos designados erros de tempo. Se eles se verificarem serão listados conjuntos

geradores com o título indicando-o, se não, serão listadas as bases do sistema também com o título a indicá-lo, estando-se a assumir para tal, a correcção das provas encontradas pelo processo *Prover9.exe*. De facto, como já referido na introdução, a correcção das provas encontradas é sempre assumida, não tendo sido utilizado nenhum verificador de provas tal como o programa *IVY* (ver [6] e [26]).

Após estas novas implementações terem testado para cada uma das estruturas algébricas do quadro 3-1 os cinquenta conjuntos foi feita uma análise estatística à variável número de chamadas ao oráculo com o auxílio do software vocacionado *Statistical Packtage for Social Sciences (SPSS)*. Esta análise constituiu-se na comparação das diferentes médias apresentadas por cada uma das implementações pelo método não paramétrico de *Friedman*, tendo a escolha deste método sido justificada por se considerar estarmos a efectuar para cada estrutura algébrica, um conjunto de cinquenta observações de nove emparelhamentos.

Nos próximos subcapítulos será apresentado individualmente para cada uma das estruturas estudadas um quadro contendo:

- Axiomas representando a estrutura na página *Web* de referência e conjuntos iniciais de axiomas utilizados:
- Diferentes estatísticas de tendência central e dispersão;
- Resultados do teste de *Friedman*;
- Alguns dos conjuntos geradores ou bases do sistema encontrados.

Também para algumas das estruturas serão feitas considerações sobre a eficiência das diferentes implementações baseadas nos resultados do método de *Friedman*, definindo-se como implementação mais eficiente a que requer um menor número de chamadas ao oráculo, sendo no último subcapítulo feita uma análise ao conjunto total de resultados obtidos independentemente da estrutura algébrica de origem e de novo tecidas considerações sobre a eficiência das diversas implementações.

3.2. Álgebras BCK

Axiomas				
((x * y) * (x * z)) * (z * y) = 0. x * 0 = x. 0 * x = 0. (((x * y) = (y * x)) & ((x * y) = 0)) -> x = y.				
Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	76	639	362,56	168,658
SA	10	445	209,90	123,689
Tietze-SA	10	491	255,32	153,135
Tietze (+)	76	560	333,14	137,367
SA (+)	10	250	124,40	58,553
Tietze-SA (+)	10	406	211,76	122,892
Tietze (V)	44	171	121,58	33,939
SA (V)	10	208	125,94	56,300
Tietze-SA (V)	10	148	92,66	42,986

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	8,82	9º	N	50
SA	5,46	6º	Chi-Square	334,417
Tietze-SA	6,30	7º	g.l.	8
Tietze (+)	8,10	8º	Sig.Asymp.	0,000
SA (+)	2,95	2º		
Tietze-SA (+)	5,19	5º	W de Kendall	0,836
Tietze (V)	3,28	3º		
SA (V)	3,75	4º		
Tietze-SA (V)	1,15	1º		

Conjuntos geradores	
[node] ((x * y) * (x * z)) * (z * y) = 0. x * (y * (y * (x * z))) != 0 | y * (y * (x * z)) = x. 0 * x = 0. [node] ((x * y) * (x * z)) * (z * y) = 0. x * (y * (y * (x * z))) != 0 | y * (y * (x * z)) = x. (((x * (x * (y * z))) * u) * w) * ((y * u) * w) = 0.	[node] x * 0 = x. ((((x * y) * (x * z)) * u) * w) * ((z * y) * u) = 0. (((x * y) = (y * x)) & ((x * y) = 0)) -> x = y.

Quadro 3-4 – Axiomas, estatísticas, resultados teste de *Friedman* e conjuntos de geradores para as álgebras BCK.

Para estas álgebras, a utilização de uma das implementações do capítulo anterior não originou um conjunto gerador de cardinal inferior ao inicial tendo sido este o utilizado, se tal tivesse sucedido a parte superior do quadro 3-4 teria sido subdividida em duas células onde na céluda esquerda estariam listados os axiomas constantes na página *Web* e na célula direita os utilizados pelas aplicações.

A análise dos valores deste quadro sugere para esta estrutura algébrica uma maior eficiência das implementações do algoritmo *Tietze-SA (V)* e do algoritmo *SA (+)*, sendo também de reter o diferente efeito da utilização do acelerador *(+)*, reduzido nos algoritmos *Tietze* e *Tietze-SA* e bastante notório no algoritmo *SA*, o que não surpreende, bastando para tal analisar o código em *C#* deste último algoritmo onde se verifica a eliminação de três chamadas ao oráculo por cada conjunto de axiomas que contenha um subconjunto previamente salvaguardado no contentor de geradores.

No que se refere ao indicador estatístico do teste de *Friedman*, permitem-nos estabelecer a não rejeição da hipótese nula, ou seja, os diferentes algoritmos apresentam diferentes eficiências, indicando-nos também o valor *W* de *Kendall* uma similitude na eficiência de cada algoritmo para os diferentes conjuntos testados.

Ainda relativamente aos algoritmos refere-se também a maior eficiência que para este tipo de estrutura algébrica o algoritmo *SA* demonstra comparativamente aos algoritmos de *Tietze* e de *Tietze-SA* quando não existe o acréscimo de um acelerador à sua implementação.

3.3. Grupos booleanos

Axiomas			Conjunto utilizado		
x * (y * z) = (x * y) * z. x * e = x. e * x = x. x * x = e.			x * (y * z) = (x * y) * z. x * e = x. x * x = e.		
Algoritmo	Mínimo	Máximo	Média		Desvio Padrão
Tietze	110	435	310,20		111,978
SA	82	409	271,46		116,533
Tietze-SA	62	398	217,50		89,074
Tietze (+)	104	345	265,80		82,814
SA (+)	44	187	120,38		42,468
Tietze-SA (+)	52	339	163,58		66,233
Tietze (V)	55	123	92,78		19,663
SA (V)	62	185	128,50		35,525
Tietze-SA (V)	40	135	79,02		23,561
Algoritmo	Média Níveis		Ordem	Estatísticas	
Tietze	8,84		9º	N	50
SA	7,28		7º	Chi-Square	356,948
Tietze-SA	6,29		6º	g.l.	8
Tietze (+)	7,32		8º	Sig.Asymp.	0,000
SA (+)	3,28		3º		
Tietze-SA (+)	4,72		5º	W de Kendall	0,892
Tietze (V)	2,36		2º		
SA (V)	3,79		4º		
Tietze-SA (V)	1,12		1º		
Conjuntos de geradores					
[node] x * (y * (z * x)) = y * z. x * e = x. [node] x * (y * (z * (x * u))) = y * (z * u). x * e = x. [node] x * (y * (z * (u * (w * x)))) = y * (z * (u * w)). x * e = x.					

Quadro 3-5 – Axiomas, estatísticas, resultados teste de *Friedman* e conjuntos de geradores para os grupos booleanos.

Uma analise do quadro 3-5 sugere que para este tipo de estruturas álgebricas o algoritmo *SA* não apresenta a eficiência demonstrada no caso das álgebras *BCK*, embora a sua implementação quando acrescida do acelerador *(+)* seja a terceira mais eficiente.

Existem duas explicações para as diferenças de comportamento deste algoritmo:

- A primeira e mais obvia é a de que os conjuntos de regras de derivação gerados e salvaguardados no contentor geradores não são subconjuntos daqueles que vão sendo derivados, o que frequentemente sucede quando as regras de derivação contêm vários elementos como premissas:
- A outra, é a existência de diversas consequências redundantes no conjunto inicial o que parece ser o caso deste exemplo, dada a diferença de eficiência registada entre as implementações dos algoritmos *Tietze* e *Tietze-SA*.

De facto, a ordem das implementações destes três algoritmos poderá servir como um indicador do tipo de eliminação que o algoritmo *SA* efectua, sugerindo uma ordenação do tipo *Tietze* > *SA* > *Tietze-SA* um maior numero de redundantes no conjunto inicial que uma ordenação do tipo *Tietze* > *Tietze-SA* > *SA*, que, por sua vez, sugere um menor cardinal do conjunto de premissas das regras de derivação.

Em relação aos indicadores estatísticos verifica-se tal como no caso das álgebras *BCK* a não rejeição da hipótese nula e um valor *W* de *Kendall* que sugere similitude de tratamento entre os diferentes conjuntos tratados por um mesmo algoritmo, embora com um valor inferior ao da estrutura algébrica anterior.

3.4. Monoides cancelativos comutativos

Axiomas	Conjunto utilizado
x * (y * z) = (x * y) * z. x * e = x. e * x = x. z * x = z * y -> x = y. x * z = y * z -> x = y. x * y = y * x.	x * (y * z) = (x * y) * z. x * e = x. z * x = z * y -> x = y. x * y = y * x.

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	155	526	327,88	104,579
SA	9	492	312,84	126,541
Tietze-SA	21	538	280,70	100,490
Tietze (+)	129	427	308,02	58,355
SA (+)	9	288	159,74	46,942
Tietze-SA (+)	21	433	234,60	88,414
Tietze (V)	23	137	102,24	21,399
SA (V)	9	213	147,98	45,635
Tietze-SA (V)	9	149	95,02	32,165

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	7,77	9º	N	50
SA	7,14	8º	Chi-Square	316,497
Tietze-SA	6,97	7º	g.l.	8
Tietze (+)	6,89	6º	Sig.Asymp.	0,000
SA (+)	3,96	4º		
Tietze-SA (+)	5,74	5º	W de Kendall	0,791
Tietze (V)	1,82	2º		
SA (V)	3,35	3º		
Tietze-SA (V)	1,36	1º		

Bases	
[node] x * (y * (z * (u * x))) = y * (z * u). x * e = x. [node] x * (y * (x * z)) = y * z. x * e = x.	[node] x * (y * (z * (u * (w * x)))) = y * (z * (u * w)). x * e = x.

Quadro 3-6 – Axiomas, estatísticas, resultados do teste de *Friedman* e bases para os monoides cancelativos comutativos

3.5. Semigrupos de *Clifford*

Axiomas – Conjunto utilizado					
x * (y * z) = (x * y) * z. x * (x' * x) = x. (x')' = x. x * x' = x' * x. (x * x') * (y' * y) = (y' * y) * (x * x').					
Algoritmo	Mínimo	Máximo	Média		Desvio Padrão
Tietze	156	896	506,28		181,514
SA	18	498	236,94		110,601
Tietze-SA	16	540	244,66		134,926
Tietze (+)	150	797	461,62		159,203
SA (+)	16	233	119,06		50,578
Tietze-SA (+)	16	435	195,02		110,110
Tietze (V)	76	211	145,26		31,105
SA (V)	18	221	127,50		48,146
Tietze-SA (V)	16	151	85,54		35,609
Algoritmo	Média Níveis	Ordem	Estatísticas		
Tietze	9,00		9º	N	50
SA	6,14		6º	Chi-Square	361,808
Tietze-SA	6,14		6º	g.l.	8
Tietze (+)	8,00		8º	Sig.Asymp.	0,000
SA (+)	2,25		2º		
Tietze-SA (+)	4,93		5º	W de Kendall	0,905
Tietze (V)	4,12		4º		
SA (V)	3,33		3º		
Tietze-SA (V)	1,09		1º		
Conjuntos de geradores					
[node] x''' * (x' * x) = x'. x * (y * z) = (x * y) * z. (x')' = x. x * (x' * (y' * y)) = y' * (y * (x * x')). [node] x''' * (x' * x) = x'. x * (y * z) = (x * y) * z. x * (x' * x) = x. x * (x' * (y' * y)) = y' * (y * (x * x')).			[node] x * (y * (y' * (x' * (x' * z)))) = y * (y' * (x' * z)). x * (y * z) = (x * y) * z. x * (x' * x) = x. x * (x' * (y' * y)) = y' * (y * (x * x')).		

