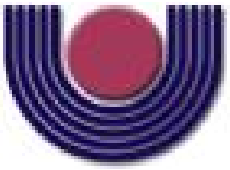

Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Módulo Simulador de Máquinas de Estados Finitos para o sistema PIRAMIDE

Allysson Sanciani Rodrigues

CASCAVEL
2016

ALLYSSON SANCIANI RODRIGUES

MÓDULO SIMULADOR DE MÁQUINAS DE ESTADOS FINITOS PARA O SISTEMA PIRAMIDE

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Adriana Postal

CASCAVEL
2016

ALLYSSON SANCIANI RODRIGUES

MÓDULO SIMULADOR DE MÁQUINAS DE ESTADOS FINITOS PARA O SISTEMA PIRAMIDE

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Adriana Postal (Orientadora)
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Josué Pereira de Castro
Colegiado de Ciência da Computação,
UNIOESTE

Julio Cesar Lazzarim
Consulfarma Informática e Assessoria em Saúde
Ltda

Cascavel, 22 de fevereiro de 2017

AGRADECIMENTOS

Primeiramente a Deus, por todas as bençãos que me deu e por todas as preces que atendeu durante todos esses anos, principalmente nesses últimos quatro, que foram tão difíceis, porém muito recompensadores.

Também a minha família como um todo, mas de forma mais especial a meu pai, minha mãe e meu irmão, que estiveram ao meu lado, me ajudando, que me auxiliaram em todas as etapas possíveis, muitas vezes impossíveis e que acima de tudo estiveram sempre lá quando eu precisei. Por financiarem essa graduação, por cuidarem de mim, por cancelarem planos e rotinas para me socorrer todas as vezes em que precisei.

A minha namorada, Camila, que me auxiliou tanto emocionalmente quanto de formas específicas durante a graduação, me ajudando em diversas situações, além de estar sempre me dando forças a continuar, me alegrando quando estava infeliz e também abrindo meus olhos quando estava empolgado demais com algo que provavelmente não daria certo. Por me oferecer esse amor gigante, que muitas vezes não consigo retribuir, esse amor que me aconchega, que me abraça e me da forças e vontade de sempre continuar motivado a seguir em frente.

A todos os meus professores, pelos conhecimentos repassados, mas de forma especial aos membros da minha banca Josué Pereira Castro e Julio Cezar Lazzarin, pelo tempo gasto e pelos conhecimentos passados. E de forma ainda mais especial a minha orientadora Adriana Postal, que foi sempre muito competente e nunca mediu esforços para me auxiliar e me indicar as melhores formas de prosseguir, não só apenas durante a execução do trabalho, como em outras situações também.

A todos os meus amigos, principalmente os que fiz durante a graduação, que com toda certeza levarei para o resto da vida, que de várias formas diferentes se fizeram presentes, seja na realização de trabalhos ou em estudos em equipe ou de qualquer forma durante esses 4 anos. Pelos momentos alegres proporcionados, pelos momentos divertidos e principalmente pelas boas lembranças que guardarei de cada um. Citando alguns dos mais importantes agradeço a Alysson Girotto, Mateus José(xico), Hamã Cândido, Vitor De Angelis, Alexandre Barreiro (eterno calouro), Gabriel Bruscatto, Guilherme Zabot, Murilo Marcelino (giriloo), Carina Dalsasso, Mauricio Ishida (japa), Irian Rockenbach, Vinicius de Lima (Sheldon), Frank Tominc, Murilo Schaefer e Cristhian Andreani.

As circunstâncias do nascimento de alguém são irrelevantes; é o que você faz com o dom da vida que determina quem você é.

Mewtwo, Pokemon - O filme

Lista de Figuras

Lista de Tabelas

Lista de Símbolos

Σ	Alfabeto de símbolos dos Autômatos
w	String de entrada das Máquinas de estado finais
Q	Conjunto dos estados das Máquinas
δ	Função de transição dos Autômatos Finito Determinístico, de Pilha e Máquina de Turing
q_0	Estado inicial das Máquinas
F	Conjunto de estados finais dos Autômatos Finitos, Autômatos de Pilha e Máquina de Turing
Δ	Função de transição do Autômato Não-Determinístico
Γ	Alfabeto de pilha para o Autômato de Pilha e alfabeto da fita para a Máquina de Turing
β	Espaço em branco na fita de uma Máquina de Turing

Sumário

Resumo

Nesse trabalho será apresentado o software PIRAMIDE, que tem por objetivo auxiliar o ensino de Máquinas de estados finitos na disciplina de Teoria da Computação no curso de Ciência da Computação. No projeto será implementado o módulo de Simuladores do software, sendo eles os simuladores de Autômato Finito, Autômato de Pilha e Máquina de Turing. Ainda interno ao módulo do simulador de Autômato Finito foram implementadas duas funções, uma para transformar Autômato Finito Não-Determinístico em Autômato Finito Determinístico e outra para minimização. Para alcançar esse objetivo foi necessário trabalhar a comunicação entre a linguagem Prolog e a linguagem Java, pois os simuladores estão implementados em Prolog e a interface com o usuário está feita em Java. Foi necessário também encontrar uma implementação da linguagem Prolog que atendesse os requisitos do sistema e possuísse suporte da empresa que o distribui.

Palavras-chave: Autômato Finito, Autômato de Pilha, Máquina de Turing, Simuladores.

Capítulo 1

Introdução

O PIRAMIDE (Programa Interpretador e Resolvedor de Autômatos e MáquInas De Estados finitos) é um sistema que tem como objetivo simular e resolver os modelos computacionais representados pelas máquinas de estados finitos (HOPCROFT; MOTWANI; ULLMAN, 2001).

O sistema atualmente conta com dois grandes módulos, o Resolvedor e o Simulador. O módulo Resolvedor utiliza algoritmos genéticos para resolver os autômatos. No momento estão implementados os resolvedores de autômato finito e de autômato de pilha.

O módulo de simuladores, como o próprio nome sugere, simula autômatos que são criados pelos usuários utilizando a interface gráfica e são capazes de testar esses autômatos a partir de uma string de teste fornecida também pelo usuário. Até o momento os simuladores implementados são os que simulam Autômatos Finitos, Autômatos de Pilha e Máquinas de Turing Padrão que aceita por estado final. Dentro dos simuladores de Autômatos Finitos ainda existem dois componentes que não estão completos, o módulo de minimização de AFs e o transformador de AFN para AFD. A Figura 1.1 representa a forma como está o sistema no momento em um diagrama de componentes.

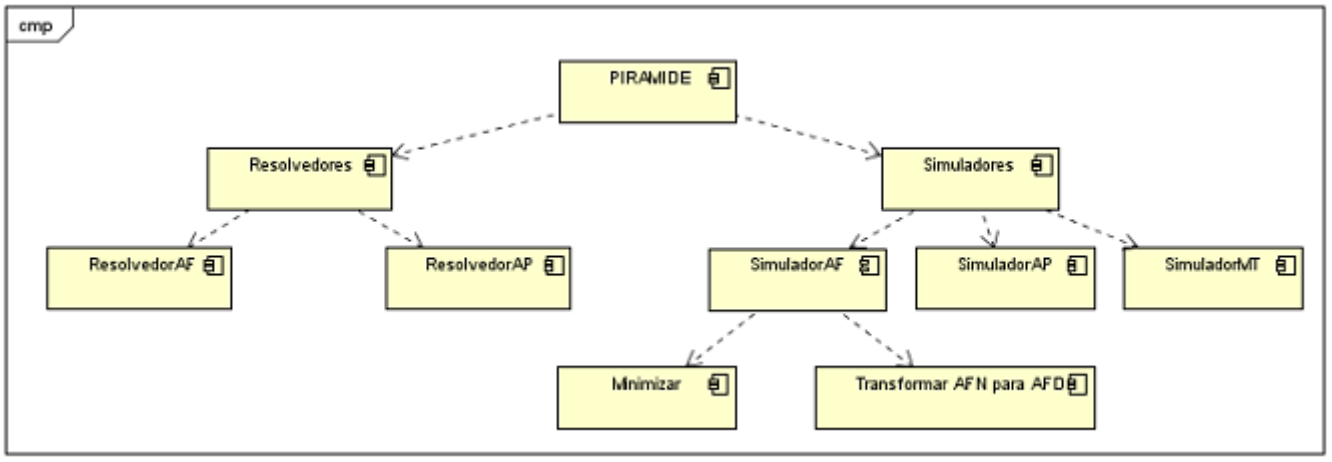

Figura 1.1: Diagrama de Componentes do PIRAMIDE

O software tem como intuito ser usado como ferramenta didática para auxiliar no ensino da disciplina de Teoria da Computação. A disciplina tem por finalidade estudar os paradigmas fundamentais da Ciência da Computação. Grande parte desses modelos que são estudados são puramente matemáticos, mesmo que o computador seja uma máquina física todos os seus modelos são teóricos e são baseados e validados através da matemática (LEWIS; PAPADIMITRIOU, 2000). A partir de uma análise é possível perceber que os alunos, em geral, sentem grande dificuldade em compreender os modelos matemáticos e abstraí-los, seja pela dificuldade em abstrair esses conteúdos ou pela falta de maturidade matemática por parte dos mesmos. Como esses modelos são a base teórica da Ciência da Computação, é de extrema importância que os mesmos sejam bem abstraídos e compreendidos pelos alunos.

A interface de comunicação com o usuário foi desenvolvida na linguagem Java e os algoritmos que simulam as máquinas de estado foram desenvolvidos em Prolog e é utilizado a biblioteca externa JPL para comunicação entre as duas linguagens. Porém a versão que antes era usada no sistema não é mais suportada pelo fabricante, então foi necessário encontrar uma nova máquina e uma nova biblioteca para realizar a comunicação entre o Java e o Prolog.

Os componentes que fazem parte do módulo de simulador de AFs apresentam problemas, pois os resultados obtidos a partir dos algoritmos não são corretos, porém não foi possível identificar qual é o erro, sendo assim será necessário refazer os códigos de transformação de AFN para AFD e de minimização de autômatos.

Logo o objetivo desse trabalho é fazer com que o módulo de Simuladores fique completa-

mente funcional.

1.1 Objetivos

De maneira geral o objetivo desse trabalho é reestruturar o módulo Simulador do sistema PIRAMIDE.

De forma mais específica a implementação ocorreu a partir dos seguintes passos:

- Busca da nova máquina Prolog que atenda aos requisitos do sistema;
- Adequação dos códigos Java e Prolog para essa nova máquina;
- Desenvolvimento dos algoritmos de transformação de AFN para AFD;
- Desenvolvimento do algoritmo de minimização;
- Realização dos testes da implementação;
- Integração do novo módulo ao sistema existente.

1.2 Trabalhos Correlatos

Como trabalhos correlatos temos o software JFLAP (RODGER; FINLEY, 2006). O JFLAP é um software para simular diversos tipos de linguagens formais e autômatos. O mesmo possui uma ferramenta gráfica que pode ser usada como um assistente no processo de aprendizagem de linguagens formais e Teoria dos Autômatos.

Porém o software JFLAP também não recebe mais suporte do fabricante do mesmo e a versão disponível está descontinuada desde maio de 2011.

Além do JFLAP temos o Auger (Ambiente para construção e simulação de autômatos finitos) (BOVET, 2005). O software foi criado com o intuito de auxiliar o ensino de linguagens formais e principalmente o ensino de autômatos finitos. O problema do Auger é que, além de simular apenas autômatos finitos, ele está sem atualização desde novembro de 2009.

Temos também o GAM (*Ginux Abstract Machine*) (JUKEMURA; NASCIMENTO; UCHôA, 2014). Esse trabalho também tem por intuito o auxílio no estudo de Linguagens Formais e Autômatos nos Cursos de Ciência da Computação. É uma ferramenta de código aberto,

se originou de um trabalho de Especialização do Departamento de Computação da Universidade Federal de Lavras, Minas Gerais.

1.3 Organização do Texto

O texto desse trabalho será organizado da seguinte forma:

Capítulo 2 - Revisão bibliográfica que irá descrever as características e conceitos das linguagens formais e dos autômatos, sendo tratado sobre Autômatos Finitos, Autômatos de Pilha e Máquina de Turing.

Capítulo 3 - Descreverá todos os recursos e materiais que foram utilizados visando atingir o objetivo do projeto.

Capítulo 4 - Irá apresentar as formas como foram implementadas as funcionalidades do PIRAMIDE e os testes que foram realizados.

Capítulo 5 - Apresentará as considerações finais do projeto e os quais serão os trabalhos futuros.

Capítulo 2

Introdução à Teoria das Linguagens e Autômatos

Nas próximas seções, iremos definir os conceitos gerais relacionados às Linguagens Formais e aos Autômatos, conceitos esses baseados em (HOPCROFT; MOTWANI; ULLMAN, 2001), (VIEIRA, 2006) e (SUDKAMP, 1997).

2.1 Linguagens Formais

Toda linguagem formal tem um alfabeto associado. Um alfabeto é um conjunto não-vazio e finito de símbolos, geralmente representado pelo símbolo Σ.

Símbolos, por sua vez, são os componentes do alfabeto e são os que constituem a string de entrada.

Uma string (ou palavra ou cadeia) é uma sequência finita de símbolos que estão contidos no alfabeto Σ. Toda string possui um tamanho. O tamanho de uma string é definido pelo número de símbolos que a compõe.

Uma linguagem formal é definida por:

1. uma sintaxe bem definida, de forma que, ao ser dado uma string, é sempre possível dizer se a mesma pertence ou não a linguagem;

2. uma semântica precisa, de modo que não contenha sentenças sem significado ou ambíguas.

2.2 Autômatos Finitos

Os autômatos finitos são máquinas de memória limitada que tem a capacidade de realizar o reconhecimento de Linguagens Regulares. Para isso os mesmos recebem uma string de entrada em uma fita e a partir dessa fita geram uma saída de aceitação ou rejeição da mesma.

Podem ser chamados de forma genérica de máquinas abstratas e são divididos em Autômatos Finitos Determinísticos e Autômatos Finitos Não-Determinísticos. As relações e diferenças entre esses dois modelos serão apresentadas com mais detalhes nas seções 2.2.7 e 2.2.8.

2.2.1 Composição básica de um AF

Um Autômato Finito pode ser considerado uma máquina com basicamente as seguintes divisões:

Fita: Dispositivo que contém a string de entrada que será processada pela máquina.

Unidade de Controle: Representa o estado atual da máquina. É composta por um cabeçote que anda, exclusivamente para a direita, na fita, fazendo a leitura das células da mesma.

Programa ou Função de Transição: Função que controla as leituras da máquina e define o estado da mesma.

2.2.2 Diagrama de transições

É um grafo orientado onde os nós representam as entradas e as arestas representam as transições sobre determinados símbolos de entrada, aceitos pela máquina. Na Figura 2.1 é mostrado um diagrama de transições para um Autômato capaz de reconhecer um símbolo **a** e vários símbolos **b** seguintes.

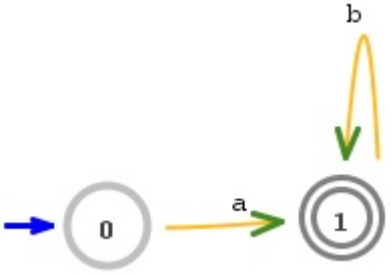

Figura 2.1: Diagrama de Transições que reconhece a expressão regular ab*

2.2.3 Tabela de transições

É um modo de representar as transições presentes em um Autômato Finito. Basicamente pode ser descrito por uma tabela com três colunas: estado atual, símbolo da fita que pode ser lido nesse estado do Autômato e o estado para onde a computação será levada ao ler o símbolo. Um exemplo dessa tabela está representado na Tabela 2.1.

Tabela 2.1: Tabela de Transições

Estado atual	Símbolo da fita	Próximo estado
Q0	1	Q0
Q0	0	Q1
Q1	0	Q1
Q1	1	Q1

2.2.4 Estado Inicial

O estado inicial representa o ponto de início para a computação do Autômato. Esse estado é obrigatório em todo Autômato, e está representado na Figura 2.2.

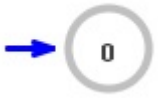

Figura 2.2: Estado Inicial

2.2.5 Estado Final

O estado final é onde é dada a aceitação da entrada do Autômato. Um Autômato necessita obrigatoriamente de um estado final, porém nada impede que existam mais ou que todos os estados do Autômato sejam finais. A representação desse estado pode ser vista na Figura 2.3.

Figura 2.3: Estado Final

A teoria de Autômatos não impede que um estado inicial também possa ser um estado final. A representação do mesmo é dada pela união das características gráficas de ambos os estados,

como pode ser visto na Figura 2.4.

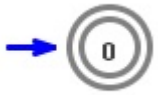

Figura 2.4: Estado Final e Inicial

2.2.6 Critérios de parada

A parada de uma computação pode ser de aceitação ou de rejeição dependendo da entrada informada. Consideremos a entrada como w $\in \Sigma^*$. As condições de parada podem ser as seguintes:

1. Após processar o último símbolo da fita o Autômato atinge um estado final. A computação é encerrada e a entrada w é aceita;

2. Após processar o último símbolo da fita o Autômato não atingiu um estado final. A computação é encerrada e a entrada w não é aceita;

3. A função de transição é indefinida para o argumento (estado atual e símbolo lido). A computação é encerrada e a entrada w não é aceita.

2.2.7 Autômatos Finitos Determinísticos (AFD)

São máquinas de poder computacional limitado, porém de extrema importância para aplicações de reconhecimentos de padrões, como a construção do analisador léxico de um compilador.

Um AFD é definido por uma quíntupla M = (Q, Σ, δ, q_0, F) onde:

Q: é um conjunto finito e não vazio de um ou mais elementos chamados de estados;

Σ: é chamado de alfabeto e representa um conjunto finito e não vazio de símbolos que são reconhecidos pelo Autômato;

δ: é a função de transição de estados e sua função é identificar as transições possíveis em cada configuração do Autômato, ou seja, para cada par (estado atual e símbolo lido) a função informa qual o novo estado que o Autômato deve seguir;

q_0: é um estado que pertence a Q. É o estado inicial do Autômato, ou seja, o estado onde deve-se começar o processamento;

F: é um subconjunto de Q que consiste em todos os estados finais do Autômato.

Para que um Autômato Finito possa ser classificado como determinístico é necessário que o mesmo siga um conjunto de três regras:

1. Não deve possuir transições sobre strings;
2. Não deve possuir transições sobre o vazio;
3. Não deve possuir mais de uma transição sobre o mesmo símbolo partindo de um mesmo estado.

Na Figura 2.5 temos uma representação de um Autômato Finito Determinístico.

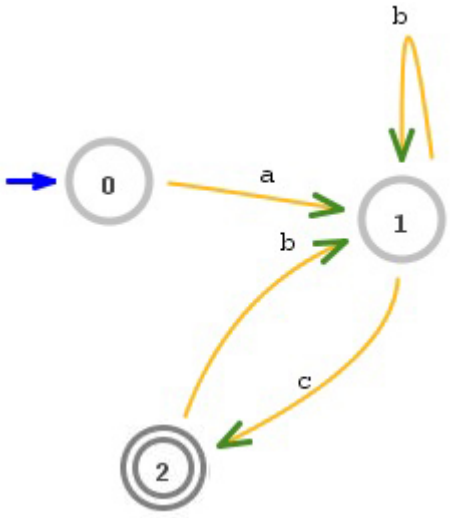

Figura 2.5: Autômato Finito Determinístico

2.2.8 Autômatos Finitos Não-Determinísticos (AFN)

São máquinas que possuem mais liberdade que os AFDs, por possuírem alguns aprimoramentos em relação aos mesmos. Porém, mesmo com esses novos recursos o poder computacional dos AFNs são os mesmos dos AFDs.

Um AFN é definido por uma quíntupla (Q, Σ, Δ, q_0, F) onde:

Q: é um conjunto finito e não vazio de um ou mais elementos chamados de estados;

Σ: é chamado de alfabeto e representa um conjunto finito e não vazio de símbolos que são reconhecidos pelo Autômato;

Δ: é a função de transição de estados e sua função é identificar as transições possíveis em cada configuração do Autômato, ou seja, para cada par (estado atual e símbolo lido) a função informa qual o novo estado ou conjunto de estados que o Autômato deve seguir;

q_0: é um estado que pertence a Q. É o estado inicial do Autômato, ou seja, o estado onde deve-se começar o processamento;

F: é um subconjunto de Q que consiste em todos os estados finais do Autômato.

Como dito anteriormente o AFN possui outros recursos que não estão presentes no AFD. Esses recursos são:

1. Possuir transição sobre o símbolo vazio: nesse caso a máquina não lê nenhum símbolo, porém ela avança para um próximo estado. No software PIRAMIDE o vazio é representado por [ ]. Um exemplo de uma transição sobre o símbolo vazio pode ser visto na Figura 2.6.

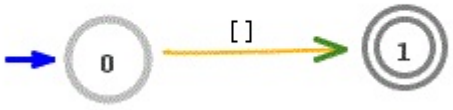

Figura 2.6: Transição sobre vazio

2. Possuir transição sobre string: nesse caso a transição é feita sobre uma sequência de símbolos e não apenas sobre um único símbolo, como apresentado na Figura 2.7.

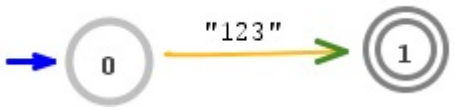

Figura 2.7: Transição sobre string

3. Possuir duas ou mais computações diferentes para uma mesma entrada: nesse caso temos a utilização da capacidade do AFN de ir para estados diferentes lendo o mesmo símbolo, assim é possível reconhecer uma mesma entrada por computações diferentes. Podemos ver um exemplo disso na Figura 2.8. Nesse exemplo podemos perceber que a partir de três computações diferentes é possível reconhecer a string "123".

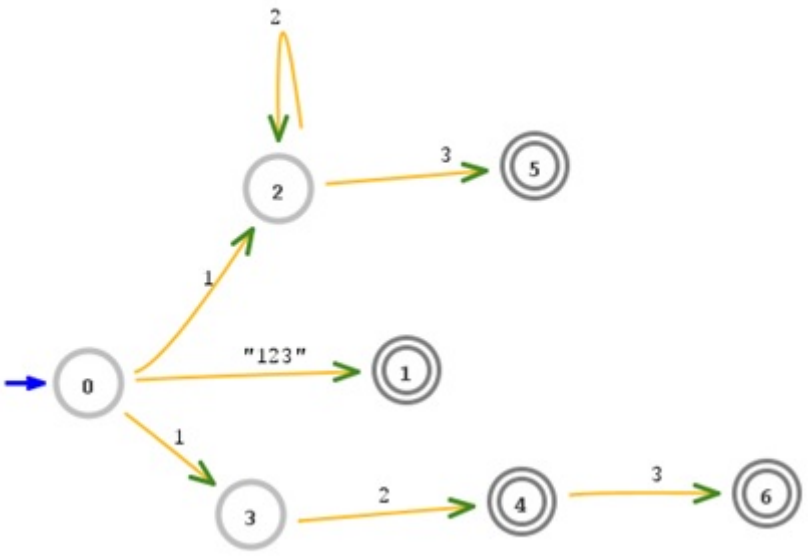

Figura 2.8: Autômato que reconhece a string "123"de três maneiras diferentes

2.2.9 Transformação de AFN para AFD

Em alguns momentos a construção de um Autômato Determinístico pode ser complexa, sendo mais simples a construção de um Não-Determinístico. Porém para alguns casos é necessário que o autômato seja determinístico, para isso temos o algoritmo de transformação de AFN para AFD.

A realização da transformação de AFN para AFD é feita quando se deseja retirar o não-determinismo de um AFN, tornado-o assim um AFD. Para que isso ocorra é necessário que o novo autômato criado após a transformação atenda às três regras que define um autômato finito determinístico, como visto na seção 2.2.7.

Para essa transformação utilizamos um algoritmo de três passos, que visa realizar a retirada do não-determinismo.

Passo 01: Nessa etapa do algoritmo deve-se retirar as transições sobre string. Ou seja, dado uma transição A, tal que A = (1, 'abb', 4), ela seria transformada por esse algoritmo em:

A = (1, a, 2);

B = (2, b, 3);

C = (3, b, 4);

Passo M 1 : Retirada de transições sobre strings

Q1 = Q;
$\Delta 1 = \Delta$;
Para cada { Transição (q, y, q') pertencente a Δ com |y| > 1}{
  //remove de $\Delta 1$ a transição sobre strings:
  $\Delta 1 = \Delta 1 - \{(q, y, q')\}$;
  //suponha |y| = k para k > 1
  Q1 = Q1 $\cup \{q1, ..., qk - 1\}$
  $\Delta 1 = \Delta 1 \cup \{(q, y(1), q1), (q1, y(2), q2), ..., (qk - 1, y(k), q')\}$;
}

Passo 02: Nessa etapa do algoritmo deve-se retirar as transições que são realizadas sobre o vazio, quando o estado do autômato é trocado sem que nada seja lido. Um exemplo da aplicação desse algoritmo pode ser visto nas figuras 2.9 e 2.10. Na primeira temos um autômato que apresenta uma transição sobre o vazio e na segunda temos o mesmo autômato só que após a retirada dessa transição.

