

Relatório Parcial de Iniciação Científica

Uma Ferramenta de Software para a Predição de Desempenho de Workflows Científicos

Aluno: Lucas Magno
Instituto de Física (IF)
Universidade de São Paulo

Orientadora: Kelly Rosa Braghetto
Departamento de Ciência da Computação (DCC)
Instituto de Matemática e Estatística (IME)
Universidade de São Paulo

1 Introdução

Um *workflow científico* é a descrição completa ou parcial de um experimento científico em termo de suas atividades, controles de fluxo e dependência de dados [1]. Há várias maneiras de se representar um *workflow científico*, entre elas Redes de Petri, Linguagem Unificada de Modelagem, Álgebra de Processos e grafos direcionados [2], mas somente estas duas últimas serão utilizadas neste projeto. Como uma das formas mais comuns de representação, grafos direcionados permitem uma visualização gráfica do *workflow*, facilitando sua descrição. Já a álgebra de processos é uma descrição algébrica textual que permite verificações, análises algébricas e aperfeiçoamento de processos por meio de transformações [2].

Um *workflow* pode ser composto de diversas estruturas, mas as relevantes aqui são atividades, que representam atividades reais de um experimento e estruturas para controle de fluxo, que descrevem o fluxo dos dados através do *workflow*. Temos como exemplos de controles de fluxo sequência, paralelismo, escolha e sincronização.

É comum em experimentos científicos a manipulação de enormes quantidades de dados e processos muito demorados, o que estimula o cientista a aperfeiçoar o experimento antes de sua execução, pois esta pode demandar muitos recursos e tempo. Daí a necessidade da análise do desempenho de um *workflow*, que pode ser feita através de três métodos [3]:

- *medição*: consiste na execução do *workflow* uma quantidade estatisticamente relevante de vezes e então no cálculo dos tempos médios de interesse. Logo, só pode ser aplicada a sistemas já implementados, não tendo caráter preditivo;
- *simulação*: baseada em modelos matemáticos cuja solução é dada por um programa que simula o comportamento modelado;
- *modelagem analítica*: também baseada em modelos, mas analisa numericamente determinados aspectos de interesse em um sistema.

Neste projeto, faremos uso da modelagem analítica, por ser preditiva, rápida e não muito difícil de se implementar, embora menos precisa que os outros métodos.

-> Depois de falar que o projeto optou pelo uso de modelagem analítica, você também precisa mencionar a PEPA. Você pode escrever algo como:

"Tanto as redes de Petri quanto as álgebras de processos são formalismos que possuem extensões estocásticas e que, portanto, podem ser usados na modelagem analítica de workflows científicos.

Neste trabalho usaremos uma álgebra de processos estocástica - a PEPA (Performance Evaluation Process Algebra), porque o uso desse formalismo ainda não foi profundamente explorado para a análise de desempenho preditiva de workflows científicos. "

2 Objetivos

Uma desvantagem da modelagem analítica ^{usando PEPA} é a necessidade da descrição do *workflow* em uma ^{????} linguagem própria e utilização de programas específicos para a análise, exigindo do usuário um certo nível de conhecimento sobre álgebras de processo. No entanto, *workflows* científicos são utilizados em diversas áreas da ciência que não necessitam de um grande aprofundamento em computação, o que pode inviabilizar a aplicação deste método. Portanto, é interessante que exista uma ferramenta capaz de automatizar todo o processo a partir da descrição do *workflow* em uma linguagem textual simples, o que pretendemos neste projeto.

que processo??? "o processo de predição de desempenho"

3 Metodologia

Experimentos científicos podem ser muito complicados, com inúmeras atividades diferentes que podem não ser executadas de forma linear. Por isso consideraremos, num primeiro momento, que os *workflows* a serem analisados serão bem comportados, isto é, apresentam somente um ponto de entrada e um ponto de saída, têm sua estrutura em forma de "blocos" e não apresentam ciclos, ou laços, o que permite uma implementação mais simples.

3.1 O Programa (*)

Para automatizar o processo de predição de desempenho, ^{estamos implementando} implementaremos um programa em *Python* capaz de ler descrições textuais em linguagens simples a serem definidas do *workflow* e dos recursos utilizados por cada atividade deste através de analisadores léxicos e sintáticos disponíveis na biblioteca *PLY*, *Python Lex-Yacc*¹, com dependência na biblioteca *pyParsing*². Então, gerar uma estrutura de dados baseada em grafo que represente o *workflow* utilizando a biblioteca *python-graph*³ a qual ^{está faltando coisa na frase?} permite a criação e manipulação de diversos tipos de grafos através de classes.