Quadro 3-7 – Axiomas, estatísticas, resultados teste de *Friedman* e conjuntos geradores para os semigrupos de *Clifford*.

3.6. Semigrupos inversos comutativos

Axiomas – Conjunto utilizado				
x * (y * z) = (x * y) * z. x * (x' * x) = x. (x')' = x. (x * x') * (y' * y) = (y' * y) * (x * x').				
Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	45	615	185,88	112,152
SA	9	433	87,14	87,745
Tietze-SA	9	437	93,62	89,043
Tietze (+)	29	497	135,00	101,388
SA (+)	9	326	64,28	63,171
Tietze-SA (+)	9	361	83,78	85,528
Tietze (V)	24	159	68,58	31,170
SA (V)	9	204	67,76	44,966
Tietze-SA (V)	9	177	58,22	41,381

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	8,85	9º	N	50
SA	5,18	7º	Chi-Square	239,227
Tietze-SA	4,92	6º	g.l.	8
Tietze (+)	7,86	8º	Sig.Asymp.	0,000
SA (+)	2,40	1º		
Tietze-SA (+)	3,56	3º	W de Kendall	0,598
Tietze (V)	4,82	5º		
SA (V)	4,10	4º		
Tietze-SA (V)	3,31	2º		

Conjuntos geradores	
[node] x * (y * (x' * (z * x'))) = y * (x' * z). (x * y) * z = x * (y * z). x * (x' * x) = x. [node] x' * (y * (x' * (z * x))) = y * (x' * z). (x * y) * z = x * (y * z). x * (x' * x) = x.	[node] x' * (x' * (y * (z * x))) = x' * (y * z). x * (y * (x * (z * (x' * u)))) = y * (x * (z * u)). x * (x' * x) = x.

Quadro 3-8 – Axiomas, estatísticas, resultados teste de *Friedman* e conjuntos geradores para os semigrupos inversos comutativos.

Para esta estrutura a implementação *SA (+)* apresenta a maior eficiência suplantando mesmo a *Tietze-SA (V)* que para as estruturas algébricas anteriores tem sido a mais eficiente, destacando-se também o baixo valor do *W* de *Kendall* comparativamente aos apresentados nas estruturas anteriores.

3.7. Anéis comutativos

Axiomas	Conjunto utilizado
x * (y * z) = (x * y) * z. x + y = y + x. x + (y + z) = (x + y) + z. 0 + x = x. x + x' = 0. x * (y + z) = (x * y) + (x * z). (x + y) * z = (x * z) + (y * z). x * y = y * x.	x * (y * z) = (x * y) * z. x * y = y * x. x + y = y + x. x + (y + z) = (x + y) + z. 0 + x = x. x + x' = 0. x * (y + z) = (x * y) + (x * z).

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	127	922	271,76	181,670
SA	13	318	72,40	74,719
Tietze-SA	13	406	76,44	85,365
Tietze (+)	124	855	260,88	168,604
SA (+)	13	145	44,44	37,504
Tietze-SA (+)	13	333	64,64	70,909
Tietze (V)	68	235	111,38	44,599
SA (V)	13	173	54,86	45,478
Tietze-SA (V)	13	131	41,28	32,201

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	8,98	9º	N	50
SA	4,74	6º	Chi-Square	343,846
Tietze-SA	4,61	5º	g.l.	8
Tietze (+)	8,02	8º	Sig.Asymp.	0,000
SA (+)	2,30	2º		
Tietze-SA (+)	3,65	3º	W de Kendall	0,860
Tietze (V)	6,63	7º		
SA (V)	3,87	4º		
Tietze-SA (V)	2,20	1º		

Conjuntos de geradores	
[node] (x + (x * y)) * z = (z + (z * y)) * x. x + y = y + x. x + (y + z) = (x + y) + z. 0 + x = x. x' + x = 0. x * (y + z) = (x * y) + (x * z).	[node] (x + (y + z)')' = y + (z + x'). (x + (x * y)) * z = (z + (z * y)) * x. x + y = y + x. 0 + x = x. x' + x = 0. x * (y + z) = (x * y) + (x * z).

Quadro 3-9 – Axiomas, estatísticas, resultados teste de *Friedman* e conjuntos geradores para os anéis comutativos.

3.8. Directóides

Axiomas – Conjunto utilizado					
x * x = x. (x * y) * x = x * y. y * (x * y) = x * y. x * ((x * y) * z) = (x * y) * z.					
Algoritmo	Mínimo	Máximo	Média		Desvio Padrão
Tietze	343	896	599,00		147,776
SA	65	653	378,26		151,457
Tietze-SA	187	671	417,58		173,154
Tietze (+)	305	758	517,66		125,500
SA (+)	65	319	190,54		68,235
Tietze-SA (+)	144	520	305,58		145,551
Tietze (V)	109	188	145,98		22,178
SA (V)	65	240	166,32		40,138
Tietze-SA (V)	77	160	105,66		31,722
Algoritmo	Média Níveis		Ordem	Estatísticas	
Tietze	8,98		9º	N	50
SA	6,10		6º	Chi-Square	376,635
Tietze-SA	6,74		7º	g.l.	8
Tietze (+)	7,84		8º	Sig.Asymp.	0,000
SA (+)	3,62		4º		
Tietze-SA (+)	5,20		5º	W de Kendall	0,942
Tietze (V)	2,28		2º		
SA (V)	3,22		3º		
Tietze-SA (V)	1,02		1º		
Conjuntos de geradores					
[node] x * x = x. (x * ((y * z) * u)) * z = x * ((y * z) * u). y * (x * y) = x * y. [node] x * x = x. (x * ((y * z) * u)) * z = x * ((y * z) * u). x * ((y * (z * (u * x))) * w) = (y * (z * (u * x))) * w.			[node] x * x = x. y * (x * y) = x * y. ((x * (y * (z * (u * w)))) * v5) * u = (x * (y * (z * (u * w)))) * v5.		

Quadro 3-10 – Axiomas, estatísticas, resultados teste de *Friedman* e conjuntos geradores para os directóides.

3.9. Corpos

Axiomas	Conjunto utilizado
(x + y) + z = x + (y + z). 0 + x = x. x' + x = 0. x + y = y + x. x * (y * z) = (x * y) * z. x * (y + z) = (x * y) + (x * z). (y + z) * x = (y * x) + (z * x). x * 1 = x. 1 * x = x x * y = y * x. (x != 0) -> exists y(x * y = 1). 0 != 1.	(x + y) + z = x + (y + z). 0 + x = x. x' + x = 0. x * (y * z) = (x * y) * z. x * (y + z) = (x * y) + (x * z). x * 1 = x. x * y = y * x. (x != 0) -> exists y(x * y = 1). 0 != 1.

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	124	1557	294,04	234,896
SA	13	1335	127,90	212,184
Tietze-SA	15	908	114,00	138,945
Tietze (+)	75	1465	288,78	216,635
SA (+)	13	805	90,06	136,105
Tietze-SA (+)	13	333	68,28	77,413
Tietze (V)	54	345	119,10	51,149
SA (V)	13	788	84,20	123,020
Tietze-SA (V)	13	227	45,22	39,219

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	8,88	9º	N	50
SA	5,32	6º	Chi-Square	317,004
Tietze-SA	5,08	5º	g.l.	8
Tietze (+)	8,00	8º	Sig.Asymp.	0,000
SA (+)	3,54	3º		
Tietze-SA (+)	2,70	2º	W de Kendall	0,793
Tietze (V)	6,14	7º		
SA (V)	3,73	4º		
Tietze-SA (V)	1,61	1º		

Conjuntos de geradores	
[node] (x + y) + z = x + (y + z). x * 1 = x. (x != 0) -> exists y(x * y = 1). (x * (y * (z * u))) + (u * (x * (z * y)))' != 1. (x * y) + (z + (y * x)') = z. ((x * y) + (z * x)) * u = x * ((y + z) * u).	[node] (x + y) + z = x + (y + z). x * 1 = x. (x != 0) -> exists y(x * y = 1). ((x * y) + (z * (x * u))) * w = x * ((y + (z * u)) * w). (x * (y * (z * u))) + (u * (x * (z * y)))' != 1. (x * y) + (z + (y * x)') = z.

Quadro 3-11 – Axiomas, estatísticas, teste de *Friedman* e conjuntos de geradores para os corpos.

3.10. Grupos

Axiomas – Conjunto utilizado				
x * (y * z) = (x * y) * z. x' * x = e. e * x = x.				
Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	63	250	93,56	46,436
SA	9	236	48,00	55,880
Tietze-SA	9	262	48,26	62,377
Tietze (+)	60	229	88,06	42,375
SA (+)	9	143	29,82	31,243
Tietze-SA (+)	9	235	42,64	55,394
Tietze (V)	36	101	48,00	17,418
SA (V)	9	149	38,34	39,097
Tietze-SA (V)	9	113	29,00	28,673

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	8,90	9º	N	50
SA	4,74	5º	Chi-Square	290,680
Tietze-SA	4,76	6º	g.l.	8
Tietze (+)	7,84	8º	Sig.Asymp.	0,000
SA (+)	2,67	1º		
Tietze-SA (+)	3,82	3º	W de Kendall	0,727
Tietze (V)	5,45	7º		
SA (V)	4,08	4º		
Tietze-SA (V)	2,74	2º		

Quadro 3-12 – Axiomas, estatísticas e resultados teste de *Friedman* para os grupos.

3.11. Semigrupos inversos

Axiomas	Conjunto utilizado
x = (x * x') * x. (x * x') * (y' * y) = (y' * y) * (x * x'). (x * y) * z = x * (y * z). (x')' = x. (x * y)’ = y’ * x’.	x = (x * x') * x. (x * x') * (y' * y) = (y' * y) * (x * x'). (x * y) * z = x * (y * z). (x')' = x.

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	53	367	150,72	61,781
SA	9	210	72,78	55,439
Tietze-SA	10	219	70,40	54,265
Tietze (+)	29	319	144,78	55,489
SA (+)	9	160	46,06	34,504
Tietze-SA (+)	9	125	47,16	37,433
Tietze (V)	24	113	70,12	19,890
SA (V)	9	133	51,54	34,232
Tietze-SA (V)	9	104	39,26	23,857

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	8,82	9º	N	50
SA	5,74	7º	Chi-Square	309,865
Tietze-SA	5,01	5º	g.l.	8
Tietze (+)	8,16	8º	Sig.Asymp.	0,000
SA (+)	2,63	2º		
Tietze-SA (+)	3,15	3º	W de Kendall	0,775
Tietze (V)	5,53	6º		
SA (V)	3,96	4º		
Tietze-SA (V)	2,00	1º		

Conjuntos de geradores	
[node] x' * (y * (x * y)')' = y * (y' * (x' * x)). x = (x * x') * x. (x * y) * z = x * (y * z).	[node] x = (x * x') * x. (x * y) * z = x * (y * (z')'). (x * x') * (y' * y) = (y' * y) * (x * x').