Passo M 2 : Retirada de transições sobre vazio

$\Delta 2 = \emptyset$;
//elimine as transições vazias
Para cada {Estado q pertencente a Q1}{
  Para cada {símbolo a pertecente a Σ} {
    $\Delta 2 = \Delta 2 \cup \{(q, a, q') \mid (q, [1, a])$ produz a partir de vários passos $(q', [2, a])\}$;
  }
}

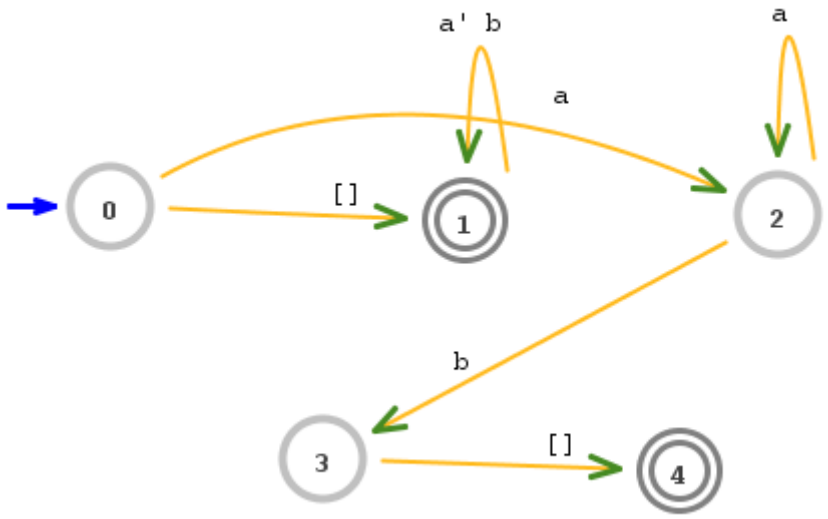

Figura 2.9: Autômato que possui transição sobre vazio

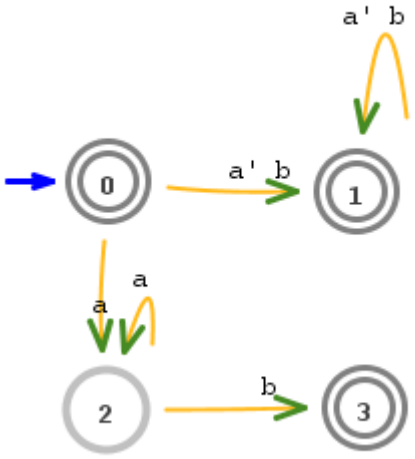

Figura 2.10: Autômato da Figura 2.9 com a transição vazia eliminada

Passo 03: Nessa etapa do algoritmo retiramos as transições que levam para estados lendo o mesmo símbolo. Assim um autômato como o da Figura 2.11 se torna igual ao autômato da Figura 2.12.

Passo M 3 : Retirada de transições para estados diferentes sobre o mesmo símbolo

```
//estados de M3 são conjuntos de estados de M2
Q3 = {{q0}};
Δ3 = ∅;
//Qu tem estados sucessores não processados de Q3
Qu = Q3;
Enquanto{Qu <> ∅}{
   //pegue um estado não processado de Qu
   Pegue qualquer Q' ∈ Qu:
   Qu = Qu - {Q'};
   //compute os sucessores de Q' para cada símbolo de entrada a
   Para cada{Símbolo ∈ Σ}{
      R = ∅;
      Para cada{Estado q' ∈ Q'}{
         Para cada{Estado q2 ∈ Q2}{
            Se{(q', a, q) ∈ Δ2 }{
               R = R ∪ {q2};
            }
            Q3 = Q3 ∪ {R};
            Δ3 = Δ3 ∪ {(Q', a R)};
         }
      }
   }
}
```

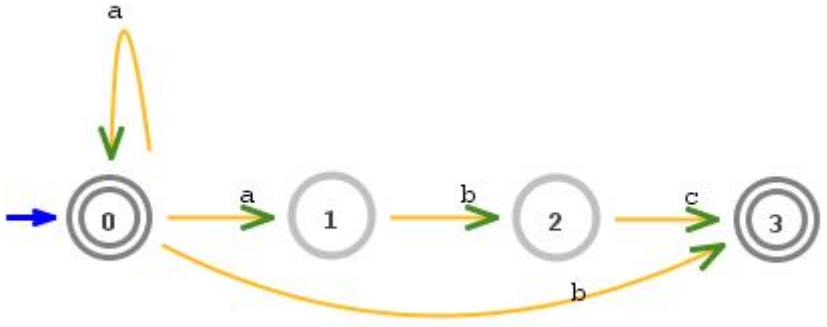

Figura 2.11: Autômato que possui transição sobre mesmo símbolo

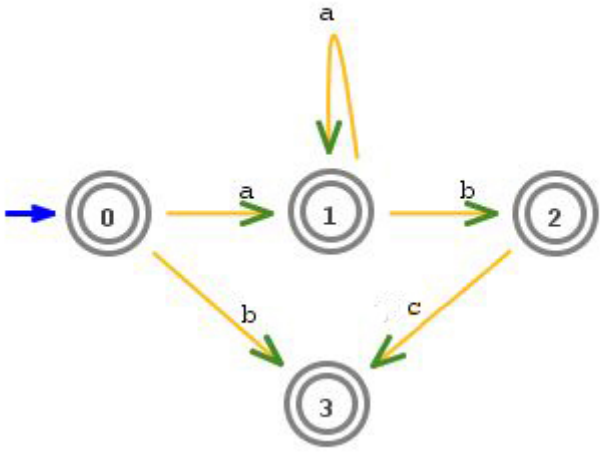

Figura 2.12: Autômato da Figura 2.11 com a transição sobre o mesmo símbolo eliminada

2.2.10 Minimização de autômatos

Na construção de um autômato pode ocorrer a geração de uma quantidade excessiva de estados e algumas vezes esses estados podem ser desnecessários no autômato.

O processo de minimização de autômatos tem por finalidade, portanto, retirar os estados equivalentes, ou seja, construir um autômato com o mínimo de estados possíveis. Dois estados p e q são ditos equivalentes se, e somente se, para qualquer w pertencente a Σ, δ(q, w) e δ(p, w) resultam simultaneamente a estados finais, ou não finais.

Para que um autômato possa ser minimizado ele precisa atender a três requisitos:

1. Deve ser determinístico;

2. Não pode possuir estados inacessíveis;

3. A função de transição deve ser total, ou seja, a partir de qualquer estado são previstas transições para todos os símbolos do alfabeto. Caso a função de transição não seja total deve ser adicionado um estado não-final `d`, tal que todos os símbolos que não são lidos pelos outros estados agora serão lidos e irão para esse estado.

Para realização da minimização de um autômato podemos utilizar o algoritmo de minimização, que ocorre em três passos:

1. Tabela: Construir uma tabela relacionando os estados distintos, onde cada par de estados ocorre somente uma vez, como mostrado na Figura 2.13;

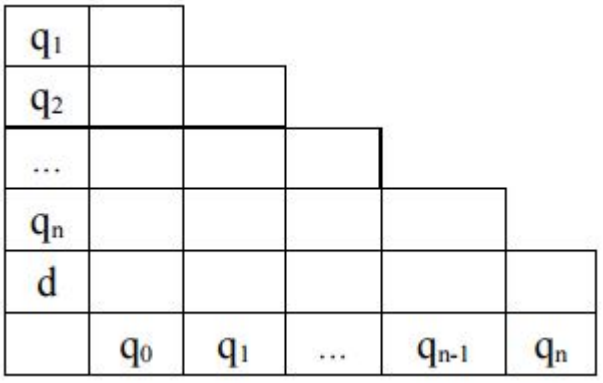

Figura 2.13: Tabela relacionando todos os estados

2. Marcação dos estados trivialmente não-equivalentes: Marcar todos os pares do tipo (estado final, estado não-final), pois, obviamente estados finais não são equivalentes a não-finais;

3. Marcação dos estados não-equivalentes: Para cada par qu, qv não-marcado e para cada símbolo a $\in \Sigma$, suponha que δ(qu, a) = pu e δ(qv, a) pv e:

   (a) se pu = pv, então qu é equivalente a qv para o símbolo a e não deve ser marcado;

   (b) se pu $\neq$ pv e o par $\{pu, pv\}$ não está marcado, então $\{qu, qv\}$ é incluído em uma lista a partir de $\{pu, pv\}$ para posterior análise;

   (c) pu $\neq$ pv e o par $\{pu, pv\}$ está marcado, então:

      i. $\{qu, qv\}$ não são equivalentes e devem ser marcados;

      ii. se $\{qu, qv\}$ encabeça uma lista de pares, então marcar todos os pares da lista (e, recursivamente, se algum par da lista encabeça outra lista);

4. Unificação dos estados equivalentes: Os estados dos pares não-marcados são equivalentes e podem ser unificados como segue:

   (a) a equivalência de estados é transitiva;

   (b) pares de estados não-finais equivalentes podem ser unificados como um estado não-final;

(c) pares de estados finais equivalentes podem ser unificados como um único estado final;

(d) se algum dos estados equivalentes é inicial, então o correspondente estado unificado é inicial.

5. Exclusão dos estados inúteis: Por fim, os estados chamados inúteis devem ser excluídos. Um estado `q` é inútil se é não-final e a partir de `q` não é possível atingir um estado final. Caso tenha sido adicionado um estado de falha `d` para tornar a função de transição completa, ele será retirado aqui.

Temos um exemplo, portanto, de um autômato ainda não minimizado (Figura 2.14) e o mesmo autômato após o processo de minimização (Figura 2.15)

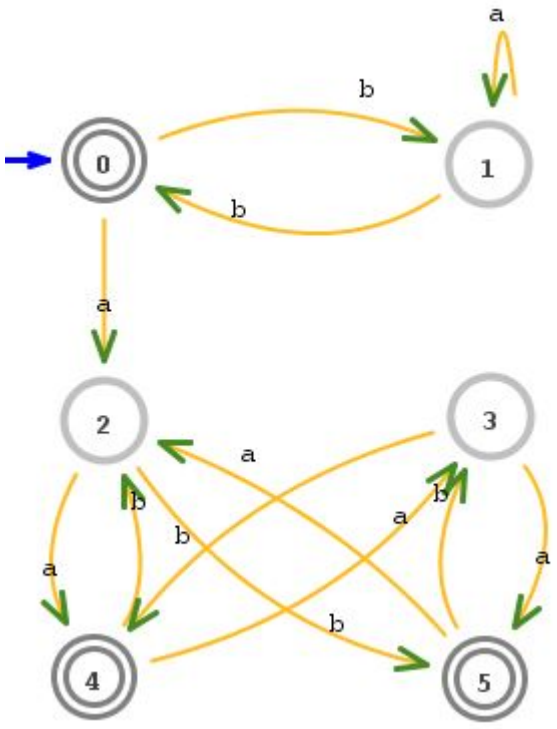

Figura 2.14: Autômato não minimizado

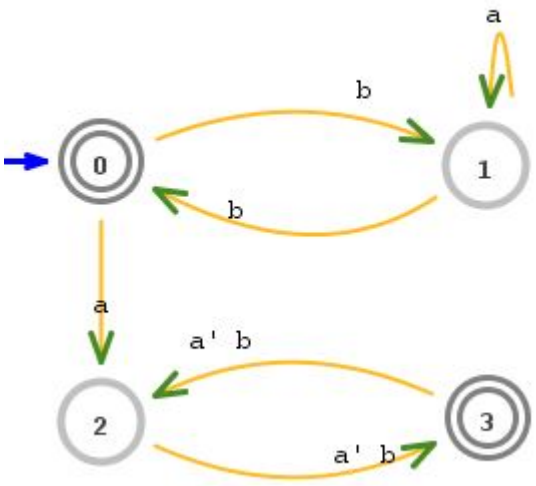

Figura 2.15: Autômato após processo de minimização

2.3 Autômatos de Pilha

O autômato de pilha em sua essência é um autômato finito não determinístico com transições sobre vazio e com uma característica adicional: uma pilha na qual podem ser armazenados símbolos. A presença da pilha permite a esse tipo de autômato "lembrar" uma quantidade infinita de informações. Porém, assim como a estrutura de dados chamada de pilha, a pilha só pode acessar as informações da forma *last-in-first-out*.

O formato LIFO corresponde ao estilo de manipulação de listas onde o último a ser inserido é o primeiro a ser retirado, ou seja, se a inserção é feita pela esquerda a retirada também deve ser feita a esquerda e caso a inserção ocorra pela direita, a remoção também deve ser pela direita.

A notação formal de um autômato de pilha envolve 6 componentes. A especificação do mesmo é dada da seguinte forma: M = (Q, Σ, Γ, δ, q_0, F), onde:

Q: Um conjunto finito de estados, assim como os do Autômato Finito.

Σ: Um conjunto finito de símbolos de entrada, também análogos ao conjunto do Autômato Finito.

Γ: É o alfabeto da pilha. Esse componente não possui um análogo no Autômato Finito e é o conjunto de símbolos permitidos na pilha. São representados usualmente por letras

maiúsculas.

δ: É a função de transição. Assim como no Autômato Finito, ela controla o estado do autômato. Formalmente, δ pode ser escrito como δ(q, a, A), onde:

(a) q é um estado de Q;

(b) a é um símbolo de Σ ou é uma string vazia;

(c) A é um símbolo da pilha e membro de Γ. Representa o que será desempilhado da pilha. Se A for vazio, nada será desempilhado.

A saída de δ é um conjunto finito de pares (qj, B), onde qj é o novo estado e B é o símbolo que substitui A no topo da pilha. Por definição, se B for vazio, nada será empilhado.

q_0: É o estado inicial. O autômato está nesse estado antes de fazer qualquer transição e q0 deve ser um estado de Q.

F: Conjunto de estados finais, onde são dadas as computações de aceitação.

2.3.1 Representação de um Autômato de Pilha

Assim como os autômatos finitos é necessário ter uma representação gráfica dos autômatos de pilha. As notações são muito semelhantes entre ambos, porém no autômato de pilha é necessário identificar os símbolos da pilha que são empilhados e desempilhados entre cada transição. Um exemplo de representação gráfica de um Autômato de Pilha pode ser visto na Figura 2.16.

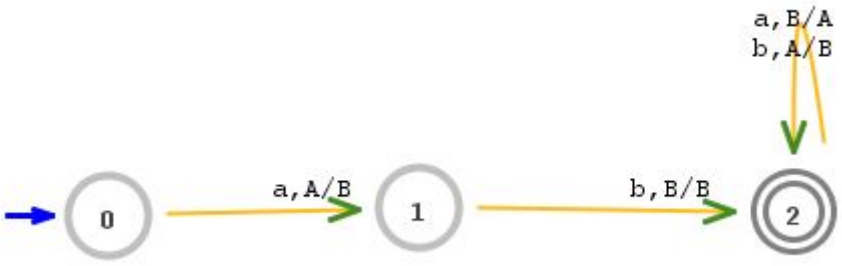

Figura 2.16: Representação gráfica de um Autômato de Pilha

2.3.2 Autômato de Pilha Determinístico (APD)

Um AP pode ser considerado determinístico caso o mesmo não possua transições compatíveis.

Duas transições $\delta(e, a, b)$ e $\delta(e, a', b')$ são ditas compatíveis se, e somente se: $(a = a'$ ou $a = \lambda$ ou $a' = \lambda)$ e $(b = b'$ ou $b = \lambda$ ou $b' = \lambda)$.

São muito importantes visto que esses autômatos lidam com uma classe de linguagens para as quais há reconhecedores eficientes.

Autômatos de Pilha Determinístico e Não-Determinísticos não são equivalentes.

2.3.3 Autômato de duas pilhas (A2P)

O Autômato de duas pilhas também é representado pelo conjunto M = (Q, Σ, Γ, δ, q0, F), porém existe uma alteração da função de transição δ sendo necessário agora representar em cada transição as movimentações que ocorrem em ambas as pilhas. Ou seja, em um A2P δ é representado por (q, a, A, B), onde:

1. q é um estado de Q;
2. a é um símbolo de Σ ou é uma string vazia;
3. A é um símbolo da pilha e membro de Γ. Representa o que será desempilhado na primeira pilha. Caso A seja vazio nada será desempilhado;
4. B é um símbolo da pilha e membro de Γ. Representa o que será desempilhado na segunda pilha. Caso B seja vazio nada será desempilhado.

A saída de δ agora não é mais um par e sim [qj, C, D], onde C é o valor substituído por A na primeira pilha e D é o valor substituído por B na segunda pilha.

O sistema do projeto implementa apenas Autômatos de Duas Pilhas, pelo fato do poder computacional de um Autômato de Múltiplas Pilhas (AnP, n > 2) ser equivalente ao de um A2P (MENEZES, 2002).

2.4 Máquinas de Turing

Uma Máquina de Turing nada mais é do que uma máquina que opera assim como os AFs e os APs, porém, além da capacidade de ler da fita de entrada possui a capacidade de escrever nessa fita (Figura 2.17).

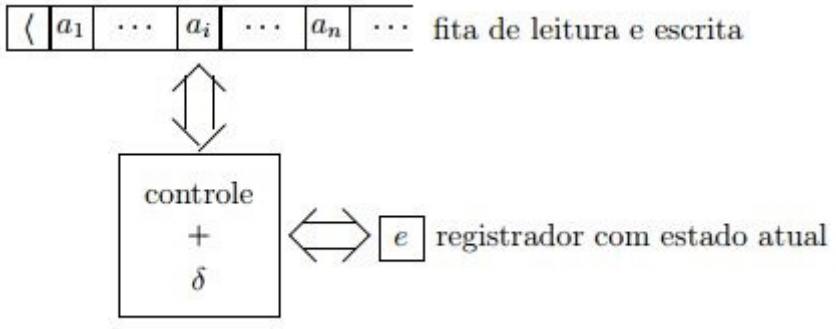

Figura 2.17: Arquitetura de uma Máquina de Turing

O cabeçote de leitura da Máquina de Turing pode se mover para direita e para esquerda na fita e a mesma possui uma divisão por células na qual, em cada célula existe apenas um símbolo, sendo limitada a esquerda. Além da fita essa máquina possui um registrador para armazenar o estado atual, possui um conjunto de instruções, que nada mais é do que a função de transição e uma unidade de controle.

Como o cabeçote da fita pode se movimentar para direita e para esquerda e a fita não é limitada a direita, no início da mesma existe um símbolo representando o início da fita.

2.4.1 Máquina de Turing Padrão (MT padrão)

A definição que será dada a seguir é a de uma Máquina de Turing Padrão que aceita por estado final. Esse foi o tipo escolhido para ser implementado no sistema desse projeto. Existem muitas outras variações da MT, porém as mesmas não alteram seu poder computacional sendo equivalentes a MT padrão, sendo esse o motivo da escolha da mesma para implementação.

Formalmente uma Máquina de Turing Padrão é definida por M = (Q, Σ, Γ, δ, q_0, F). Onde:

Q: Um conjunto finito de estados.

Σ: Um conjunto finito de símbolos de entrada.

Γ: É o alfabeto da fita. Possui todos os símbolos que podem aparecer na fita, incluindo o Σ e o símbolo β, que representa uma célula vazia.

δ: É a função de transição.

q_0: É o estado inicial, deve estar contido em Q.

F: Conjunto de estados finais, onde são dadas as computações de aceitação.

2.4.2 Representação de uma Máquina de Turing

Assim como os autômatos finitos e os autômatos de pilha é necessário ter uma representação gráfica para as Máquinas de Turing. As notações são muito semelhantes às anteriores, porém na Máquina de Turing é necessário identificar símbolo lido, símbolo escrito e direção do cabeçote na fita (onde D é direita e E é esquerda). Um exemplo de representação gráfica de uma Máquina de Turing pode ser visto na Figura 2.18.

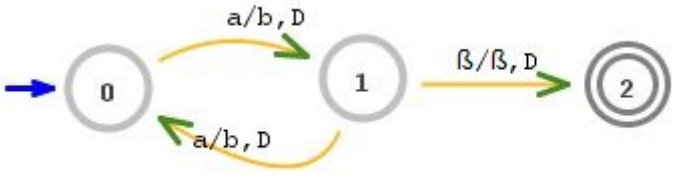

Figura 2.18: Representação gráfica de uma Máquina de Turing

2.4.3 Critérios de Parada

Na MT padrão a aceitação é dada quando a computação encontra-se em um estado final e não possui mais transições referentes ao estado atual da fita e o símbolo lido. Uma entrada será rejeitada em todos os outros casos.

Capítulo 3

Tecnologias Utilizadas

Neste capítulo serão apresentados os recursos que foram utilizados para o desenvolvimento do projeto.

3.1 Prolog

O primeiro passo no desenvolvimento desse projeto foi a troca do interpretador Prolog, visto que o que era usado estava obsoleto.

Anteriormente o sistema utilizava o Amzi Prolog para suporte dos códigos em Prolog e usava a própria máquina para fazer as comunicações entre o Java e o Prolog, a qual deveria ser substituída por uma nova máquina com as mesmas características fornecidas pelo Amzi e também permitisse a comunicação com o Java.

Foi realizada a pesquisa então, de um Prolog que tivesse essas características e a busca foi refinada em três implementações do Prolog: SWI Prolog, tuProlog e o Jlog.

A ferramenta tuProlog (DENTI; OMICINI; CALEGARI, 2013) é desenvolvida pelo laboratório de pesquisa APICE, o mesmo foi atrativo a primeira vista por possuir uma interface de comunicação com o Java, porém possuía pouca organização quanto a sua documentação, tanto do Prolog quanto da interface com o Java e acabaria se tornando difícil trabalhar com o mesmo.

A ferramenta Jlog (HOLST; MACKWORTH, 2005) chamou atenção também pelo fato de possuir a ligação entre o Prolog e o Java, porém o sistema não possui suporte pela empresa que o distribui desde 2007, além de várias partes da documentação já não estarem disponíveis.

A que mais se destacou entre as três foi a SWI Prolog (WIELEMAKER et al., 2012), que possui uma sintaxe muito parecida com a do Amzi e também a biblioteca JPL para comunicação

entre o Java e o Prolog, portanto foi a escolhida para o projeto.

3.2 SWI Prolog

O SWI-Prolog é uma implementação gratuita da linguagem Prolog criada pelo professor e cientista da computação Jan Wielemaker. Essa implementação é bastante usada no ambiente de ensino pela intuitividade da implementação e facilidade no uso.

A mesma está em contínua implementação desde 1987 e ainda hoje recebe atualizações em sua implementação. O SWI-Prolog é compatível com as plataformas Unix, Windows, Macintosh e Linux. Possui uma ampla gama de funcionalidades implementadas, como suporte a múltiplas tarefas, suporte a GUI e principalmente para esse trabalho, uma interface de comunicação com a linguagem Java, a biblioteca JPL que será descrita na seção 3.4.

3.3 Implementação Prolog

Com a troca do Prolog foi necessário rever e atualizar as implementações em Prolog para o SWI-Prolog para que fosse possível continuar a implementação do módulo de Simuladores. No apêndice A temos a implementação dos três simuladores do PIRAMIDE, o simulador de Autômato Finito, o simulador de Autômato de Pilha e o simulador de Máquina de Turing.

3.4 JPL

A JPL é um conjunto de classes Java e funções em C que fornecem uma interface entre o Java e o Prolog. A JPL não é uma implementação pura do Prolog para o Java, ela pode ser usada em diversas outras plataformas suportadas pelo Prolog.

A JPL é desenvolvida em duas camadas. Uma interface de mais baixo nível com o Prolog, a FLI (*Foreign Language Interface*) e outra de mais alto nível com o Java, para programadores Java que não querem se preocupar com os detalhes do Prolog FLI. É utilizando as bibliotecas desse nível mais alto que a comunicação entre o Java e o Prolog ocorre.

Para utilização da JPL no Windows foi necessário um grande estudo de configuração, visto que a mesma não foi muito trivia. Os detalhes desse processo de configuração estão descritos no apêndice B.