Tendo a estrutura, o programa ~~deve~~ traduzir para a linguagem *DOT*⁴ e gerar automaticamente um arquivo *pdf* com a visualização gráfica da estrutura, para que possa ser feita a conferência entre esta e o *workflow* descrito inicialmente. Ambos processos são executados por ferramentas já implementadas na biblioteca *python-graph*, com dependência nas bibliotecas *pydot*⁵ e *Graphviz*⁶.

A partir daí, basta que o programa traduza o *workflow* em grafo para um modelo ~~de modelagem~~ analítico⁰. No caso, usaremos o PEPA, ou *Performance Evaluation Process Algebra*⁷ uma álgebra de processos estocásticos⁸, e sua implementação em *Python*, a *pyPEPA*⁸. Utilizando esta biblioteca, pode-se executar automaticamente a análise numérica e, então, apresentar os resultados ao usuário.

Pretende-se, também, utilizar outras ferramentas para auxiliar a compreensão dos conceitos envolvidos, entre elas *Eclipse*⁹ e *Taverna*¹⁰.

3.2 Nota Técnica

O projeto será desenvolvido no sistema operacional *Lubuntu* 13.10 e testado a partir de um terminal *linux*.

(*) Lucas, antes de sair dizendo como o programa foi feito, seria melhor explicar sucintamente o que o programa se propõe a fazer. Algo como: "Para automatizar o processo de predição de desempenho, implementaremos um programa que: (i) lê como entrada uma descrição textual de um *workflow*; (ii) gera uma visualização do *workflow* de entrada, (iii) gera um modelo analítico (estocástico) do *workflow*, (iv) obtém a solução numérica desse modelo, e (v) extrai índices de desempenho a partir dessa solução."

Aí, depois você pode dizer qual foi a estratégia usada para implementar as etapas (i) e (ii) - que já estão concluídas - e qual será a estratégia para implementar as demais etapas. Tudo isso já está escrito aí na seção 3. Você só precisa organizar melhor as ideias.

¹ <http://www.dabeaz.com/ply/>, visualizado em 14 de janeiro de 2014

² <http://pyparsing.wikispaces.com/>, visualizado em 14 de janeiro de 2014

³ <http://code.google.com/p/python-graph/>, visualizado em 14 de janeiro de 2014

⁴ <http://www.graphviz.org/content/dot-language>, visualizado em 14 de janeiro de 2014

⁵ <https://code.google.com/p/pydot/>, visualizado em 14 de janeiro de 2014

⁶ <http://www.graphviz.org/>, visualizado em 14 de janeiro de 2014

⁷ <http://www.dcs.ed.ac.uk/pepa/>, visualizado em 14 de janeiro de 2014

⁸ <https://github.com/tdi/pyPEPA>, visualizado em 14 de janeiro de 2014

⁹ <http://www.eclipse.org/>, visualizado em 14 de janeiro de 2014

¹⁰ <http://www.taverna.org.uk/>, visualizado em 14 de janeiro de 2014

4 Resultados parciais

Até agora, o programa executa todos os passos desde a leitura da descrição textual do *workflow* à visualização gráfica da estrutura em grafo criada, exceto ler a descrição dos recursos utilizados por cada atividade, o que ainda não foi implementado. Consequentemente, apenas a linguagem de descrição do *workflow* já foi definida. Para demonstrar as saídas do programa, serão utilizados exemplos de um mesmo experimento, enquanto que o código do programa em si ficará em um apêndice.

pode ser visto no

4.1 Descrição Textual ^{do} Workflow

Baseada na linguagem DOT, a linguagem aqui definida busca ser simples, pois não necessita de tanta generalidade quanto aquela, apenas descrevendo experimentos científicos, ao invés de qualquer tipo de grafo.

```
1 graph          : digraph [ ID ] '{' stmt_list '}'
2 stmt_list      : stmt ';' [ stmt_list ]
3 stmt           : node [ "->" edge_list ]
4 node           : ID [ node_attr ]
5 node_attr      : number | operator
6 edge_list      : edge [ ',' edge_list ]
7 edge           : [ edge_prob ] ID
8 edge_prob      : '[' number ']'
9 number         : DIGIT* [ '.' DIGIT* ]
10 operator       : "AND" | "XOR" | "OR"
```

"Criamos uma linguagem simples para descrição textual de workflows que se baseia na linguagem DOT. Como a intenção dessa linguagem é permitir apenas a descrição de grafos que representam experimentos científicos, e não de qualquer tipo de grafo, ela é muito mais concisa do que a linguagem DOT."

