RESATYRUS: GERAÇÃO AUTOMATIZADA DE GRAFOS DE COMPARTILHAMENTO

Daniel Santos Ferreira Alves

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Priscila Machado Vieira Lima

Rio de Janeiro
Maio de 2014

RESATYRUS: GERAÇÃO AUTOMATIZADA DE GRAFOS DE COMPARTILHAMENTO

Daniel Santos Ferreira Alves

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Priscila Machado Vieira Lima, Ph.D.

Prof. Nelson Maculan Filho, D.H.R.

Prof. Luiz Marcos Garcia Gonçalves, Ph.D.

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 2014

Alves, Daniel Santos Ferreira

ReSATyrus: Geração Automatizada de Grafos de Compartilhamento/Daniel Santos Ferreira Alves. – Rio de Janeiro: UFRJ/COPPE, 2014.

XIII, 74 p.: il.; 29, 7cm.

Orientadores: Felipe Maia Galvão França
Priscila Machado Vieira Lima

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Referências Bibliográficas: p. 66 – 68.

1. Escalonamento por reversão de arestas. 2. Compartilhamento de recursos. 3. Agentes inteligentes. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Agradecimentos

Gostaria de agradecer à minha família, a meus orientadores, meus professores, meus amigos, meus colegas e todos que me ajudaram neste caminho.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

RESATYRUS: GERAÇÃO AUTOMATIZADA DE GRAFOS DE COMPARTILHAMENTO

Daniel Santos Ferreira Alves

Maio/2014

Orientadores: Felipe Maia Galvão França
Priscila Machado Vieira Lima

Programa: Engenharia de Sistemas e Computação

Propomos nesta dissertação um gerador de dinâmicas de escalonamento por reversão de arestas (*Scheduling by Edge Reversal – SER*) e de escalonamento por reversão de múltiplas arestas (*Scheduling by Multiple Edge Reversal – SMER*), o ReSATyrus. O ReSATyrus possibilita gerar grafos utilizados pela modelagem *SER*/*SMER* a partir de uma descrição de compartilhamento dos recursos envolvidos em linguagem mais compacta que os grafos resultantes. Este estudo é acompanhado de um exemplo de aplicação do *SER* como sistema de coordenação para um conjunto de agentes em movimento, que serve ainda como teste para as capacidades do ReSATyrus. Finalmente, também é desmonstrado como obter orientações iniciais para o *SMER* a partir de uma orientação válida para o *SER*, assim como um estudo básico dos resultados de usar as principais heurísticas para gerar orientação acíclica do *SER*. Este mecanismo é interessante por poder ser utilizado de forma distribuída, que é o foco de aplicação do SMER.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

RESATYRUS: AUTOMATIC GENERATION OF SHARING GRAPHS

Daniel Santos Ferreira Alves

May/2014

Advisors: Felipe Maia Galvão França
Priscila Machado Vieira Lima

Department: Systems Engineering and Computer Science

In this work, we present a generator of dynamics for use with the mechanisms of Scheduling by Edge Reversal (SER) and Scheduling by Multiple Edge Reversal (SMER), the ReSATyrus. The ReSATyrus allows the creation of graphs used for SER/SMER modelling from a resource sharing description in a language that is more compact than the resulting graphs. This study follows with an example of the application of the SER as a coordination system for a group of moving agents, which also works as a test for the capabilities of the ReSATyrus. Finally, we also show how to generate initial setups for the SMER from a valid initial setup for the SER, as well as a basic study of the results from the use of the main heuristics used to create the initial of the SER. This mechanism is interesting as it can be used in a distributed form, where lies the main uses for the SMER.

Sumário

Lista de Figuras

Lista de Símbolos

D	digrafo gerado a cada etapa do *SER*, p. 5
D^M	multidigrafo gerado a cada etapa do *SMER*, p. 21
G	grafo conexo usado como base para o *SER*, p. 9
G^M	multigrafo usado como base para o *SMER*, p. 21
V^+	agentes que esperam prioridade do agente, também chamados de *vizinhos de entrada*, p. 5
V^-	agentes que possuem prioridade sobre o agente, também chamados de *vizinhos de saída*, p. 5
$Vizinhos$	agentes com os quais um agente mantém comunicações, p. 5
α	tempo total de atuação de um agente, p. 10
$\emptyset$	conjunto vazio, p. 5
λ	número-λ de um nó. Nós com mesmo número-λ estão na mesma camada-λ, p. 7
$\leftarrow$	atribuição de valor, p. 5
π	tempo de duração de um período *SER* ou *SMER*, p. 9
τ	tempo total de atividade de uma tarefa dentro de uma dinâmica *SER* ou *SMER*, p. 10
$\varnothing$	valor nulo para variável, p. 5
a_{xy}	quantidade de arestas no multidigrafo D^M orientadas de x para y, p. 21
e_y	equivalente a $a_y v$, quando v é o nó respondendo a um evento em um algoritmo, p. 5

e_{xy}	total de arestas entre x e y no multidigrafo D^M, p. 19
r	equivalente a r_v, quando v é o nó respondendo a um evento em um algoritmo, p. 5
r_v	reversibilidade de um vértice v, p. 19
t	tempo de atividade total *SER* ou *SMER*, p. 10

Capítulo 1

Introdução

Propomos neste trabalho uma plataforma para a modelagem de sistemas baseados em compartilhamento de recursos. Esta plataforma permite descrever sistemas que disputam recursos de forma concisa e não ambígua. Com o uso da plataforma pode-se facilitar o estudo dos sistemas envolvidos ou automatizar a geração de sistemas com compartilhamento de recursos.

1.1 Motivação

O compartilhamento de recursos é um problema comum em aplicações distribuídas. Exemplos de compartilhamento incluem acesso a arquivos ou uso de máquinas em uma linha de produção. Um problema clássico para ilustrar as dificuldades envolvidas é o jantar dos filósofos [1].

Neste problema um grupo de filósofos está sentado a mesa para jantar e pensar. Há um garfo por filósofo, posicionados entre cada dois filósofos, mas cada filósofo necessita de dois garfos para comer. Soluções simples, por exemplo, agarrar primeiro o garfo à esquerda depois o garfo à direita, comer e devolvê-los na mesma ordem, podem resultar em complicações adicionais. Neste caso, se todos os filósofos agarrem simultaneamente o garfo à esquerda, não haverá garfos à direita para serem agarrados e nenhum poderá comer. Outra situação que deve ser evitada é o caso em que um filósofo nunca consegue comer pois seus vizinhos sempre impedem seu acesso total aos garfos.

Um mecanismo que oferece uma solução evitando essas complicações é o escalonamento por reversão de arestas (*Scheduling by Edge Reversal – SER*) [2]. O *SER* trata particularmente da situação em que os filósofos só jantam, sem pausas para pensar. Ele possui ainda uma variante, o escalonamento por reversão de múltiplas arestas (*Scheduling by Multiple Edge Reversal – SMER*) [3], que lida com uma variante adicional do problema inicial, em que alguns filósofos desejam comer menos que outros. Ambos os mecanismos podem lidar com variações do problema inicial,

dada a modelagem adequada.

Tanto o *SER* como o *SMER* utilizam um grafo para especificar as restrições de compartilhamento de recursos entre participantes. A composição do grafo de restrições, contudo, pode apresentar complicações à medida que aumentam a quantidade de participantes e recursos envolvidos. Embora existam relações conhecidas que podem simplificar a descrição do modelo, não há um sistema formal para descrever por meio dessas relações.

O *SMER* possui ainda poucos métodos para inicialização distribuída [3, 4]. É importante diversificar os métodos para inicializar o *SMER* para permitir a adaptação para diferentes situações. O próprio *SER* possui diferente heurísticas para inicialização que contribuem para um ajuste da concorrência no compartilhamento de recursos conforme a necessidade do problema.

1.2 Metodologia

O trabalho foi desenvolvido inicialmente com estudo da literatura existente sobre compartilhamento de recursos com *SER* e com *SMER*, em particular em sistemas de manufatura. Ao estudo seguiu-se experimentação da aplicação do SER para controle de veículos. A partir da necessidade de testar diferentes modelos para o sistema de controle foi desenvolvido o ReSATyrus.

Mais experimentos foram realizados, estes utilizando o ReSATyrus. Para que o ReSATyrus pudesse gerar orientações para o *SMER* foi desenvolvido ainda o multi-Alg, baseado em uma heurística apresentada no trabalho original do *SMER* que não pode ser utilizada de forma automatizada. O multi-Alg também foi testado para comparar-se com a outra heurística disponível para *SMER*.

1.3 Contribuições

Este trabalho introduz a plataforma ReSATyrus para tratamento de dinâmicas *SER* e *SMER* utilizando uma representação baseada em relações semelhantes às existentes em lógica Booleana. A representação deve ser de tal modo que o grafo correspondente possa ser gerado a partir dela. Tal grafo deve ainda ser único por representação, isto é, esta não pode ser ambígua. Para demonstrar a viabilidade dessa representação este trabalho apresenta ainda uma aplicação desta plataforma para coordenação de veículos em movimento.

Além disso é introduzido um novo método para inicializar o *SMER*. Este método busca aproveitar as ideias por trás dos métodos de inicialização do *SER* de modo a poder adaptá-los para o *SMER*. Este trabalho apresenta ainda testes para comparar com métodos alternativos já existentes.

1.4 Aplicações

O ReSATyrus pode ser aplicado para estudo de modelagens de compartilhamento de recursos e para geração automatizada de modelos. Ele foi usado da primeira forma durante este trabalho para testar diferentes ideias para aplicação do *SER* como sistema de controle de veículos e depois como parte de um sistema de controle de veículos longos. Neste segundo caso o ReSATyrus foi empregado para gerar de forma automatizada dinâmicas diferentes para alterações nas rotas dos veículos.

1.5 Estrutura do documento

Esta dissertação está organizada da seguinte forma: O Capítulo 2 apresenta os fundamentos teóricos e ferramentas utilizadas. Os fundamentos incluem os mecanismos *SER* e *SMER* que serviram como base para os trabalhos desenvolvidos e o SATyrus, que serviu como inspiração para o ReSATyrus. As ferramentas incluem o Graphviz, que é importante para a visualização dos grafos gerados, e o NetLogo, utilizado para implementação de exemplos dos estudos. O Capítulo 3 apresenta o ReSATyrus, uma ferramenta para geração de grafos para uso com *SER* ou *SMER* a partir de descrições de processos. O Capítulo 4 contém a apresentação do multi-Alg, uma heurística para gerar orientações compatíveis com o *SMER* a partir de orientações acíclicas. Este capítulo contém ainda resultados de testes de medida de concorrência finalmente, o Capítulo 5 contém os conclusões e possíveis continuações dos trabalhos.

Além disso, este documento apresenta alguns apêndices: o primeiro, Apêndice A, trata dos trabalhos desenvolvidos durante este período de estudos; segue o Apêndice B, sobre a gramática utilizada pelo programa ReSATyrus, apresentado no Capítulo 3, e o Apêndice C, sobre o uso do ReSATyrus.

Capítulo 2

Fundamentos Teóricos e Ferramentas

SER e *SMER* são dois mecanismos desenvolvidos para permitir o uso de recursos em ambientes com concorrência, garantindo que todos os participantes terão acesso justo e não ocorrerão travamentos por má distribuição de recursos. O *SER* foi desenvolvido, inicialmente, focalizando a distribuição igualitária. O *SMER* é uma modificação do *SER* implementando diferentes taxas de acesso aos recursos para os participantes.