Quadro 3-13 – Axiomas, estatísticas, resultados teste de *Friedman* e conjuntos de geradores para os semigrupos inversos.

3.12. Reticulados

Axiomas – Conjunto utilizado
(x + y) + z = x + (y + z). (x ^ y) ^ z = x ^ (y ^ z). x + y = y + x. x ^ y = y ^ x. (x + y) ^ x = x. (x ^ y) + x = x.

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	111	11074	1416,70	1644,428
SA	12	8029	808,72	1225,051
Tietze-SA	95	8815	1039,42	1306,105
Tietze (+)	111	9382	1286,72	1412,767
SA (+)	12	3479	416,32	568,492
Tietze-SA (+)	11	6404	734,18	1007,186
Tietze (V)	60	633	270,60	121,314
SA (V)	12	941	283,34	174,052
Tietze-SA (V)	12	609	218,02	121,662

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	8,41	9º	N	50
SA	5,01	6º	Chi-Square	283,122
Tietze-SA	7,10	7º	g.l.	8
Tietze (+)	7,83	8º	Sig.Asymp.	0,000
SA (+)	2,93	2º		
Tietze-SA (+)	4,84	5º	W de Kendall	0,708
Tietze (V)	3,62	4º		
SA (V)	3,17	3º		
Tietze-SA (V)	2,09	1º		

Conjuntos de geradores	
[node]	[node]
(x ^ y) + x = x.	(x ^ y) + x = x.
(x + y) ^ (x + (z + (y + u))) = x + y.	(x + (y + z)) ^ (y + (x + (z + u))) = y + (x + z).
(x + y) ^ (z ^ (x + (y + u))) = z ^ (x + y).	(x + y) ^ (z ^ (x + (y + u))) = z ^ (x + y).

Quadro 3-14 – Axiomas, estatísticas, resultados teste de *Friedman* e conjuntos de geradores para os reticulados.

3.13. Álgebras MV

Axiomas			Conjunto utilizado		
x + (y + z) = (x + y) + z. x + 0 = x. 0 + x = x. (x')' = x. x + 0' = 0'. ((x)' + y)' + y = ((y)' + x)' + x.			x + (y + z) = (x + y) + z. x + 0 = x. (x')' = x. x + 0' = 0'. ((x)' + y)' + y = ((y)' + x)' + x.		
Algoritmo	Mínimo	Máximo	Média		Desvio Padrão
Tietze	15	686	174,72		135,512
SA	9	291	77,28		67,568
Tietze-SA	10	471	100,08		87,792
Tietze (+)	15	616	182,26		113,446
SA (+)	9	204	56,68		43,937
Tietze-SA (+)	9	346	68,78		65,045
Tietze (V)	15	180	80,42		39,078
SA (V)	9	175	57,74		41,713
Tietze-SA (V)	10	132	42,14		29,733
Algoritmo	Média Níveis		Ordem	Estatísticas	
Tietze	8,19		9º	N	50
SA	4,65		5º	Chi-Square	264,342
Tietze-SA	6,09		7º	g.l.	8
Tietze (+)	8,12		8º	Sig.Asymp.	0,000
SA (+)	3,19		2º		
Tietze-SA (+)	3,85		4º	W de Kendall	0,661
Tietze (V)	5,63		6º		
SA (V)	3,23		3º		
Tietze-SA (V)	2,05		1º		
Conjuntos de geradores					
[node] x + (y + z) = (x + y) + z. (x' + y')' + (y + (x + y)')' = x. x + 0' = 0'.			[node] x + (y + z) = (x + y) + z. (x' + y')' + (x + (x + y)')' = y. x + ((y' + x)' + (z + (y + z)')) = 0'.		

Quadro 3-15 – Axiomas, estatísticas, resultados teste de *Friedman* e conjuntos de geradores para as álgebras MV.

3.14. Anéis

Axiomas – Conjunto utilizado
x + y = y + x. (x + y) + z = x + (y + z). 0 + x = x. x' + x = 0. (x * y) * z = x * (y * z). x * (y + z) = (x * y) + (x * z). (y + z) * x = (y * x) + (z * x).

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	124	3568	347,06	530,480
SA	13	526	89,62	119,727
Tietze-SA	13	1792	132,02	286,834
Tietze (+)	120	3133	327,18	469,085
SA (+)	13	343	53,26	68,322
Tietze-SA (+)	13	1325	105,06	223,606
Tietze (V)	68	399	119,98	72,133
SA (V)	13	287	63,18	67,873
Tietze-SA (V)	13	254	49,78	56,198

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	8,98	9º	N	50
SA	4,86	6º	Chi-Square	338,079
Tietze-SA	4,62	5º	g.l.	8
Tietze (+)	8,02	8º	Sig.Asymp.	0,000
SA (+)	2,50	2º		
Tietze-SA (+)	3,54	3º	W de Kendall	0,845
Tietze (V)	6,54	7º		
SA (V)	3,67	4º		
Tietze-SA (V)	2,27	1º		

Conjuntos de geradores	
[node]	[node]
0 + x = x.	0 + x = x.
(x * y) * z = x * (y * z).	(x * y) * z = x * (y * z).
(y + z) * x = (y * x) + (z * x).	(y + z) * x = (y * x) + (z * x).
x + (y + (z + x')) = y + z.	x + (y + (z + x')) = y + z.
x * (y + z) = (x * y) + (x * z).	(x * y) + (z + (x * u)) = z + (x * (y + u)).

Quadro 3-16 – Axiomas, estatísticas, resultados teste de *Friedman* e conjuntos de geradores para os anéis.

3.15. Semi-reticulados

Axiomas – Conjunto utilizado					
x * (y * z) = (x * y) * z. x * y = y * x. x * x = x.					
Algoritmo	Mínimo	Máximo	Média		Desvio Padrão
Tietze	60	128	69,38		16,412
SA	9	114	21,42		28,586
Tietze-SA	9	75	39,62		24,547
Tietze (+)	44	119	67,40		16,473
SA (+)	9	72	16,06		16,171
Tietze-SA (+)	9	55	16,30		15,788
Tietze (V)	36	59	39,02		6,579
SA (V)	9	70	18,16		20,149
Tietze-SA (V)	9	41	14,06		11,024
Algoritmo	Média Níveis		Ordem	Estatísticas	
Tietze	8,71		9º	N	50
SA	3,90		5º	Chi-Square	304,653
Tietze-SA	5,84		6º	g.l.	8
Tietze (+)	7,79		8º	Sig.Asymp.	0,000
SA (+)	3,16		2º		
Tietze-SA (+)	3,20		3º	W de Kendall	0,762
Tietze (V)	5,96		7º		
SA (V)	3,56		4º		
Tietze-SA (V)	2,88		1º		

Quadro 3-17 – Axiomas, estatísticas e resultados teste de *Friedman* para os semi-reticulados.

3.16. Semi-anéis

Axiomas – Conjunto utilizado				
(x * y) * z = x * (y * z). x + y = y + x. (x + y) + z = x + (y + z). x * (y + z) = (x * y) + (x * z). (y + z) * x = (y * x) + (z * x).				
Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	95	434	106,66	50,103
SA	11	248	18,10	34,751
Tietze-SA	11	290	18,74	40,330
Tietze (+)	92	410	103,00	47,063
SA (+)	11	140	14,86	18,846
Tietze-SA (+)	11	260	17,74	35,806
Tietze (V)	52	162	56,12	16,632
SA (V)	11	148	15,98	21,395
Tietze-SA (V)	11	122	14,58	16,544

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	9,00	9º	N	50
SA	3,69	6º	Chi-Square	383,738
Tietze-SA	3,58	4º	g.l.	8
Tietze (+)	8,00	8º	Sig.Asymp.	0,000
SA (+)	3,33	1º		
Tietze-SA (+)	3,49	3º	W de Kendall	0,959
Tietze (V)	6,94	7º		
SA (V)	3,62	5º		
Tietze-SA (V)	3,35	2º		

Quadro 3-18 – Axiomas, estatísticas e resultados teste de *Friedman* para os semi-anéis.

3.17. Estruturas algébricas

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
Tietze	15	11074	347,76	559,689
SA	9	8029	188,85	384,825
Tietze-SA	9	8815	209,89	432,332
Tietze (+)	15	9382	318,02	489,829
SA (+)	9	3479	103,06	184,047
Tietze-SA (+)	9	6404	157,27	326,084
Tietze (V)	15	633	106,08	71,045
SA (V)	9	941	95,42	95,664
Tietze-SA (V)	9	609	67,30	66,448

Algoritmo	Média Níveis	Ordem	Estatísticas	
Tietze	8,74	9º	N	750
SA	5,33	6º	Chi-Square	4101,347
Tietze-SA	5,60	7º	g.l.	8
Tietze (+)	7,85	8º	Sig.Asymp.	0,000
SA (+)	2,98	2º		
Tietze-SA (+)	4,11	4º	W de Kendall	0,684
Tietze (V)	4,74	5º		
SA (V)	3,63	3º		
Tietze-SA (V)	2,02	1º		

Quadro 3-19 – Estatísticas e resultados do teste de *Friedman* para as estruturas algébricas.

A análise das estatísticas e valores obtidos pelo teste de Friedman expressos no quadro 3-19 parece confirmar o que se ia tornando perceptível ao longo dos diversos subcapítulos indicando as implementações *Tietze-SA (V)* e *SA (+)* como as mais eficientes.

Podemos apontar como as causas desta maior eficiência, o melhor desempenho destes dois últimos algoritmos em relação ao primeiro e a elevada redução no número de chamadas ao oráculo efectuada pelo acelerador *(V)*, sendo este número reduzido em cerca de dois terços comparativamente às implementações dos algoritmos de *Tietze* e de *Tietze-SA* e em metade comparativamente à do algoritmo *SA*. No que respeita à redução obtida pelo acelerador *(+)*, embora não tão elevada no caso dos algoritmos de *Tietze* e de *Tietze-SA* como a apresentada pelo acelerador *(V)*, supera-a no caso do algoritmo *SA*.

De facto, após a análise dos resultados obtidos ao longo deste capítulo foi decidida a criação de duas novas implementações, *Tietze-SA (V+)* e *SA (V+)* com o objectivo de maximizar a eficiência e reduzir a complexidade temporal da solução encontrada.

4. *Tietze-SA (V+)* e *SA (V+)*

4.1. Introdução

Os resultados obtidos no capítulo anterior originaram a hipótese de a adição simultânea dos dois aceleradores, *(+)* e *(V)*, aos algoritmos cujas implementações apresentaram um melhor desempenho, *Tietze-SA* e *SA*, conseguir ainda aumentar a eficiência da aplicação criada.

Para validar esta hipótese, foram criadas duas novas aplicações baseadas nas implementações de *Tietze-SA (V+)* e a *SA (V+)* e para cada estrutura algébrica, testados os mesmos conjuntos de axiomas. Os resultados obtidos foram novamente comparados pelo método não paramétrico de *Friedman*, mas desta vez considerando-se só quatro emparelhamentos, correspondendo aos resultados de *SA (+)*, de *Tietze-SA (V)* e das duas recém-criadas implementações *Tietze-SA (V+)* e *SA (V+)*.