Código 1: Gramática abstrata da linguagem de descrição textual dos *workflows*

```
1 digraph workflow1 {
2     A          -> B;
3     B          -> AND1;
4     AND1 [AND]  -> E, OR1;
5     OR1 [OR]    -> [0.15] C, [0.85] D;
6     E [0.5]    -> AND2;
7     C          -> OR2;
8     D          -> OR2;
9     OR2 [OR]    -> AND2;
10    AND2 [AND]  -> F;
11 }
```

Código 2: Exemplo de descrição textual de um *workflow* na linguagem definida

Embora isso possa ser meio intuitivo, acho importante você explicar aqui o significado desses nós "especiais" (AND, OR, XOR) e das probabilidades no grafo de um *workflow*.

4.2 Estrutura de Dados Baseada em Grafo

Utilizando os analisadores léxicos e sintáticos juntamente com a biblioteca *python-graph*, foi possível criar uma estrutura de dados baseada em grafo na memória, que utiliza classes para representar grafos, grafos direcionados e hipergrafos. Quando requisitada a impressão de um grafo para a tela, a biblioteca retorna uma lista contendo os vértices do grafo e outra contendo ^{tuplas} com dois vértices cada, que, ^{no nosso caso} ~~nesses exemplos~~, representam uma aresta direcionada. Para fins de demonstração, incluiremos esta saída aqui.

```
1 [ 'A', 'C', 'B', 'E', 'D', 'F', 'OR2', 'OR1', 'AND1', 'AND2' ]
2 [ ('A', 'B'), ('C', 'OR2'), ('B', 'AND1'), ('E', 'AND2'), ('D', 'OR2'),
3   ('OR2', 'AND2'), ('OR1', 'C'), ('OR1', 'D'), ('AND1', 'E'),
4   ('AND1', 'OR1'), ('AND2', 'F') ]
```

"A seguir, a saída da impressão é ilustrada."

4.3 Linguagem DOT

A partir do grafo ^{em memória do workflow} o programa ~~consegue~~ ^{automaticamente} gerar arquivos ~~externos~~ contendo o código equivalente na linguagem DOT e um *pdf* com a visualização do mesmo.

troque esse nome para "workflow"

```

1 digraph graphname {
2   A [shape=box, attr="1.0"];
3   C [shape=box, attr="1.0"];
4   B [shape=box, attr="1.0"];
5   E [shape=box, attr="0.5"];
6   D [shape=box, attr="1.0"];
7   F [shape=box, attr="1.0"];
8   OR2 [shape=diamond, attr=OR];
9   OR1 [shape=diamond, attr=OR];
10  AND1 [shape=diamond, attr=AND];
11  AND2 [shape=diamond, attr=AND];
12  A -> B [label="1.0"];
13  C -> OR2 [label="1.0"];
14  B -> AND1 [label="1.0"];
15  E -> AND2 [label="1.0"];
16  D -> OR2 [label="1.0"];
17  OR2 -> AND2 [label="1.0"];
18  OR1 -> C [label="0.15"];
19  OR1 -> D [label="0.85"];
20  AND1 -> E [label="1.0"];
21  AND1 -> OR1 [label="1.0"];
22  AND2 -> F [label="1.0"];
23 }