Ambos os mecanismos lidam com o compartilhamento de recursos através de uma modelagem em grafo das partes envolvidas. As partes envolvidas são chamadas de agentes e recursos. Os agentes realizam ações para as quais eles necessitam de um subconjunto dos recursos disponíveis. Uma entidade com uma ou mais ações possíveis é denominada um processo e os agentes correspondentes a suas ações são chamados de subprocessos e podem compartilhar recursos entre si.

2.1 Algoritmos

A linguagem adotada para algoritmos nesta dissertação enfoca a representação orientada a eventos de processos com múltiplos agentes. Cada agente executará sua cópia do algoritmo em simultâneo com outros, com diferenças apenas nos valores das variáveis disponíveis e nas ocorrências de eventos. Os algoritmos apresentados normalmente executam de forma assíncrona, mas exemplos podem usar sincronia para simplificar a explicação.

Cada algoritmo contém duas partes principais, declaração de variáveis e eventos. A declaração de variáveis atribui os valores iniciais para as variáveis de cada agente, esses valores podem variar por agente. Eventos consistem de duas partes: primeiro a entrada, que indica a condição do evento, usualmente a chegada de alguma men-

sagem, seguida por ações desencadeadas por esse evento. Seguem os símbolos mais utilizados na representação dos algoritmos:

$Vizinhos$ agentes com os quais um agente mantém comunicações;

V^+ agentes que esperam prioridade do agente, também chamados de *vizinhos de entrada*;

V^- agentes que possuem prioridade sobre o agente, também chamados de *vizinhos de saída*;

r equivalente a r_v, quando v é o nó respondendo a um evento;

e_y equivalente a $a_y v$, quando v é o nó respondendo a um evento;

$\varnothing$ valor nulo para variável;

$\emptyset$ conjunto vazio;

$\leftarrow$ atribuição de valor;

2.2 Escalonamento por reversão de arestas (*SER*)

O escalonamento por reversão de arestas (*Scheduling by Edge Reversal – SER*) é um mecanismo desenvolvido para organizar concorrência por recursos [2]. O *SER* foi desenvolvido para situações de concorrência em alta carga, isto é, onde os processos participantes desejam sempre ter acesso aos recursos. Ele funciona com os processos gerando uma alternância para o acesso.

Para gerar a dinâmica de compartilhamento o *SER* parte de um digrafo acíclico D. Na descrição original, cada processo corresponde a um vértice v no grafo, com arcos ligando processos que compartilham um ou mais recursos. O sentido do arco indica qual processo tem prioridade no acesso aos recursos.

Como D é acíclico, necessariamente pelo menos um processo terá prioridade sobre todos os vizinhos. Esse tipo de nó é denominado *sorvedouro* (*sink*). Ele poderá então operar sobre os recursos por um tempo, revertendo as arestas ao final deste, resultando em um novo digrafo. Dessa forma ele perde prioridade sobre os recursos, tornando-se uma *fonte* (*source*) e seus vizinhos ganham. Novos processos poderão então atuar, perpetuando a concorrência. A Figura 2.1 mostra um exemplo.

Usos mais complexos do *SER* podem considerar cada processo como tendo diversos subprocessos correspondentes a diferentes comportamentos. Esses usariam grupos de recursos diferentes, mas não necessariamente concorreriam entre si. Nesse caso, cada processo corresponde a um subconjunto dos vértices de D.

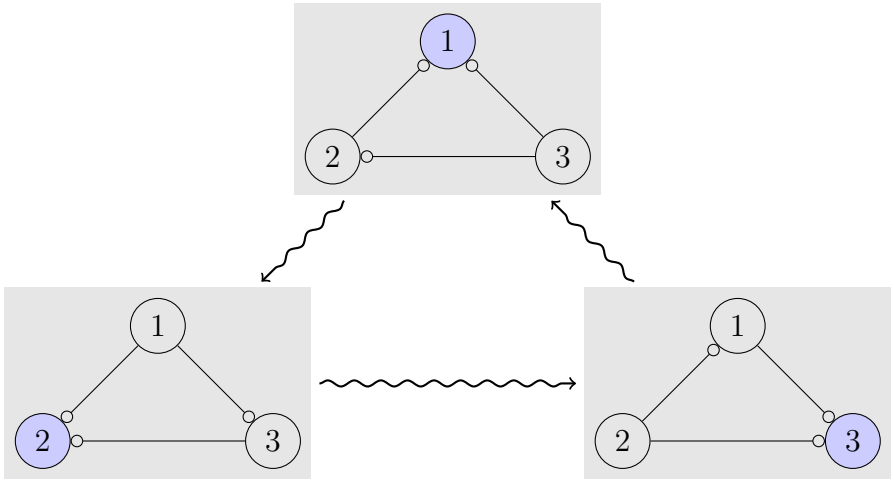

Figura 2.1: Exemplo de execução do *SER*

2.2.1 Algoritmo de reversão de arestas

Partindo do digrafo D com uma orientação inicial acíclica, o Algoritmo 2.1 representa a dinâmica de reversão de arestas.

Algoritmo 2.1: Escalonamento por reversão de arestas

Variáveis:

V^+, inicializado pela orientação

V^-, inicializado pela orientação

Entrada:

$msg_i = \emptyset$, $V^- = \emptyset$

Ação:

Opera

Para cada $v \in V^+$**:**

Enviar reverte **para** v

$V^- \leftarrow V^+$

$V^+ \leftarrow \emptyset$

Entrada:

$msg_i =$ reverte, origem$(msg_i) = v$

Ação:

$V^+ \leftarrow V^+ \cup \{v\}$

$V^- \leftarrow V^- \setminus \{v\}$

2.2.2 Conceitos

Nesta seção serão apresentados conceitos importantes para a compreensão do SER. Esses conceitos abrangem número-λ e camada-λ, a prova da corretude da prevenção de travamentos e ausência de acesso a recursos, assim como a repetição de orientações.

2.2.2.1 Camadas-λ

Cada nó no digrafo D tornar-se-á sorvedouro (sem arestas de saída) após um certo número de reversões. Esse número pode ser medido pelo maior caminho possível do nó até um sorvedouro qualquer. Chamamos esse número de λ. Assim sendo, todo sorvedouro tem naturalmente número-λ igual a 0. Outros nós têm então número-λ igual ao maior dentre os vizinhos de saída mais um. O Algoritmo 2.2 apresenta uma forma simples para calcular números-λ.

Algoritmo 2.2: Cálculo de Número-λ

Variáveis:

V^+, inicializado pela orientação
V^-, inicializado pela orientação
$espera \leftarrow |V^-|$
$\lambda \leftarrow \varnothing$

Entrada:

$msg_i = \emptyset$, $V^- = \emptyset$
Ação:
$\lambda \leftarrow 0$
Para cada $v \in V^+$**:**
Enviar lambda(λ) **para** v

Entrada:

$msg_i = \text{lambda}(\lambda_i)$
Ação:
$espera \leftarrow espera - 1$
Se $\lambda_i \geq \lambda$**:**
$\lambda \leftarrow \lambda_i + 1$
Fim Se
Se $espera = 0$**:**
Para cada $v \in V^+$**:**

Enviar lambda(λ) **para** v
Fim Se

Com os números λ calculados é possível separar os nós em grupos que tenham o mesmo valor. A Figure 2.2 ilustra esse conceito. A esses grupos chamamos *camadas* λ. Pela própria natureza do cálculo, não há nós adjacentes na mesma camada. Considerando o caso em que todos os sorvedouros revertem suas arestas simultaneamente, nós na mesma camada permanecem na mesma camada até tornarem-se sorvedouros, assumindo que todos os sorvedouros sempre revertam suas arestas simultaneamente.

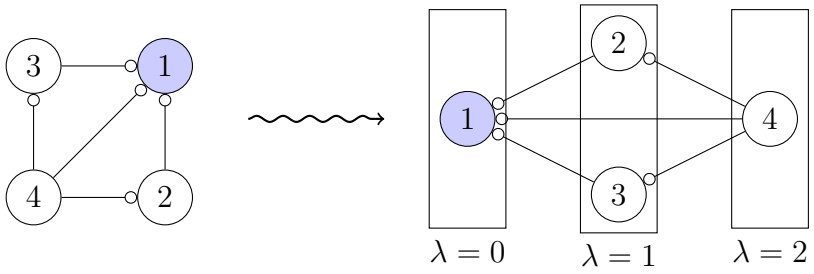

Figura 2.2: Decomposição em camadas-λ.

2.2.2.2 Corretude

Um problema comum em algoritmos distribuídos é haver travamentos (*deadlocks*) ou ausência de acesso a recursos (*starvation*). Travamentos ocorrem quando há um ciclo de dependências, isto é, pelo menos um processo espera uma liberação de recursos a qual depende de liberação pelo próprio para ocorrer. Já o problema de ausência de acesso ocorre quando o sistema não oferece garantias de que todos os processos poderão acessar os recursos necessários em tempo finito. Os Teoremas 1 e 2 demonstram que o *SER* garante a não ocorrência desses fenômenos.

Teorema 1. *SER impede travamentos*

Demonstração. Prova por absurdo. Assume-se que de uma orientação acíclica através da aplicação de reversão de arestas pode-se chegar a um ciclo, caracterizando travamento. Como o processo de reversão modifica apenas as arestas conectadas a um sorvedouro, para ocorrer a formação do ciclo, em um certo momento um digrafo D deve possuir pelo menos um sorvedouro u que fará parte do novo ciclo. O processo de reversão, contudo, troca todas as arestas que entravam em u por arestas que saem de u. Logo, não é possível que um caminho direcionado passe através de u completando o ciclo. Assim sendo, o processo de reversão não pode formar ciclos com os sorvedouros, resultando em absurdo. □

Teorema 2. *SER promove acesso a recursos a todos os nós*

Demonstração. Prova por indução em λ.

Caso base ($\lambda = 0$)**:** O nó é um sorvedouro, portanto tem acesso a recursos.

Caso geral: Assumindo que funciona para todo $\lambda \leq i$, mostraremos que funciona para $\lambda = i + 1$.

Dado o mecanismo de cálculo para o λ de um nó v (Algoritmo 2.2), todos os seus vizinhos possuirão λ diferentes, isto é, nenhum vizinho está na mesma camada-λ. Todos os vizinhos com λ maior que o do nó esperam receber prioridade dele, enquanto ele, por sua vez, espera os vizinhos com λ menor. Pela hipótese do caso geral, todos os vizinhos com λ menor terão acesso aos recursos. Após esse acesso, eles reverterão as arestas e esperarão pelo acesso de seus vizinhos antes de repetir. Assim, depois que um vizinho com λ menor tiver acesso a recursos, ele reverterá as suas arestas, cedendo a prioridade a v. Como todos os vizinhos com λ menor que o de v terão acesso, todos eles cederão prioridade, então v por sua vez também terá acesso aos recursos.

Seguindo a indução proposta, todos os nós em todas as camadas-λ terão acesso. □

2.2.2.3 Períodos

O processo de reversão de arestas aplicado a uma orientação acíclica resulta em uma nova orientação acíclica para o grafo. Considerando, para simplificação, que todos os nós na mesma camada-λ revertem suas arestas simultaneamente, cada orientação resulta em uma única outra orientação. O caso genérico, com reversões não sincronizadas, permite atingir um conjunto de orientações a partir de cada orientação. Após um número finito de reversões, ocorrerá a repetição de orientações previamente utilizadas.