Neste capítulo abordar-se-á em primeiro lugar os aspectos relacionados com o código das novas implementações, e em seguida os resultados obtidos quer por estrutura algébrica quer pela sua totalidade onde também serão tecidas considerações sobre a eficiência destas novas implementações. Por último, serão discutidos aspectos relacionados com a complexidade temporal da implementação considerada mais eficiente.

4.2. Implementação de *Tietze-SA (V+)* e *SA (V+)*

O pseudocódigo necessário para a criação das duas novas implementações *Tietze-SA (V+)* e *SA (V+)*, encontra-se listado nas figuras 4-1 e 4-2, resumindo-se à alteração do método *LstNode.ln_eXecute* e inserção do método *LstNode.ln_Exists* nas implementações *Tietze-SA (+)* e *SA (+)*, como foi discutido no segundo capítulo a propósito da adição do acelerador *(V)*.

```
1:  List⟨Node⟩listanosrepositorio ← newList⟨Node⟩()
2:  List⟨Node⟩listanostemporarios ← newList⟨Node⟩()
3:  for all Node n ∈ this · lst_nos do
4:    if n ∉ listanosrepositorio then
5:      List⟨Node⟩listanostemporarios2 ← newList⟨Node⟩ · n.no_eXecute
6:      if listanostemporarios2 · Count ≠ 0 then
7:        for all Node o ∈ listanostemporarios2 do
8:          if o ∉ listanosrepositorio then
9:            adiciona o nó listanosrepositorio
10:         end if
11:         if o ∉ listanostemporarios) then
12:           adiciona o nó listanostemporarios
13:         end if
14:       end for
15:     else if ¬listaSNodes · lSN_Exists(n) then
16:       adiciona nó n a listaSNodes · lst_tmp
17:     end if
18:   end if
19: end for
20: retorna o contentor listanosrepositorio
```

Figura 4-1 – Método *LstNode.ln_eXecute* nas implementações *Tietze-SA (V+)* e *SA (V+)*.

```
1: for all Node n ∈ this · lst_nos do
2:   if n · no_IsEqual(nó passado como argumento) then
3:     retorna true
4:   end if
5: end for
6: retorna false
```

Figura 4-2 – Método *LstNode.Is_Exists* nas implementações *Tietze-SA (V+)* e *SA (V+)*.

4.3. Resultados Obtidos

Neste subcapítulo serão apresentados os resultados obtidos para o teste das estruturas algébricas listadas no quadro 3-1, de um modo semelhante ao efectuado no capítulo anterior, com a diferença de serem unicamente indicadas as estatísticas de dispersão e tendência central e os resultados obtidos pelo teste de *Friedman*, estando realçados a cinzento os valores que se podem considerar mais inabituais.

Tal como já foi referido na introdução, foram utilizados os conjuntos de axiomas utilizados no capítulo anterior, ou seja, conjuntos de cinquenta axiomas por estrutura algébrica constituídos por um conjunto de axiomas equivalente e independente ao conjunto que define a estrutura algébrica em união a três consequências dele, geradas pelo processo Prover9.exe.

4.3.1. Álgebras BCK

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	10	250	124,40	58,553
Tietze-SA (V)	10	148	92,66	42,986
SA (V+)	10	157	95,68	39,682
Tietze-SA (V+)	10	142	90,42	39,150
Algoritmo	**Média Níveis**	**Ordem**	**Estatísticas**	
SA (+)	3,88	4º	N	50
Tietze-SA (V)	2,51	3º	Chi-Square	102,886
SA (V+)	2,20	2º	g.l.	3
Tietze-SA (V+)	1,41	1º	Sig.Asymp.	0,000
			W de Kendall	0,686

Quadro 4-1 – Estatísticas e resultados do teste de *Friedman* para as álgebras BCK.

4.3.2. Grupos booleanos

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	44	187	120,38	42,468
Tietze-SA (V)	40	135	79,02	23,561
SA (V+)	36	136	76,50	24,186
Tietze-SA (V+)	36	129	74,90	22,789
Algoritmo	**Média Níveis**	**Ordem**	**Estatísticas**	
SA (+)	3,88	4º	N	50
Tietze-SA (V)	2,94	3º	Chi-Square	122,708
SA (V+)	1,83	2º	g.l.	3
Tietze-SA (V+)	1,35	1º	Sig.Asymp.	0,000
			W de Kendall	0,818

Quadro 4-2 – Estatísticas e resultados do teste de *Friedman* para os grupos booleanos.

4.3.3. Monoides cancelativos comutativos

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	12	793	388,04	179,098
Tietze-SA (V)	12	302	177,62	77,401
SA (V+)	12	389	195,32	77,256
Tietze-SA (V+)	12	295	187,34	66,411
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	3,95	4º	N	50
Tietze-SA (V)	2,35	3º	Chi-Square	101,955
SA (V+)	2,18	2º	g.l.	3
Tietze-SA (V+)	1,52	1º	Sig.Asymp.	0,000
			W de Kendall	0,680

Quadro 4-3 – Estatísticas e teste de *Friedman* para os monoides cancelativos comutativos.

4.3.4. Semigrupos de *Clifford*

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	16	233	119,06	50,578
Tietze-SA (V)	16	151	85,54	35,609
SA (V+)	16	174	83,02	36,239
Tietze-SA (V+)	16	145	79,72	32,248
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	3,91	4º	N	50
Tietze-SA (V)	2,66	3º	Chi-Square	113,828
SA (V+)	2,12	2º	g.l.	3
Tietze-SA (V+)	1,31	1º	Sig.Asymp.	0,000
			W de Kendall	0,759

Quadro 4-4 – Estatísticas e resultados do teste de *Friedman* para os semigrupos de Clifford.

4.3.5. Semigrupos inversos comutativos

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	9	326	64,28	63,171
Tietze-SA (V)	10	177	58,22	41,381
SA (V+)	9	178	56,50	46,279
Tietze-SA (V+)	9	153	52,72	38,736
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	2,89	3º	N	50
Tietze-SA (V)	2,96	4º	Chi-Square	24,900
SA (V+)	2,18	2º	g.l.	3
Tietze-SA (V+)	1,97	1º	Sig.Asymp.	0,000
			W de Kendall	0,166

Quadro 4-5 – Estatísticas e de teste de *Friedman* para os semigrupos de inversos comutativos.

4.3.6. Anéis comutativos

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	13	145	37,504	44,44
Tietze-SA (V)	13	131	32,201	41,28
SA (V+)	13	119	30,236	39,44
Tietze-SA (V+)	13	125	30,002	39,16
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	2,92	4º	N	50
Tietze-SA (V)	2,87	3º	Chi-Square	40,787
SA (V+)	2,11	2º	g.l.	3
Tietze-SA (V+)	2,10	1º	Sig.Asymp.	0,000
			W de Kendall	0,272

Quadro 4-6 – Estatísticas e resultados do teste de *Friedman* para os anéis comutativos.

4.3.7. Directóides

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	86	319	190,54	68,235
Tietze-SA (V)	77	160	105,66	31,722
SA (V+)	74	173	113,56	38,279
Tietze-SA (V+)	72	154	109,92	33,597
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	4,00	4º	N	50
Tietze-SA (V)	2,51	3º	Chi-Square	115,265
SA (V+)	2,16	2º	g.l.	3
Tietze-SA (V+)	1,33	1º	Sig.Asymp.	0,000
			W de Kendall	0,768

Quadro 4-7 – Estatísticas e resultados do teste de *Friedman* para os directóides.

4.3.8. Corpos

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	13	805	90,06	136,105
Tietze-SA (V)	13	227	45,22	39,219
SA (V+)	13	220	44,06	37,620
Tietze-SA (V+)	13	220	45,14	38,530
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	3,42	4º	N	50
Tietze-SA (V)	2,62	3º	Chi-Square	61,373
SA (V+)	1,95	1º	g.l.	3
Tietze-SA (V+)	2,01	2º	Sig.Asymp.	0,000
			W de Kendall	0,409

Quadro 4-8 – Estatísticas e resultados do teste de *Friedman* para os corpos.

4.3.9. Grupos

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	9	143	29,82	31,243
Tietze-SA (V)	9	113	29,00	28,673
SA (V+)	9	111	26,56	25,636
Tietze-SA (V+)	9	107	27,04	26,535
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	2,78	3º	N	50
Tietze-SA (V)	2,96	4º	Chi-Square	43,904
SA (V+)	2,05	1º	g.l.	3
Tietze-SA (V+)	2,21	2º	Sig.Asymp.	0,000
			W de Kendall	0,293

Quadro 4-9 – Estatísticas e resultados do teste de *Friedman* para os grupos.

4.3.10. Semigrupos Inversos

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	9	160	46,06	34,504
Tietze-SA (V)	9	104	39,26	23,857
SA (V+)	9	101	39,88	23,316
Tietze-SA (V+)	9	98	38,20	21,277
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	3,24	4º	N	50
Tietze-SA (V)	2,74	3º	Chi-Square	47,483
SA (V+)	2,10	2º	g.l.	3
Tietze-SA (V+)	1,92	1º	Sig.Asymp	0,000
			W de Kendall	0,317

Quadro 4-10 – Estatísticas e resultados do teste de *Friedman* para semigrupos inversos.

4.3.11. Reticulados

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	12	3479	416,32	568,492
Tietze-SA (V)	12	609	218,02	121,662
SA (V+)	12	760	207,56	162,327
Tietze-SA (V+)	12	542	204,88	121,905
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	3,39	4º	N	50
Tietze-SA (V)	2,75	3º	Chi-Square	49,032
SA (V+)	2,08	2º	g.l.	3
Tietze-SA (V+)	1,78	1º	Sig.Asymp.	0,000
			W de Kendall	0,327

Quadro 4-11 – Estatísticas e resultados do teste de *Friedman* para os reticulados.

4.3.12. Álgebras MV

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	9	204	56,68	43,937
Tietze-SA (V)	10	132	42,14	29,733
SA (V+)	9	122	45,64	30,165
Tietze-SA (V+)	9	126	43,16	28,641
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	3,36	4º	N	50
Tietze-SA (V)	2,59	3º	Chi-Square	45,503
SA (V+)	2,11	2º	g.l.	3
Tietze-SA (V+)	1,94	1º	Sig.Asymp.	0,000
			W de Kendall	0,303

Quadro 4-12 – Estatísticas e resultados do teste de *Friedman* para as álgebras MV.

4.3.13. Anéis

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	13	343	53,26	68,322
Tietze-SA (V)	13	254	49,78	56,198
SA (V+)	13	229	47,94	54,052
Tietze-SA (V+)	13	247	47,52	54,006
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	2,90	4º	N	50
Tietze-SA (V)	2,78	3º	Chi-Square	30,342
SA (V+)	2,23	2º	g.l.	3
Tietze-SA (V+)	2,09	1º	Sig.Asymp.	0,000
			W de Kendall	0,202

Quadro 4-13 – Estatísticas e resultados do teste de *Friedman* para os anéis.