A implementação dessas configurações de comunicação foram trabalhadas e foi identificada uma grande dificuldade para realizar a mesma, visto que as informações da documentação da biblioteca, não aparecem bem documentadas gerando essa dificuldade de compreensão da forma do uso da mesma.

3.5 JAVA

Utilizando a JPL, foi possível trabalhar a comunicação com o Java. Essa comunicação é necessária pois é no Java que está implementada a interface com o usuário e onde o mesmo realiza as entradas dos dados que serão utilizados nos simuladores. Portanto é na interface em Java que as informações do Autômato a ser simulado e as entradas a serem testadas serão informadas pelo usuário e repassados ao simulador.

A versão que está sendo utilizada do Java é o Java 8 e para construção das interfaces está sendo utilizado o conjunto de ferramentas de construção de GUI AWT (*Abstract Window Toolkit*).

3.6 Funcionamento geral do sistema de Simuladores

Na Figura 3.1 temos uma tela do PIRAMIDE com a interface do Simulador de Autômato Finito aberta.

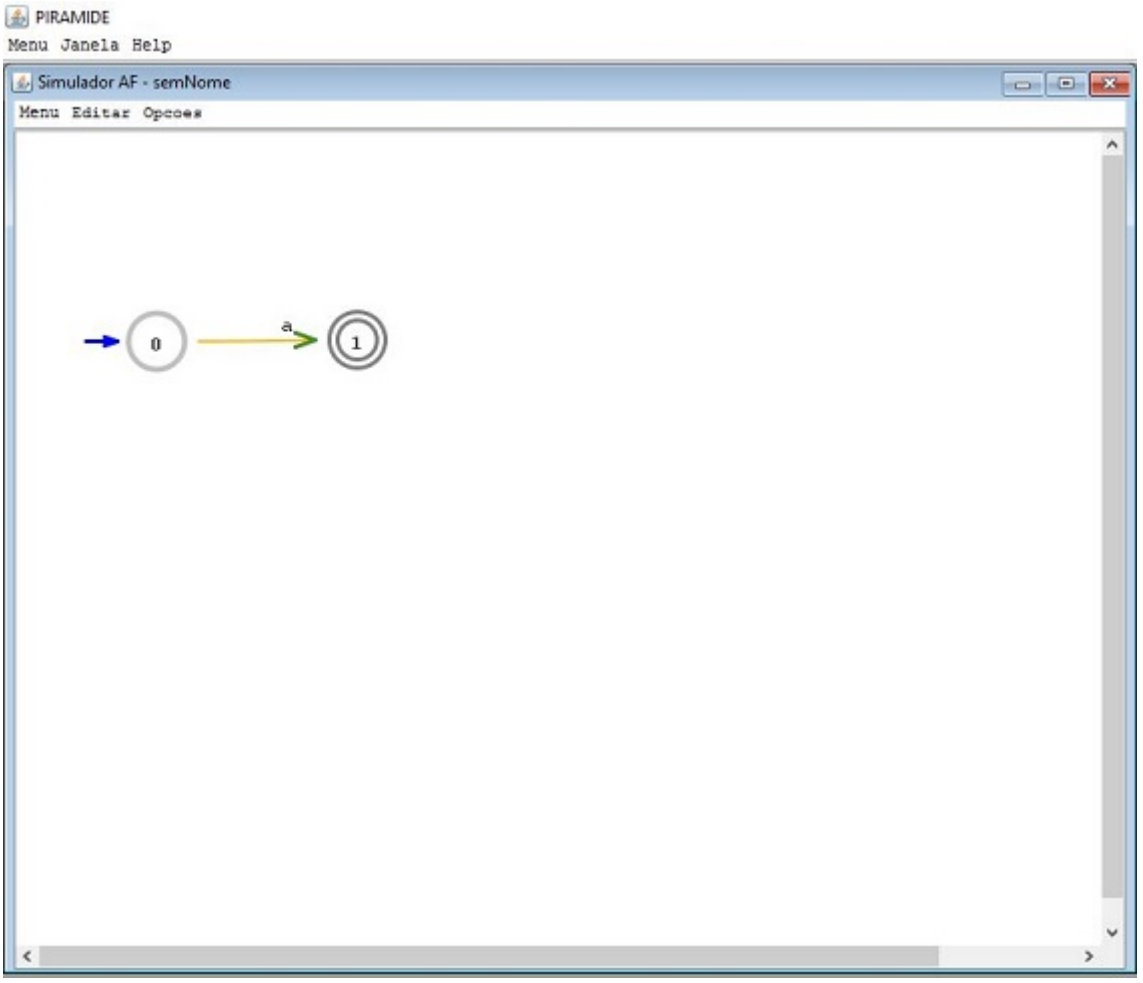

Figura 3.1: Tela do Simulador de Autômato Finito do PIRAMIDE

Para melhor compreensão do atual funcionamento do sistema de Simuladores o fluxograma da Figura 3.2 resume a execução geral das três máquinas.

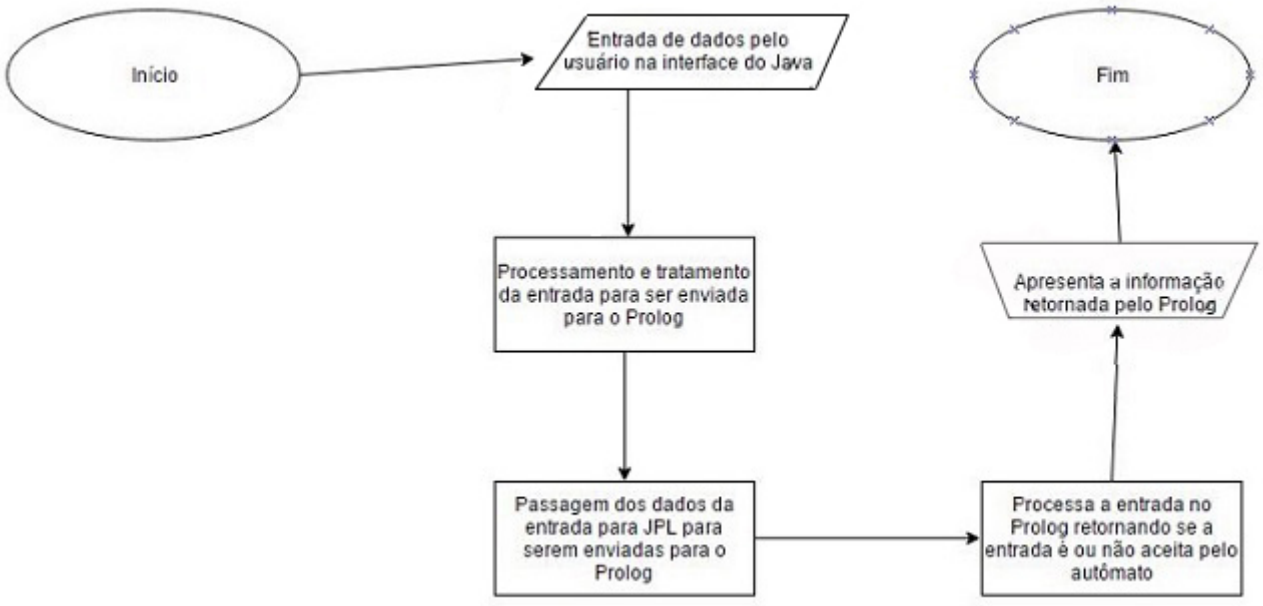

Figura 3.2: Fluxograma de execução do sistema de Simuladores

Esse fluxograma ocorre da seguinte forma:

- Inicialmente o usuário deve fazer a entrada do autômato que deseja simular, desenhando um diagrama de transições em tela. Após isso ele pode simular entradas nesse autômato. O usuário faz a entrada da string a ser simulada.
- As transições e estados são processados e a string de entrada é tratada para ser analisada pelo algoritmo Prolog.
- Depois disso é passado para a JPL os dados do autômato e da string e eles são enviados para a máquina do Prolog para ser analisada.
- Dentro do Prolog os dados são então processados e analisados para retornar a resposta se a string é reconhecida ou não pelo autômato.
- Por fim o resultado volta pela JPL para o Java e é apresentado para o usuário.

Capítulo 4

Desenvolvimento

Nesse capítulo será apresentado como foram desenvolvidas as implementações dos simuladores e das funcionalidades do simulador de Autômatos Finitos.

4.1 Implementação

4.1.1 Simuladores

De maneira geral os três simuladores (de Autômatos Finitos, de Autômatos de Pilha e de Máquina de Turing) foram implementados de forma muito semelhante possuindo como maior diferença a implementação dos mesmos na linguagem Prolog, implementação essa que pode ser vista no Apêndice A.

A parte da codificação que faz a ligação dos algoritmos em Prolog com a interface do usuário, implementada na linguagem Java, foi feita utilizando a JPL e alguns recursos de gerenciamento de arquivos da própria linguagem Java para realizar essa comunicação.

O primeiro problema encontrado foi a necessidade de realizar o *assert*[1] das informações dos autômatos para o Prolog. Para isso foram utilizados os recursos de manipulações de arquivos do Java, onde as informações são escritas no corpo do arquivo do código Prolog para que a computação da string de entrada seja realizada utilizando a JPL e após essas informações serem utilizadas o arquivo do código é refeito retirando as informações que foram adicionadas.

Essa solução foi utilizada pois não foi encontrada uma maneira de realizar o *assert* das informações utilizando exclusivamente a JPL.

Depois de resolver o problema do *assert* de informações era necessário então fazer o envio

[1] Método que adiciona um novo fato ou cláusula à base de dados do Prolog.

para o Prolog da string que seria testada e conseguir a resposta do mesmo. Para as aceitações foi utilizado a classe *Query* da JPL que faz todo o processo de envio e recepção da resposta, porém os códigos Prolog quando não aceitam a string de entrada entram em *loop*, sendo essa a forma de marcação que a string foi rejeitada, então será necessário identificar esses casos.

Para isso, utilizamos a classe *Threads* do Java, para controlar o tempo que o método leva para reconhecer a string, e caso esse tempo exceda um limite a string é identificada como rejeitada.

De forma mais específica é criada uma *thread* paralela, que chamaremos de *thread* A, em relação a *thread* principal do programa, que chamaremos de *thread* B. Na *thread* A será feito o envio da string para o Prolog para ser processada, enquanto na *thread* B temos um contador que ficará monitorando o tempo que a *thread* A está levando para processar. Quando o contador na *thread* B atingir um valor limite, o mesmo envia um sinal de interrupção para a *thread* A, finalizando a mesma. Caso seja finalizada antes de completar a execução, a *thread* A não é capaz de alterar o valor da *flag* que marca se a string é ou não aceita, sendo assim a ela é dada como rejeitada. Na Figura 4.1 temos a representação do funcionamento do reconhecimento de strings, tanto para aceitação quanto para rejeição.

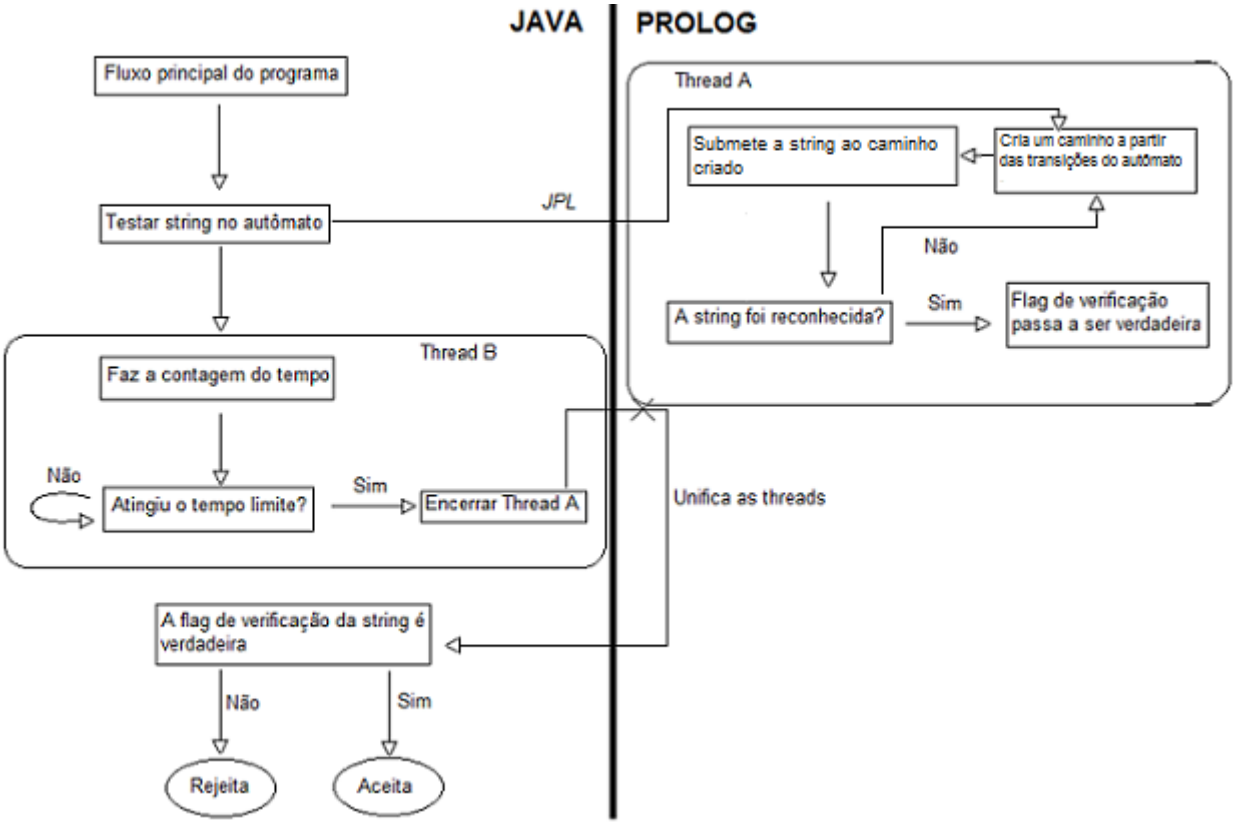

Figura 4.1: Funcionamento do simulador de strings utilizando *Threads*

O valor limite utilizado para forçar a parada do Prolog, foi o valor de máxima representação de um inteiro, que foi testado empiricamente. Utilizamos dessa forma, pois, estimar o valor limite ótimo depende de vários fatores como a quantidade de estados do autômato, o tamanho da string, a quantidade de transições, entre outros.

Outra parte da implementação foram os dois componentes do Simulador de AFs, que são apresentados nas seções 4.1.2 e 4.1.3.

4.1.2 Transformação de AFN para AFD

Para a Transformação de Autômato Não-Determinístico para Determinístico dividiu-se o algoritmo em três métodos separados, um para cada passo do processo, ou seja, Passo M1, Passo M2 e Passo M3. O algoritmo utilizado como base é o que pode ser visto na seção 2.2.9, sendo que os mesmos foram adaptados para melhor atender os requisitos do projeto

Para o Passo M1 foi feito a retirada das transições sobre string. De forma geral, quando o algoritmo identificava uma transição sobre mais de um símbolo ele fazia a separação da transição

em novas transições, uma para cada símbolo da string.

Para o Passo M2 foi feita a retirada das transições sobre o símbolo vazio, que no sistema é representado pelo símbolo `[]`. Ao ser identificada a transição sobre esse símbolo é feito, portanto, a retirada ou adaptação, dependendo do caso, da transição.

No Passo M3 são retiradas as transições sobre mesmo símbolo partindo do mesmo estado. Para essa etapa foi utilizado o método da tabela, que foi construída utilizando um vetor de vetores para cada estado, sendo feito primeiro, a criação da tabela e preenchimento da mesma. Após a tabela ser criada, a mesma é varrida e são eliminados os estados não acalçáveis. Por fim a tabela é processada pelo algoritmo e o novo autômato é montado com os novos estados e as novas transições já adaptadas.

A implementação do algoritmo está descrita em forma de pseudocódigo no Apêndice D para melhor entendimento.

4.1.3 Minimização de AF

Esse processo foi implementado todo no mesmo método, porém dividido em quatro etapas. A implementação foi baseada no algoritmo de minimização utilizando tabela.

Antes do começo do processo é feito a verificação se o autômato é determinístico e se é ou não completo. Caso não seja determinístico, uma janela aparece para o usuário, informando que é necessário fazer o processo de transformação primeiramente e como realizá-lo no sistema. Para o caso de não ser completo cria-se o estado `d` que irá receber as transições sobre os símbolos que não existem no autômato.

A primeira etapa foi a construção da tabela e a marcação dos estados triviais na mesma. A tabela, assim como no algoritmo de transformação, foi feita utilizando um vetor de vetores. A segunda etapa foi o processamento das células da tabela, que representavam a relação entre cada estado, para marcar os estado que não eram trivialmente equivalentes. Após isso, com a tabela construída e preenchida foi realizada a etapa de unificação dos estados equivalentes (não marcados na tabela). Por fim são excluídos os estados inúteis do autômato (que não atingem um estado final). Também é nessa última etapa que o estado `d` é retirado, se foi criado.

A implementação do algoritmo está descrita em forma de pseudocódigo no Apêndice D para melhor entendimento.

4.2 Testes

4.2.1 Descrição do experimento

Para realizar os testes utilizamos como base o Teste de Funcionalidade (NGUYEN; KANER; FALK, 1999), onde são testadas as funcionalidades que o sistema possui, verificando se elas realizam o que se pretendia que fosse feito. É importante enfatizar que o teste não buscava problemas de usabilidade ou testar a otimização da execução das tarefas, mas sim se as tarefas eram realizadas de forma correta. Como o teste foi feito com usuários, por consequência foi possível detectar alguns desses problemas, porém esse não era o foco do teste.

De forma concreta o teste foi realizado fazendo um experimento com a turma de alunos de Teoria da Computação do ano de 2016. Foram escolhidos esses usuários, pois eles são os usuários alvo do sistema.

Da turma toda 28 alunos se voluntariaram para realizar o experimento. O grupo foi dividido em quatro grupos de oito alunos cada um para realizar o experimento. Cada grupo realizou o experimento em um horário diferente, porém utilizando as mesmas máquinas e a mesma versão do sistema.

Foi entregue aos alunos então, um conjunto de testes que eles deveriam realizar no sistema, visando testar os simuladores e os componentes do simulador de autômatos finitos. Esse conjunto de testes pode ser consultado no Apêndice E. Antes de começar o experimento foi demonstrado rapidamente a tela do software e onde eles poderiam encontrar os simuladores que deveriam testar.

Foi feita uma divisão em três passos para o teste. Na primeira parte eles deveriam testar o simulador de AFs e os componentes, sendo dado um limite de 20 minutos de tempo para essa parte. As outras duas partes se referiam aos testes com o simulador de APs e MTs sendo dado 10 minutos para cada parte.

Para documentar o experimento foi realizado a gravação da tela dos alunos com todas as ações que eles realizaram durante o experimento e esse foi material base para a análise do experimento.

4.2.2 Resultados dos testes

Após a realização do experimento analisou-se todos os vídeos coletados, observando as funcionalidades de cada função do sistema.

Para o simulador de Autômatos Finitos não foi registrado nenhum tipo de erro, podendo então validar o funcionamento do mesmo. O mesmo ocorreu para o simulador de Autômatos de Pilha, funcionando corretamente para todos os testes em que foi submetido.

No simulador de Máquina de Turing foi registrado um problema quanto a transição inicial $\beta/\beta, D$, para encaixar o cabeçote. Quando se utilizava essa transição no autômato era necessário acrescentar o β no início da string a ser testada para que a mesma funcionasse. Para corrigir esse problema foi concatenado o símbolo no início de toda string que vai ser avaliada pelo sistema.

Quanto aos componentes do simulador de AFs, no caso do minimizador ele funcionou corretamente para todos os casos em que foi testado pelos alunos não apresentando nenhum tipo de erro.

Para a transformação de AFN para AFD os Passos M1 e M3 funcionaram corretamente para todos os casos de testes a que foram submetidos, porém o passo M2 apresentou um erro: quando a transição sobre o vazio é a partir do estado inicial, ao executar o passo a transformação ocorre, porém o estado inicial perde sua propriedade tornando-se apenas um estado comum.

Além desses problemas de funcionalidades, os alunos também identificaram vários problemas de interface.

Vários deles tiveram problemas para identificar a ferramenta que lhes permitiria editar as transições e estados, assim como modificar os estados para inicial e final. Tiveram problemas também para compreender como se declarava uma transição em todos os três simuladores. Também reportaram dificuldades na área de edição que se movimenta quando o mouse se aproxima da borda da tela, explicando que a movimentação é realizada de forma muito brusca.

Tentando solucionar os problemas de intuitividade do sistema, foi descrito no manual do sistema como realizar cada funcionalidade do sistema de forma detalhada. O manual pode ser consultado no Apêndice C. Quanto ao problema dos movimentos da tela, os valores que definiam a quantidade que a janela se movimenta quando o mouse toca a borda da janela, foram alterados para valores menores, fazendo com que a ação ficasse menos agressiva.

Capítulo 5

Considerações Finais e Trabalhos Futuros

Após as implementações e testes realizados, podemos concluir que os simuladores estão todos funcionais, realizando todas as atividades que devem realizar. Para os componentes do simulador de AF, temos a minimização funcionando totalmente, porém na transformação de AFN para AFD temos os passos M1 e M3 funcionando e apenas o passo M2 apresentando alguns problemas quanto ao seu funcionamento.

Portanto, com os simuladores funcionais, o outro grande módulo do sistema pode ser implementado, os Resolvedores. Para implementação desse módulo era necessário que os Simuladores estivessem funcionais e validados e nesse momento temos todos funcionando, deixando a possibilidade para que, em um trabalho futuro, seja implementado o módulo de Resolvedores para as três máquinas.

Além disso também fica proposto para próximos trabalhos a melhoria da interface do sistema, torná-la mais intuitiva e responsiva, visto que, durante o experimento realizado, tivemos indicações de problemas dos usuários do sistema.

Outra proposta seria a de tentar realizar a implementação exclusivamente em Java, utilizando uma abordagem para o autômato de grafo e realizando buscas em profundidade ou largura no mesmo para resolver a questão da simulação das strings nos autômatos.

Também se propõe, em um âmbito mais técnico, a melhoria do código do sistema, refatorando-o, adicionando uma arquitetura ao mesmo, atualizando as funções que estão sendo utilizadas no sistema, para versões mais atuais, oferecidas pelas novas versões do Java.

Outra proposta de trabalho futuro seria incorporar ao sistema do simulador de Máquina de Turing a MKT, que é a Máquina de Turing com suporte a múltiplas fitas, que atualmente não está implementada no sistema.

Mais uma proposta é a de estudar como configurar a JPL para outros sistemas operacionais, pois nesse trabalho apenas apresentamos a configuração para o Windows, sendo que é necessário investigar como realizar o mesmo em outras plataformas.

Por fim outra proposta seria corrigir o problema da forma como os autômatos são organizados na tela, após serem transformados em AFDs e/ou minimizados, tornando-os mais legíveis. Esse é um problema de otimização de layout e poderia ser até mesmo um novo trabalho.

Apêndice A

Implementações dos Simuladores em Prolog

Neste apêndice, mostraremos os códigos de cada uma das implementações dos simuladores.