No caso genérico, com reversões não sincronizadas, também ocorre repetição de orientações, com argumentação semelhante, mas não há garantias de formar um ciclo. Para o caso simplificado, a repetição resulta em um conjunto de orientações que ocorrem em uma ordem definida, uma após a outra. Esse ciclo é o chamado período, com comprimento π. A partir do período pode ser calculada a concorrência da orientação, a ser explicada a seguir.

Teorema 3. *SER promove a formação de períodos*

Demonstração. Prova por absurdo. Assume-se que existe alguma orientação acíclica inicial ω' do grafo G que não resulta na formação de um período. Considere um digrafo $\Gamma(\Omega, \Theta)$, onde cada $\omega \in \Omega$ corresponde a uma orientação acíclica possível

de G, e existe $(\omega, \nu) \in \Theta$ se e somente se a reversão θ aplicada em ω resulta em ν. Como as reversões ocorrem em simultâneo, então cada ω possui no máximo uma aresta de saída. Como G é finito, Γ também é finito.

Para não haver período, é necessário que exista um caminho partindo de ω' tal que ele chegue a um ω sem arestas de saída ou prossiga infinitamente sem repetir orientações. Pelo Teorema 1, todo vértice $\omega \in \Omega$ possui exatamente uma aresta de saída (ω, ν), logo o caminho nunca encontrará um vértice sem arestas de saída. Como também existe um número finito de orientações, o caminho também não pode prosseguir infinitamente sem repetir vértices de Ω. Logo, há formação de período, resultando em contradição. □

2.2.2.4 Concorrência

Concorrência é uma medida de quanto os processos submetidos ao escalonamento por reversão de arestas podem agir comparado ao tempo total. Considerando então o modelo do período de duração π, a concorrência pode ser obtida dividindo-se o tempo τ em que um processo está ativo durante um período pela duração deste. O tempo τ é o mesmo para qualquer um dos n processos, assumindo tempo de operação igual a cada acesso aos recursos. A concorrência também pode ser aproximada dividindo-se o tempo α total de atuação de um processo pelo tempo t à medida que este tende a infinito. Devido à dificuldade em medir o período, por vezes esse método é mais prático. A Equação 2.1 resume essa relação.

$$\lim_{t \to \infty} \frac{\alpha}{t} = \frac{\tau}{\pi} \tag{2.1}$$

Logo, a concorrência c de um sistema *SER* pode ser definida como $\frac{1}{n} \leq c \leq \frac{1}{2}$. A maior concorrência possível é de $1/2$, possível em grafos bipartidos. Como todos os nós no sistema *SER* possuem igual acesso aos recursos disputados, havendo disputa por recursos um nó pode atuar no máximo em metade do tempo. Já a menor concorrência é de $1/n$, onde n é o total de nós do grafo, facilmente visível em grafos completos. Nesse caso dois nós não podem trabalhar concorrentemente, então o tempo será distribuído entre todos os nós operantes.

2.2.3 Geração do grafo subjacente

Dado um conjunto de processos P e um conjunto de recursos R, o grafo subjacente do mecanismo *SER* pode ser gerado com uma série de regras. Primeiro é preciso definir apropriadamente P e R. R possui uma definição mais simples, é um conjunto dos identificadores dos recursos, todos diferentes entre si. Cada processo $p \in P$ é composto de subprocessos s, cada subprocesso necessitando de um subconjunto ρ de R. Os subprocessos indicam diferentes grupos de recursos que o processo

original pode utilizar e podem ser inclusivos ou exclusivos entre si, dependendo de como eles se relacionam com relação a compartilhar recursos entre si.

Dois subprocessos são ditos exclusivos entre si se eles não podem compartilhar recursos entre si. De forma semelhante, os dois subprocessos são ditos inclusivos se eles podem compartilhar recursos. Um exemplo seria um processo que tenha, dentre outras tarefas, subprocessos para ler e para escrever em arquivos. Dois subprocessos de leitura são inclusivos entre si, enquanto um subprocesso de leitura e outro de leitura ou de escrita são exclusivos entre si, pois escrita envolve modificação do arquivo, e isso pode interferir na operação do outro subprocesso.

O grafo subjacente $G(V, E)$ pode ser construído da seguinte forma: V contém um vértice correspondente a cada subprocesso s de cada processo p de P; para cada dois vértices v e u, a aresta (v, u) pertence a E só e somente se v e u compartilham pelo menos um recurso e ou correspondem a subprocessos de processos diferentes ou se correspondem a subprocessos do mesmo processo e são exclusivos entre si. Uma vez obtido G, D pode ser encontrado gerando-se uma orientação acíclica inicial ω, o que será explicado na Subseção 2.2.4.

O sistema de processos pode ser descrito por expressões estruturalmente semelhantes a expressões Booleanas, mas com restrições quanto à sua interpretação. A declaração da composição do uso de recursos utilizada por um subprocesso é análoga ao operador lógico E, isto é, o subprocesso só pode executar com todos os recursos disponíveis. De forma semelhante, subprocessos de um mesmo processos são comparáveis a termos unidos por operadores lógicos OU inclusivo e exclusivo, de acordo com a independência entre eles. Esse refere-se a existência ou não de restrições entre subprocessos. É possível ainda que um subprocesso possua recursos "negados", indicando que ele não pode operar enquanto outro subprocesso do mesmo processo tiver acesso a um desses recursos. As Figuras 2.3, 2.4, 2.5 e 2.6 ilustram melhor esses operadores.

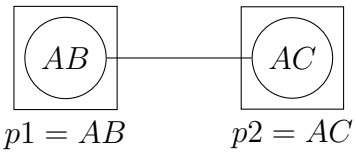

Figura 2.3: Exemplo de operador E.

2.2.4 Geração da orientação acíclica

Obter a orientação que maximize ou minimize a concorrência é um problema NP-completo [2], podendo ser comparado ao problema de coloração em grafo. Existem contudo diferentes heurísticas disponíveis, que proporcionam aproximações distintas

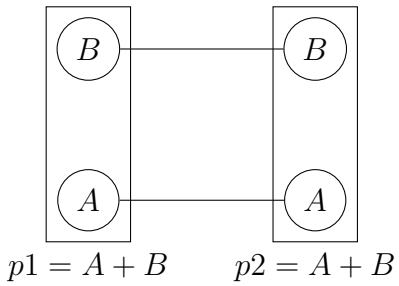

Figura 2.4: Exemplo do operador OU inclusivo.

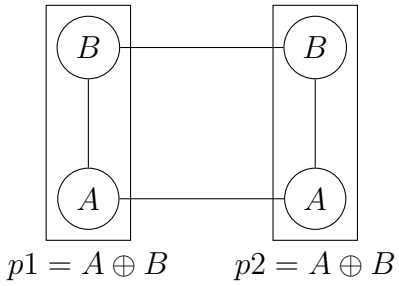

Figura 2.5: Exemplo do operador OU exclusivo.

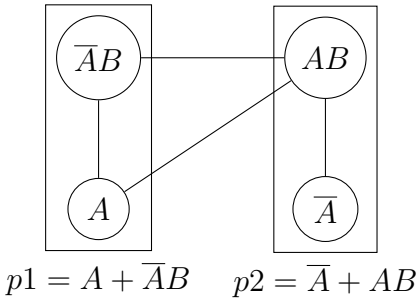

Figura 2.6: Exemplo de uso da negação de recursos.

para a concorrência. De modo geral, Alg-Cores tende a gerar as maiores concorrências, enquanto Alg-Arestas gera as menores [5]. Todas as heurísticas listadas a seguir podem ser implementadas de forma distribuída, o que é uma característica importante nas aplicações consideradas.

2.2.4.1 Alg-Neighbors

Uma das primeiras heurísticas desenvolvidas para orientação do *SER*, onde cada nó tenta obter prioridade sobre todos os vizinhos. Em termos de implementação, cada nó obtém um valor aleatório de um dado com n faces, que é comparado com os valores dos vizinhos. Se esse valor for superior aos valores obtidos por todos os vizinhos, o nó obtém prioridade sobre eles e sai do processo de orientação. O processo continua com os nós remanescentes, até que não restem arestas para orientar. Uma implementação mais detalhada está presente no Algoritmo 2.3.

Algoritmo 2.3: Alg-Neighbors

Variáveis:

$d \leftarrow \varnothing$
$espera \leftarrow 0$
$vencidos \leftarrow 0$
$V \leftarrow Vizinhos$
$V^+ \leftarrow \emptyset$
$V^- \leftarrow \emptyset$

Entrada:

$msg_i = \emptyset$, $espera = 0$, $V \neq \emptyset$
Ação:
$d \leftarrow r$ aleatório, tal que $1 \leq r \leq n$
$vencidos \leftarrow 0$
$espera \leftarrow |V|$
Para cada $v \in V$**:**
Enviar d **para** v

Entrada:

$msg_i = d_i$
Ação:
$espera \leftarrow espera - 1$
Se $d_i < d$**:**
$vencidos \leftarrow vencidos + 1$

Se $vencidos = |V|$**:**
$V^+ \leftarrow V$
$V \leftarrow \emptyset$
Para cada $v \in V^+$**:**
Enviar orientar **para** v
Fim Se
Fim Se

Entrada:
msg_i = orientar, origem$(msg_i) = v$
Ação:
$V^- \leftarrow V^- \cup \{v\}$
$V \leftarrow V \setminus \{v\}$

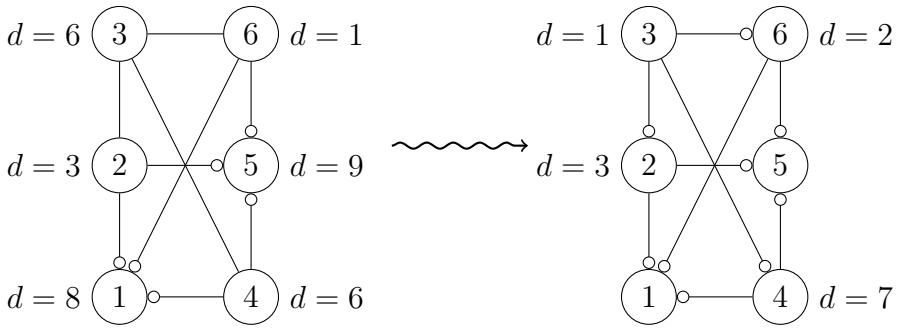

Figura 2.7: Exemplo da execução do Alg-Neighbors.

2.2.4.2 Alg-Colors

O Alg-Colors segue a ideia estabelecida pelo Alg-Neighbors, mas utiliza primeiro uma fase de coloração para depois gerar a orientação. As cores podem ser consideradas valores discretos com uma única ordenação possível, assim, distribuindo-se as cores de modo que dois nós vizinhos não tenham a mesma cor, é possível orientar todas as arestas de forma acíclica. Para obter as cores para cada nó, o algoritmo funciona de forma semelhante, mas cada nó que obtém um resultado melhor que seus vizinhos recebe a menor cor que ainda não foi utilizada por um vizinho ao invés de obter as prioridades. Cada nó que obtém sua cor pode então orientar suas arestas relativamente aos vizinhos já coloridos. O Algoritmo 2.4 exemplifica a aplicação. Para simplificar a definição do algoritmo, assumimos que cada nó tem acesso a um conjunto *Cores* com todas as cores existentes para a coloração.