4.3.14. Semi-reticulados

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	9	72	16,06	16,171
Tietze-SA (V)	9	41	14,06	11,024
SA (V+)	9	44	14,14	11,205
Tietze-SA (V+)	9	37	13,62	9,975
Algoritmo	Média Níveis	Ordem	Estatísticas	
SA (+)	2,69	4º	N	50
Tietze-SA (V)	2,57	3º	Chi-Square	14,643
SA (V+)	2,43	2º	g.l.	3
Tietze-SA (V+)	2,31	1º	Sig.Asymp.	0,002
			W de Kendall	0,098

Quadro 4-14 – Estatísticas e resultados do teste de *Friedman* para os semi-reticulados.

4.3.15. Semi-anéis

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	11	140	14,86	18,846
Tietze-SA (V)	11	122	14,58	16,544
SA (V+)	11	116	14,70	15,805
Tietze-SA (V+)	11	116	14,30	15,505
Algoritmo	**Média Níveis**		**Estatísticas**	
SA (+)	2,54		N	50
Tietze-SA (V)	2,56		Chi-Square	5,000
SA (V+)	2,47		g.l.	3
Tietze-SA (V+)	2,43		Sig.Asymp.	0,172
			W de Kendall	0,033

Quadro 4-15 – Estatísticas e resultados do teste de *Friedman* para os semi-anéis.

4.3.16. Estruturas algébricas

Algoritmo	Mínimo	Máximo	Média	Desvio Padrão
SA (+)	9	3479	125,67	207,067
Tietze-SA (V)	9	609	76,96	74,717
SA (V+)	9	760	77,56	80,826
Tietze-SA (V+)	9	542	75,27	73,027
Algoritmo	**Média Níveis**	**Ordem**	**Estatísticas**	
SA (+)	3,37	4º	N	700
Tietze-SA (V)	2,70	3º	Chi-Square	818,813
SA (V+)	2,12	2º	g.l.	3
Tietze-SA (V+)	1,80	1º	Sig.Asymp.	0,000
			W de Kendall	0,390

Quadro 4-16 – Estatísticas e resultados do teste de *Friedman* para a totalidade dos dados.

Para a totalidade dos dados expressos no quadro 4-16 consideraram-se os obtidos de todas as estruturas algébricas à excepção dos semi-anéis devido ao valor da significância do método de *Friedman* para a estrutura algébrica não validar o tratamento proposto.

No que respeita à eficiência, embora não exista uma grande diferença entre as implementações testadas neste capítulo, em particular entre as três mais eficientes, *Tietze-SA (V)*, *SA (V+)* e *Tietze-SA (V+)*, os valores obtidos por intermédio do teste de *Friedman* indiciam um incremento dela nas duas últimas implementações criadas, sendo destas, a *Tietze-SA (V+)* aquela que sempre apresentou os melhores resultados, quer em relação a globalidade dos dados, quer em relação ao teste individual de cada uma das estruturas algébricas excluindo os grupos e dos corpos nas quais a implementação *SA (V+)* a superou.

4.4. Complexidade temporal

Embora o objectivo deste trabalho tenha sido o de criar uma aplicação prática para a determinação das bases de um sistema de fecho sem ter como preocupação principal a complexidade temporal da solução encontrada tal como é frequente em outros trabalhos do género (ver [8], [11], [14], [15], [16], [20] e [30]), devido à sua importância no desenvolvimento de uma ferramenta útil, será analisada neste capítulo, a complexidade temporal da implementação *Tietze-SA (V+)*.

Tomando como ponto de partida o que foi dito no segundo capítulo relativamente ao acelerador *(V)*, sendo na altura referido que o número máximo de conjuntos derivados seria 2^n para qualquer implementação que o utilize, com n igual ao cardinal do conjunto inicial.

Assumindo, de modo a obtermos o número máximo de chamadas ao oráculo, que cada axioma é premissa única de uma regra de dedutibilidade dos restantes, ou seja, que existe para cada axioma a_n o conjunto das n-1 regras de dedução $\{a_n \rightarrow a_{(n-1)}, \dots, a_n \rightarrow a_{(1)}\}$.

Tendo em atenção que por cada nível de derivação podem existir $\binom{n}{k}$ nós com n igual ao cardinal do conjunto inicial e k igual ao cardinal do nó do nível e sendo por cada um destes nós efectuadas k chamadas ao oráculo, obtemos:

- $\sum_{k=1}^{n} \mathrm{k}\binom{n}{k}$ chamadas ao oráculo para a implementação *Tietze (V)*;
- $2\mathrm{n} + \sum_{k=1}^{n-1} 3\mathrm{k}\binom{n-1}{k}$ chamadas ao oráculo para a implementação *SA (V);*
- $2\mathrm{n} + \sum_{k=1}^{n-1} \mathrm{k}\binom{n-1}{k}$ chamadas ao oráculo para a implementação *Tietze-SA (V)*.

No que respeita ao acelerador *(+)*, este por seu lado não nos permite estabelecer um limite para o número máximo de chamadas ao oráculo devido à redução que efectua no número das mesmas, depender das provas encontradas ao longo da sua execução, podendo-se obter, para as assumpções previamente referidas (todos os axiomas implicam todos os outros sem existirem axiomas redundantes), um resultado compreendido entre n e os limites máximos atrás apontados, dependendo do tempo de execução decorrido até ser fornecida pelo processo *Prover9.exe* a prova que contenha a regra de dedutibilidade requerida.

No quadro 4-17 encontram-se listados para diferentes cardinais do conjunto inicial os limites máximos das diferentes implementações.

#	T (V)	T-SA (V+)	SA (V+)
6	9786	1317	3927
7	82201	9800	29372
8	767208	82217	246619
9	7891281	767226	2301642
10	88776900	7891292	23673836
...	...	...	...
n	$\sum_{k=1}^{n} \mathrm{k}\binom{n}{k}$	$2\mathrm{n} + \sum_{k=1}^{n-1} \mathrm{k}\binom{n-1}{k}$	$2\mathrm{n} + \sum_{k=1}^{n-1} 3\mathrm{k}\binom{n-1}{k}$

Quadro 4-17 – Valores do número máximo de chamadas ao oráculo para diferentes cardinais do conjunto inicial.

Não obstante os valores impressos neste quadro sejam bastante elevados e possam constituir um factor de apreensão para a utilização das aplicações criadas, refere-se que eles só serão atingidos numa situação deveras hipotética, nunca encontrada nos resultados fornecidos pelos testes efectuados.

De facto, os resultados destes sempre apresentaram valores bastante menores que os listados no quadro 4-17 tal como é visível no quadro 4-18 onde estão impressos os valores máximos e da média obtidos pela execução das duas implementações criadas neste capítulo bem como o cardinal dos conjuntos iniciais para cada uma das estruturas algébricas testadas.

Designação	#	SA (V+)		T-SA (V+)	
		Max.	Média	Max.	Média
Grupos booleanos	6	136	76,50	129	74,90
Grupos	6	111	26,56	107	27,04
Semi-reticulados	6	44	14,14	37	13,62
Álgebras BCK	7	157	95,68	142	90,42
Monoides cancelativos comutativos	7	389	195,32	295	187,34
Semigrupos inversos comutativos	7	178	56,50	153	52,72
Directóides	7	173	113,56	154	109,92
Semigrupos inversos	7	101	39,88	98	38,20
Semigrupos de Clifford	8	174	83,02	145	79,72
Álgebras MV	8	122	45,64	126	43,16
Reticulados	9	760	207,56	542	204,88
Anéis comutativos	10	119	30,236	125	30,002
Anéis	10	229	47,94	247	47,52
Corpos	12	220	44,06	220	45,14

Quadro 4-18 – Número máximo de chamadas ao oráculo e média de chamadas para as diferentes estruturas algébricas.

Convém no entanto referir que a inserção neste quadro da coluna contendo o cardinal do conjunto inicial é feita não para demonstrar aspectos relacionados com a complexidade temporal, mas sim com uma certa elasticidade que as implementações apresentam quando a variação do número de axiomas no conjunto inicial é constituída pela variação do número de axiomas pressupostamente invariantes, sendo a maior parte introduzida no contentor que armazena os axiomas invariantes por intermédio do construtor da classe *Node*, evitando a futura construção de nós que os contenham como elementos sobre os quais se tenha de questionar o oráculo.

Voltando à questão da complexidade, e tendo sido estabelecido que os resultados práticos se afastam bastante dos limites máximos teóricos, de modo a que se tenha uma noção estatística do tipo de complexidade que a implementação apresenta, foram efectuadas observações para diferentes conjuntos iniciais de axiomas da estrutura algébrica grupos, contendo cada um deles os axiomas listados no quadro 3-1 em união a conjuntos contendo três a nove consequências da teoria, tendo a escolha desta estrutura sido motivada por dois factores:

- Durante a realização dos testes nunca foi encontrada uma base de cardinal inferior à inicial;
- A execução decorre sem erros para um valor baixo de tempo atribuído aos processos (±3 segundos).

Os valores máximos e da média obtidos para estes diferentes conjuntos estão listados nos quadro 4-19, contendo a figura 4-3 uma representação gráfica dos mesmos.

Nº Consequências	n	Máximos	Média	Desvio Padrão
Três	40	107	27,04	26,535
Quatro	40	237	72,25	72,139
Cinco	40	639	213,45	184,941
Seis	40	1501	580,63	407,151
Sete	40	3464	1300,90	928,852
Oito	40	7346	3262,18	2326,909
Nove	40	15713	7386,81	4396,872

Quadro 4-19 – Máximos, média e desvio padrão obtidos para conjuntos inicias de diferentes cardinais.

Não obstante os cardinais dos conjuntos iniciais ser baixo e o teste ter sido efectuado a um reduzido número de cardinais do conjunto inicial, os resultados parecem sugerir um crescimento exponencial do número de chamadas ao oráculo com o incremento do número de consequências nos conjuntos iniciais.

A validar o referido no parágrafo anterior o teste de ajustamento feito com o auxílio de software específico (ver [37]) forneceu os valores:

- Número máximo de chamadas ao oráculo - a equação $y = 0.7194 * (2.317)^x$, com um coeficiente de correlação de 0.9991;
- Média do número de chamadas ao oráculo - a equação $y = 0.1076 * (2.554)^x$, com um coeficiente de correlação de 0.9988.

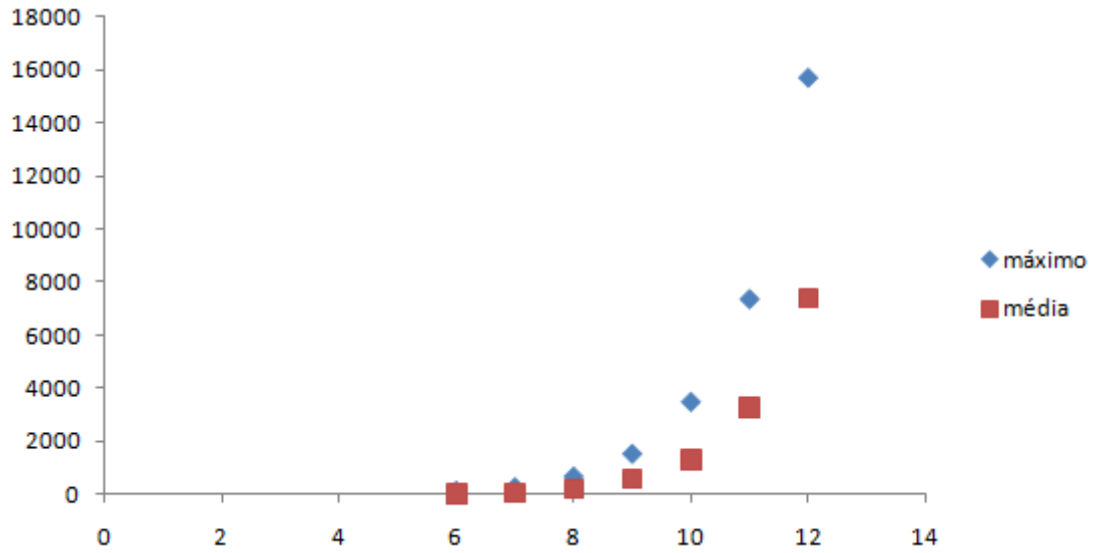

Figura 4-3 – Valores máximos e da média obtidos para conjuntos inicias de diferentes cardinais.