```

Código 3: Exemplo de descrição do *Workflow* em linguagem DOT

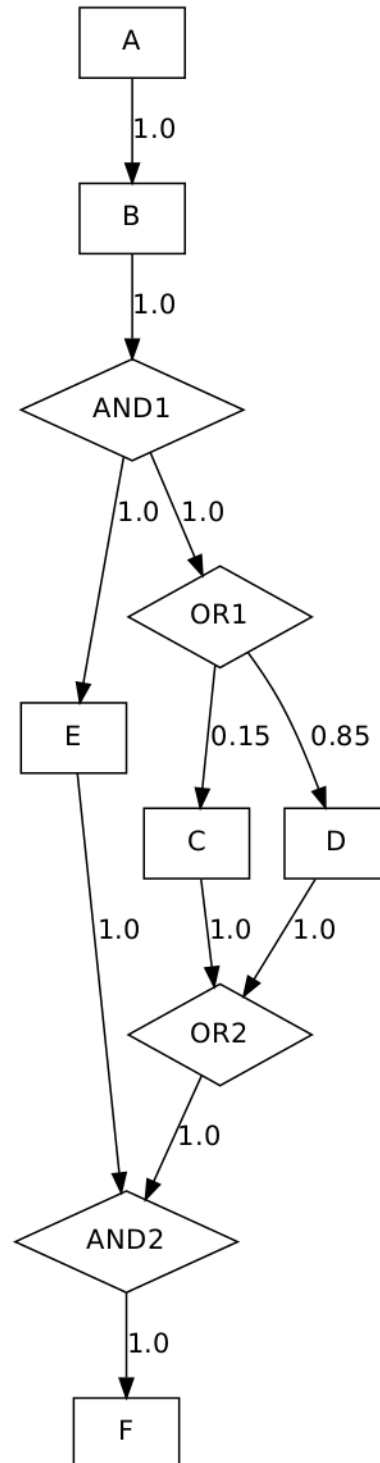


Figura 1: Exemplo de visualização do *workflow* criada a partir do código em DOT


5 Análises

5.1 Descrição Textual do *Workflow*

Como dito anteriormente, a gramática de descrição do *workflow* definida é baseada na linguagem DOT, pois esta é amplamente utilizada na representação de grafos em geral. Porém, essa mesma generalidade implica em maior complexidade sintática, por isso a necessidade da definição de uma nova linguagem, capaz de lidar com as formas mais comuns de *workflows* científicos, mas ainda assim minimalista. Portanto, definimos uma linguagem com as seguintes regras (ver o exemplo do Código 2 para maior esclarecimento):

"Toda descrição textual de workflow deve ser iniciada com ..."

- ~~Texto iniciando~~ com a palavra "digraph", para manter similaridade com a linguagem DOT.
- A seguir, ~~o nome do~~ ^{deve aparecer} *workflow*, que será usado para nomear os arquivos de saída.
- A descrição ~~em si~~ ^{"do workflow propriamente dita fica ..."} fica entre chaves.
- Espaços e tabulações são ignorados.
- Todas as linhas da descrição terminam com ponto e vírgula ";".
- Não há declaração explícita de vértices e arestas, esta é feita de forma implícita.
- Nomes de vértices e arestas devem ser alfanuméricos e podem conter o símbolo *underline* "_", mas não podem iniciar com um número.
- Cada linha representa uma ou mais arestas, utilizando o símbolo "->" (seta) para separar o vértice de partida de seus vértices de destino.
- À esquerda da seta são declarados os vértices de partida com seus respectivos atributos após seu nome e entre colchetes: um número, caso seja uma atividade, representando sua taxa de execução¹¹, ou uma palavra (OR, XOR ou AND), caso seja um operador, representando seu tipo. Cada linha só pode conter um vértice de partida e seu respectivo atributo.
- À direita são declaradas os vértices de destino e seus respectivos atributos, separados por vírgula. Os atributos são números representando a probabilidade daquele caminho (aresta) ser tomado pelos dados e deve estar entre colchetes e antes do nome do vértice. Aqui há uma criação de vértices ainda não declarados, mas sem atributos, que serão adicionados posteriormente quando estes forem declarados como vértices de partida.
- Toda linha deve conter a seta, isto é, não se pode declarar somente um vértice em uma linha. No entanto, na declaração de arestas, o vértice de destino é criado automaticamente. Isto implica no fato de que vértices sem arestas partindo deles não podem ter atributos, exigindo em alguns casos a utilização de um vértice especial para simbolizar o final do *workflow*.

Entretanto, a gramática pode ser mudada eventualmente, caso alguma das partes ainda não implementadas exijam isso. Além disso, ainda não é feita a verificação de erro ^{do programa} sobre o texto, o que pode ser implementado no futuro caso isso não comprometa o cronograma do projeto.  "na descrição textual do workflow"

5.2 Descrição Textual dos Recursos

Nesta etapa, busca-se permitir que o usuário descreva sucintamente os recursos utilizados por cada atividade de seu *workflow* de forma simples e independente, permitindo melhor representar seu experimento sem afetar a abstração da descrição do próprio *workflow*, além de facilitar alterações. Entretanto, por dificuldade de implementação e por não ser fundamental para o projeto, foi decidido adiar esta etapa até o programa estar completo.

Opá, precisamos ser cuidadosos ao dizer isso, afinal essa é sim uma etapa importante do seu projeto. Você pode substituir esse trecho por algo como: "A definição de uma linguagem para a descrição dos requisitos de recursos e a incorporação das informações de recursos no modelo analítico do workflow gerado pelo programa são atividades que estão previstas para serem desenvolvidas no último trimestre do projeto."

Ainda não definido se este atributo representará mesmo a taxa de execução, pois isso só vai afetar o programa no seu final, mas, de qualquer forma, este valor deverá ser uma informação a respeito da velocidade com que a atividade é executada em comparação com as outras.

5.3 Analisadores Léxico e Sintático

Uma vez que foi decidido partir de uma descrição textual do *workflow*, é necessário que o programa seja capaz de ler e interpretar o texto dado. Logo, é necessário o uso de analisadores léxicos e sintáticos, que têm exatamente essa função.