Algoritmo 2.4: Alg-Colors

Variáveis:

$d \leftarrow \varnothing$

$espera \leftarrow 0$

$vencidos \leftarrow 0$

$V \leftarrow Vizinhos$

$V^+ \leftarrow \emptyset$

$V^- \leftarrow \emptyset$

$cor \leftarrow \varnothing$

$Usadas \leftarrow \emptyset$

Entrada:

$msg_i = \emptyset$, $espera = 0$, $V \neq \emptyset$

Ação:

$d \leftarrow r$ aleatório, tal que $1 \leq r \leq n$

$vencidos \leftarrow 0$

$espera \leftarrow |V|$

Para cada $v \in V$**:**

Enviar d **para** v

Entrada:

$msg_i = d_i$

Ação:

$espera \leftarrow espera - 1$

Se $d_i < d$**:**

$vencidos \leftarrow vencidos + 1$

Se $vencidos = |V|$**:**

$V \leftarrow \emptyset$

$cor = \min(Cores \setminus Usadas)$

Para cada $v \in Vizinhos$**:**

Enviar $\text{cor}(cor)$ **para** v

Fim Se

Fim Se

Entrada:

$msg_i = \text{cor}(cor_i)$, $\text{origem}(msg_i) = v$

Ação se $cor = \varnothing$**:**

$Usadas \leftarrow Usadas \cup \{cor_i\}$

Ação se $cor < cor_i$**:**

$V^+ \leftarrow V^+ \cup v$

Enviar orientar$^-$ **para** v

Senão:

$V^- \leftarrow V^- \cup v$

Enviar orientar$^+$ **para** v

Entrada:

$msg_i =$ orientar$^-$, origem$(msg_i) = v$

Ação:

$V^- \leftarrow V^- \cup \{v\}$

Entrada:

$msg_i =$ orientar$^+$, origem$(msg_i) = v$

Ação:

$V^+ \leftarrow V^+ \cup \{v\}$

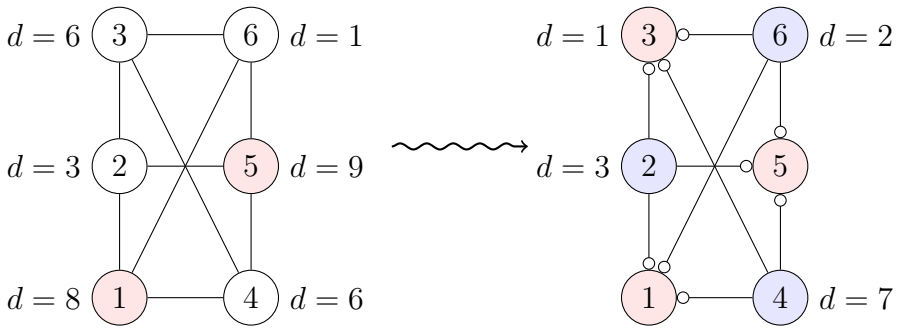

Figura 2.8: Exemplo da execução do Alg-Colors. Vermelho precede azul.

2.2.4.3 Alg-Edges

O Alg-Edge parte de uma ideia ainda mais simples do que a dos algoritmos anteriores. Agora, para orientar as arestas basta comparar os valores obtidos aleatóriamente pelos nós, orientando no sentido do maior. Os nós repetem com novos números em caso de empates, até que todas as arestas estejam orientadas. O Algoritmo 2.5 ilustra a implementação da heurística. Um exemplo da execução pode ser

visto na Figura 2.9.

Algoritmo 2.5: Alg-Edges

Variáveis:

$d \leftarrow \varnothing$

$espera \leftarrow 0$

$V \leftarrow Vizinhos$

$V^+ \leftarrow \emptyset$

$V^- \leftarrow \emptyset$

Entrada:

$msg_i = \emptyset$, $espera = 0$, $V \neq \emptyset$

Ação:

$d \leftarrow r$ aleatório, tal que $1 \leq r \leq n$

$vencidos \leftarrow 0$

$espera \leftarrow |V|$

Para cada $v \in V$**:**

Enviar d **para** v

Entrada:

$msg_i = d_i$, origem$(msg_i) = v$

Ação:

$espera \leftarrow espera - 1$

Se $d > d_i$**:**

$V \leftarrow V \setminus v$

$V^+ \leftarrow V^+ \cup v$

Fim Se

Se $d < d_i$**:**

$V \leftarrow V \setminus v$

$V^- \leftarrow V^- \cup v$

Fim Se

2.2.5 Aplicações

O escalonamento por reversão de arestas foi originalmente desenvolvido para controlar compartilhamento de recursos em sistemas de alta carga. Nessas situações, múltiplos processos precisam acessar diferentes conjuntos de recursos que não podem

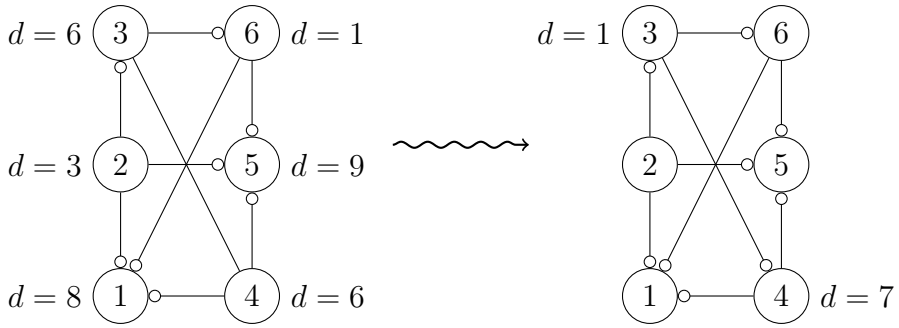

Figura 2.9: Exemplo da execução do Alg-Edges.

ser usados em simultâneo. O *SER* fornece então um mecanismo com garantia de que todos os processos envolvidos terão acesso aos recursos e não haverá travamentos.

A partir dessa estrutura básica é possível obter aplicações mais elaboradas. Um primeiro exemplo é um sistema de controle de semáforos. Cada semáforo pode ser modelado como um processo que compartilha trechos de rua com outros. Assim, um sinal em um cruzamento só estará aberto se os outros estiverem fechados. Outras aplicações menos diretas envolvem a criação de circuitos criptográficos [6], planejamento de rotas em sistemas automatizados de manufatura [7, 8], descontaminação de grafos [9–11], controle de agentes para busca [12].

2.2.5.1 Descontaminação e busca

O problema da descontaminação de grafos [13] deriva do problema da contaminação [14]. A contaminação consiste na propagação de uma propriedade pelo grafo segundo um critério previamente definido. A descontaminação é então o problema de espalhar agentes pelo grafo de modo a conter e remover a contaminação. A solução normalmente consiste em buscar o menor número de agentes necessários para eliminar a contaminação de todos os nós.

Para o tratamento de descontaminação por meio de *SER*, a implementação costuma construir o digrafo por cima do grafo a ser descontaminado. Os nós indicados para processar servem como guias para os agentes. Como os novos sorvedouros são sempre adjacentes aos anteriores, os agentes podem deslocar-se seguindo essas orientações.

Um exemplo de aplicação prática para essa interpretação é no uso desse sistema para coordenação de robôs para controle de incêndios [11]. Outro exemplo é a aplicação em coordenação de buscas com agentes distribuídos [15]. Ambos os casos traduzem a contaminação para uma propriedade que espalha-se pelo ambiente e deve ser removida. No primeiro caso é o fogo, no segundo, o desconhecimento do ambiente. Essas abordagem aproveitam-se também do estudo das propriedades da contaminação, como velocidade e intensidade da propagação [10].

2.2.5.2 Sistemas de manufatura

Outro problema abordado com escalonamento por reversão de arestas é o de planejamento de tarefas em um sistema de manufatura (*Job Shop*). Nesse ambiente há uma série de tarefas a serem realizadas, com diferentes durações utilizando uma sequência de máquinas. As tarefas possuem diversas durações e, possivelmente, prazos de término. O problema então é organizar as tarefas de modo a atingir um objetivo: minizar número total de atrasos, tempo total de atraso, tempo total de execução, ou uma combinação desses critérios

É possível tirar uma orientação a partir das regras de despacho [8]. Com essa orientação, os processos podem começar a operar. A orientação *SER* permite ainda a integração de veículos autônomos, chamados AGVs, utilizados para transporte dos produtos.

2.3 Escalonamento por reversão de múltiplas arestas (*SMER*)

O escalonamento por reversão de múltiplas arestas (*Scheduling by Multiple Edge Reversal*) é uma modificação do *SER* que permite taxas diferenciadas de acesso para os processos [3]. Ele baseia-se na ideia do jantar de filósofos zen, que comem apenas uma vez para cada duas de seus colegas. Esse mecanismo modificado pode ser implementado com o uso de múltiplas arestas e novas regras de reversão. A Figura 2.10 mostra um exemplo simples, semelhante ao exemplo da Figura 2.1 para o *SER*. Para simplificar a representação de multigrafos ilustram-se múltiplas arestas entre dois vértices por meio de uma única aresta entre os dois, com marcadores repetidos em cada terminação para indicar quantas arestas seguem aquele sentido.

2.3.1 Propriedades

Cada nó na nova definição possui agora um atributo chamado *reversibilidade*, que é inversamente proporcional à frequência de operação dele. Assim, por exemplo, em um sistema de dois nós, onde um atua duas vezes para cada ação do outro, o primeiro nó terá reversibilidade 1 e o outro, 2. A reversibilidade relaciona-se diretamente às regras de reversão, pois define quantas arestas serão revertidas após a operação de um nó. O novo problema então é definir quantas arestas devem haver entre cada par de nós.

Dados dois vértices x e y com reversibilidades r_x e r_y, é preciso definir a quantidade e_{xy} de arcos entre os dois. Para garantir que ambos possam atuar, é preciso que $e_{xy} \geq \max(r_x, r_y)$. Da mesma forma, é preciso que $e_{xy} < r_x + r_y$ para garantir