5. Conclusão

5.1. Considerações finais

O resultado final da dissertação com a construção de duas ferramentas informáticas capazes de determinar as bases de um sistema de fecho permite aumentar o arsenal que o matemático tem ao seu dispor na procura de conjuntos de identidades que definam de um modo equivalente e mais simples uma estrutura matemática, definindo-se mais simples como a utilização de um menor número de identidades, de símbolos utilizados ou do tamanho das identidades (ver [6]).

Não obstante o tipo de complexidade que a aplicação apresenta ser limitativa da sua execução para conjuntos iniciais de cardinal elevado, ambas as aplicações são bastante úteis no teste de identidades às quais pressupomos a sua validade como axioma constituinte de um conjunto equivalente mais simples (ver o segundo exemplo do subcapítulo 2.9) ou na procura de novas identidades quando os conjuntos iniciais tenham um cardinal baixo, como foi o sucedido em todos os testes efectuados no terceiro capítulo às diferentes estruturas algébricas.

Refere-se também a maleabilidade que este tipo de construção permite, possibilitando ajustes e alterações tais como a adição e substituição de aceleradores como as que se descrevem em seguida, efectuadas na segunda das aplicações apresentadas em anexo.

Na construção de consequências em vez de se executar simplesmente o processo *Prover9.exe* com uma lista de objectivos vazia utilizou-se numa primeira fase a geração de modelos do processo *Mace4.exe*, atribuindo-se um número máximo de modelos a gerar por execução. Os modelos obtidos foram salvaguardados num ficheiro de texto e utilizados na geração de consequências do processo *Prover.exe* como interpretações a adicionar ao conjunto de premissas.

Na derivação de nós a alteração consistiu na salvaguarda das provas que iam sendo encontradas pelos processos *Prover.exe* num ficheiro auxiliar sendo a cada nova chamada ao processo o ficheiro adicionado como entrada auxiliar. Esta técnica substitui a necessidade da

adição do acelerador *(+)* e é frequentemente utilizada quando se pretende axiomatizar estruturas algébricas (ver por exemplo [36])

Embora não tenha sido feito um estudo detalhado aos resultados destas novas alterações como para os algoritmos e aceleradores, a constatação empírica da sua utilização permite-nos afirmar que em média reduzem para metade o número de chamadas ao oráculo e consequentemente o tempo de execução da aplicação.

A sua utilização na construção de algumas provas matemáticas como na determinação das bases equacionais das álgebras MV (ver [5]) demonstrou a sua aplicabilidade para conjuntos axiomáticos de reduzida dimensão. De facto a expectativa criada foi grande mas em certa medida desvanecida pela incapacidade na obtenção resultados positivos nas respostas a questões relacionadas com a descoberta de uma base-1 ou uma base-2 mais simples para as álgebras MV das apresentadas no artigo já previamente citado ou ainda para outras estruturas algébricas (ver [4]).

No que respeita à sua migração para outros sistemas operativos também é necessário um estudo mais elaborado, parecendo no entanto o projecto *Mono* (ver [29]) um bom ponto de partida para a possibilidade de execução em computadores pessoais cujos sistemas operativos sejam o *Linux* ou o *Mac OS X*.

Não foi no entanto feito nenhum estudo quanto à possibilidade de estabelecer uma interface com o que se pode considerar o sistema de álgebra computacional mais usado actualmente, o interpretador *GAP* (ver [12]). Este ponto, embora não determinante no desenvolvimento das aplicações poderá ser útil na implementação de algoritmos que necessitem na sua construção de utilizar algoritmos cuja implementação nas bibliotecas do interpretador tenha já uma complexidade temporal difícil de suplementar.

Apontam-se assim como possíveis desenvolvimentos futuros:

- A implementação de outros algoritmos;
- A implementação de outros aceleradores;
- A migração para outros sistemas operativos;
- O estabelecimento de uma interface com o interpretador GAP.

Uma ultima consideração a respeito da complexidade temporal das aplicações finais apresentadas nesta dissertação é a de que parece obvio que a complexidade destas estão intimamente relacionadas com a complexidade do algoritmo que implementam não existindo

factores significantes de agravamento por um factor maior que o polinomial. Deste modo, os dois primeiros tópicos do desenvolvimento futuro e com especial relevância o primeiro adquire a maior importância se pretendermos determinar as bases de um sistema de fecho tendo como ponto de partida conjuntos iniciais de cardinal mais elevado.

Ainda relativamente a este tópico comparativamente a outras soluções apresentadas na literatura sobretudo na área da *Data Mining* onde de facto se concentra a maioria dos esforços realizados no sentido de encontrar algoritmos de complexidade temporal polinomial para problemas de um tipo similar ao apresentado na tese de dissertação (ver por exemplo [8], [14], [30]), verifica-se que os resultados apresentados se destinam a estruturas independentes de um tipo diferente, sendo apresentado em [8] uma categorização das mesmas.

De facto dos algoritmos estudados nas referências apresentadas no primeiro parágrafo do subcapítulo 4.4, é o VISOR (ver [15]) aquele que constitui um melhor factor de comparação com este algoritmo, apesar de reclamar um número de chamadas ao oráculo polinomial em função do cardinal do conjunto inicial requer para a sua execução uma enumeração de 2^n.

Conclui-se assim que embora a complexidade da solução apresentada seja elevada e bastante diferente do que se pretenderia para a resolução de um problema deste tipo, dado o panorama actual, as aplicações finais constituem uma boa ferramenta para a realização de estudos como os efectuados nos capítulos precedentes apresentando também como vantagem ser bastante maleável à realização de alterações que incrementem o seu desempenho.

Bibliografia e Referências

[1] ALTOVA ®; UML Tool; http://www.altova.com/umodel.html; consultada em Setembro 2010

[2] ARAÚJO, João & KINYON Michael; An Elegant 3-basis for Inverse Semigroups; 2010; ver também http://arxiv.org/PS_cache/arxiv/pdf/1003/1003.4028v3.pdf

[3] ARAÚJO, João & KINYON Michael; Axioms for Unary Semigroups via Division Operations; 2010; ver também http://es.arxiv.org/PS_cache/arxiv/pdf/1004/1004.0007v1.pdf

[4] ARAÚJO, João & KINYON Michael; Independent Axiom Systems for Nearlattices; 2010; ver também http://es.arxiv.org/PS_cache/arxiv/pdf/1007/1007.3120v1.pdf

[5] ARAÚJO, João & KINYON, Michael & VIGÁRIO, Edgar; Short Equational Bases for MV-Algebras, Commutative BCK-Algebras And LBCK-Algebras; Julho 2010, ver também http://arxiv.org/PS_cache/arxiv/pdf/1007/1007.1601v1.pdf

[6] ARAÚJO, João & McCUNE, William; Computer Solutions of Problems in Inverse Semigroups; Comunications in Algebra, nº 38; pg. 1104-1121; 2010

[7] BEZIEU, Jean-Yves; From Consequence Operator to Universal Logic: A Survey of General Abstract Logic; in Logica Universalis, Towards a General Theory of Logic; 2ª Ed.; Birkhäuser Verlag AG.; 2007

[8] BOLEY, Mario & HORVÁTH, Tamás & POIGNÉ, Alex & WRBEL, Stephan; Listing Closed Sets of Strongly Accessible Set Systems with Applications to Data Minining; Theorical Computing Science; 2009

[9] BURRIS, Stanley & SANKAPPANAVAR, H.P.; A Course in Universal Algebra; Springer-Verlag Graduate Texts in Mathematics; 1981; ver também http://www.math.uwaterloo.ca/~snburris/htdocs/UALG/univ-algebra.pdf

[10] CHAPMAN University; Mathematical Structures: HomePage; http://math.chapman.edu/cgi-bin/structures; consultada em Maio de 2010

[11] GANTER, Bernhard & REUTER, Klaus; Finding All Closed Sets: A General Approach; Order nº 8; pg. 283-290; 1991

[12] GAP Group; GAP – System for computational discrete algebra; http://www.gap-system.org/; consultada em Setembro 2010

[13] GIBBONS, J. Dickison & CHAKRABORTI, Subhabrata; Nonparametric Statistical Inference; 4ª Ed.; Marcel Dekker, 2003

[14] HERMANN, Miki & SERTKAYA, Baris; On the Complexity of Computing Generators of Closed Sets; Proceedings of the 6th international conference on Formal concept analysis; 2008; ver também http://www.lix.polytechnique.fr/~hermann/publications/icfca08final.pdf

[15] JAGER, B. & BANENS, J.; VISOR: Vast Independence System Optimization Routine; Algorithmica 30; pg. 630–644; 2001

[16] JOHNSON, David S. & PAPADIMITRIOU, Christos H. & YANNAKAKIS, Mihalis; On Generating All Maximal Independent Sets; Information Processing Letters, 27; pg. 119-123; 1988; ver também http://www.cs.huji.ac.il/course/2005/mssys/independent.pdf

[17] KINYON, Michael; Automated Deduction and Algebra - Lecture I [PPT]; Universidade de Lisboa Junho 2009; http://www.cs.unm.edu/~veroff/ADAM/2009/LectureI.pdf; consultada em Abril de 2010

[18] KINYON, Michael; Automated Deduction and Algebra - Lecture II [PPT]; Universidade de Lisboa Junho 2009; http://www.cs.unm.edu/~veroff/ADAM/2009/LectureII.pdf; consultada em Abril de 2010

[19] KINYON, Michael; Automated Deduction and Algebra - Lecture III [PPT]; Universidade de Lisboa Junho 2009; http://www.cs.unm.edu/~veroff/ADAM/2009/LectureIII.pdf; consultada em Abril de 2010

[20] LAWLER, E.L. & LENSTRA, J.K. & RINOOY Kan, A.H.G.; Generating all Maximal Independent Sets: NP-Hardness and Polynomial-Time Algorithms; SIAM J. Computer, Vol. 9, nº 3; pg. 558-565; 1980

[21] MARQUES, Paulo & PEDROSO, Hernâni; C# 2.0; FCA; 2005

[22] McCUNE, William W.; Mace4 Reference Manual and Guide; Argonne National Laboratory, Technical Memorandum No. 264; 2003; ver também http://www.mcs.anl.gov/research/projects/AR/mace4/July-2005/doc/mace4.pdf

[23] McCUNE, William; Prover9 and Mace4; http://www.cs.unm.edu/~mccune/mace4/; consultada em Abril de 2010

[24] McCUNE, William; Prover9-Mace4 v05; http://www.cs.unm.edu/~mccune/prover9/gui/v05.html; consultada em Junho de 2010