O analisador léxico, ou *lexer*, quebra o texto em pequenos fragmentos, ou *tokens*, seguindo regras definidas pelo usuário. A seguir, passa esses *tokens* ao analisador sintático, ou *parser*, que, também a partir de regras, interpreta o papel de cada *token* na sintaxe geral em relação aos anteriores, executando uma ação específica a cada *token* novo. Apesar de não serem muito simples de se implementar, os analisadores permitem uma grande flexibilidade na definição da gramática do texto.

Escolheu-se, então, utilizar o Lex, *A Lexical Analyzer Generator*, e o Yacc, *Yet Another Compiler-Compiler*¹². Apesar de antigos (1975 e 1970, respectivamente), ainda são amplamente utilizados através de reimplementações e frequentemente juntos. O que é exemplificado pela biblioteca PLY, *Python Lex-Yacc*, uma implementação recente (2001) inteiramente em *Python*, a qual busca entregar toda a funcionalidade do Lex/Yacc somada a uma extensa verificação de erro.

Portanto, ao utilizar a biblioteca PLY, é possível definir uma sintaxe abstrata para o *workflow* independente das outras partes do programa e, ao mesmo tempo, construir o grafo representante em tempo real, isto é, ao longo da leitura do texto, e de forma automática.

5.4 Estrutura de Dados Baseada em Grafo

O motivo pela escolha de se representar o *workflow* por meio de uma estrutura de dados na memória, em forma de grafo, se dá pela generalidade dessa estrutura, que não depende de nenhuma linguagem, facilitando sua manipulação e eventuais traduções para linguagens específicas.

Por o projeto ser em *Python*, era necessária uma biblioteca desta linguagem que trabalhasse com grafos de uma maneira leve e flexível. Foi encontrada, então, a *python-graph*, que preenche esses requisitos e oferece um grande número de algoritmos úteis ao se lidar com grafos. Por ser baseada em classes (por exemplo, a classe *digraph*, que representa um grafo direcionado), sua utilização é muito simples, como demonstrado no seguinte exemplo¹³.

```
1 >>> # Import the module and instantiate a graph object
2 >>> from pygraph.classes.graph import graph
3 >>> from pygraph.algorithms.searching import depth_first_search
4 >>> gr = graph()
5 >>> # Add nodes
6 >>> gr.add_nodes(['X', 'Y', 'Z'])
7 >>> gr.add_nodes(['A', 'B', 'C'])
8 >>> # Add edges
9 >>> gr.add_edge(('X', 'Y'))
10 >>> gr.add_edge(('X', 'Z'))
11 >>> gr.add_edge(('A', 'B'))
12 >>> gr.add_edge(('A', 'C'))
13 >>> gr.add_edge(('Y', 'B'))
14 >>> # Depth first search rooted on node X
15 >>> st, pre, post = depth_first_search(gr, root='X')
16 >>> # Print the spanning tree
17 >>> print st
18 {'A': 'B', 'C': 'A', 'B': 'Y', 'Y': 'X', 'X': None, 'Z': 'X'}
```

Código 4: Exemplo de uso da biblioteca *python-graph* fornecido em sua documentação oficial

¹²<http://dinosaur.compilertools.net/> visualizado em 14 de janeiro de 2014

¹³<http://dl.dropboxusercontent.com/u/1823095/python-graph/docs/index.html> visualizado em 14 de janeiro de 2014

5.5 Linguagem DOT

Além das funcionalidades descritas anteriormente, a biblioteca *python-graph* ainda conta com ferramentas que lidam com a linguagem DOT, um dos motivos pelos quais ela foi escolhida. Associada às bibliotecas *pydot* e *Graphviz*, ela é capaz de traduzir automaticamente um grafo seu para DOT, além de gerar gráficos para visualização desse grafo. Novamente, será utilizado um exemplo para a demonstração, desta vez adaptado de sua documentação oficial¹⁴.

```
1 # 'gr' é uma instância da classe graph
2
3 # traduzir o grafo 'gr' para DOT
4 dot = write(gr)
5
6 # gravar a visualização gráfica no arquivo externo 'europe.png'
7 gvv = gv.readstring(dot)
8 gv.layout(gvv, 'dot')
9 gv.render(gvv, 'png', 'europe.png')
```

Código 5: Exemplo adaptado de uso das ferramentas para DOT da biblioteca *python-graph*

No exemplo, o formato de imagem utilizado é o *png*, porém neste projeto foi adotado o formato *pdf*, por sua qualidade, versatilidade e facilidade de inclusão em documentos.