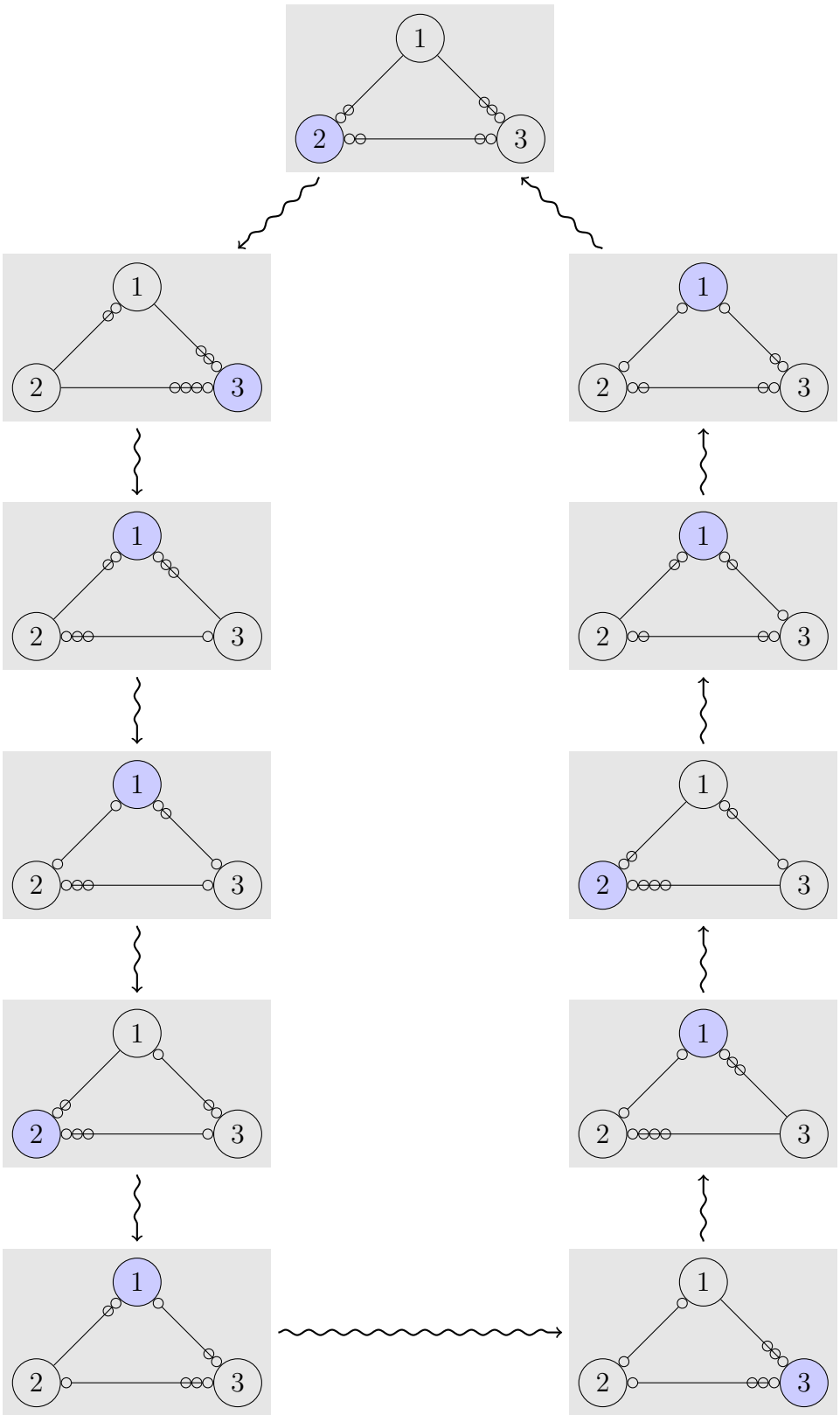

Figura 2.10: Exemplo de aplicação do *SMER*.

a exclusão entre os processos correspondentes. Apesar desse valor ser suficiente, o valor necessário e suficiente é

$$e_{xy} = r_x + r_y - \text{mdc}(r_x, r_y) \tag{2.2}$$

Com esta quantidade de arestas entre dois nós há garantia de que a operação entre eles funcionará [3].

A concorrência no *SMER* pode ser medida de forma semelhante ao *SER*, mas com alterações para lidar com a diferença de reversibilidades. Considere n o número total de tarefas, $\tau(v)$ o tempo em que a tarefa v está ativa durante um período de duração π e $a(v)$ o tempo total de atividade da v. A concorrência do *SMER* pode ser dada então pela Equação 2.3.

$$\lim_{t\to\infty} \frac{\sum_{\forall v} t(v)}{nt} = \frac{\sum_{\forall v} \tau(v)}{np} \tag{2.3}$$

2.3.2 Geração de orientação inicial de *SMER*

Para o *SER*, a existência de ciclos no grafo configura um problema de travamento. Devido às múltiplas arestas, é mais complicado detectar o travamento no *SMER*. Agora o travamento depende do total de arestas orientadas em cada sentido para cada ciclo existente no *SMER*.

De modo análogo ao *SER*, o *SMER* utiliza um multigrafo G^M como base, que será orientado de forma "acíclica". Contudo, como há múltiplas arestas, é preciso definir um "multiciclo". Primeiro considera-se que para cada par de vértices x e y do multigrafo D^M usado pelo *SMER*, há a_{xy} arcos no sentindo de x para y e a_{yx} no contrário. Para cada ciclo C em G^M, há "multiciclo" se e somente se

$$\max(\sum_{(x,y)\in C} a_{xy}, \sum_{(x,y)\in C} a_{yx}) < \sum_{x\in C} r_x \tag{2.4}$$

resultando em travamento em D^M [3].

Um algoritmo de orientação existente e válido baseia-se nas reversibilidades. Para cada par de vértices x, y, com $r_x > r_y$, são orientadas r_x arestas para x e o restante para y, ou seja, de acordo com a Equação 2.2:

$$\begin{cases} a_{xy} = r_x \\ a_{yx} = r_y - \text{mdc}(r_x, r_y) \end{cases} \tag{2.5}$$

Arestas entre nós com mesma reversibilidade devem ser orientadas de modo a evitar atingir as condições da Equação 2.4. Pode-se notar que a ideia por trás dessa heurística de orientação é semelhante à do Alg-Edges, com adaptações para múltiplas

arestas.

Em [4], Santos apresenta um método alternativo para obter uma orientação *SMER* baseado em uma orientação *SER* prévia. O Algoritmo 2.6 ilustra essa abordagem. Este modelo busca implementar mais facilmente alterações de reversibilidade durante a operação do *SMER* e para isso utiliza o conceito de "hiper-nós": cada nó do grafo é substituído por um "hiper-nó" que contém um nó para cada aresta. Esses nós só podem reverter se o todos os nós do "hiper-nó" tiverem acesso à quantidade de arestas correspondentes a suas reversibilidades. Para alterar a reversibilidade, um nó precisa ter a quantidade de arestas correspondente à reversibilidade anterior, mas envia uma quantidade de arestas correspondente à nova. Gerar a orientação nesse contexto corresponde, então, a modificar as reversibilidades de todos os nós.

Algoritmo 2.6: Orientação *SMER*

Variáveis:

V^+, inicializado pela orientação

V^-, inicializado pela orientação

$V \leftarrow V^+ \cup V^-$

$e_j \leftarrow 1, \forall j \in V^+$

$e_j \leftarrow 0, \forall j \in V^-$

$r_j, \forall j \in V$, já inicializado

$status \leftarrow SER$

Entrada:

$msg_i = \emptyset$, $e_j \leq r_j, \forall j \in V$

Ação:

Para cada $v \in V^+$**:**

Se $status = SER$ **:**

$e_v \leftarrow r_v + e_v - \text{mdc}(r_v, e_v)$

$status \leftarrow SMER$

Fim Se

$e_v \leftarrow e_v - r_v$

Enviar reversão(r_v) **para** v

Entrada:

$msg_i = \text{reversão}(r)$, $\text{origem}(msg_i) = v$

Ação:

$e_v \leftarrow e_v + r$

2.4 SATyrus

O SATyrus foi uma plataforma desenvolvida para facilitar a expressão da resolução de problemas de otimização como problemas de Satisfatibilidade [16]. Problemas podem ser descritos numa linguagem própria, SATish, a partir da qual será convertido de restrições em lógica proposicional com pesos, ou expressões pseudo-Booleanas, para uma função de energia e desta para uma rede estocástica de Hopfield. Ele foi depois reescrito como SATyrus2 [17] com intuito de superar limitações impostas pela arquitetura original.

O SATyrus2 serviu como modelo para o ReSATyrus, inclusive contribuindo com código. Apesar dos objetivos diferentes, o funcionamento básico de ambos envolve transformar uma forma simplificada de código para outra. Esta nova representação pode ser então tratada por um programa especializado em resolver problemas de otimização combinatória, por exemplo, Xpress [18] ou módulos do AMPL [19]. O ReSATyrus será apresentado no Capítulo 3.

2.4.1 Propósito

O SATyrus original possuía duas funções: compilar a especificação do problema, e otimizar o modelo resultante. Essa otimização era feita com a criação de uma função de energia e correspondente rede estocástica de Hopfield, que é utilizada para encontrar o mínimo global do problema descrito por meio do algoritmo de *Simulated Annealing*. O SATyrus2 expandiu a linguagem SATish e introduziu a conversão para outros sistemas, com conversão para as linguagens AMPL [19] e MOSEL [18], sendo assim mais flexível.

A nova estrutura do SATyrus2 foi desenvolvida em Python, com um foco em módulos. A linguagem Python foi escolhida por ser uma linguagem interpretada e dinâmica, permitindo avaliar trechos de código durante execução. Assim, é possível transferir parte da avaliação do código SATish, particularmente operações matemáticas, diretamente para Python. Apesar de mais lenta que uma linguagem compilada, por exemplo C++, o tempo de compilação é pequeno comparado com o tempo para busca do ótimo global, realizada por resolvedores externos.

A estrutura do SATyrus2 foi dividida em quatro módulos para melhor organizar o código. Os módulos são responsáveis pela interface com o usuário, interpretação do código, geração da função de energia e conversão para a linguagem dos resolvedores. O módulo de interface com os resolvedores é composto de submódulos correspondentes aos resolvedores. Dessa forma o SATyrus pode ser ampliado com a adição de submódulos para poder trabalhar com novos resolvedores. A relação entre esses módulos está presente na Figura 2.11.

A interface com o usuário é responsável por receber o arquivo fonte, transferi-lo

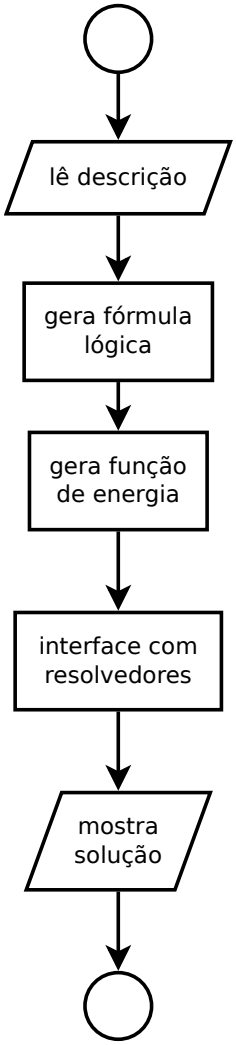

Figura 2.11: Processo básico do SATyrus.

para o interpretador, receber os resultados, sejam eles respostas ou erros, e devolvê-los. Essa interface também lida com as opções recebidas do usuário e organiza a execução para adequar-se a estas.

O interpretador recebe, verifica e interpreta o código dos problemas descritos, gerando uma modelagem interna que é passada ao gerador da função de energia se não houver erros no código. Ele cuida da análise lexical e sintática do código, convertendo-o para fórmulas lógicas. Essas fórmulas podem ser manipuladas, até mesmo simplificadas se o usuário requisitar, e são usadas como base para geração da função de energia

O gerador converte a formulação lida para uma expressão algébrica de energia por meio de regras aperfeiçoadas como parte do projeto do SATyrus [17]. Essas regras permitem reformular expressões Booleanas na forma normal conjuntiva como expressões aritméticas utilizando os inteiros 1 e 0. Essa modificação é importante para o passo de conversão seguinte.

A interface para os resolvedores é composta por submódulos que utilizam a expressão de energia para criar modelos nas linguagens que os resolvedores compreendem. Para gerar esses modelos é importante observar algumas restrições: (i) as variáveis da função de energia devem ser binárias, isto é, aceitar apenas os valores 1 ou 0; (ii) uma estratégia de otimização apropriada para funções não lineares e não convexas (como a função de energia) deve ser escolhida; (iii) a função de energia deve ser minimizada pelo resolvedor; (iv) a solução encontrada deve ser exibida para o usuário. O submódulo converte a função de energia segundo as restrições listadas, envia para o resolvedor correspondente, recebe a resposta e retorna ao usuário de forma transparente.

2.5 Graphviz

O pacote Graphviz [20] consiste de uma linguagem para expressar grafos e um conjunto de programas que geram representações gráficas a partir de descrições nessa linguagem. A linguagem pode ser usada para expressar grafos gerados durante o trabalho. A visualização dos grafos permitirá examinar ideias para dar continuidade ao desenvolvimento da pesquisa.