[25] McCUNE, William W.; OTTER 3.3 Reference Manual; Argonne National Laboratory, Technical Memorandum No. 263; 2003; ver também http://www.cs.unm.edu/~mccune/otter/Otter33.pdf

[26] McCUNE, William & SHUMSKY, Olga; 1999; Ivy – A Preprocessor and Proof Checker for First Order Logic; ver também ftp://info.mcs.anl.gov/pub/tech_reports/reports/P775.ps.Z

[27] McMILLAN, Michael; Data Structures and Algorithms Using C#; Cambridge University Press; 2007

[28] MICROSOFT Corporation; Setup Projects; http://msdn.microsoft.com/en-us/library/996a3fxs(VS.80).aspx; consultada em Junho 2010

[29] NOVELL Inc.; MonoDevolop: Main page; http://monodevelop.com/; consultada em Setembro de 2010

[30] OBIEDKOV, S. & DUQUENNE, V.; Attribute-incremental Construction of the Canonical Implication Basis; Annals Math. Artificial Intelligence 49; pg. 77–99; 2007

[31] ROSS, Sheldon M.; Introduction to Probability and Statistics for Engineers and Scientists; 3ª ed.; Academic Press; 2004

[32] SHESKIN, David J.; Handbook of Parametric & Non-parametric Statistical procedures; 2ª Ed.; Chapman & Hall; 2002

[33] SILVA, Alberto & VIDEIRA, Carlos; UML Metodologias e Ferramentas CASE; Centro Atlântico; 2001

[34] SIMONIC, Aleksander; WINEDT; http://www.winedt.com/index.html; consultada em Setembro de 2010

[35] UNIVERSITY of St Andrews; Tietze biography; http://www-history.mcs.st-and.ac.uk/Biographies/Tietze.html; consultada em Junho de 2010

[36] VEROFF, R. & SPINKS, M.; Axiomatizing the Skew Boolean Propositional Calculus; Journal of Automated Reasoning, 37; pg.3-20; 2006; ver também http://www.cs.unm.edu/~veroff/DOCS/sbpc.pdf

[37] WANER, Stefan & COSTENOBLE, Steven; Simple Regression Page; http://www.zweigmedia.com/RealWorld/newgraph/regressionframes.html; consultada em Junho de 2010

ANEXOS

ANEXO A – Manuais de utilização

A.1 Introdução

Neste anexo são apresentados os manuais de utilização das aplicações que resultaram da dissertação. A primeira destas aplicações quer no seu modo de utilização, quer na sua interface gráfica, segue o descrito no subcapítulo 2.9 para as nove implementações dos três algoritmos, e o segundo, incorpora características das aplicações criadas no terceiro capítulo ao programa anterior, tendo para ambas sido utilizada a implementação *Tietze-SA (V+)* devido a ser a que estatisticamente apresentou a maior eficiência.

O subcapítulo seguinte descreverá do modo de instalação, semelhante para ambas as aplicações sendo em seguida descritas as interfaces gráficas de cada uma delas em dois subcapítulos separados.

Antes de terminar esta introdução alerta-se para o facto de que tudo o que é dito neste anexo, o é assumindo que no computador onde vão ser instaladas e executadas as aplicações, esteja presente na directoria *C:\Program Files\Prover9-Mace4,* o programa *Prover9-Mace4.exe* (ver [24]).

A.2. Instalação das aplicações

Ambas as aplicações são instaladas por intermédio de um módulo de software disponibilizado pela interface de programação *Visual Studio 2008 PE* (ver [28]), tendo como produto final um conjunto de dois ficheiros executáveis designados *setup.exe* e *[nome do programa].msi*.

De modo a facilitar a sua distribuição estes dois ficheiros são comprimidos num ficheiro de extensão *.zip*, iniciando-se a instalação do aplicação num computador, extraindo os ficheiros para o *Ambiente de Trabalho* ou qualquer outro directório do sistema de ficheiros e clicando duas vezes o ícone *[nome do programa].msi.*

Após ter sido efectuado o passo anterior aparecerá uma caixa de diálogo igual à do lado esquerdo da figura A-1, onde o utilizador escolherá a opção *Everyone* clicando de seguida em *Next*.

Figura A-1 – Caixas de diálogo na instalação da aplicação MIND.

Nas sucessivas caixas de diálogo que aparecerem o utilizador continuará clicando em "*Next*" até obter uma caixa de diálogo igual à do lado direito da figura A-1 onde clicará em "*Close*" terminando o processo de instalação da aplicação.

Este término é assinalado com a criação no *Ambiente de Trabalho* de uma nova pasta onde estarão localizados os ficheiros de trabalho do programa e de um ícone consistindo num apontador para o executável do programa tal como é exemplificado na figura A-2.

Figura A-2 – Novas pastas e ícones originadas pelas instalações das aplicações *MIND* e *BMIND*.

A instalação no sistema operativo *Windows 7 (64 bits)* necessita a alteração da "*string*" que aparece por defeito na caixa de texto "*Folder*" (ver figura A-1) de *C:\Program Files(x64)\MIND* para *C:\Program Files\MIND*, o que também e válido para a instalação da aplicação *Prover9/Mace4 v05*.

A3. Interface gráfica da aplicação MIND

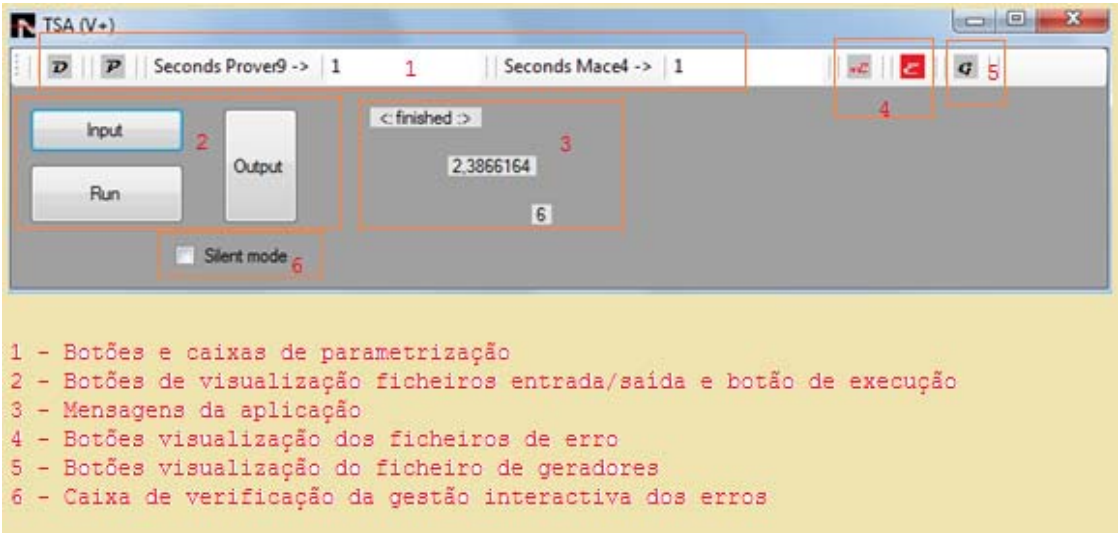

Figura A-3 – Interface gráfica da aplicação MIND.

A primeira destas aplicações é praticamente idêntica às discutidas no subcapítulo 2.9 sendo listada na figura A-3 a interface gráfica da aplicação com seis grupos assinalados, os quais:

- O primeiro grupo contém botões e caixas de texto para parametrizar o tempo máximo em segundos dos processos executados, assumindo um segundo como defeito. O botão P salvaguarda os novos valores inseridos nas caixas de texto e D restaura os valores de defeito;
- O segundo grupo inclui botões para acesso às janelas gráficas dos ficheiros de entrada e saída. Estes são ficheiros de texto, designados, respectivamente, por *entrada.txt* e *fim.txt* cujo conteúdo consiste em "*strings*" representando os axiomas num formato compatível com os processos *Prover9.exe* e *Mace4.exe*. A compatibilidade acima referida exclui a inserção de etiquetas do tipo "*label #*" (utilizadas no armazenamento das regras de dedução), sendo construídas pelo programa como facilmente se verifica comparando na figura A-4, as janelas de entrada e saída;
- O terceiro grupo contém caixas de mensagens que indicam o status de execução, o tempo de execução total e o número de chamadas ao oráculo;
- O quarto grupo contém botões que acedem aos ficheiros onde são salvaguardadas as mensagens de erro, já previamente referidos;

- O quinto grupo é constituído exclusivamente por um botão que acede à janela gráfica do ficheiro geradores.txt, que contém, para cada axioma do nó inicial as regras de derivação encontradas durante a execução (ver figura 2-27);
- O sexto grupo contém uma caixa de verificação que permite ao utilizador escolher se deseja ou não uma gestão interactiva dos erros. Assim, se a caixa estiver verificada, o programa nunca é interrompido e as mensagens de erro são redireccionadas para o ficheiro *erros.txt*. Se não, quando um erro deste tipo ocorrer uma caixa de diálogo é exibida e o utilizador tem a possibilidade de cancelar imediatamente a execução.

Assim, para executar o programa o utilizador deve executar os seguintes passos:

- Introduzir as linhas de texto que representam os axiomas no ficheiro *entrada.txt* num formato compatível ao processos Prover9 e Mace4;
- Definir o tempo máximo para a execução dos processos e pressionar P para salvaguardar os novos valores (opcional);
- Pressionar o botão *Run*.

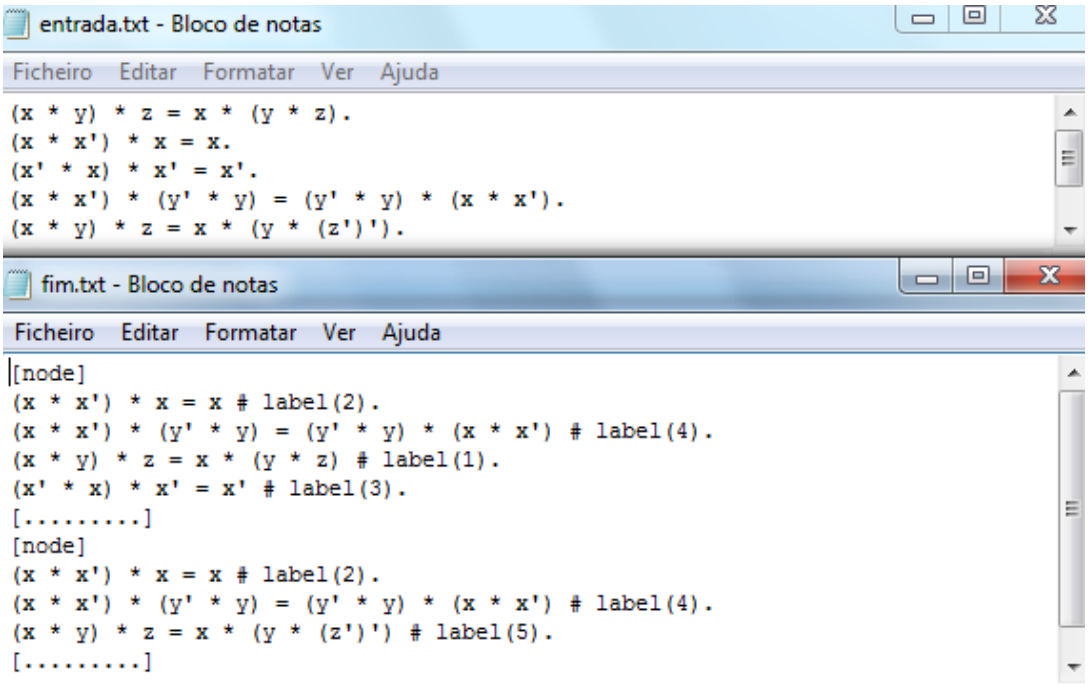

Figura A-4 – Ficheiros *entrada.txt* e *fim.txt*.