Apesar de algumas restrições, como a ausência de atributos dos vértices nas imagens (Figura 1), ainda é vantajosa sua utilização, pois suas funções são relativamente simples de implementar e dão ao usuário a possibilidade de verificar se a estrutura em memória corresponde a seu *workflow* original.

5.6 Modelagem Analítica

Uma vez que já existe o grafo na memória, só resta sua tradução para um modelo de modelagem analítica^a e então efetuar a análise numérica. Embora esta etapa ainda não tenha sido implementada, algumas escolhas já foram feitas, como, por exemplo, o modelo de álgebra de processos estocásticos^a, por apresentar vantagens em relação a outros modelos, dentre as quais as mais importantes são¹⁵.

- *Composicionalidade*: a habilidade de modelar um sistema como a interação de subsistemas.
- *Formalismo*: dar um significado preciso para todos os termos na linguagem.
- *Abstração*: a habilidade de construir modelos complexos a partir de componentes detalhadas, desconsiderando os detalhes quando apropriado.

Dentre as linguagens disponíveis, será usada a PEPA, uma álgebra de processos estocásticos^a bem desenvolvida e que conta com várias ferramentas de apoio, como um complemento para o ambiente integrado de desenvolvimento *Eclipse* e, recentemente, uma biblioteca para a linguagem *Python*, a *pyPEPA*. Essa biblioteca, porém, ainda está em desenvolvimento e, portanto, não apresenta todas as funcionalidades da PEPA. Sendo assim, há a possibilidade dela não corresponder aos objetivos do projeto e, neste caso, a ferramenta *Eclipse* teria de ser utilizada, comprometendo a automatização do processo, mas ainda assim permitindo obter os resultados de predição.

No entanto, a tradução para PEPA pode se provar complicada, pois envolve uso de algoritmos para percorrer o grafo e estruturas de dados como pilhas e filas, cuja aplicação neste contexto não é muito comum. A análise numérica, em contrapartida, espera-se que seja relativamente mais fácil, por já ter sido desenvolvida com esse objetivo.

¹⁴<https://code.google.com/p/python-graph/wiki/Example> visualizado em 14 de janeiro de 2014

¹⁵<http://www.dcs.ed.ac.uk/pepa/about/> visualizado em 14 de janeiro de 2014

Em textos técnicos, a melhor estratégia de escrita é a sobriedade: "A Tabela 1 mostra as atividades já concluídas no projeto. A mesma tabela mostra as atividades previstas para os próximos seis meses do projeto."

5.7 Cronograma

~~Por ser um relatório parcial,~~ **Muito informal!** vale a apresentação do cronograma do projeto, explicitando as etapas completas e as futuras. Embora não haja registro da data exata do término de cada etapa, a Tabela 1 as dispõe em ordem cronológica. a data exata não precisa, mas o mês de início e fim precisa sim!

<i>Etapas</i>	<i>Status</i>
Estudo de <i>Python</i>	Completa
Descrição Textual do <i>Workflow</i>	Completa
Analísadores Léxico e Sintático	Completa
Estrutura de Dados Baseada em Grafo na Memória	Completa
Tradução da Estrutura para a Linguagem DOT	Completa
Criação de Gráficos a partir do Código DOT	Completa
Tradução da Estrutura para a Linguagem PEPA	Não iniciada
Análise Numérica do Modelo em PEPA	Não iniciada
Descrição Textual dos Recursos	Não iniciada
Obtenção de Resultados de Predição com Recursos	Não iniciada
Extensão do Programa para <i>Workflows</i> mais Complexos	Não iniciada

Indicar que essa é "opcional"!

Tabela 1: Cronograma resumido das etapas do projeto

Lucas, no cronograma você precisa dar uma estimativa de tempo (pode ser em meses) para a realização de cada etapa ainda não concluída. Você pode fazer isso incluindo 2 colunas adicionais na sua tabela: "mês de início" e "mês de fim".

6 Conclusões parciais

Apesar de não estar completo, o programa, a partir de uma descrição textual simples de um *workflow*, já é capaz de automaticamente gerar um grafo em *Python*, além do código equivalente na linguagem DOT e um gráfico com a representação do *workflow*, o que é útil na verificação de erros. Existem também as dificuldades já discutidas em relação à linguagem PEPA, como também outras que podem surgir nas etapas posteriores, que são difíceis de se analisar agora, por exigirem alterações no programa atual.

De qualquer forma, espera-se que todo o cronograma seja cumprido, podendo, talvez, haver tempo para implementar funções extras, como a verificação de erros.

na descrição textual do workflow.

Essa seção de conclusão ficou bem estranha!

Você pode dizer as mesmas coisas, mas de forma mais "sofisticada":

"Neste projeto, nos propomos a desenvolver um programa que, a partir de uma descrição textual de um workflow e dos recursos usados por ele, gere índices preditivos do desempenho do referido workflow.