2.5.1 A linguagem DOT

A linguagem utilizada para descrever os grafos é denominada linguagem DOT. Com ela é possível criar arquivos de texto simples contendo a estrutura do grafo. Embora a linguagem seja simples o suficiente para ser escrita a mão, ela apresenta funcionalidades que permitem definir atributos como forma dos nós, das arestas,

texto, dentre outros, possibilitando criar grafos complexos.

Cada arquivo deve conter idealmente apenas um grafo ou digrafo. O arquivo inicia com a declaração do grafo, indicando tipo e nome, seguido pelos atributos entre chaves. Os tipos de grafo são definidos pelas palavras chaves `graph` ou `digraph`. Além disso o tipo pode ser precedido pela palavra `strict` para indicar que não é um multigrafo ou multidigrafo. As Figuras 2.12 e 2.13 mostram dois exemplos.

```
graph G {
    subgraph S {
        a -- b;
        a -- c;
    }
    a -- b;
}
```

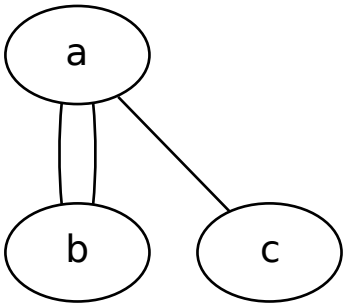

Figura 2.12: Exemplo de multigrafo não-orientado.

```
strict digraph D {
    subgraph clusterS {
        a -> b;
    }
    a -> c;
    a -> b;
}
```

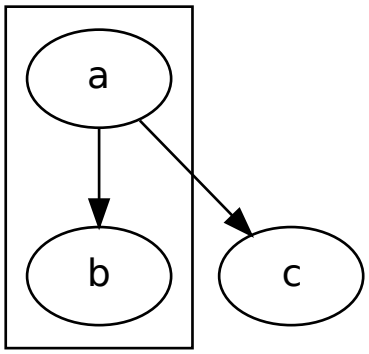

Figura 2.13: Exemplo com digrafo.

2.5.2 Os componentos do Graphviz

O pacote Graphviz inclui programas para gerar visualizações dos grafos, desenhar grafos ou modificar a estrutura (por exemplo, unir dois grafos ou tornar um

grafo direcionado acíclico). Para este trabalho interessam apenas os programas de representação. Cada um deles apresenta diferentes métodos para posicionar os nós, com diferentes objetivos para aplicação. Particularmente, o `dot` foi adotado para a maioria dos exemplos presentes nesta dissertação, por causa da capacidade de poder demarcar agrupamentos de nós (*clusters*), evidenciada na Figura 2.13.

2.6 NetLogo

O NetLogo [21] é uma ferramenta para modelagem de sistemas com múltiplos agentes. O sistema inclui facilidades para gerar interfaces para a simulação e suporte a uma linguagem para programação do ambiente baseada em Logo [22]. A linguagem fornece estruturas para permitir que agentes ajam de forma independente entre si.

2.6.1 Descrição

O modelo baseia-se em três tipos básicos de agentes: agentes móveis (chamados de *tartarugas*), trechos de terreno, e arestas. Tartarugas possuem, dentre outras características, posição, cor e orientação, e também podem acessar as características do terreno onde se encontram. Trechos de terreno possuem também posição e cor, mas não podem modificar a posição e são todos criados ao início da simulação. Arestas podem ser direcionadas ou não e conectam duas tartarugas diferentes. A linguagem permite estender esses tipos básicos para ter novos, com possível adição de novos atributos. A Figura 2.14 ilustra um exemplo com elementos diversos do NetLogo.

Apesar de não haver estruturas para comunicação entre agentes, comunicação entre tartarugas pode ser simulada com arestas. Essas capacidades são suficientes para desenvolver simulações básicas sobre o comportamento dos mecanismos estudados. O próprio sistema NetLogo já vem com uma biblioteca de modelos previamente desenvolvidos, incluindo sistemas clássicos como o jantar dos filósofos [1].

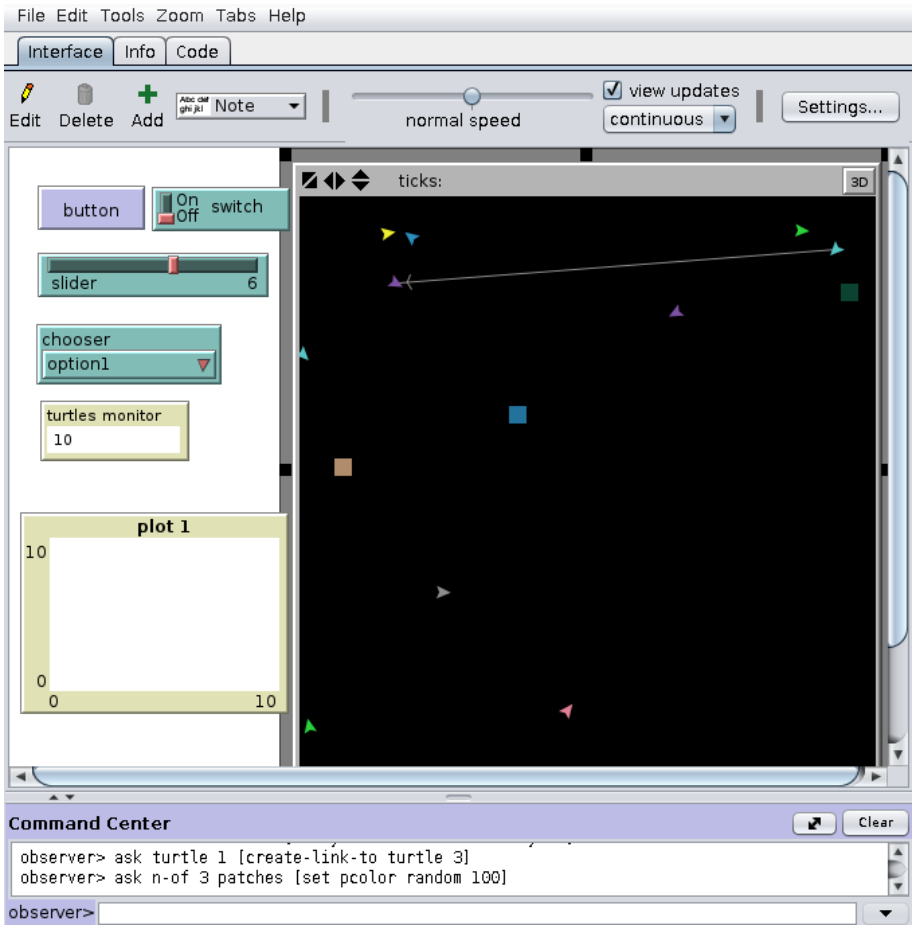

Figura 2.14: Exemplo da interface do NetLogo

Capítulo 3

ReSATyrus

O ReSATyrus é um programa criado para converter descrições de processos com uso de recursos para representação em grafo utilizável pela mecânica *SER*. Ele foi desenvolvido a partir do SATyrus, inclusive compartilhando a base de código. Durante o desenvolvimento o ReSATyrus foi estendido para poder criar também grafos para uso com *SMER*.

3.1 Motivação para criação de um compilador de compartilhamento de recursos

Os estudos para definir uma modelagem envolvem a criação de diferentes grafos para avaliar as interações criadas pelo *SER*. Contudo, mesmo um sistema relativamente simples com poucos processos e recursos pode resultar em um grafo comparavelmente maior. O tamanho dos grafos envolvidos aumenta a chance de algum erro durante a elaboração manual dos mesmos. Como o processo de criar esses grafos consiste de etapas bem definidas, é interessante então automatizá-lo com um programa.

A automação dessas etapas fornece uma economia de tempo que permite desenvolver mais modelos. Com diversos modelos a disposição, é possível testar mais teorias e achar aquela que melhor adequa-se às necessidades do problema em mãos. O compilador então serve como uma etapa inicial no planejamento de soluções *SER*/*SMER* para problemas. Com ele é possível descrever problemas em uma linguagem própria (ReSATish) mais compacta e obter descrições completas de grafos expressos na linguagem DOT do Graphviz.

3.2 Interpretação da modelagem de processos

O uso de recursos é modelado usando as relações explicadas na Seção 2.2.3, utilizando os operadores `and`, `or` e `xor`, com parênteses para permitir relações entre grupos de recursos, por exemplo: `(a and b) or c`. Existe ainda o modificador `not` para indicar que um subprocesso exige que seu processo não tenha acesso a um recurso em qualquer outro subprocesso. Os operadores possuem a propriedade distributiva e as seguintes responsabilidades:

`and` indica que dois recursos precisam ser obtidos. Um conjunto de recursos relacionados apenas pelo operador `and` corresponde a um subprocesso;

`or` indica que basta um de dos dois recursos relacionados para operar. Conjuntos de subprocessos relacionados pelo operador `or` podem estar ativos concorrentemente;

`not` indica um recurso negado. Um subprocesso que tiver um recurso negado só pode estar ativo enquanto nenhum outro subprocesso do mesmo processo usar esse recurso. Assim, o `not` serve para definir restrições dentro de um processo, lembrando que subprocessos não disputam recursos entre si.

`xor` indica que o processo pode operar tendo apenas um dos recursos relacionados. Aqui é importante notar outra vez as diferenças para operadores Booleanos tradicionais. O `xor` é tradicionalmente definido como $A \oplus B = A\bar{B} + \bar{A}B$. Contudo, para especificação de subprocessos normalmente é mais comum desejar utilizar o `xor` em um grupo de subprocessos que não podem operar em simultâneo. A implementação então é `xor(A, B, ..., Z) = (A and not(B, ..., Z)) or (B and not(A, ..., Z)) or ...or (Z and not(A, B, ...))`.

A descrição de processos assemelha-se a descrições em lógica Booleana, contudo apresenta algumas restrições. Como o `and` representa relações internas de um subprocesso, enquanto o `or` representa relações entre subprocessos, para melhores resultados o ideal é representar um processo na forma normal disjuntiva. Como exemplo de comportamentos inesperados que podem acontecer, podemos considerar a representação de um processo como `not(A and B)`, supondo que isso resulte em um processo que atue apenas quando os recursos `A` e `B` não estiverem sendo utilizados simultaneamente por outros subprocessos. Contudo, após a conversão para a forma normal disjuntiva, o processo seria expresso como `not(A) or not(B)`, resultando em dois subprocessos, cada um atuando quando na ausência de um dos recursos mencionados, o que pode não corresponder ao esperado.

O `xor` também deve ser utilizado com cautela. Devido à forma como foi implementado, a quantidade de termos unidos pelo `xor` não é necessariamente igual

à quantidade de subprocessos resultantes. Para exemplo, `xor(A, B, C, D)` possui quatro termos e resulta em quatro subprocessos, como esperado. Por outro lado, `xor(A and B, C and D)` possui apenas dois termos, mas resulta em quatro subprocessos depois de convertido para a forma normal disjuntiva.

Outra diferença importante está na aplicação da propriedade distributiva e de simplificação. Em lógica Booleana, as expressões

- `(A and B) or (A and B and C)`
- `A and (B or (B and C))`
- `A and B`

são equivalentes. As mesmas expressões, contudo, possuem uma diferente quantidade de subprocessos quando descrevendo um processo. Disso observa-se que a simplificação possibilitada pela propriedade distributiva força uma sincronização entre subprocessos, assim suas atividades não podem ocorrer mais independentemente. Dessa forma a quantidade de subprocessos diminui.