A.4. Interface gráfica da aplicação BMIND

A aplicação *BMIND* resulta das aplicações criadas no terceiro capítulo possibilitando um processamento encadeado de vários conjuntos iniciais de axiomas, contendo a figura A-5 a interface gráfica desta aplicação.

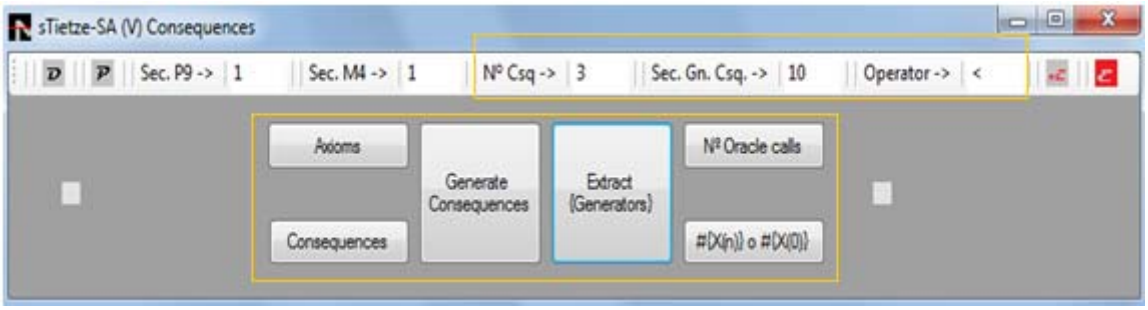

Figura A-5 – Interface gráfica da aplicação BMIND.

Nesta figura estão assinalados a laranja dois grupos de componentes, um constituído por caixas de texto que parametrizam aspectos da aplicação e outro constituído exclusivamente por são botões, os quais:

- *Axioms* – que permite o acesso ao ficheiro de texto *entrada.txt* onde são salvaguardados os axiomas da teoria;
- *Consequences* - que permite o acesso ao ficheiro de texto *axiomas.txt* onde são salvaguardadas as consequências da teoria;
- *Generate Consequences* – que gera consequências da teoria a partir dos axiomas contidos no ficheiro *entrada.txt* durante o tempo especificado pelo utilizador na caixa de texto *Sec.Gn.Csq.* salvaguardando o resultado no ficheiro *axiomas.txt*;
- *Extract {Generators}* – que quando clicado extrai os conjuntos geradores ou as bases dos diferentes conjuntos iniciais.
- *Nº Oracle Calls* – que permite o acesso ao ficheiro de texto onde são salvaguardados os números de chamadas ao oráculo para cada um dos conjuntos iniciais, sendo estes conjuntos inicias a união dos axiomas contidos no ficheiro *entrada.txt* com um conjunto de consequências de cardinal determinado pelo valor inscrito na caixa de texto *Nº Csq*;

- $\#(X_n)$ *o* $\#(X_0)$ - que permite o acesso ao ficheiro de texto *fim.txt* onde são salvaguardados os conjuntos de geradores de cardinal menor maior ou igual os do conjunto inicial dependendo do operador inserido na caixa de texto *Operator*.

No segundo grupo temos as caixas de texto:

- *Nº Csq* – que indica o número de consequências da teoria a acrescentar aos axiomas da teoria de modo a se constituir o conjunto inicial;
- *Sec. Gn. Csq* – que indica o número de segundos durante os quais o programa gera consequências da teoria;
- *Operator* – que especifica qual a relação de ordem entre o cardinal dos conjuntos finais e o do conjunto de axiomas da teoria.

Como feito no subcapítulo 2.9, será em seguida apresentado um exemplo prático de modo a clarificarmos a utilização e objectivo da aplicação recorrendo ao conjunto de axiomas para os semigrupos inversos do subcapítulo 3.11 e a três consequências desta estrutura estando ambos os conjuntos listados na figura A-6 nos ficheiros *entrada.txt* e *axiomas.txt*.

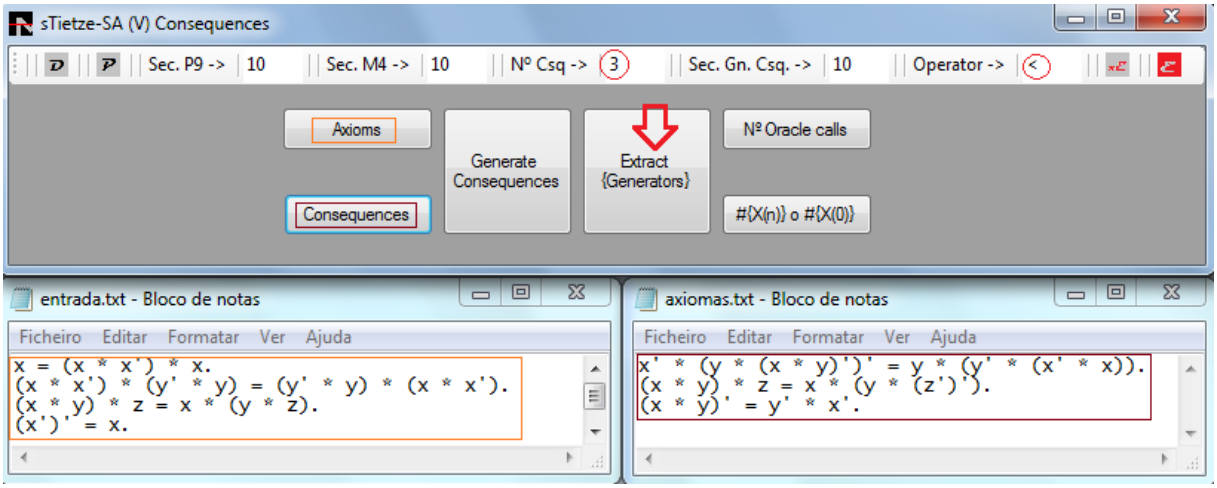

Figura A-6 – Interface gráfica da aplicação *BMIND* e os ficheiros *entrada.txt* e *axiomas.txt*.

Nesta mesma figura, chama-se a atenção para as caixas de texto assinaladas pelos círculos vermelhos *Nº Csq* e *Sec.Gn.Csq*, contendo os valores 3 e <. Deste modo o conjunto inicial conterá sete axiomas, quatro da teoria e três consequências e serão listados os conjuntos finais de cardinal inferior ao número de axiomas que definem a estrutura, ou seja, quatro.

Se, por outro lado tivéssemos nas caixas de texto *Nº Csq* e *Sec.Gn.Csq*, introduzido os valores 1 e <= respectivamente, obteríamos três conjuntos iniciais de cinco axiomas e seriam listados

os conjuntos finais de cardinal igual ou inferior a quatro, como representado no quadro A-1 onde são listados os conteúdos do ficheiro *fim.txt* para ambas configurações.

Nº Csq – 3; Operator - <	Nº Csq – 1; Operator - <=
[node] x = (x * x') * x. (x * y) * z = x * (y * (z')'). (x * x')*(y'* y)=(y' * y)*(x * x'). [node] x = (x * x') * x. x'*(y*(x * y)')'= y *(y'*(x' * x)). (x * y) * z = x * (y * z).	[node] x = (x * x') * x. (x * x')* (y' * y) = (y' * y) * (x * x'). (x * y) * z = x * (y * z). (x')' = x. [node] x = (x * x') * x. (x * y) * z = x * (y * (z')'). (x * x')* (y' * y) = (y' * y) * (x * x'). [node] x = (x * x') * x. x'* (y * (x * y)')' = y * (y' *(x' * x)). (x * y) * z = x * (y * z).

Quadro A-1 – Conjuntos axiomas finais obtidos para diferentes valores dos parâmetros número de consequências e operador.

Quanto à manipulação do ficheiro contendo as consequências da teoria, *axiomas.txt*, convém referir que o utilizador o pode manipular directamente pela sua introdução ou recorrer ao processo *Prover.exe* para as gerar automaticamente durante o período de tempo parametrizado em *Sec.Gn.Csq* clicando o botão *Generate Consequences*, tal como é exemplificado na figura A-7.

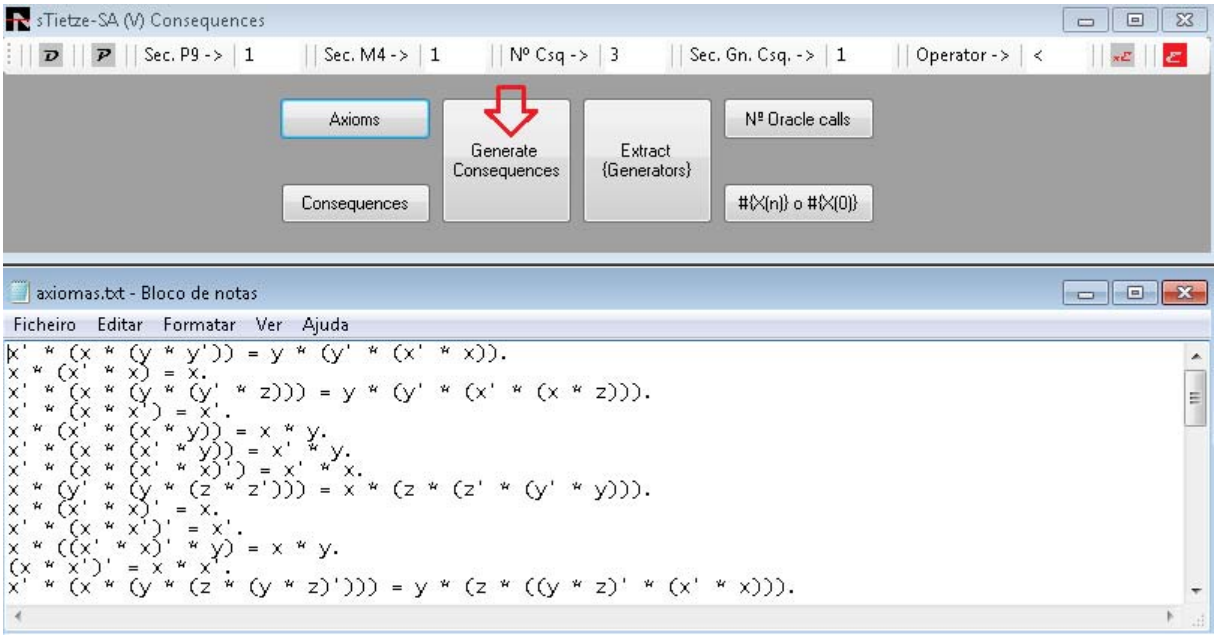

Figura A-7 – Interface gráfica da aplicação *BMIND* e ficheiro *axiomas.txt* contendo consequências.