Os seis primeiros meses do projeto foram dedicados a:

- 1) O estudo dos conceitos relacionados ao tema do projeto, como "workflows científicos", "linguagens de modelagem de workflows" e "modelos estocásticos", como a PEPA.
- 2) Criação de um algoritmo para a conversão de um grafo de workflow para um modelo analítico em PEPA
- 3) Criação de um lexer e um parser ...
- 4) ...
- 5)

A escolha de Python como linguagem de implementação para o projeto se mostrou bastante acertada, já que as estruturas de dados e bibliotecas existentes nessa linguagem agilizaram o desenvolvimento das funcionalidades primárias do programa.

Os principais desafios para os próximos do projeto são:

- 1) Criação de um algoritmo para a conversão de um grafo de workflow para um modelo analítico em PEPA
- 2) Incorporação de informações sobre recursos nos modelos analíticos dos workflows
- 3) O uso da biblioteca pyPepa para a obtenção da solução numérica dos modelos analíticos "

Referências

- [1] Gadelha, L., *Gerência de Proveniência em Workflows Científicos Paralelos e Distribuídos*, Tese de Doutorado, COPPE, UFRJ, 2012.
- [2] Ogasawara, E., *Uma Abordagem Algébrica para Workflows Científicos com Dados em Larga Escala*, Tese de Doutorado, COPPE, UFRJ, 2011.
- [3] Braghetto, K., *Técnicas de Modelagem para a Análise de Desempenho de Processos de Negócio*, Tese de Doutorado, IME, USP, 2011.

A Avaliação do Orientador

B Código do Programa

```
1 import gv
2 import sys
3 import ply.lex as lex
4 import ply.yacc as yacc
5 import pygraph.mixins.labeling
6 from pygraph.readwrite.dot import write
7 from pygraph.classes.digraph import digraph
8
9 # Lista de tokens
10 reserved = {
11     'digraph' : 'DIGRAPH',
12     'OR' : 'OR',
13     'XOR' : 'XOR',
14     'AND' : 'AND'
15 }
16
17 tokens = [
18     'OPEN_BRACE',
19     'CLOSE_BRACE',
20     'OPEN_BRACKET',
21     'CLOSE_BRACKET',
22     'EDGEOP',
23     'ID',
24     'NUMBER',
25     'COMMA',
26     'SEMICOLON'
27 ] + list(reserved.values())
28
29 # Regras de expressão regular para tokens simples
30 t_OPEN_BRACE = r'\{'
31 t_CLOSE_BRACE = r'\}'
32 t_OPEN_BRACKET = r'\['
33 t_CLOSE_BRACKET = r'\]'
34 t_EDGEOP = r'>'
35 t_COMMA = r','
36 t_SEMICOLON = r';'
37
38 # Regras de expressões regulares com alguma ação
39 def t_ID(t):
40     r'[a-zA-Z_][a-zA-Z_0-9]*'
41     t.type = reserved.get(t.value, 'ID')
42     return t
43
44 def t_NUMBER(t):
45     r'\d+|\.\d+|\d+'
46     t.value = float(t.value)
47     return t
48
49 # Definir uma regra pra registrar o número de linhas
50 def t_newline(t):
```

```

51     r'\n+'
52     t.lexer.lineno += len(t.value)
53
54     # Uma string contendo caracteres ignorados (espaços e tabs)
55     t_ignore = '\t'
56
57     # Regra para tratamento de erro
58     def t_error(t):
59         print "Illegal_character_%s" % t.value[0]
60         t.lexer.skip(1)
61
62     # Compilar o analisador léxico
63     lexer = lex.lex()
64
65     # Obter dados do arquivo
66     arq = open(sys.argv[1], "r")
67     data = arq.read()
68
69     # Dar uma entrada para o analisador léxico
70     lexer.input(data)
71
72     # Criar o digrafo
73     dgr = digraph()
74
75     # Expressões do analisador sintático
76
77     # Definir nome do digrafo e o atribuir à variável global 'dgrname'
78     def p_digraph_id(t):
79         'digraph_: _DIGRAPH_ID_OPEN_BRACE_stmt_CLOSE_BRACE'
80         global dgrname
81         dgrname = t[2]
82
83     def p_digraph_no_id(t):
84         'digraph_: _DIGRAPH_OPEN_BRACE_stmt_CLOSE_BRACE'
85         global dgrname
86         dgrname = 'workflow'
87
88
89     def p_stmtlist(t):
90         'stmt_: _stmt_stmt'
91
92     # Criar as arestas
93     def p_stmt(t):
94         'stmt_: _node_edge_SEMICOLON'
95         for edge in t[2]:
96             dgr.add_edge((t[1]['name'], edge['node']), 1, edge['label'])
97
98
99     # Criar cada vértice ou atualizar seus atributos caso este já exista
100    def p_node_attr_op_or(t):
101        'node_: _ID_OPEN_BRACKET_OR_CLOSE_BRACKET_EDGEOP'
102        t[0] = {'name': t[1], 'attr': 'OR'}
103        if dgr.has_node(t[1]):
104            dgr.add_node_attribute(t[1], ('attr', 'OR'))
105            dgr.add_node_attribute(t[1], ('shape', 'diamond'))