A Figura 3.1 ilustra esse exemplo: cada retângulo indica um processo, enquanto as elipses são seus subprocessos. Os processos são identificados como `_p1`, `_p2` e `_p3`, conforme a ordem que suas expressões apareceram. Nota-se que `_p1` e `_p2` possuem a mesma quantidade de subprocessos, enquanto `_p3` possui apenas um. Da mesma forma pode-se observar pelas arestas que apenas um processo pode ter seus subprocessos ativos de cada vez, e que `_p1`, assim como `_p2`, pode ter seus subprocessos internos ativos concorrentemente.

3.3 Linguagem ReSATish

A linguagem do ReSATyrus é inspirada na SATish usada pelo SATyrus, mas com várias modificações. Há dois blocos principais na linguagem: declaração de processos e de restrições. As restrições podem ser divididas ainda em dois grupos, relativas a recursos e relativas a orientação. Cada linha não vazia deve conter uma expressão terminada com ponto e vírgula.

A declaração de processos consiste em uma série de atribuições de valores inteiros para identificadores de processos. Cada identificador deve ser único e iniciado por uma letra, podendo incluir também números no resto de seu nome. Os valores inteiros atribuídos são as reversibilidades usadas na modelagem do *SMER*.

As restrições são declaradas primeiro com o tipo de restrição, seguido do identificador do processo. Segue-se uma especificação da restrição após dois pontos. Restrições de recursos são especificadas como `intgroup`. Cada recurso possui um

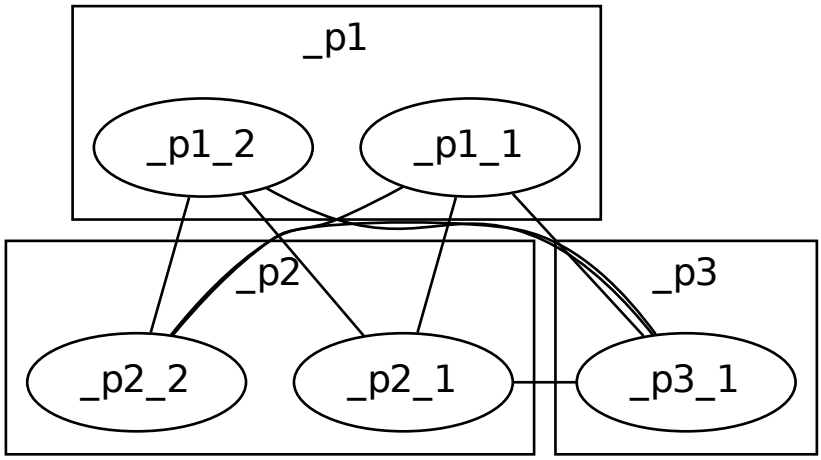

Figura 3.1: Diferenças causadas por descrição

identificador em formato semelhante ao de um processo. Os recursos relacionam-se entre si utilizando os operadores apresentados na Seção 3.2.

Para situações em que há uma ordem preferencial para uso dos recursos, é possível usar outro tipo de restrição. A declaração é de forma semelhante, utilizando o tipo de restrição `optgroup`. A ordem é indicada após os dois pontos com uma lista simples dos recursos separados por `->`. A preferência de uso dos recursos é definida da esquerda para a direita.

O compilador usa então as preferências dos recursos não negados de cada subprocesso para orientar as arestas correspondendo a relações dentro do processo. Se todos os recursos de um subprocesso superam os de outro em preferência, então ele tem prioridade de ação e a aresta está orientada para ele. Quando as preferências não fornecem uma ordenação clara, ele ignora essa restrição e deixa a aresta sem orientação. A gramática completa do ReSATyrus está no Apêndice B.

3.4 Exemplos de ReSATish

Esta seção apresenta exemplos de aplicação da linguagem. Cada exemplo busca demonstrar a recriação de um dos controles de compartilhamento de recursos apresentados na Seção 3.2, reproduzindo os exemplos iniciais apresentados na Seção 2.2.3. Cada controle é acompanhado pelo grafo correspondente gerado.

Nesses exemplos os retângulos delimitam os subprocessos de um processo, identificado pelo nome utilizado na especificação ReSATish precedido por um subtraço

('_'). Os subprocessos, por sua vez, são representados por elipses contendo um identificador composto do indentificador do processo, unido a um identificador numérico próprio do subprocesso por outro subtraço. Devido ao modo como o ReSATyrus gera a representação a representação em linguagem DOT e o modo como os componentes do Graphviz trabalham, não há uma garantia de que a ordenação de processos e subprocessos siga um padrão. Por vezes o Graphviz também não consegue dispor automaticamente os vértices de modo a poder separá-los em subprocessos, nesses casos os processos não estarão delimitados.

- `and`:

```
p1 = 1;
p2 = 1;

intgroup p1: A and B;

intgroup p2: A and B;
```

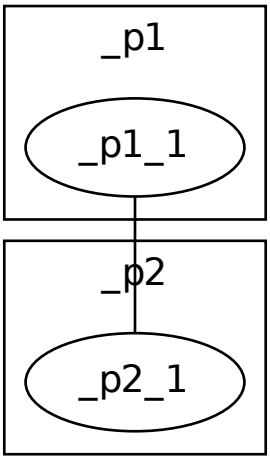

Figura 3.2: Estilo de compartilhamento E: Como ambos processos usam apenas o estilo de compartilhamento E, todos os recursos listados são necessários conjuntamente para operar, logo não pode-se dividir as tarefas e existe apenas um subprocesso por processo. Como há recursos em comum entre os subprocessos dos diferentes processos, há restrição de operação entre eles.

- or:

```
p1 = 1;
p2 = 1;

intgroup p1: A or B;

intgroup p2: A or B;
```

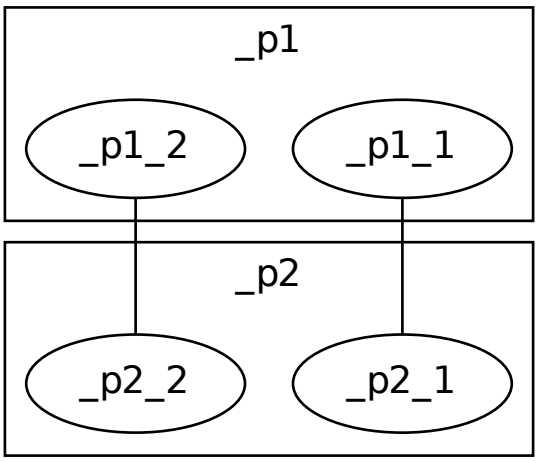

Figura 3.3: Estilo de compartilhamento OU inclusivo: O estilo de compartilhamento OU inclusivo permite a divisão de tarefas e a existência de dois subprocessos por processo, um para cada recursos listado, pois estes não precisam ser obtidos conjuntamente. Os subprocessos possuem a operação restrita apenas com subprocessos de outro processo que utilizem recursos em comum.

- `not`:

```
p1 = 1;
p2 = 1;

intgroup p1: A or (not A and B);

intgroup p2: not A or (A and B);
```

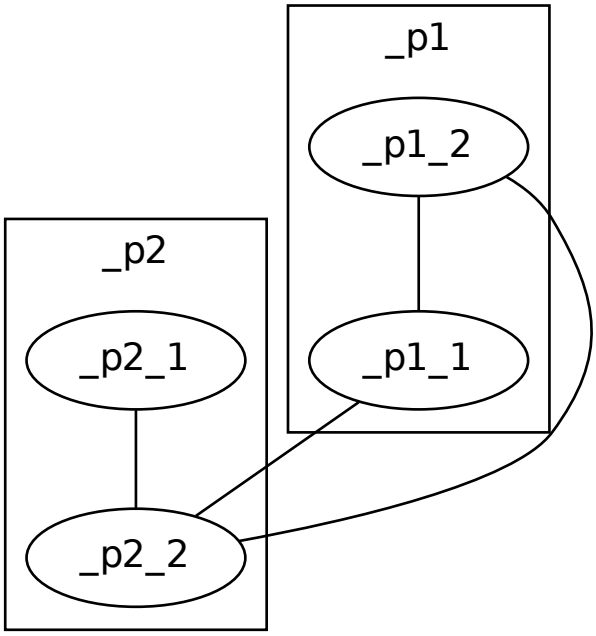

Figura 3.4: Uso de recursos negados: A função da negação é inibir a operação de um subprocesso enquanto outro subprocesso do mesmo processo tem acesso ao recurso negado. Este exemplo reforça ainda que a negação atua apenas entre subprocessos do mesmo processo.

- `xor`:

```
p1 = 1;
p2 = 1;

intgroup p1: xor(A, B);

intgroup p2: xor(A, B);
```

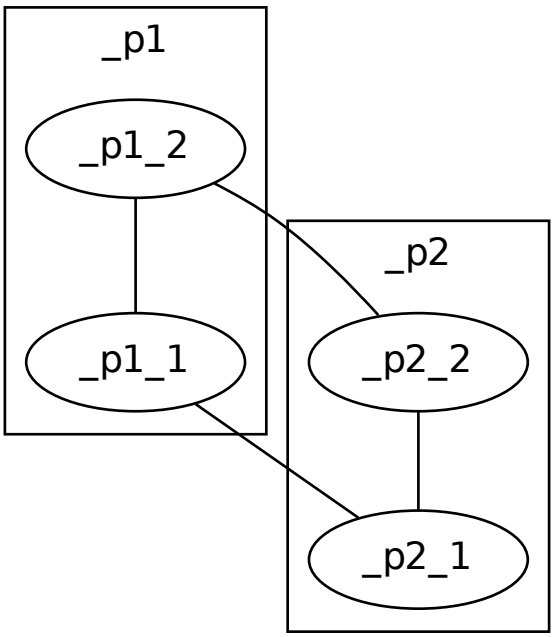

Figura 3.5: Estilo de compartilhamento OU exclusivo: Este exemplo é semelhante ao exemplo para OU inclusivo, mas com negação implícita entre os subprocessos listados pelo OU exclusivo. Apenas um subprocesso resultante de um conjunto de recursos relacionados pelo compartilhamento por OU exclusivo pode estar ativo por vez.

3.5 Geração de grafos

O processo para gerar todos os grafos possíveis de serem extraídos de uma descrição está ilustrado na Figura 3.6. Os tipos de grafos podem ser selecionados para o programa gerar apenas os que forem relevantes. Também deve ser mencionado que os grafos podem ser orientados para uso pelo *SER* ou pelo *SMER*, utilizando três heurísticas diferentes para cada, logo ao todo existem seis tipos principais de grafos orientados que podem ser gerados. Existe ainda uma alternativa que consegue efeitos semelhantes ao *SMER* utilizando *SER* por meio da multiplicação de vértices ao invés de arestas, oferecendo mais um tipo de grafo orientado.

Os seguintes tipos de grafos podem ser gerados, acompanhado de ilustrações para exemplificar baseados no código presente na Figura 3.7:

grafo de recursos Nesse grafo há um vértice para cada recurso e para cada subprocesso, com arestas entre cada recurso e os subprocessos que o utilizam. Subprocessos são separados em grupos de acordo com o processo correspondente; A Figura 3.8 mostra o exemplo deste grafo. Observa-se que existe um grupo apenas para os recursos e que recursos negados (criados pelo `xor`) são representados como um recurso independente dos outros, a ser relacionado depois.

grafo de restrições Gerado a com as informações obtidas do grafo de recursos. Este é o grafo usado como base pelo *SER*. Para cada vértice de recurso existente no grafo de recursos podem ocorrer duas coisas:

- se for um recurso normal, os subprocessos conectados recebem arestas entre si no grafo de concorrência, a menos que pertençam ao mesmo processo;
- se for um recurso negado, os subprocessos conectados recebem arestas no grafo de concorrência com todos os subprocessos do mesmo processo que tenha uma aresta para o recurso equivalente não negado no grafo de recursos;

A Figura 3.9 possui arestas resultantes de ambos os tipos de recursos.

grafo completo Um hipergrafo que relaciona as arestas entre subprocessos com os recursos envolvidos. Neste modo, cada aresta entre dois subprocessos conecta-se ainda aos recursos responsáveis pela sua existência. Se a relação for causada por um recurso negado, a aresta conecta-se ao vértice correspondente ao recurso não negado. A Figura 3.10 mostra o grafo completo resultante para este exemplo. Os recursos negados são indicados, mas não estão conectados pois para a relação é relevante o recurso original.