Algoritmo 4 : Implementação em Prolog do simulador de Autômato Finito

```
/*final(X). deve ser assertado*/

/*trans(X,Y,Z). deve ser assertado*/

/*trans(s2,s4).deve ser assertado- transicao sobre vazio*/

/*inicial(X).deve ser assertado*/

aceita(L ,E):- inicial(X), aceita(X,L,E).

aceita(S,[],E):- E is S, final(S).

aceita(S,[X|R],E):- trans(S,X,S1),aceita(S1,R,E).

aceita(S,L,E):- trans(S,S1),aceita(S1,L,E).
```

Algoritmo 5 : Implementação em Prolog do simulador de Autômato de Pilha

```
processa([],Q0,[],[])  :- final(Q0),!.

processa([H|T],Q0,P,A) :-d(Q0,H,'[]','[]','[]','[]',Qx),
processa(T,Qx,P,A).

processa([H|T],Q0,P,A) :- d(Q0,H,B,'[]','[]','[]',Qx),
B\='[]', pop(P,B,P1), processa(T,Qx,P1,A).

processa([H|T],Q0,P,A) :- d(Q0,H,'[]',C,'[]','[]',Qx),
C\='[]', push(C,P,P1), processa(T,Qx,P1,A).

processa([H|T],Q0,P,A) :- d(Q0,H,B,C,'[]','[]',Qx), B\='[]'
, pop(P,B,P1), C\='[]', push(C,P1,P2), processa(T,Qx,P2,A).

processa([H|T],Q0,P,A) :- d(Q0,H,'[]','[]',B1,'[]',Qx),
B1\='[]', pop(A,B1,A1), processa(T,Qx,P,A1).

processa([H|T],Q0,P,A) :- d(Q0,H,B,'[]',B1,'[]',Qx),
B\='[]', pop(P,B,P1), B1\='[]', pop(A,B1,A1), processa(T,Qx,P1,A1).

processa([H|T],Q0,P,A) :- d(Q0,H,'[]',C,B1,'[]',Qx), C\='[]',
push(C,P,P1), B1\='[]', pop(A,B1,A1), processa(T,Qx,P1,A1).

processa([H|T],Q0,P,A) :- d(Q0,H,B,C,B1,'[]',Qx), B\='[]',
pop(P,B,P1), C\='[]', push(C,P1,P2),
B1\='[]', pop(A,B1,A1), processa(T,Qx,P2,A1).

processa([H|T],Q0,P,A) :- d(Q0,H,'[]','[]','[]',C1,Qx),
C1\='[]', push(C1,A,A1), processa(T,Qx,P,A1).

processa([H|T],Q0,P,A) :- d(Q0,H,B,'[]','[]',C1,Qx),
B\='[]', pop(P,B,P1), C1\='[]', push(C1,A,A1), processa(T,Qx,P1,A1).
```

Algoritmo 6 : Implementação em Prolog do simulador de Autômato de Pilha - continuação

```
processa([H|T],Q0,P,A) :- d(Q0,H,'[]',C,'[]',C1,Qx),
C\='[]', push(C,P,P1), C1\='[]', push(C1,A,A1), processa(T,Qx,P1,A1).

processa([H|T],Q0,P,A) :- d(Q0,H,B,C,'[]',C1,Qx),
B\='[]', pop(P,B,P1), C\='[]', push(C,P1,P2),
C1\='[]', push(C1,A,A1), processa(T,Qx,P2,A1).

processa([H|T],Q0,P,A) :-d(Q0,H,'[]','[]',B1,C1,Qx),
B1\='[]', pop(A,B1,A1), C1\='[]', push(C1,A1,A2), processa(T,Qx,P,A2).

processa([H|T],Q0,P,A) :- d(Q0,H,B,'[]',B1,C1,Qx), B\='[]',
pop(P,B,P1), B1\='[]', pop(A,B1,A1), C1\='[]',
push(C1,A1,A2),  processa(T,Qx,P1,A2).

processa([H|T],Q0,P,A, Delay, Ti) :- d(Q0,H,'[]',C,B1,C1,Qx),
C\='[]', push(C,P,P1), B1\='[]', pop(A,B1,A1), C1\='[]',
push(C1,A1,A2), processa(T,Qx,P1,A2).

processa([H|T],Q0,P,A) :- d(Q0,H,B,C,B1,C1,Qx), B\='[]',
pop(P,B,P1), C\='[]', push(C,P1,P2), B1\='[]',
pop(A,B1,A1), C1\='[]', push(C1,A1,A2),
processa(T,Qx,P2,A2).

processa(T,Q0,P,A) :-d(Q0,H,'[]','[]','[]','[]',Qx),
H='[]', processa(T,Qx,P,A).

processa(T,Q0,P,A) :- d(Q0,H,B,'[]','[]','[]',Qx),
H='[]', B\='[]', pop(P,B,P1), processa(T,Qx,P1,A).

processa(T,Q0,P,A) :- d(Q0,H,'[]',C,'[]','[]',Qx),
H='[]', C\='[]', push(C,P,P1), processa(T,Qx,P1,A).
```

Algoritmo 7 : Implementação em Prolog do simulador de Autômato de Pilha - continuação

```
processa(T,Q0,P,A) :- d(Q0,H,B,C,'[]','[]',Qx), H='[]',
B\='[]', pop(P,B,P1), C\='[]', push(C,P1,P2),
processa(T,Qx,P2,A).

processa(T,Q0,P,A) :- d(Q0,H,'[]','[]',B1,'[]',Qx),
H='[]', B1\='[]', pop(A,B1,A1), processa(T,Qx,P,A1).

processa(T,Q0,P,A) :- d(Q0,H,B,'[]',B1,'[]',Qx),
H='[]', B\='[]', pop(P,B,P1), B1\='[]',
pop(A,B1,A1), processa(T,Qx,P1,A1).

processa(T,Q0,P,A) :- d(Q0,H,'[]',C,B1,'[]',Qx),
H='[]', C\='[]', push(C,P,P1), B1\='[]',
pop(A,B1,A1), processa(T,Qx,P1,A1).

processa(T,Q0,P,A) :- d(Q0,H,B,C,B1,'[]',Qx),
H='[]', B\='[]', pop(P,B,P1), C\='[]',
push(C,P1,P2), B1\='[]', pop(A,B1,A1),
processa(T,Qx,P2,A1).

processa(T,Q0,P,A) :- d(Q0,H,'[]','[]','[]',C1,Qx),
H='[]', C1\='[]', push(C1,A,A1),
processa(T,Qx,P,A1).

processa(T,Q0,P,A) :- d(Q0,H,B,'[]','[]',C1,Qx),
H='[]', B\='[]', pop(P,B,P1), C1\='[]',
push(C1,A,A1), processa(T,Qx,P1,A1).

processa(T,Q0,P,A) :- d(Q0,H,'[]',C,'[]',C1,Qx),
H='[]', C\='[]', push(C,P,P1), C1\='[]',
push(C1,A,A1),  processa(T,Qx,P1,A1).
```

Algoritmo 8 : Implementação em Prolog do simulador de Autômato de Pilha - continuação

```
processa(T,Q0,P,A) :- d(Q0,H,B,C,'[]',C1,Qx),
H='[]', B\='[]', pop(P,B,P1), C\='[]',
push(C,P1,P2), C1\='[]', push(C1,A,A1),
processa(T,Qx,P2,A1).

processa(T,Q0,P,A) :- d(Q0,H,'[]','[]',B1,C1,Qx),
H='[]', B1\='[]', pop(A,B1,A1), C1\='[]',
push(C1,A1,A2), processa(T,Qx,P,A2).

processa(T,Q0,P,A) :- d(Q0,H,B,'[]',B1,C1,Qx), H='[]',
B\='[]', pop(P,B,P1), B1\='[]', pop(A,B1,A1),
C1\='[]', push(C1,A1,A2),  processa(T,Qx,P1,A2).

processa(H,Q0,P,A, Delay, Ti) :- d(Q0,H,'[]',C,B1,C1,Qx),
H='[]', C\='[]', push(C,P,P1), B1\='[]',
pop(A,B1,A1), C1\='[]', push(C1,A1,A2),
processa(T,Qx,P1,A2).

processa(T,Q0,P,A) :-d(Q0,H,B,C,B1,C1,Qx), H='[]',
B\='[]', pop(P,B,P1), C\='[]', push(C,P1,P2),
B1\='[]', pop(A,B1,A1), C1\='[]', push(C1,A1,A2),
processa(T,Qx,P2,A2).

push(E,[],[E]) :- !.

push(E,L,[E|L]) :- !.

pop([H|T],H,T) :- !.

aceita(L, Delay) :- inicial(X),
processa(L,X,[],[]).
```

Algoritmo 9 : Implementação em Prolog do simulador de Máquina de Turing

```
processa(C,F,Q0) :- d(Q0,SL,SE,'D',Q1),
push(SE,C,C1), pop(F,SL,F1),
processa(C1,F1,Q1).

processa(C,F,Q0, Delay, Ti) :- d(Q0,SL,SE,'D',Q1),
 push(SE,C,C1), pop(F,SL,[]), push('ß',[],F1),
 processa(C1,F1,Q1).

processa(C,F,Q0) :- d(Q0,SL,SE,'E',Q1), pop(F,SL,F1),
push(SE,F1,F2), pop(C,E,C1), push(E,F2,F3),
processa(C1,F3,Q1).

processa(C,[H|F],Q0) :- not d(Q0,H,{\_},{\_},{\_}),
final(Q0).

processa(C,[],Q0) :- final(Q0).

push(E,[],[E]) :- !.

push(E,L,[E|L]) :- !.

pop([H|T],H,T) :- !.

aceita(A) :- inicial(I), processa([],A,I).
```

Apêndice B

Configurações da JPL no Sistema Operacional Windows

A configuração da JPL no sistema operacional Windows não é tão trivial e, portanto, será descrito passo a passo como deve ser feito a configuração.

Primeiramente deve ser feito o download do SWI Prolog. A biblioteca da JPL já vem inclusa com a instalação do SWI Prolog e o .jar pode ser encontrado dentro da pasta lib dentro da pasta do SWI Prolog criada na instalação da mesma.

Após isso devem ser tratadas as variáveis de ambiente do sistema. As variáveis de ambiente podem ser acessadas a partir das configurações avançadas da máquina. A variável que deve ser editada é a Path. Primeiramente deve ser checado se o caminho do Java está configurado corretamente, como mostrado na Figura B.1. Usualmente esse caminho é configurado na instalação do Java, porém é interessante garantir que a mesma está corretamente configurada.

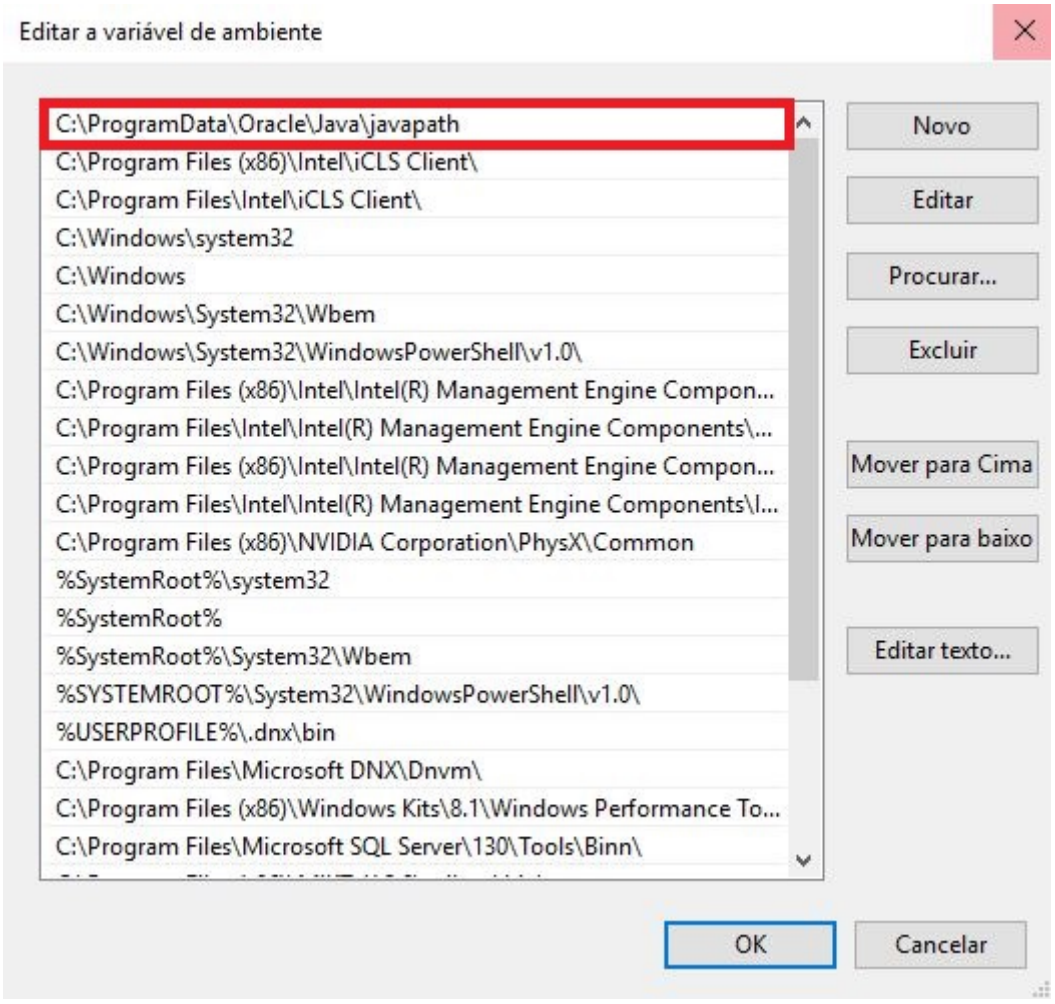

Figura B.1: Configuração do caminho do Java nas variáveis de ambiente

Posteriormente devem ser configurados os caminhos do SWI Prolog e da JPL. Os caminhos que devem estar setados estão destacados na Figura B.2.

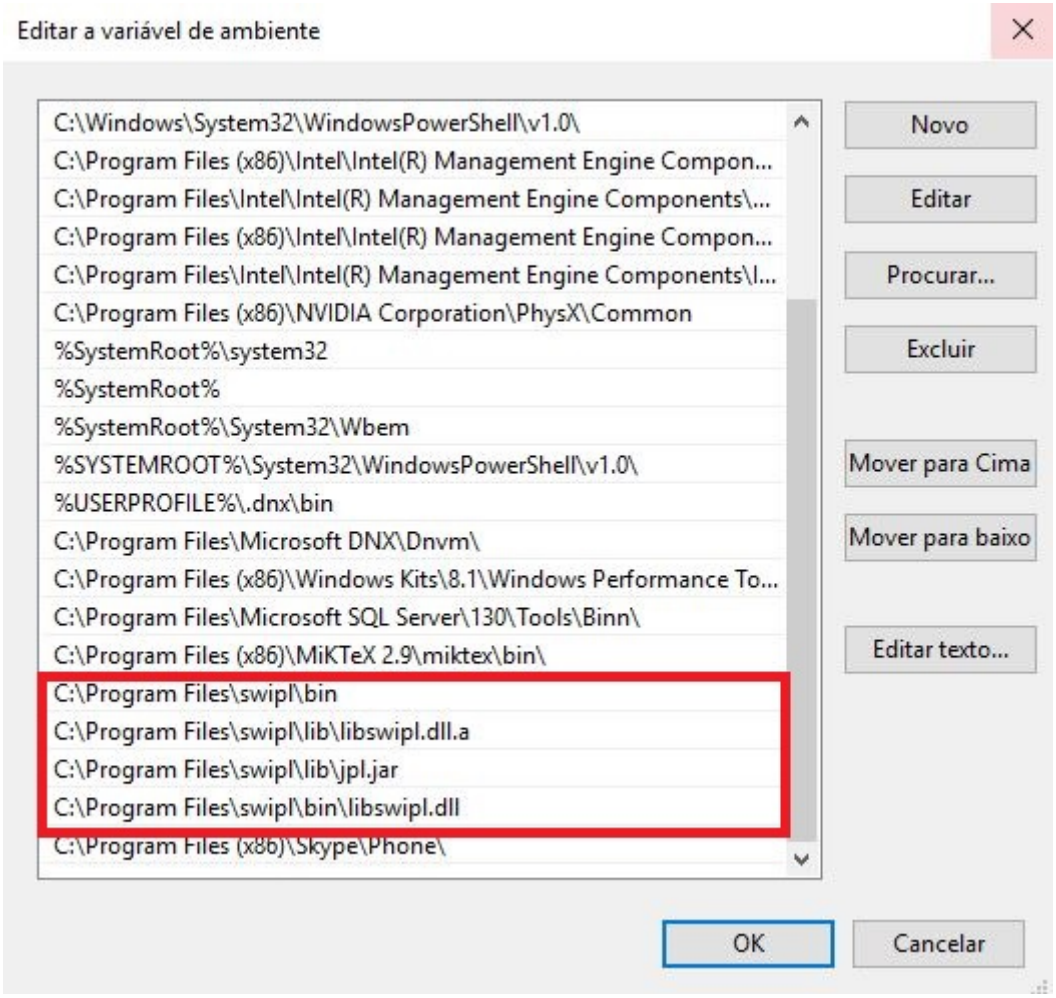

Figura B.2: Configuração do caminho do SWI Prolog e do caminho da JPL nas variáveis de ambiente

Com todas as variáveis de ambiente configuradas, deve-se então configurar o ambiente de desenvolvimento para compilar e executar utilizando a JPL. Para esse exemplo utilizamos a IDE NetBeans 8.1.

Primeiramente deve-se adicionar ao projeto o .jar citado anteriormente, como uma biblioteca externa. Após ser adicionado deve ser configurado a forma de compilação da IDE. No NetBeans 8.1 essa configuração pode ser modificada acessando o menu Executar -> Definir Configuração do Projeto -> Personalizar. Na tela que será aberta deve-se acessar a aba Executar e na caixa de texto “Opções de VM” deve ser colocado o código de configuração, assim como pode ser visto na Figura B.3.

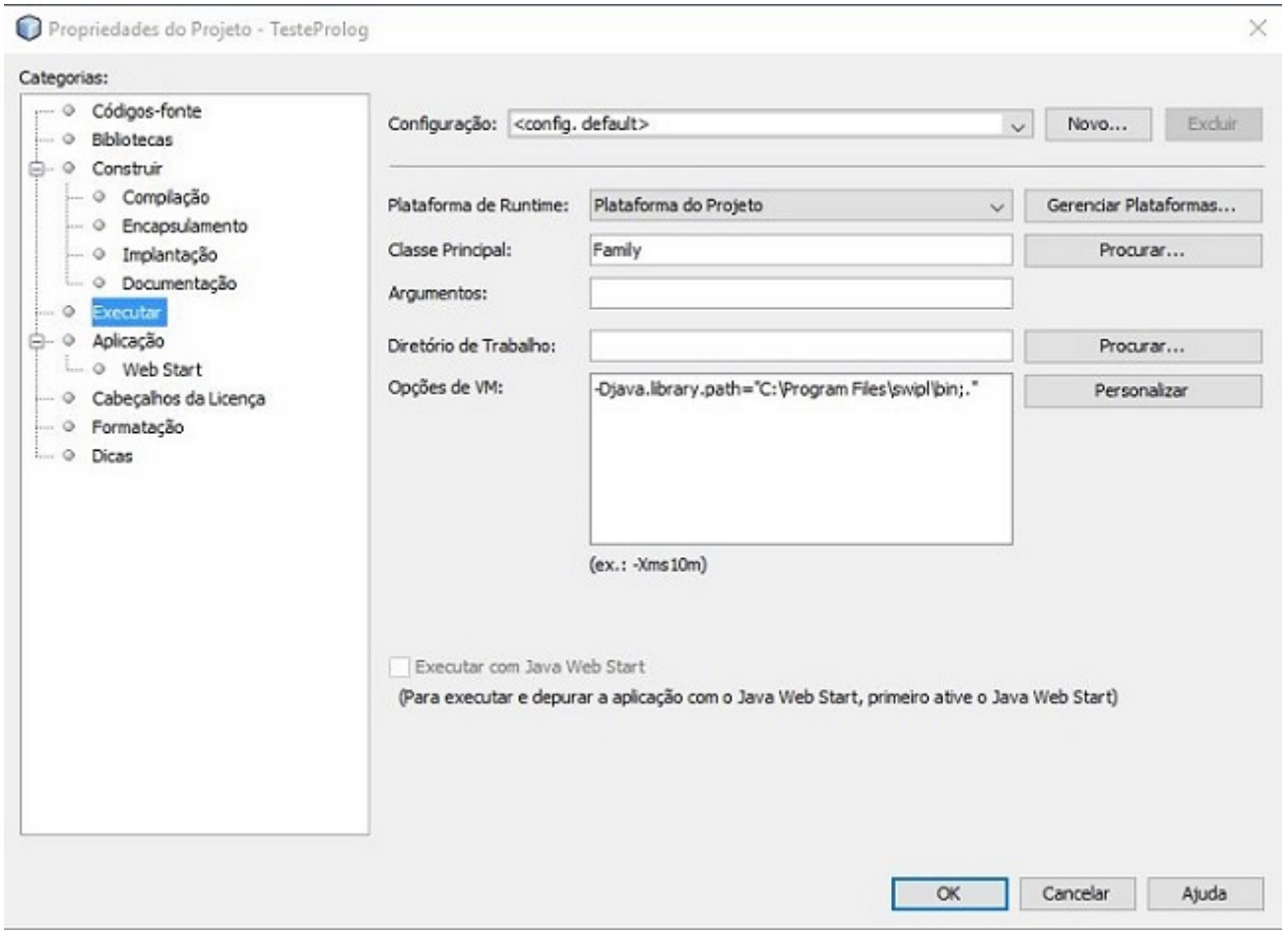

Figura B.3: Configuração de execução do NetBeans

Após isso pode ser necessário reiniciar a IDE para que ocorra o total funcionamento da JPL vinculada ao Java.

Apêndice C

Manual do Usuário do Sistema PIRAMIDE

Neste apêndice será apresentado o manual do usuário para utilização do Módulo de Simuladores do Sistema PIRAMIDE.

Ao executar o sistema a primeira tela será a apresentada na Figura C.1. Nessa tela temos três opções no menu superior onde podemos acessar `Menu`, `Janela` e `Help`. O menu `Help` apenas dará a opção de ver as informações do software PIRAMIDE e seus colaboradores. O menu `Janela` irá alternar entre todas as janelas abertas no simulador, sem encerrá-las, apenas colocando-as atrás das demais, conforme a opção é clicada. O menu pode ser visto na Figura C.2. E por fim temos a opção `Menu`, onde poderemos acessar as principais funcionalidades do PIRAMIDE.