```

```

106         else:
107             dgr.add_node(t[1], [('attr', 'OR'), ('shape', 'diamond')])
108
109     def p_node_attr_op_xor(t):
110         'node_: _ID_OPEN_BRACKET_XOR_CLOSE_BRACKET_EDGEOP'
111         t[0] = {'name': t[1], 'attr': 'XOR'}
112         if dgr.has_node(t[1]):
113             dgr.add_node_attribute(t[1], ('attr', 'XOR'))
114             dgr.add_node_attribute(t[1], ('shape', 'diamond'))
115         else:
116             dgr.add_node(t[1], [('attr', 'XOR'), ('shape', 'diamond')])
117
118     def p_node_attr_op_and(t):
119         'node_: _ID_OPEN_BRACKET_AND_CLOSE_BRACKET_EDGEOP'
120         t[0] = {'name': t[1], 'attr': 'AND'}
121         if dgr.has_node(t[1]):
122             dgr.add_node_attribute(t[1], ('attr', 'AND'))
123             dgr.add_node_attribute(t[1], ('shape', 'diamond'))
124         else:
125             dgr.add_node(t[1], [('attr', 'AND'), ('shape', 'diamond')])
126
127     def p_node_attr_num(t):
128         'node_: _ID_OPEN_BRACKET_NUMBER_CLOSE_BRACKET_EDGEOP'
129         t[0] = {'name': t[1], 'attr': t[3]}
130         if dgr.has_node(t[1]):
131             dgr.add_node_attribute(t[1], ('attr', t[3]))
132             dgr.add_node_attribute(t[1], ('shape', 'box'))
133         else:
134             dgr.add_node(t[1], [('attr', t[3]), ('shape', 'box')])
135
136     def p_node_attr_none(t):
137         'node_: _ID_EDGEOP'
138         t[0] = {'name': t[1], 'attr': 1.0}
139         if dgr.has_node(t[1]):
140             dgr.add_node_attribute(t[1], ('attr', 1.0))
141             dgr.add_node_attribute(t[1], ('shape', 'box'))
142         else:
143             dgr.add_node(t[1], [('attr', 1.0), ('shape', 'box')])
144
145     def p_edge_list(t):
146         'edge_: _edge_COMMA_edge'
147         t[0] = t[1] + t[3]
148
149     # Criar vértices ainda não existentes para
150     # posteriormente as arestas poderem ser criadas
151     # Atributos serão adicionados depois
152     def p_edge_prob(t):
153         'edge_: _OPEN_BRACKET_NUMBER_CLOSE_BRACKET_ID'
154         t[0] = [{'node': t[4], 'label': t[2]}]
155         if not dgr.has_node(t[4]):
156             dgr.add_node(t[4])
157
158     def p_edge_no_prob(t):
159         'edge_: _ID'
160         t[0] = [{'node': t[1], 'label': 1.0}]

```



```

161         if not dgr.has_node(t[1]):
162             dgr.add_node(t[1])
163
164     # Tratamento de erro
165     def p_error(t):
166         print("Syntax_error_at_ '%s' " % t.value)
167
168     # Compilar o analisador sintático
169     yacc.yacc()
170     yacc.parse(data)
171
172     # Atualizar vértices sem atributos para equivaler a um vértice padrão
173     for node in dgr.nodes():
174         if dgr.node_attributes(node) == []:
175             dgr.add_node_attribute(node, ('attr', 1.0))
176             dgr.add_node_attribute(node, ('shape', 'box'))
177
178     # Escrever para a linguagem DOT
179     dot = write(dgr)
180
181     # Gravar código em DOT em arquivo externo
182     dot_file = open(dgrname + '.dot', 'w')
183     dot_file.write(dot)
184     dot_file.close()
185
186     # Gravar visualização gráfica em arquivo pdf externo
187     gvv = gv.readstring(dot)
188     gv.layout(gvv, 'dot')
189     gv.render(gvv, 'pdf', dgrname + '.pdf')

```