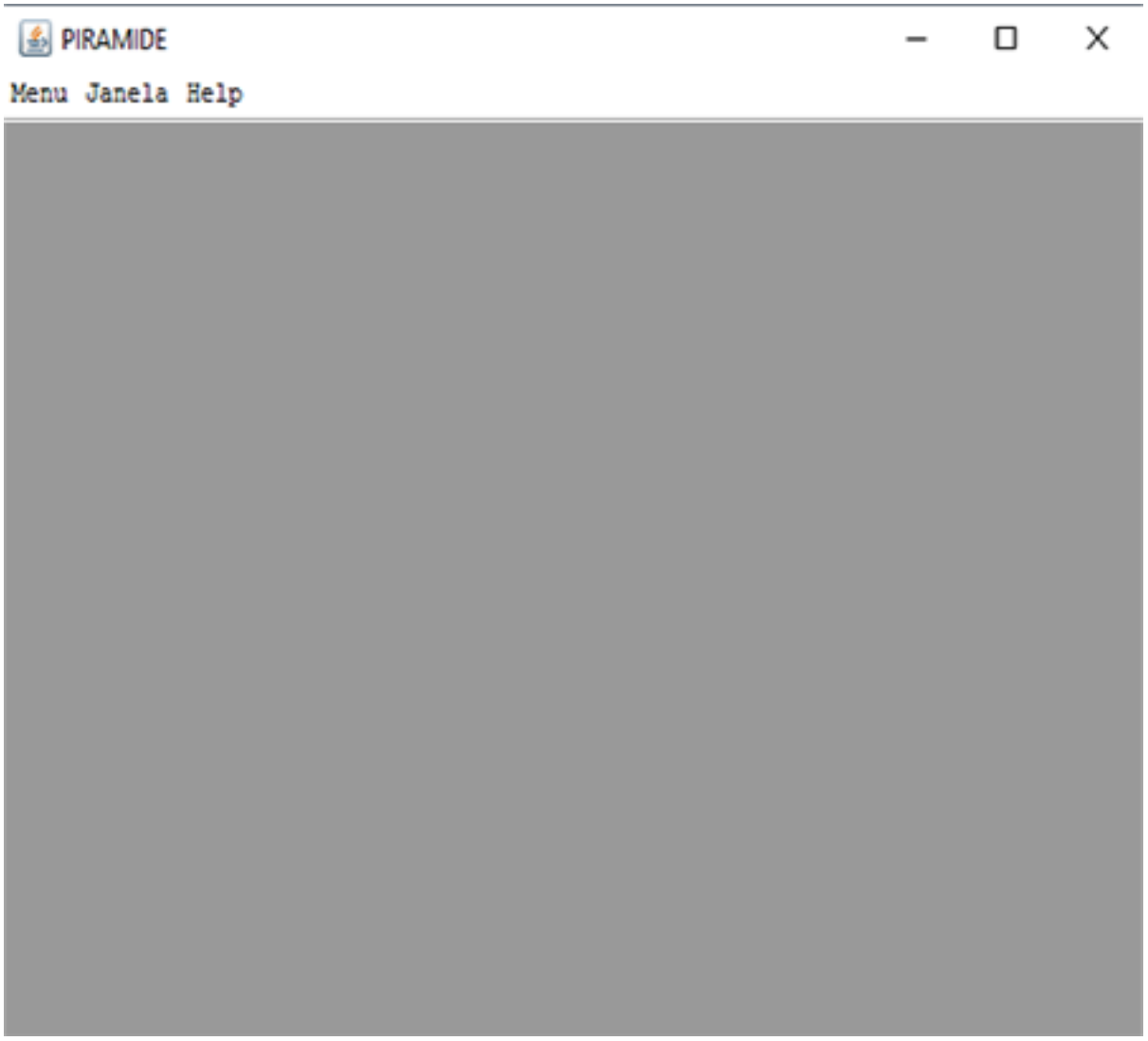

Figura C.1: Tela Inicial do Sistema PIRAMIDE

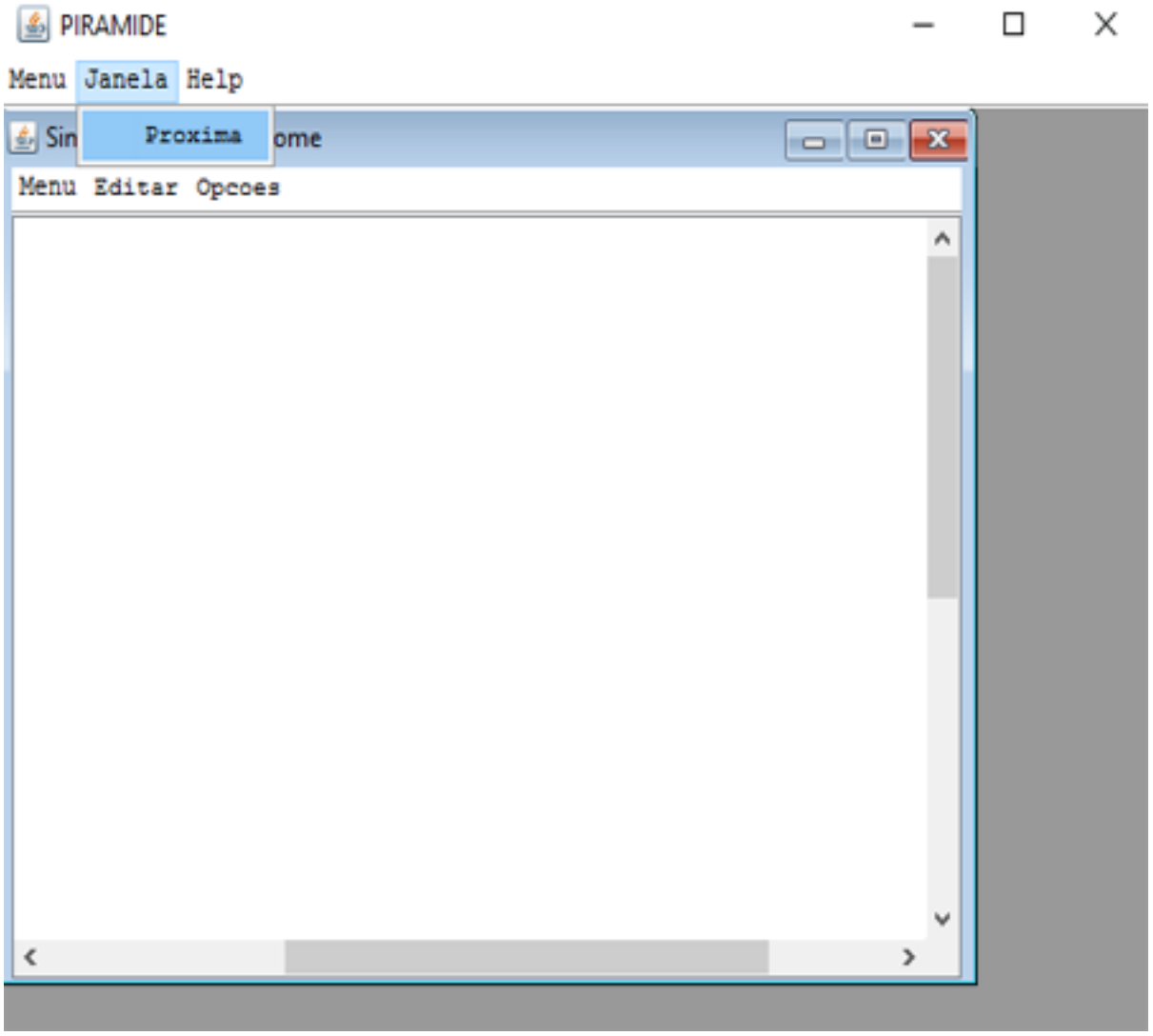

Figura C.2: Tela do sistema demonstrando o menu Janela

Ao clicar sobre a opção `Menu` você terá uma tela como a da Figura C.3 onde existem três opções, o menu de `Simuladores`, de `Resolvedores` e o botão `Sair`. Como o nome sugere o botão `Sair` encerra a execução do sistema. No menu `Resolvedores` você terá acesso aos Resolvedores do sistema, porém eles ainda não estão finalizados e completamente funcionais então não serão explicados nesse manual. Por fim temos o menu dos `Simuladores`, onde existem três opções, o `SimuladorAF`, o `SimuladorAP` e o `SimuladorMT`. Como os nomes sugerem cada uma dessas opções levará para o simulador que está descrito no nome, o de Autômato Finito ou o de Autômato de Pilha ou o de Máquina de Turing.

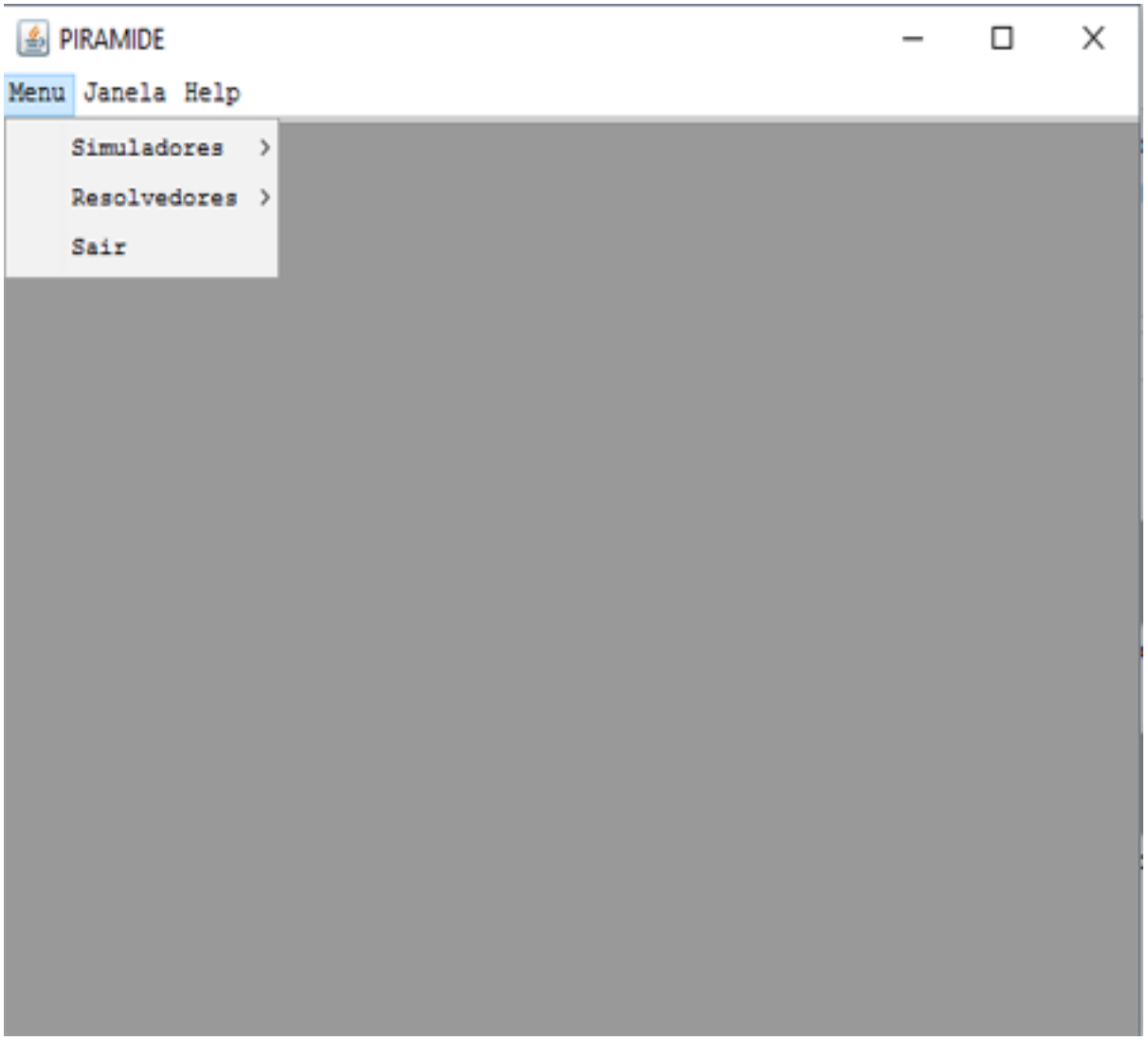

Figura C.3: Tela do Sistema PIRAMIDE com o Menu aberto

Nas Figuras C.4, C.5, C.6 temos as telas do Simuladores de AF, AP e MT abertas. Todas possuem o mesmo menu superior, possuindo as mesmas opções nos menus `Menu` e `Editar` e apenas o `SimuladorAF` possui diferença no menu `Opções`, portanto será explicada de modo único as funcionalidades que são iguais nos três simuladores e separadamente os detalhes de cada simulador.

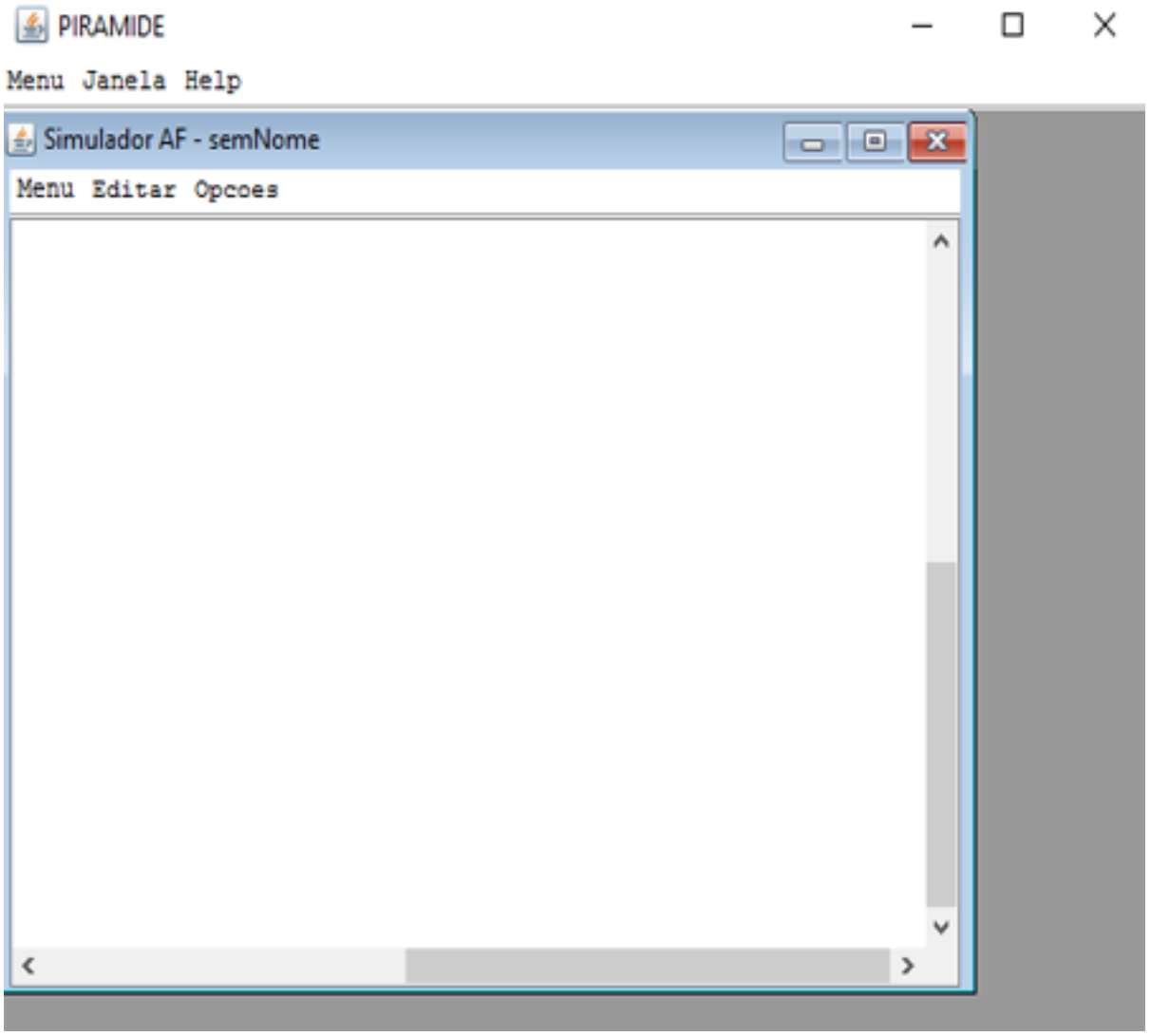

Figura C.4: Tela do Simulador de AF

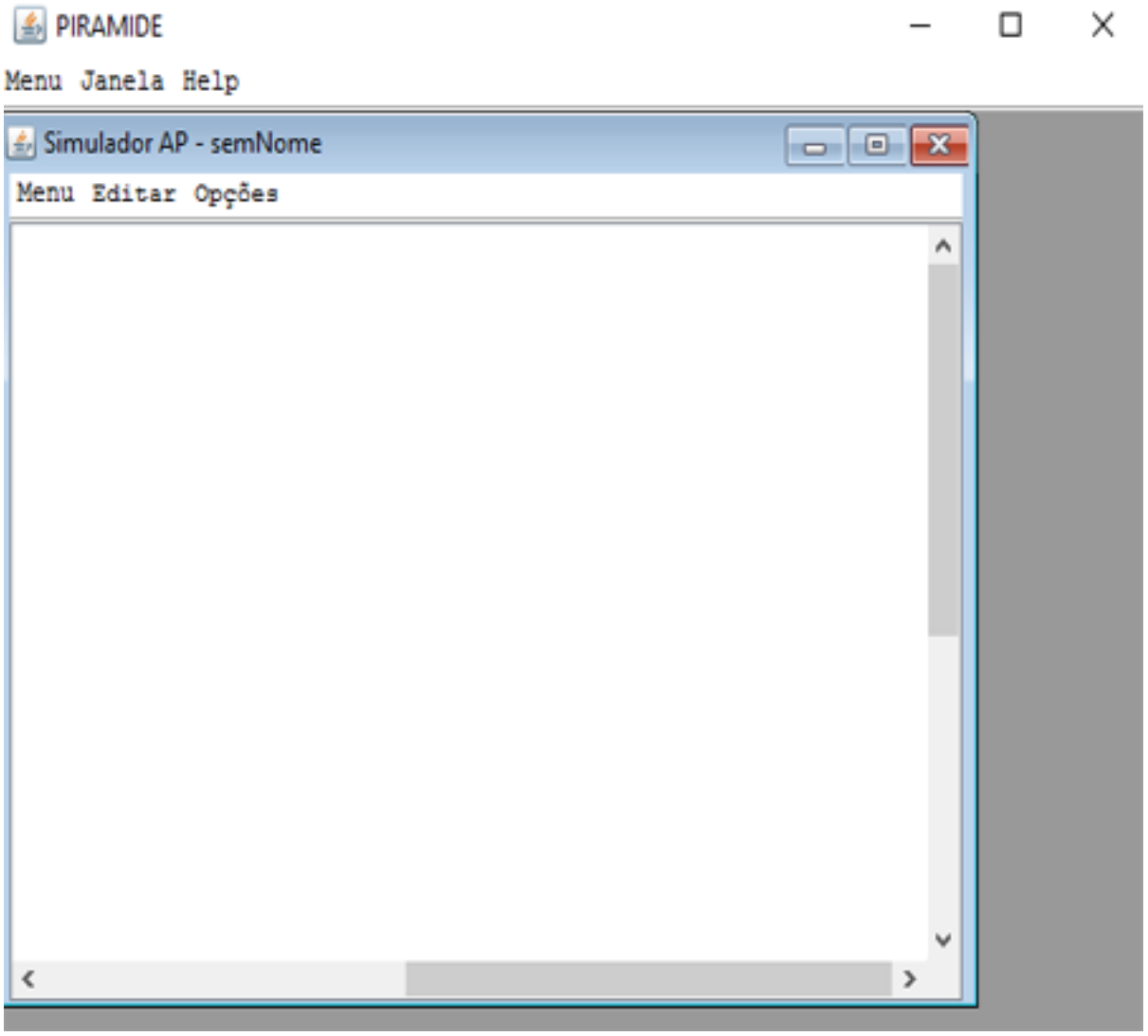

Figura C.5: Tela do Simulador de AP

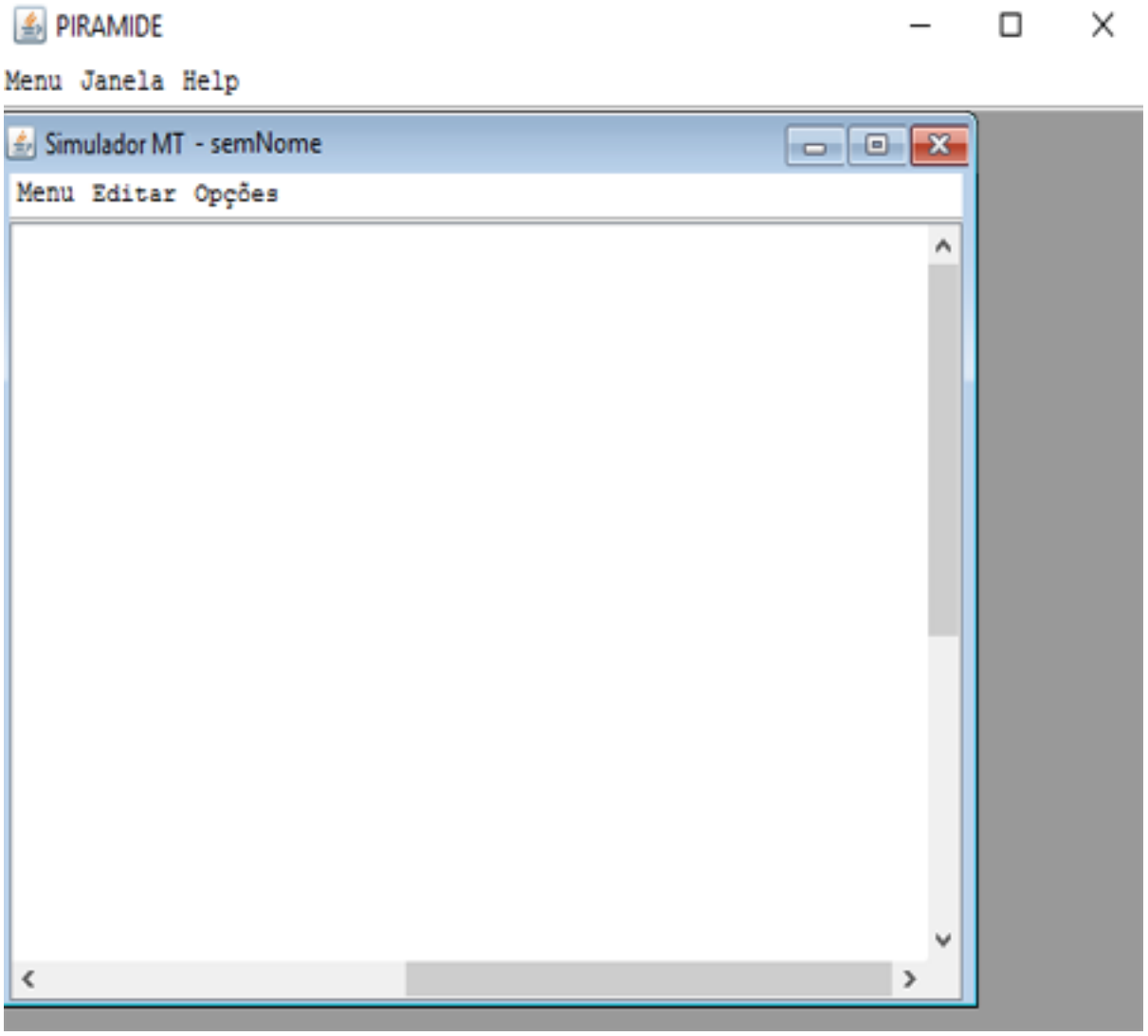

Figura C.6: Tela do Simulador de MT

Para explicação das funcionalidades que são iguais no três simuladores utilizaremos a tela do `SimuladorAF`, porém, reforçando, as funcionalidades que aqui serão explicadas são exatamente iguais nos três simuladores. Primeiramente na Figura C.7 temos o menu `Menu` aberto e todas as suas opções.

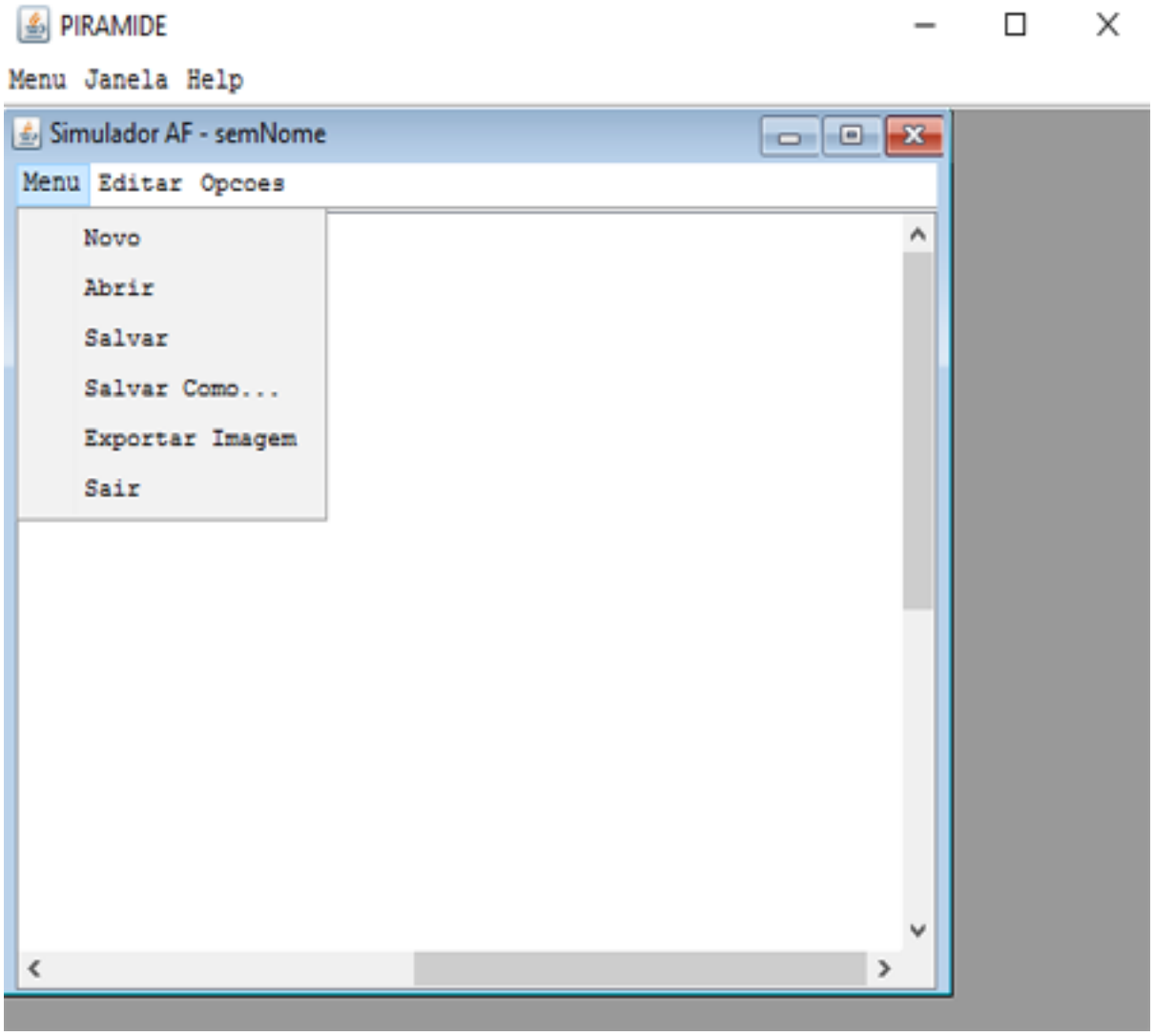

Figura C.7: Tela do Simulador com o menu Menu aberto

A primeira opção é o `Novo`, essa opção irá criar uma nova tela em branco para ser editada, caso exista algo na tela aparecerá uma janela questionando se as informações devem ser salvas.

A segunda opção é o `Abrir`. Nesse caso irá abrir uma janela de exploração para que você busque em sua máquina um arquivo já salvo para ser aberto na tela. Ao ser escolhido o arquivo, o mesmo irá ser aberto da mesma forma que estava quando foi salvo.

A seguir temos duas opções `Salvar` e `Salvar como`. Caso seja a primeira vez que você está salvando o seu autômato ambas funcionarão da mesma maneira, abrindo uma janela para que você escolha a pasta e o nome do arquivo para salvar. Porém caso você já tenha feito isso ao menos uma vez a opção `Salvar` irá apenas atualizar o arquivo que foi criado, não aparecendo

a janela.

A penúltima opção é `Exportar imagem`. Nessa opção você será capaz de exportar uma imagem do autômato que está sendo construído. Quando clicado também irá abrir uma janela para buscar a pasta e decidir o nome do arquivo a ser criado. Para que funcione de forma apropriada deve-se adicionar uma extensão de imagem ao nome do arquivo, como por exemplo teste.png, assim a imagem criada poderá ser aberta de forma direta em visualizadores de imagem.

A última opção é `Sair`, que quando clicada irá fechar a janela do simulador. Caso exista alguma informação não salva, aparecerá uma janela perguntando se deseja ou não salvar essas informações. No segundo menu temos as opções `Estado`, `Transição` e `Selecionar`. Nesse menu temos todas ferramentas para o desenho do autômato.

Na Figura C.8 temos o menu aberto com a opção `Estado` selecionada, as opções podem ser selecionadas clicando sobre seus respectivos botões. Quando selecionada a opção `Estado` você poderá clicar livremente na tela que estados serão desenhados, assim como mostrado na Figura C.9. Os índices dos estados são criados a partir do número 0 e incrementados a partir desse valor. Posteriormente será explicado como alterar o valor dos índices.

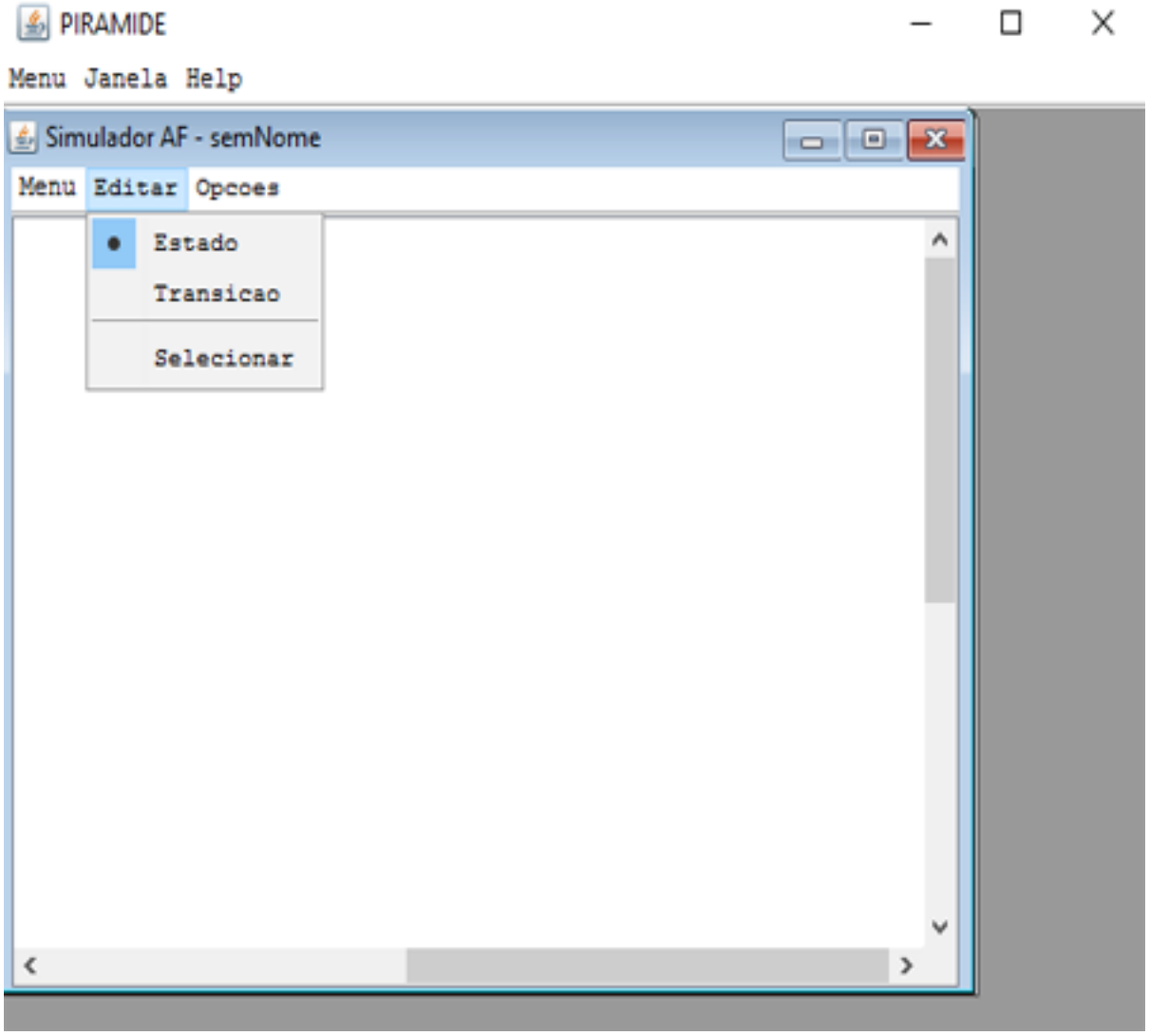

Figura C.8: Tela do Simulador com o menu `Editar` aberto e a opção `Estado` selecionada

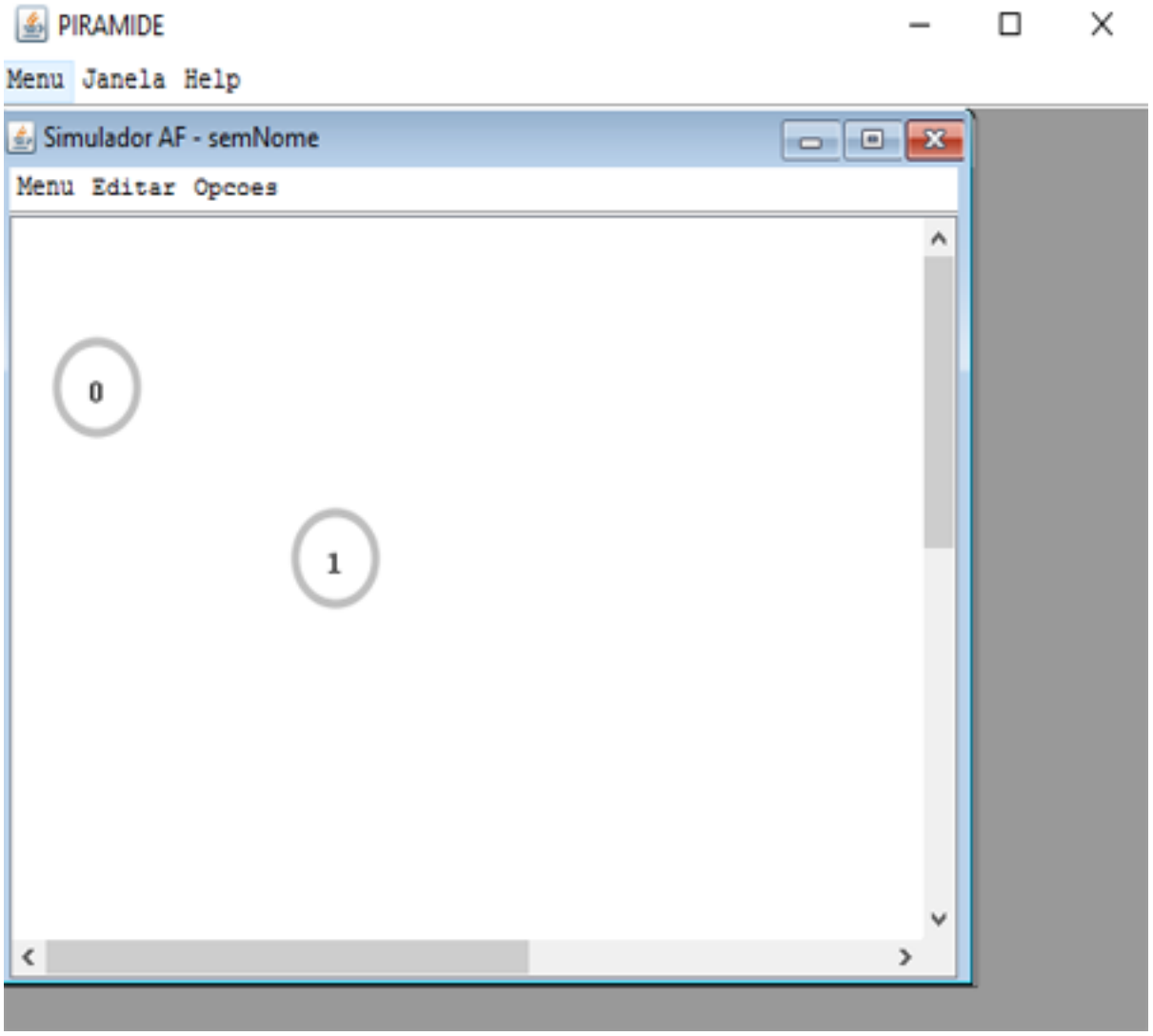

Figura C.9: Tela do Simulador demonstrando o desenho de Estados

Na Figura C.10 temos o menu aberto com a opção `Transição` selecionada. Essa opção permitirá o desenho das transições entre os estados. Para criar uma transição basta clicar primeiramente no estado que deve ser a origem e posteriormente no estado que deverá ser o destino, criando uma flecha, representando a transição, que ligará os dois estados.

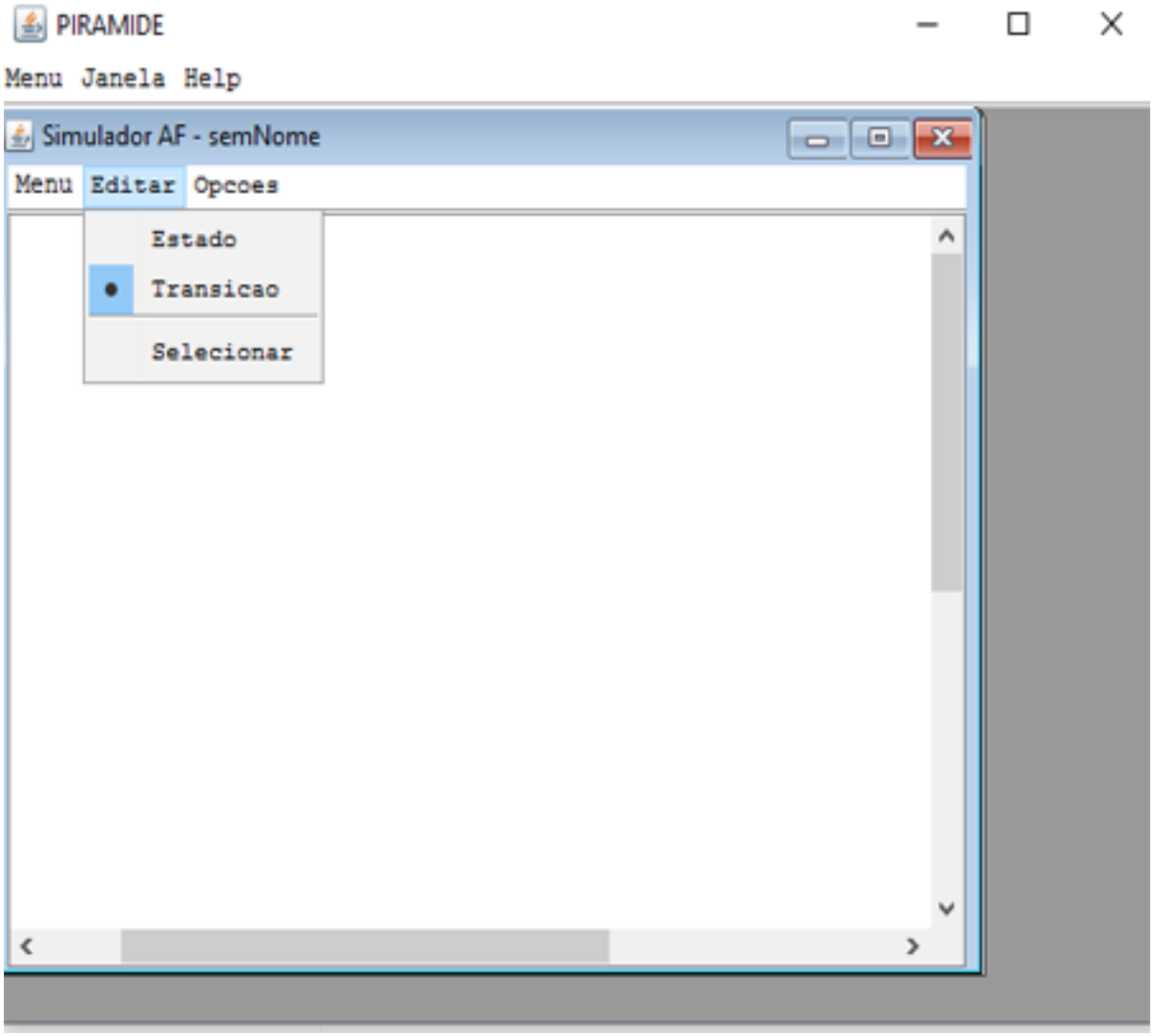

Figura C.10: Tela do Simulador com o menu `Editar` aberto e a opção `Transição` selecionada

Após clicar no estado de destino uma janela irá aparecer na tela pedindo que seja escrito o valor da transição. Nesse caso temos uma diferença entre cada simulador, visto que as transições não são as mesmas para cada máquina. Para o caso do `SimuladorAF` a tela será a representada na Figura C.11 onde para criar a transição basta digitar o caractere ou a string que deseja que a transição possua, por exemplo: `a`, `aabb` ou caso queira que a transição seja sobre vazio deve-se colocar o símbolo [].

Para o `SimuladorAP` a janela que irá abrir é a mesma da Figura C.11, porém a formatação da transição deve ser diferente. Para esse simulador a transição deve ser formatada como x,A/B, onde x é o símbolo lido, A é o valor que será desempilhado e B o valor que será empilhado. O

símbolo para representar o vazio também é o mesmo para esse simulador.

Para o `SimuladorMT` temos uma leve diferença na janela, como pode se visto na Figura C.12 onde possuímos um botão para ser inserido o símbolo β que representa o espaço em branco na célula. A formatação dessa transição é feita da seguinte forma x/X,D onde x é o símbolo lido, X o símbolo que será escrito e D é a direção que será movido o cabeçote na fita. Para movimentar o cabeçote para direita use a letra D e para esquerda use a letra E.

Em todos os três simuladores, caso se deseje fazer mais de uma transição sobre símbolos diferentes partindo de um mesmo estado e indo para um mesmo estado, basta separar cada transição com `enter` na janela de declaração de transição.

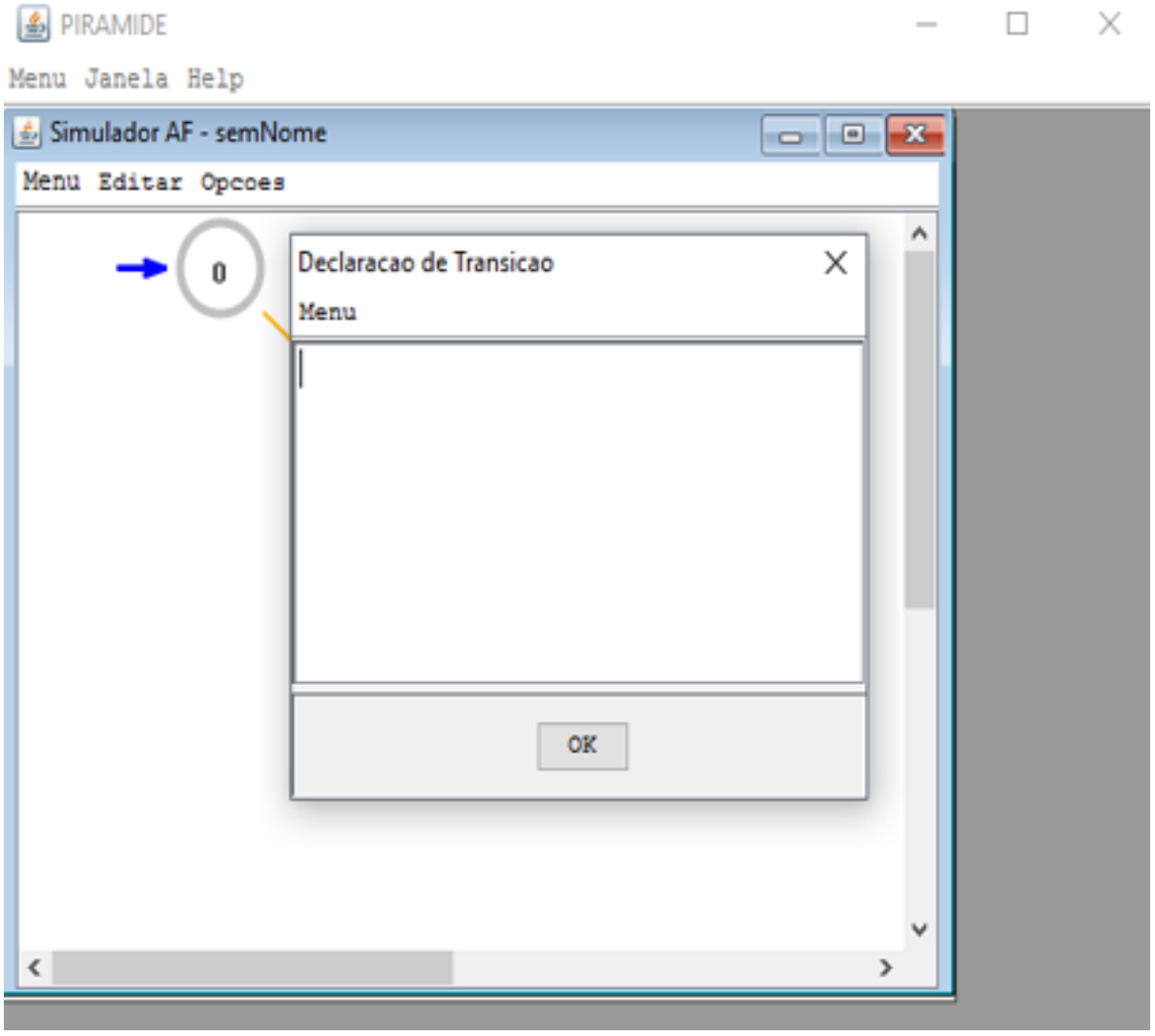

Figura C.11: Tela do Simulador de AF e AP demostrando o desenho de Transições

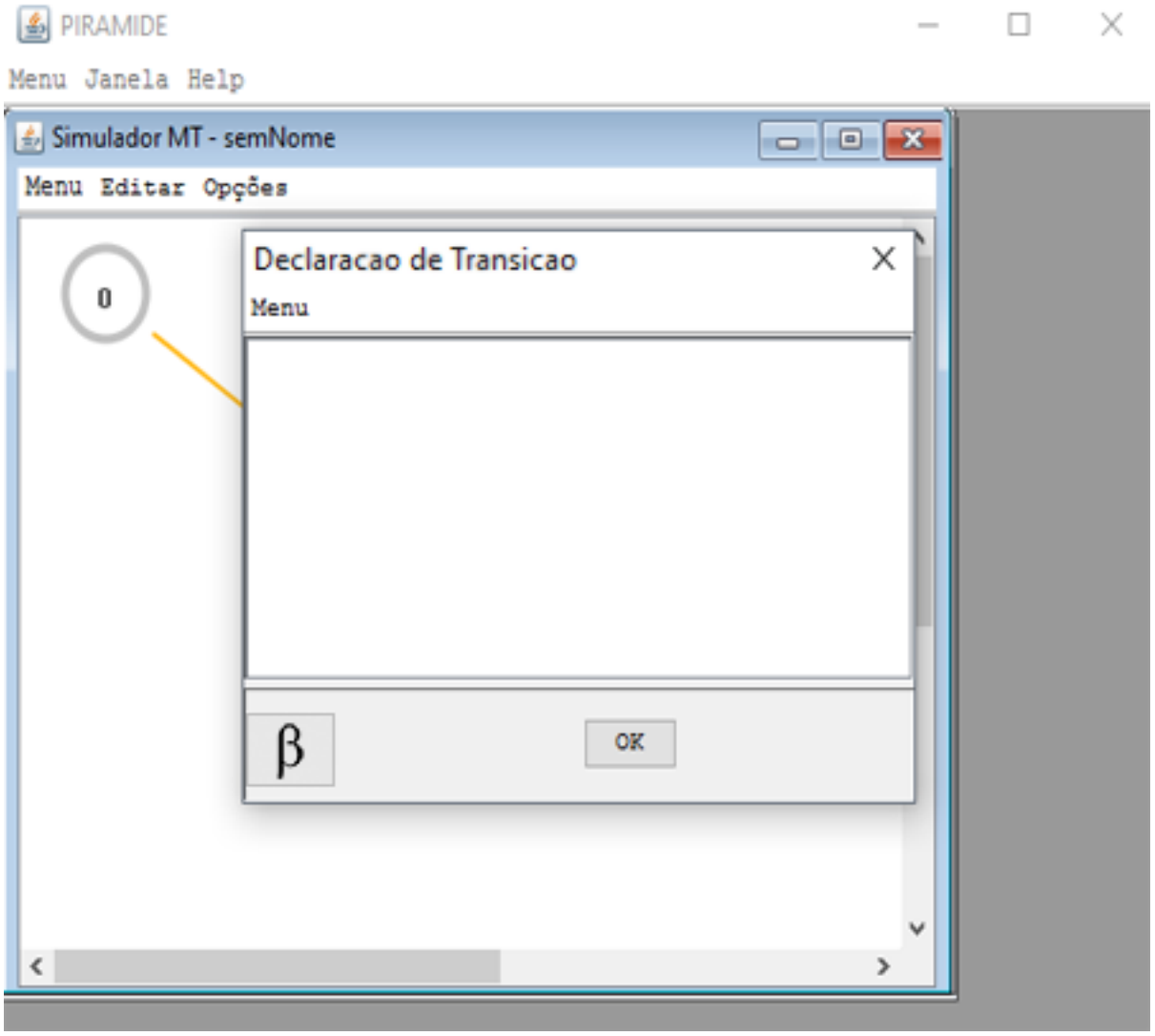

Figura C.12: Tela do Simulador de MT demostrando o desenho de Transições

Como última opção do menu temos o `Selecionar`, como vemos na Figura C.13. Com essa opção selecionada podemos executar várias ações. A primeira delas é mover um estado pela área de desenho, bastando apenas manter o mouse clicado sobre um estado e arrastá-lo pela tela.

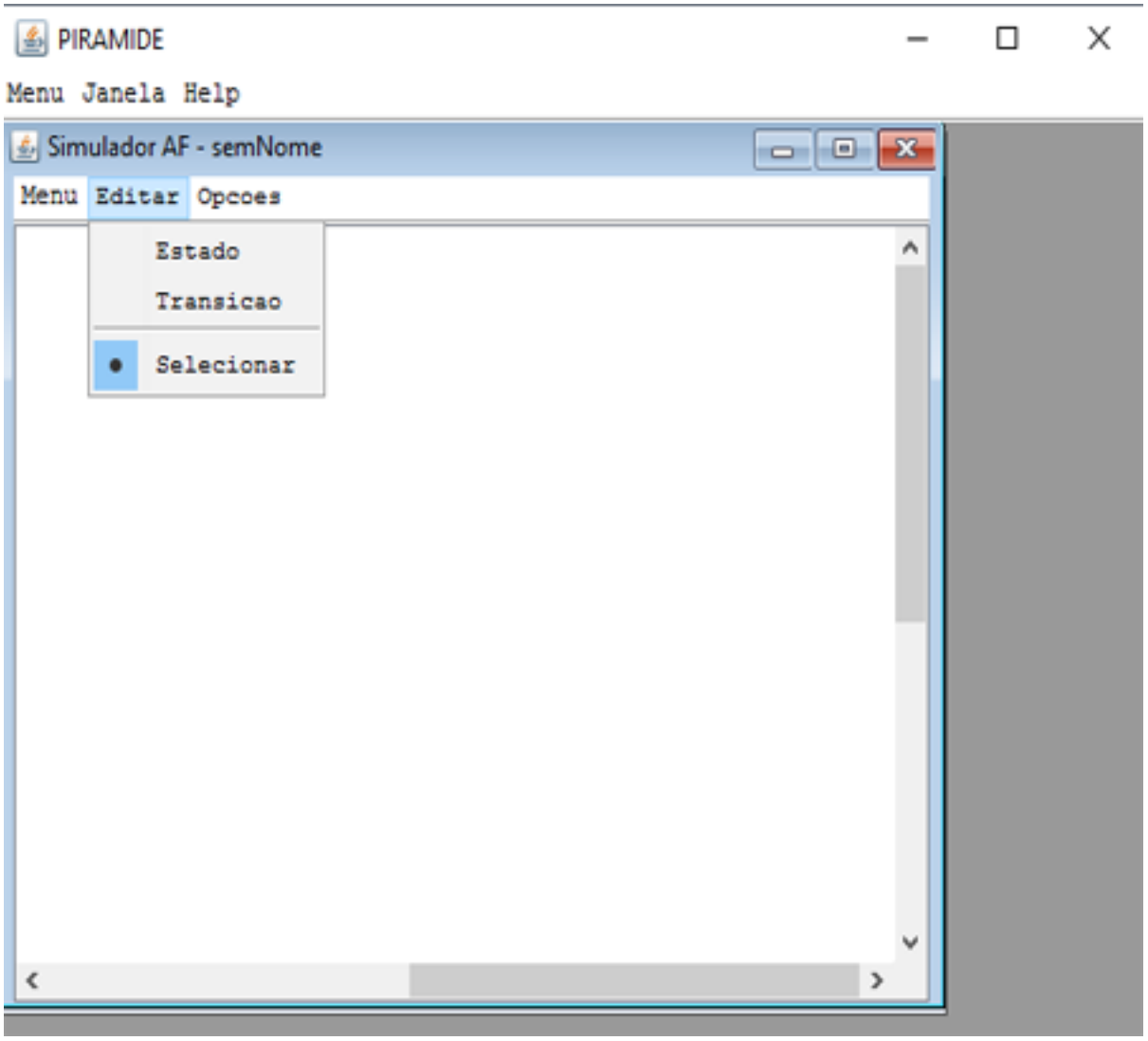

Figura C.13: Tela do Simulador com o menu `Editar` aberto e a opção `Selecionar` selecionada

Podemos também mover uma transição. Para isso, deve-se clicar sobre a mesma e então aparecerá uma imagem de um quadrado, como vemos na Figura C.14. Para movimentar a transição basta manter o mouse clicado sobre esse quadrado e arrastá-lo.

Além de mover os componentes da tela também pode-se alterar informações dos estados e das transições. Para editar um estado basta clicar com o botão direito sobre ele e algumas opções irão aparecer, como podemos ver na Figura C.15. A opção `remover` irá remover o estado e consequentemente todas as transições ligadas a ele. Na opção `Tipo de Estado` você poderá selecionar se deseja que o mesmo seja `Inicial` ou `Final` e na opção `Alterar ID` é possível alterar o índice do estado, porém só são aceitos índices numéricos.

Também é possível fazer alterações nas transições, também clicando com o botão direito sobre elas abrindo então um menu como na Figura C.16. A opção `Alterar Transição` permitirá alterar o símbolo da transição e o `Remover` remove a transição selecionada.

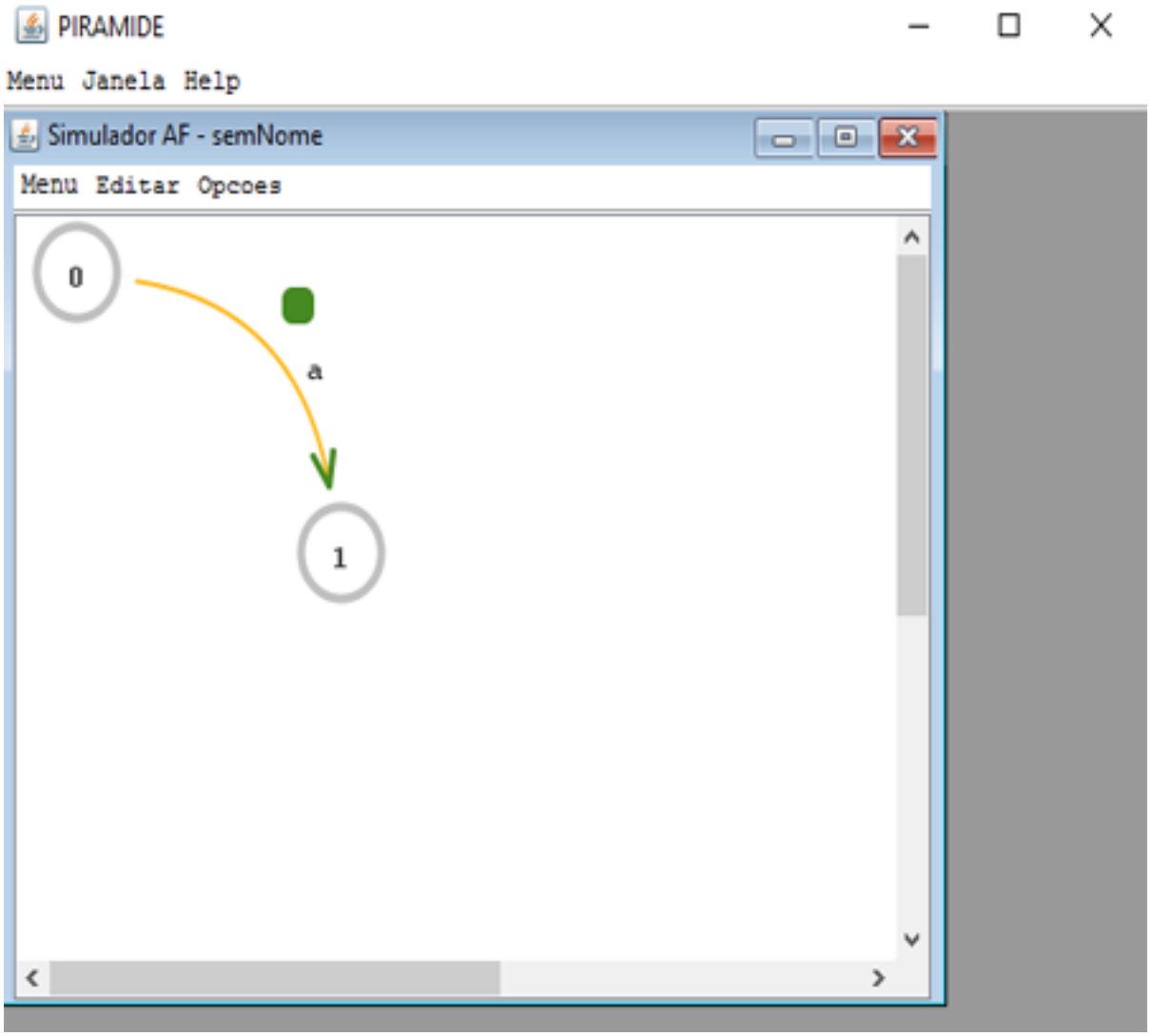

Figura C.14: Tela do Simulador demonstrando como mover uma transição

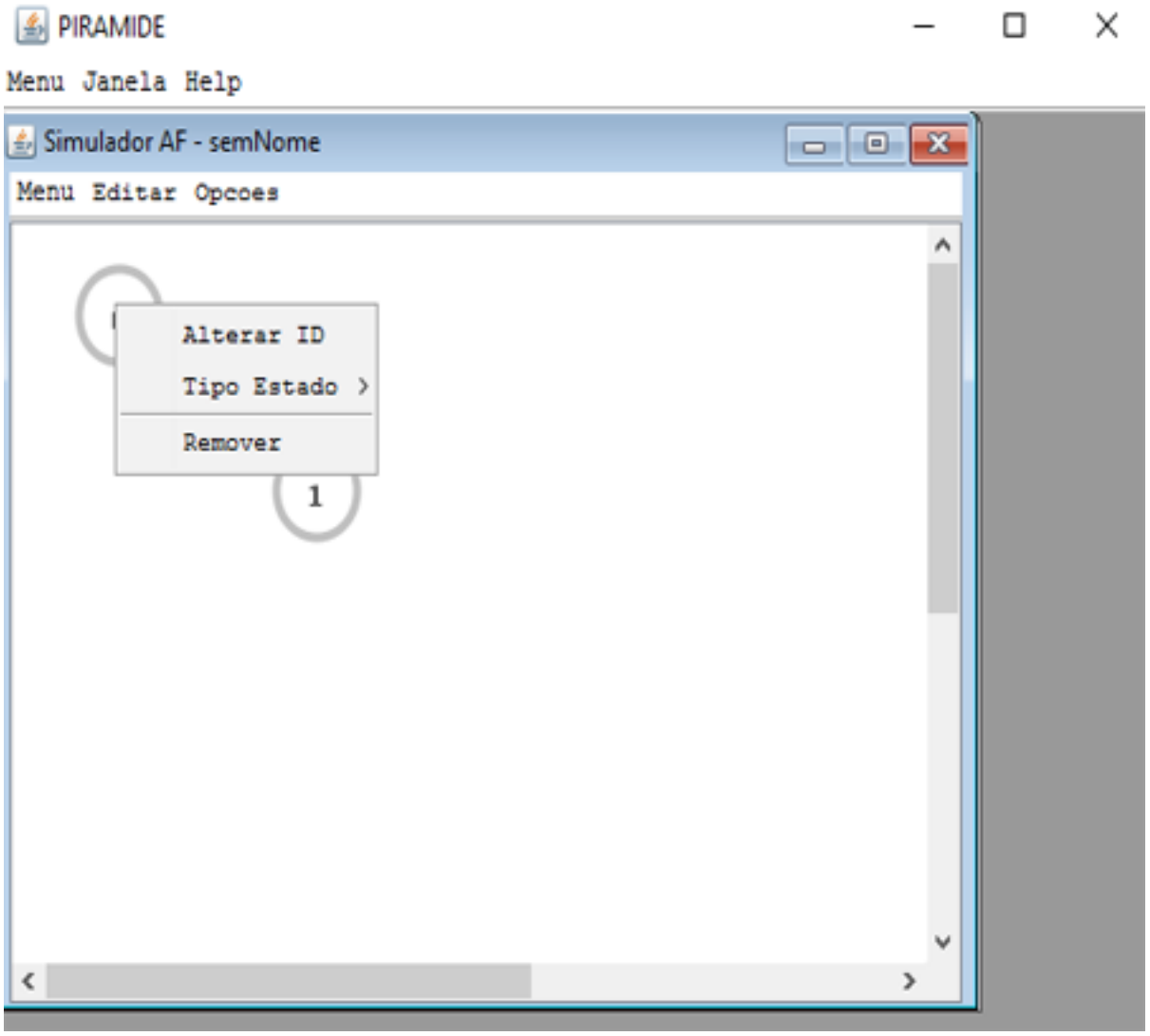

Figura C.15: Tela do Simulador demonstrando como editar um estado

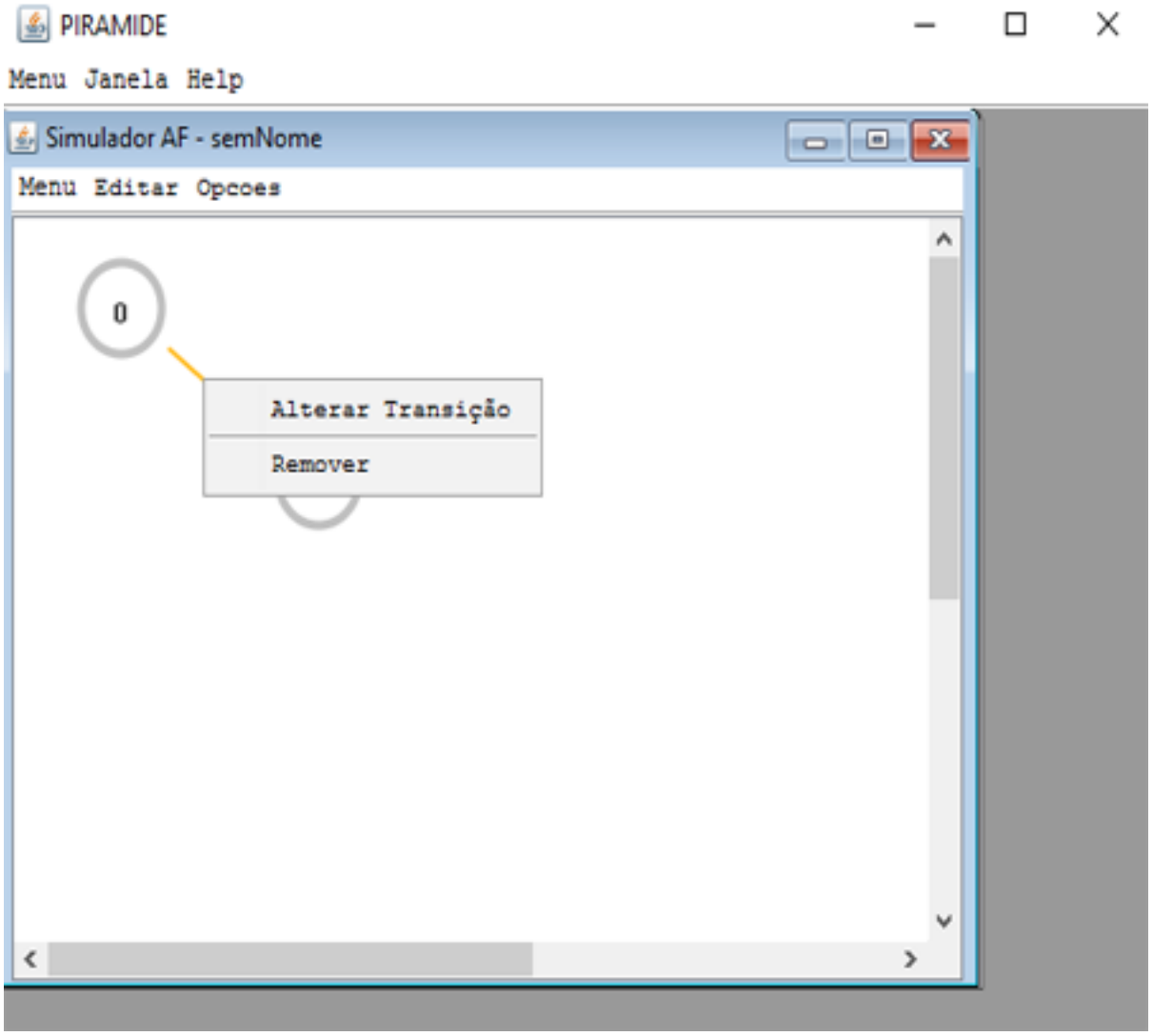

Figura C.16: Tela do Simulador demonstrando como editar uma transição

Para o menu `Opções` temos a diferença entre o `SimuladorAF` e os outros dois, por isso utilizaremos um `SimuladorAP` como exemplo para explicar as semelhanças e depois mostraremos as diferenças.

Como vemos na Figura C.17 temos duas opções nesse menu, `Testar autômato` ou `Deslocar autômato para origem`. A opção `Deslocar autômato para origem`, como o nome sugere, irá mover todo o autômato para a origem da área de desenho. A opção `Testar autômato` irá abrir uma janela na tela para que seja descrita a string a ser testada no autômato, como pode ser visto na Figura C.18. Para efetuar o teste deve-se digitar a string que se deseja testar e pressionar a tecla `enter`. A string então será processada

e aparecerá o resultado indicando se a mesma foi ou não aceita pelo autômato. Para testar uma string vazia, basta pressionar `enter` com o campo de texto vazio.

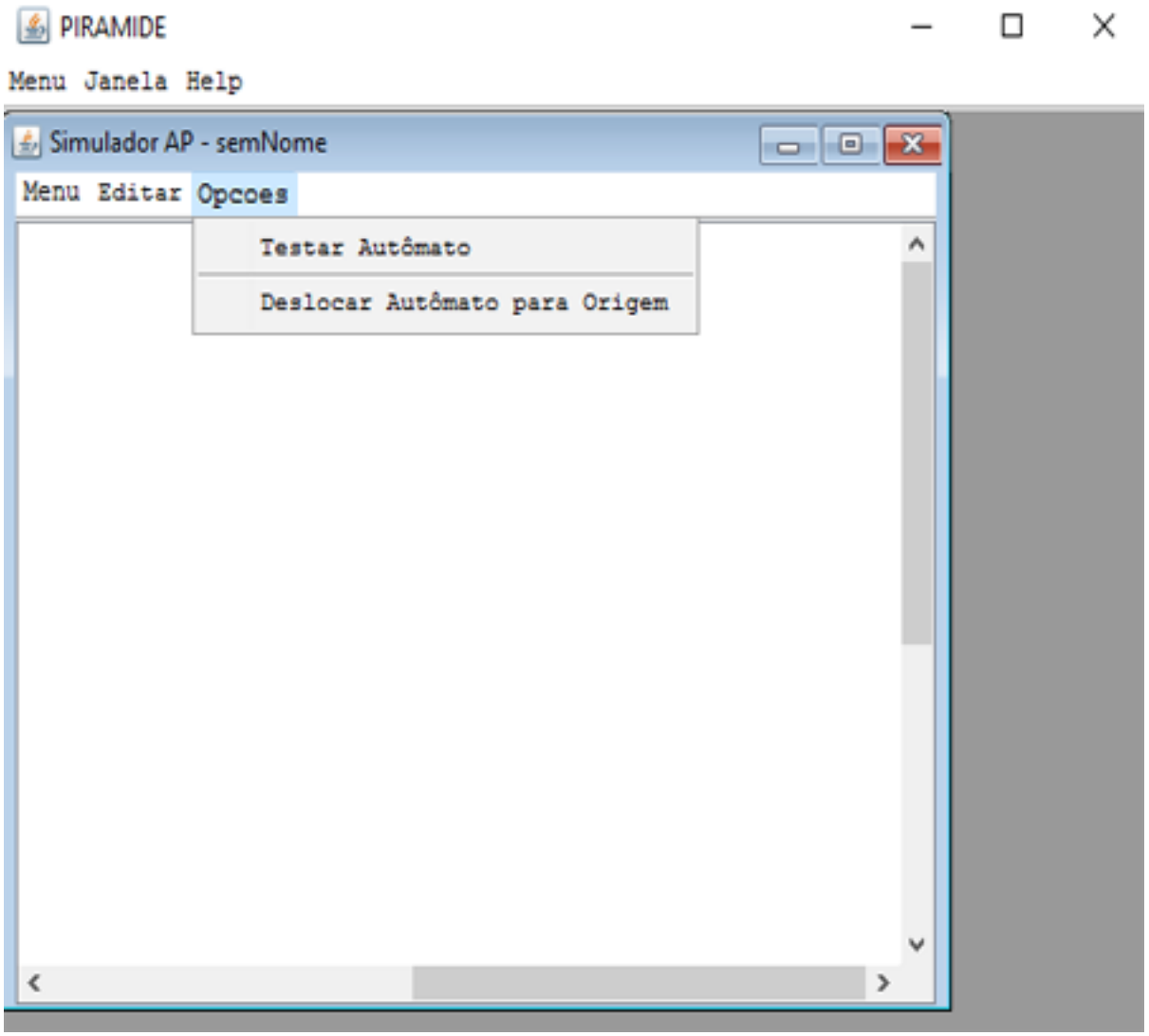

Figura C.17: Tela do Simulador demonstrando o menu `Opcoes`

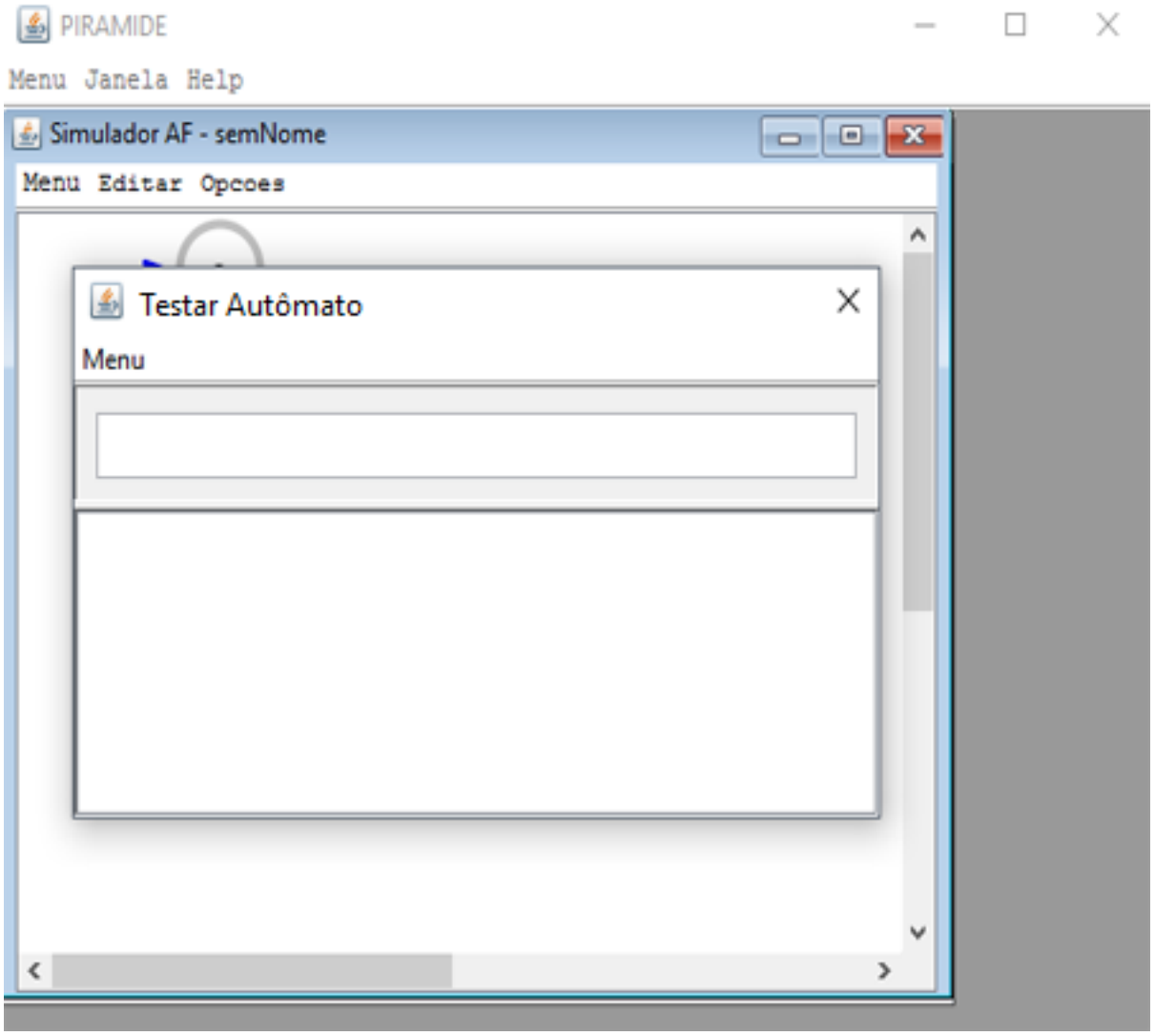

Figura C.18: Tela do Simulador demonstrando o a tela de testar autômato

Como citado anteriormente o `SimuladorAF` possui uma diferença nesse menu em relação aos outros, como vemos na Figura C.19. Em destaque na figura temos as três opções que possuímos a mais nesse Simulador. A opção `Reordenar IDs` irá refazer os valores dos índices do IDs em ordem crescente a partir do valor zero que será dado para o estado inicial.

As outras duas opções estão relacionadas aos componentes do simulador de AF. Primeiro temos a opção de fazer a transformação de AFN para AFD. Para realizar essa operação é necessário que o autômato seja não-determinístico, caso contrário irá aparecer uma mensagem na tela indicando que o mesmo é determinístico. Para realização da transformação existem duas possibilidades, onde você pode fazê-lo executando passo-a-passo clicando em cada passo da

transformação, ou suprimir a demonstração de um dos passos, por exemplo, caso clique direto na opção `Passo M2`, será realizado o `Passo M1` primeiramente, porém só será apresentado o autômato após a realização do passo M2.

Outra opção que possuímos é a de `Minimização`. Essa opção irá minimizar o autômato desenhado. Para que o processo de minimização ocorra é necessário que o autômato seja determinístico, portanto é necessário que no mínimo seja feito a transformação direto, clicando no `Passo M3`, para que a minimização ocorra.

Ambos processos, ao serem finalizados, desenharão os estados em linha reta, tornando assim, difícil a visualização do autômato, então é necessário após isso reorganizar o autômato na tela, para que ele fique mais prático de compreender.

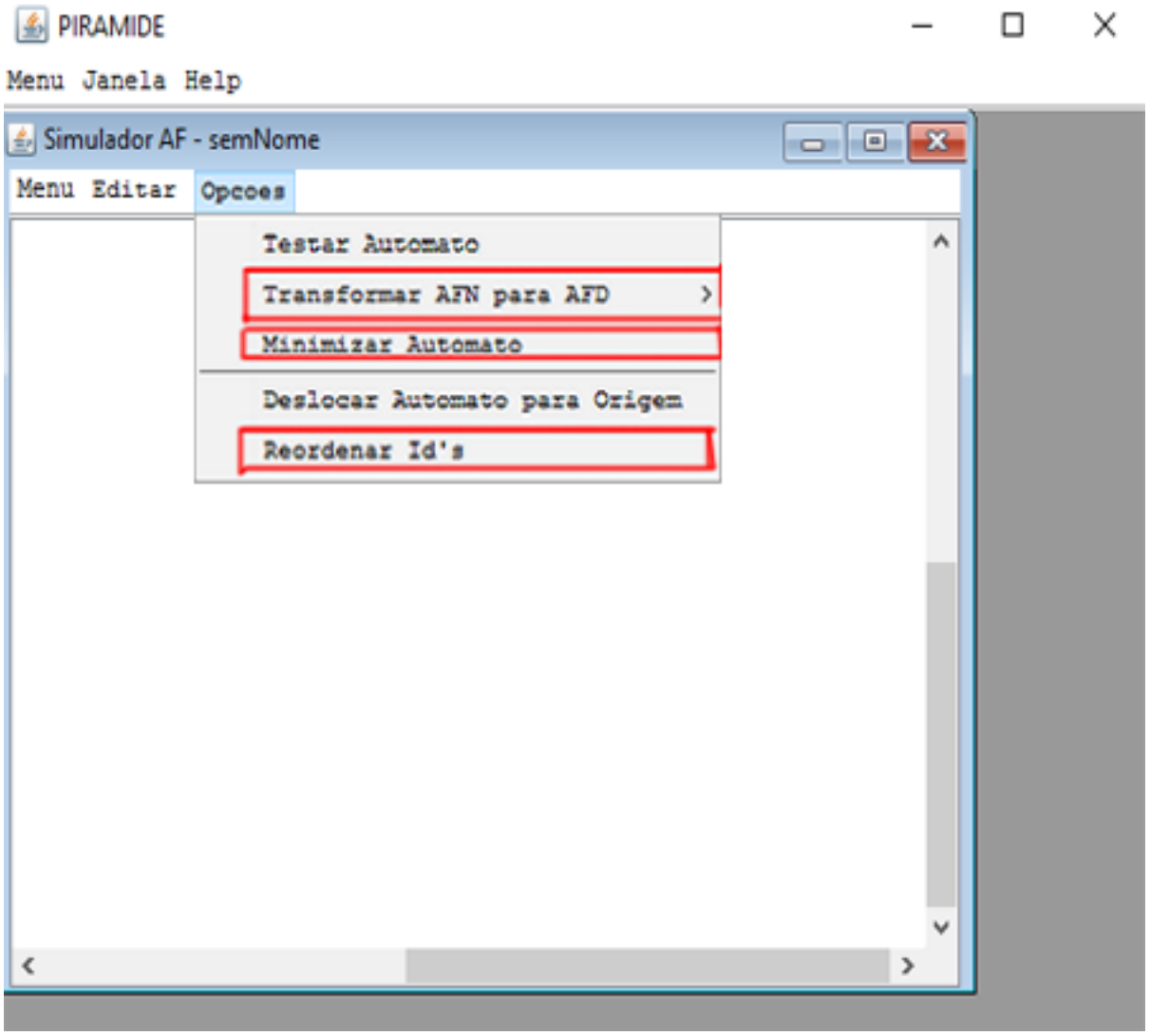

Figura C.19: Tela do Simulador demonstrando o menu `Opcoes` para o SimuladorAF

Apêndice D

Pseudocódigos da seção ??

Algoritmo 10 : Implementação do método de Transformação de AFN para AFD - Passo M1

```
Para(i = 0; i < listaTrans.tamanho(); i++){
  t = listaTrans[i]; vet = textoTransicao();
  Para(k = 0; k < vet.tamanho(); k++){
    Se ((vet[k].tamanho() > 1) E (!(vet[k] == []))) {
      estado = proximoMenorIdDesocupado();
      estado.setaTipo(ESTADOCOMUM);
      listaEstados.adiciona(estado);
      Se (j == 0) {
        tempTrans.setaIdOrigem(t.IdOrigem());
        tempTrans.setaIdDestino(estado);
        Se(!isTransicao(tempTrans.IdOrigem, tempTrans.IdDestino)
          listaTrans.adiciona(tempTrans);
      } Senao {
          tempTrans.setaIdOrigem(estadovelho);
          tempTrans.setaIdDestino(estado);
          Se(!isTransicao(tempTrans.IdOrigem,tempTrans.IdDestino)
          listaTrans.adiciona(tempTrans);
      }
      estadovelho = estado; estadovelho.setaTipo(ESTADOCOMUM);
    }
    tempTrans.setaIdOrigem(estadovelho));
    tempTrans.setaIdDestino(t.IdDestino());
    Se(!isTransicao(tempTrans.IdOrigem(), tempTrans.IdDestino()))
      listaTrans.adiciona(tempTrans);
    Se (vet.tamanho() == 1 OU t.Texto().Tamanho() == 0) {
      removerTrans(t.IdOrigem(), t.IdDestino());
      i-;
    } Senao t.setTexto(this.remove(t.getTexto(), vet[k]));
  }
}
```

Algoritmo 11 : Implementação do método de Transformação de AFN para AFD - Passo M2

```
i = 0;
Enquanto(i < listaTrans.tamanho()){
  Se listaTrans[i].possuiTransicao([])) {
    Para(j = 0; j < listaTrans.tamanho(); j++){
      Se (listaTrans[j].IdOrigem() == (listaTrans[j].IdDestino())
{
        Se (existeTransicao(listaTrans[j].IdOrigem(),
        (listaTrans[j].IdDestino())) {
          novo = nova Transicao();
          novo.IdOrigem = listaTrans.IdOrigem;
          novo.IdDestino = listaTrans.IdDestino;
          adicionaTransicao(novo);
        } Senao{
          trans = listaTrans.get(
          listaTrans[i].IdOrigem, listaTrans[i].IdDestino);
          vet = listaTrans.texto();
          Para (k = 0; k < vet.tamanho; k++) {
            Se (!contem(trans.Texto(), vet[k]))
              trans.setTexto(trans.getTexto() + vet[k]);
          }
        }
      }
    }
    Se (listaTrans[i].IdDestino.tipoVertice() == ESTADOFINAL)
      listaTrans[i].IdOrigem.setaTipoVertice();
    listaTrans[i].setaTexto(
    listaTrans[i].texto.replace([] , "");
    Se (listaTrans[i].texto.tamanho == 0)
      listaTrans.remove(i); i-;
  }
  i++;
  Para j = 0; j < listaEstados.tamanho(); j++
  for(int j = 0; j < listaEstados.size(); j++){
    e = listaEstados[j];
    Se(e.tipoVertice() != e.ESTADOINICIAL){
      Se(checaTrans(e)
        listaEstados.remove(e);
    }
  }
}
```

Algoritmo 12 : Implementação do método de Transformação de AFN para AFD - Passo M3 (Criação e preenchimento da tabela)

```
Para(i = 0; i < listaEstados.tamanho(); i++){
  tab.adiciona(lista[alfabeto.tamanho+1]);
  tab[i].adiciona(listaEstados[i]);
  Para(j = 0; j < alfabeto.tamanho; j++){
    aux = destinosAlcancaveisComCaracter(listaEstados[i],
alfabeto[j]);
    tab[i].adiciona(aux);
  }
}
Para(i = 0; i < tab.tamanho(); i++){
    Para(j = 0; j < tab[0].tamanho; j++){
      aux = tab[i][j].split(" ");
      Se ((aux.tamanho > 1) E (isTabela(tab, tab[i][j]) == -1))
{
        tabela.adiciona(alfabeto.tamanho()+1);
        tab[tab.size() - 1].adiciona(new String(tab[i][j]);
        Para (k = 0; k < aux.tamanho; k++) {
          vet = getTabelaLinha(tab, aux[k]);
          Se (tab[tab.size() - 1].tamanho() < 2) {
            Para (m = 0; m < vet.tamanho(); m++)
          tab.[tab.tamanho()-1].adiciona(vet[m]);
          } Senao{
            Para (m = 0; m < vet.tamanho(); m++){
              aux2 = tab.[tab.tamanho() - 1][m+1];
              Se (aux2 == "" aux2 = vet[m];
              Senao{
                Se (vet[m] == "") {
                  vetaux = vet[m].quebra("");
                  Para (x = 0; x < vetaux.tamanho(); x++) {
                    Se (!contem(aux2, vetaux[x])) aux2 += " "
                    + vetaux[x];
                  }
                }
              }
              tab[tab.tamanho() - 1][m + 1] = aux2;
            }
          }
        }
      }
    }
  }
}
```

Algoritmo 13 : Implementação do método de Transformação de AFN para AFD - Passo M3 (Eliminação Estados não alcançáveis)

```
Para (i = 0; i < tab.tamanho(); i++)
  vet.adiciona(tab.[i][0]);
Para (i = 0; i < vet.tamanho(); i++){
  aux = vet[i].quebra(" ");
  teste = verdadeiro;
  Se (aux.tamanho() < 2) {
    idd = vet[i];
    Se(idd.tipoVertice==ESTADOINICIAL
    OU idd.tipoVertice==ESTADOINICIALFINAL)
      teste = false;
  }
  Se (teste) {
    pos = isTabela(tab, vet[i]);
    Se (pos != -1) {
      Se (removerLinhaTabela(tab, pos) == verdadeiro) {
        linha = tabelaLinha(tab, vet[i]);
        Para (k = 0; k < linha.tamanho(); k++) {
          Se (linha[k] != "") vet.adiciona(linha[k]);
        }
        tab.remove(pos);
      }
    }
  }
}
```

Algoritmo 14 : Implementação do método de Transformação de AFN para AFD - Passo M3 (Montagem do autômato)

```
tamestados = listaEstados.tamanho();
tamtransicao = listaTrans.tamanho();
Para (i = 0; i < tab.tamanho(); i++) {
  est = tab[i][0];
  est.setaTipo(ESTADOCOMUM);
  aux = tab[i][0].quebra(" ");
  Para (j = 0; j < aux.tamanho(); j++) {
    (aux.tamanho() < 2) est.setaTipo(aux[j].tipoVertice);
    Senao {
      Se ((aux[j].tipoVertice==ESTADOFINAL) OU
(aux[j].tipoVertice==ESTADOINICIALFINAL)){
        est.setaTipo(ESTADOFINAL);
        pare;
      }
    }
  }
  listaEstados.adiciona(est);
}
Para (j = 0; j < tamestados; j++) listaEstados.remove(0);
Para (i = 0; i < tab.tamanho(); i++) {
  prim = tab[i][0];
  Para (j = 1; j < tab[0].tamanho(); j++) {
    Se (tab[i][j] != "") {
      Transicao tran;
      tran.setaIdOrigem(prim);
      seg = tab[i][j];
      tran.setIdDestino(seg);
      Se (!isTransicao(tran.IdOrigem(), tran.IdDestino()))
        listaTrans.adiciona(tran);
      Senao
        t = listaTrans.get(tran.IdOrigem(), tran.IdDestino());
    }
  }
}
Para (j = 0; j < tamtransicao; j++) listaTrans.remove(0);
```

Algoritmo 15 : Implementação do método de minimização de Autômatos Finitos - Criação da tabela e marcação dos estados trivialmente equivalentes

```
Se(automato.isAFD()){
  criaRatueiraEVerificaSeETotal();
  int tab[listaEstados.tamanho()] [];
  int cont = 2;
  Para(i = 0; i < listaEstados.tamanho()-1; i++){
    tab[i][0] = listaEstados[i+1];
    Para(j = 0; j < cont; j++){
      tab[i][j] = 0;
    }
    cont++;
  }
  tab[listaEstados.tamanho()-1][0] = -2;
  Para(i = 0; i < listaEstados.tamanho()-1; i++){
    tab[listaEstados.tamanho()-1][i+1]= listaEstados[i];
  }
  cont = 2;
  Para(i = 0; i < listaEstados.tamanho()-1; i++){
    Para(j = 0; j < cont; j++){
      id1 = tab[i][0];
      id2 = tab[listaEstados.tamanho()-1][j];
      Se ((automato.estado(id1).tipoVertice == ESTADOFINAL) OU
      (automato.estado(id1).tipoVertice == ESTADOINICIALFINAL){
        Se((automato.estado(id2).tipoVertice != ESTADOFINAL) E
        (automato.estado(id2).tipoVertice !=
        ESTADOINICIALFINAL){
          tab[i][j] = -1;
        }
      }
      Se ((automato.estado(id2).tipoVertice == ESTADOFINAL) OU
      (automato.estado(id2).tipoVertice == ESTADOINICIALFINAL){
        Se((automato.estado(id1).tipoVertice != ESTADOFINAL) E
        (automato.estado(id1).tipoVertice !=
        ESTADOINICIALFINAL){
          tab[i][j] = -1;
        }
      }
    }
    cont++;
  }
  cont = 2;
```

Algoritmo 16 : Implementação do método de minimização de Autômatos Finitos - Marcação dos estados não-trivialmente equivalentes

```
Para(i = 0; i < listaEstados.tamanho()-1; i++){
  Para(j = 0; j < cont; j++){
    Se (tab[i][j] == -1) continue;
    qu = tab[i][0];
    qv = tab[listaEstado.tamanho()-1][j];
    Para(a = 0; a < alfabeto.tamanho(); a++){
      pu = automato.transicao(qu, alfabeto[a]);
      pv = automato.transicao(qv, alfabeto[a]);
      Se(pu != pv){
        Se(!isMarcadoTabela(tab, pu, pv)){
          tem = falso;
          Para(k = 0; k < lista.size; k++){
            puaux = lista[k].quebra(,);
            pvaux = lista[k].quebra(=).quebra(,);
            Se((puaux == pu E pvaux == pv ) OU (puaux == pv
            E pvaux == pu)){
              lista[k] = lista[k];qu,qv; tem = verdadeiro;
            }
          }
          Se(tem == falso)
            lista.adiciona(pu,pv=qu,qv);
        } Senao{
          marcarTabela(tab, qu, qv);
          encabeca.adiciona(qu,qv);
          Para(m = 0; m < encabeca.tamanho(); m++){
            Para(k = 0; k < lista.tamanho(); k++){
              aux = encabeca[m].quebra(,);
              aux2 = aux[1],aux[0];
              Se((lista[k].quebra(=)[0]==encabeca[m])OU(lista[k].quebra(
                corpo = lista[k].quebra(=).quebra[;];
                Para(g = 0; g < corpo.tamanho; g++){
                  um = corpo[g].quebra(,)[0];
                  dois = corpo[g].quebra(,)[1];
                  Se(!isMarcadoTabela(tab, um, dois)){
                    marcarTabela(tab, um, dois);
                    encabeca.adiciona(corpo[g]);
                  }
                }
              }
            }
          }
        }
      }
    }
  } cont++;
```

Algoritmo 17 : Implementação do método de minimização de Autômatos Finitos - Unificação dos estados equivalentes

```
cont = 2;
numEstados = listaEstados.tamanho();
Para(i = 0; i < listaEstados.tamanho()-1; i++){
  Para(j = 0; j < cont; j++){
    Se(tab[i][j] == -1) continue;
    id1 = tab[i][0];
    id2 = tab[numEstados-1][j];
    um = automato.estado(id1);
    dois = automato.estado(id2);
    Se(um == D){
      um = automato.estado(id2);
      dois = automato.estado(id1);
      id2 = tab[i][0];
      id1 = tab[numEstados-1][j];
    }
    vet = dois.split(,);
    Para(k = 0; k < vet.tamanho; k++){
      Se(!automato.contem(um, vet[k])) um = um,vet[k];
    }
    Se(um.tippVertice == ESTADOCOMUM) um.setaTipo(dois.tipoVertice);
    Senao{
      Se(um.tipoVertice) == ESTADOINICIAL){
        Se(dois.tipoVertice == ESTADOFINAL
          OU dois.tipoVertice == ESTADOINICIALFINAL)
          um.setaTipo(ESTADOINICIALFINAL);
      }
      Senao{
        Se(um.tipoVertice == ESTADOFINAL){
          Se(dois.tipoVertice == ESTADOINICIAL
          OU dois.tipoVertice == ESTADOINICIALFINAL)
            um.setaTipo(ESTADOINICIALFINAL);
        }
      }
      arrumaTransicoes(id1, id2);
      remove.adiciona(id2);
    }
  } cont++;
}
Para(i = 0; i < remove.tamanho(); i++){
  automato.removeEstado(remove[i]);
}
```

Algoritmo 18 : Implementação do método de minimização de Autômatos Finitos - Exclusão dos estados inúteis

```
    entra = falso; sai = falso; idsVerif = listaEstados;
    Para(k = 0; k < idsVerif.tamanho; k++){
      e = idsVerif[k];
      Se(e.tipoVertice == ESTADOCOMUM){
        Para(l = 0; l < listaTrans.tamanho(); l++){
          t = listaTrans[l];
          Se(t.idOrigem == e) sai = verdadeiro;
          Se(t.idDestino == e) entra = verdadeiro;
        }
        Se((sai == verdadeiro E entra == falso) OU sai == falso
        E entra == verdadeiro){
          automato.removeEstado(listaEstados[k]);
          Se(k-1 >= 0) k = k-1;
          Senao k = 0;
        } Senao{
          inutil = verdadeiro; list.adiciona(e);
          Para(i = k; i < list.tamanho(); i++){
            e2 = automato.estado[list[i]];
            inutil = verdadeiro;
            Para(j = 0; j < listaTrans.tamanho(); j++){
              t = listaTrans[j];
              Se(t.idOrigem == e2)
                Se(!(automato.estado((t.idDestino).tipoVertice ==
                ESTADOFINAL) OU(automato.estado((t.idDestino).tipoVertice
                == ESTADOINICIALFINAL))
                  Se(!list.contem(t.idDestino))
                  list.adiciona(t.idDestino);
                Senao{
                Para(g = 0; g < list.tamanho(); g++)
                idsVerif.remove(list[g]); inutil = falso;
                }
            }
          }
          Se(inutil E e.tipoVertice !=ESTADOINICIAL)
            automato.removeEstado(listaEstados[k]);
          Se(k-1 >= 0) k = k-1;
          Senao k = 0;
        }
      }
    }
  }
  ratu = automato.estado(D);
  Se(ratu != NULL) automato.removeEstado(ratu);
}
```

Apêndice E

Conjunto de testes realizado pelos acadêmicos

PARTE 1: Autômatos Finitos (20 minutos)

Passo 1.1: Teste do simulador

1- Abra o simulador de autômatos finitos.

2- Encontre em suas anotações um autômato finito e desenhe-o (seus estados e transições) no Simulador de Autômatos Finitos. Lembre-se de selecionar o estado inicial e os estados finais.

3- Procure a opção de testar strings no autômato.

4- Teste strings nesse autômato, que sejam válidas (que o autômato reconheça) e também teste strings que não sejam válidas (que o autômato não reconheça).

5- Feche a janela e o simulador.

Passo 1.2: Teste da transformação de AFN para AFD

1- Abra o simulador de autômatos finitos

2- Encontre em suas anotações um autômato finito e desenhe-o (seus estados e transições) no Simulador de Autômatos Finitos. Lembre-se de selecionar o estado inicial e os estados finais.

3- Procure a opção de transformação de AFN para AFD.

4- Realize a transformação do autômato.

5- Organize o autômato na tela.

6- Feche o simulador

Obs.: Execute esse passo pelo menos duas vezes.

Passo 1.3: Teste da minimização de AF

1- Abra o simulador de autômatos finitos

2- Encontre em suas anotações um autômato finito desenhe-o (seus estados e transições) no Simulador de Autômatos Finitos. Lembre-se de selecionar o estado inicial e os estados finais.

3- Procure a opção de minimização.

4- Realize a minimização do autômato.

5- Organize o autômato na tela.

6- Feche o simulador

PARTE 2: Autômatos de Pilha (10 minutos)

1- Abra o simulador de autômatos de pilha.

2- Encontre em suas anotações um autômato de pilha e desenhe-o (seus estados e transições) no Simulador de Autômatos de Pilha. Lembre-se de selecionar o estado inicial e os estados finais.

3- Procure a opção de testar strings no autômato.

4- Teste strings nesse autômato, que sejam válidas (que o autômato reconheça) e também teste strings que não sejam válidas (que o autômato não reconheça).

5- Feche a janela e o simulador.

PARTE 3: Máquina de Turing (10 minutos)

1- Abra o simulador de Máquina de Turing.

2- Encontre em suas anotações uma Máquina de Turing (que possua estado final) e desenhe-o (seus estados e transições) no Simulador de Máquinas de Turing. Lembre-se de selecionar o estado inicial e os estados finais.

3- Procure a opção de testar strings no autômato.

4- Teste strings nesse autômato, que sejam válidas (que o autômato reconheça) e também teste strings que não sejam válidas (que o autômato não reconheça).

5- Feche a janela e o simulador.

Referências Bibliográficas

BOVET, J. *Auger - Ambiente para construção e simulação de autômatos finitos*. 2005. Consultado na INTERNET: http://www.evoluma.com/auger/index.html, 17 de agosto de 2016.

DENTI, E.; OMICINI, A.; CALEGARI, R. tuProlog: Making Prolog ubiquitous. *ALP Newsletter*, Association for Logic Programming, out. 2013. Disponível em: <http://www.cs.nmsu.edu/ALP/2013/10/tuprolog-making-prolog-ubiquitous/>.

HOLST, G.; MACKWORTH, A. *Computational Intelligence: A Logical Approach*. 2005. Consultado na INTERNET: http://jlogic.sourceforge.net, 13 de dezembro de 2016.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. 2. ed. USA: Addison Wesley, 2001.

JUKEMURA, A. S.; NASCIMENTO, H. A. D. do; UCHôA, J. Q. Gam - um simulador para auxiliar o ensino de linguagens formais e de autômatos. In: *XXV Congresso da Sociedade Brasileira de Computação*. São Leopoldo: [s.n.], 2014. p. 2432–2443.

LEWIS, H. R.; PAPADIMITRIOU, C. *Elementos de teoria da computação*. 2. ed. Porto Alegre: Bookman, 2000.

MENEZES, P. B. *Linguagens formais e autômatos*. 4. ed. Porto Alegre: Bookman, 2002.

NGUYEN, H. Q.; KANER, C.; FALK, J. *Testing Computer Software*. 2. ed. Hoboken, New Jersey: Wiley Computer Publishing, 1999.

RODGER, S. H.; FINLEY, T. W. *JFLAP: An Interactive Formal Languages and Automata Package*. 1. ed. Sudbury, MA: Jones and Bartlett Publishers, 2006.

SUDKAMP, T. A. *Languages and Machines: An Introduction to the Theory of Computer Science*. 2. ed. [S.l.]: Addison Wesley, 1997.

VIEIRA, N. J. *Introdução aos Fundamento da Computação - Linguagens e máquinas*. 1. ed. SP: Thomson, 2006.

WIELEMAKER, J. et al. Swi-prolog. In: *Theory and Practice of Logic Programming*. [S.l.: s.n.], 2012. p. 67–96.