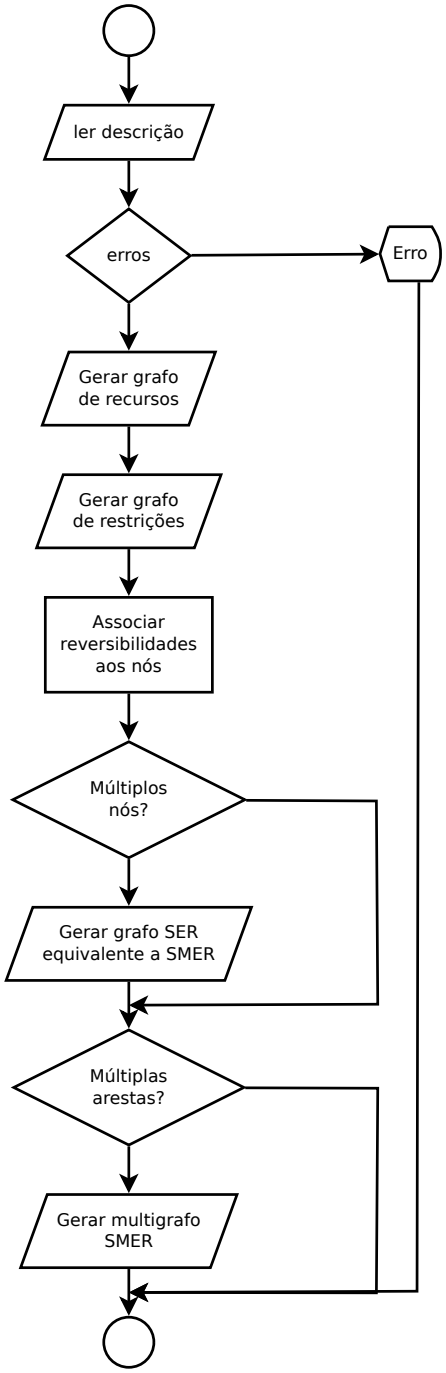

Figura 3.6: Processo de geração de grafos.

```
p1 = 1;
p2 = 2;

intgroup p1: xor(A, B) or (C and D);
optgroup p1: A -> B;

intgroup p2: (A and B and C) or D;
```

Figura 3.7: Código de exemplo

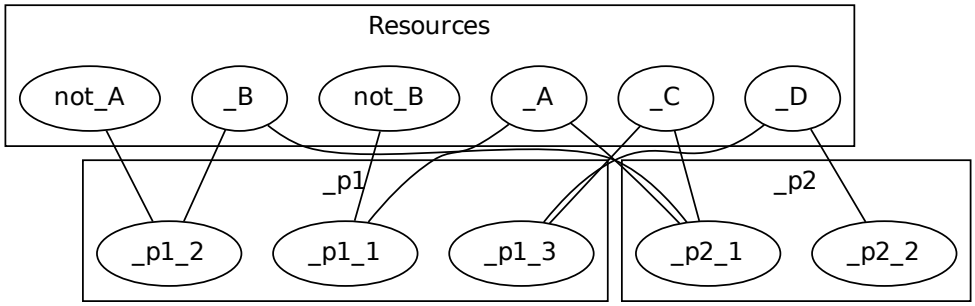

Figura 3.8: Exemplo do grafo de recursos

grafo de dicas de orientação Usando as dicas de orientação do `optgroup` é possível ainda criar um grafo para identificar as preferências de orientação. Este grafo conterá apenas arestas orientadas obtidas pelas dicas. Logo, para cada aresta no grafo de concorrência pode haver uma aresta orientada no grafo de dicas se a aresta for entre dois subprocessos do mesmo processo e se os todos recursos de um tiverem preferência de uso menor ou igual ao do outro. A criação deste grafo pode causar erros, por exemplo, se as preferências indicadas contiverem um ciclo. A Figura 3.11 mostra o resultado das dicas de orientação para o exemplo da Figura 3.7. Observa-se que apenas uma aresta resulta dessas dicas, em apenas um processo, o que confere com a existência de um termo `optgroup` apenas para o processo `_p1`.

grafos orientados para o *SER* O ReSATyrus pode também gerar orientações usando as heurísticas apresentadas anteriormente, Alg-Neighbors, Alg-Edges e Alg-Colors. Esses grafos possuem uma aresta orientada para cada aresta do grafo de concorrência. Devido à natureza aleatória das heurísticas, não é possível garantir que dois grafos orientados por uma mesma heurística em duas execuções do programa serão iguais. A Figura 3.12 mostra uma orientação, no caso utilizando, para exemplo, o Alg-Colors. Nota-se que a aresta entre os

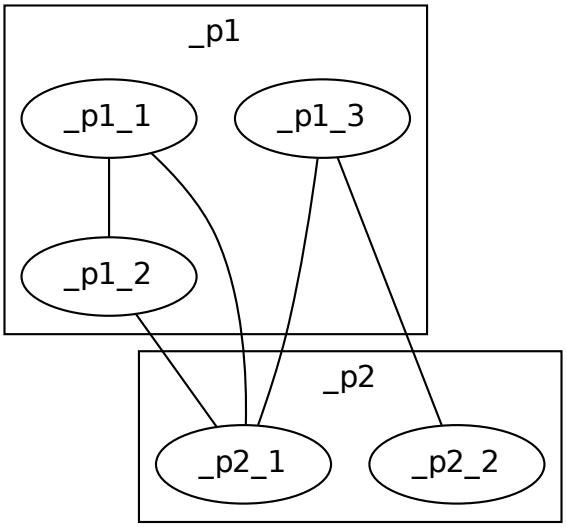

Figura 3.9: Exemplo do grafo de restrições

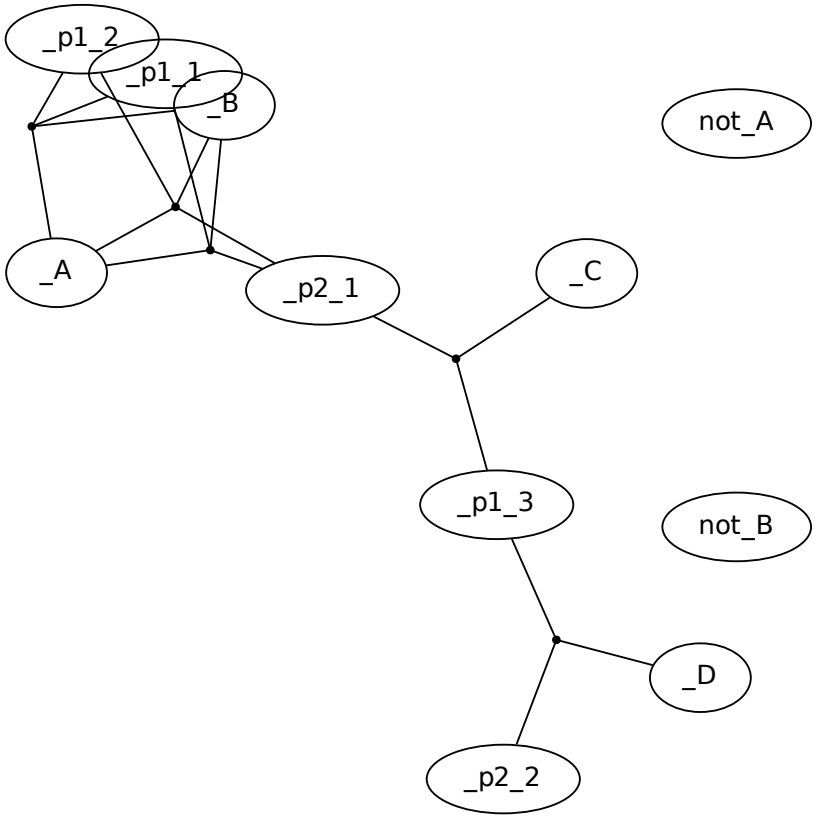

Figura 3.10: Exemplo do grafo completo

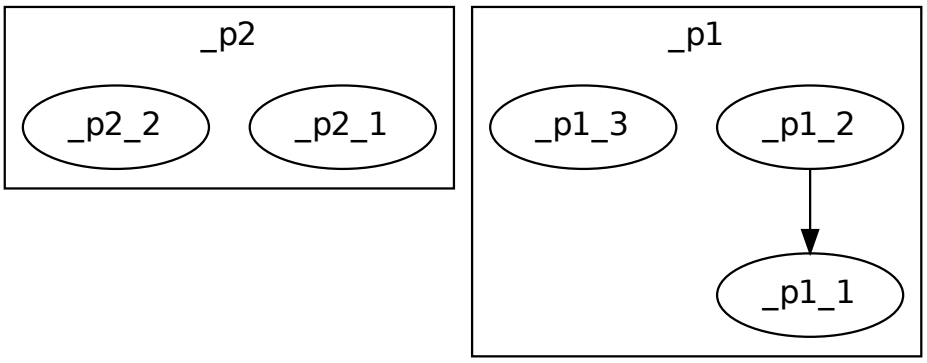

Figura 3.11: Exemplo do grafo de dicas de orientação

subprocessos `_p1_2` e `_p1_1` segue a orientação originária das dicas na Figura 3.11.

multigrafo de restrições o ReSATyrus pode criar grafos para uso com o *SMER*. Ele precisa primeiro gerar um grafo de concorrência do *SMER*, com múltiplas arestas entre dois subprocessos que compartilham recursos. Cada subprocesso tem sua reversibilidade definida pelo seu processo. Entre dois subprocessos com reversibilidades r_i e r_j há $r_i + r_j - \text{mdc}(r_i, r_j)$, conforme é definido como necessário e suficiente [3]. A Figura 3.13 mostra o multigrafo de restrições gerado.

multigrafos orientados para o *SMER* Usando o grafo de concorrência do *SMER*, o ReSATyrus consegue ainda gerar grafos orientados usando versões modificadas das heurísticas já apresentadas. Essa modificação é detalhada no Capítulo 4. A Figura 3.14 mostra o multigrafo orientado para o *SMER*.

grafo de restrições de múltiplos nós O ReSATyrus pode ainda gerar um grafo de concorrência para compartilhamento de recursos em taxas desiguais usando múltiplos vértices ao invés de múltiplas arestas [3]. Nesta variante, cada subprocesso possui r vértices, onde r é a sua reversibilidade. O grafo é construído combinando ciclos que alternam todos os vértice de cada dois subprocessos que disputam recursos. Dessa forma respeitam-se as propriedades requisitadas: (i) garantir que apenas um dos subprocessos em cada par opere de cada vez e (ii) manter as taxas diferenciadas de acesso aos recursos. A Figura 3.15 mostra o grafo com múltiplos nós equivalente ao multigrafo de restrições do *SMER*. Observa-se que agora os subprocessos estão representados por retângulos inter-

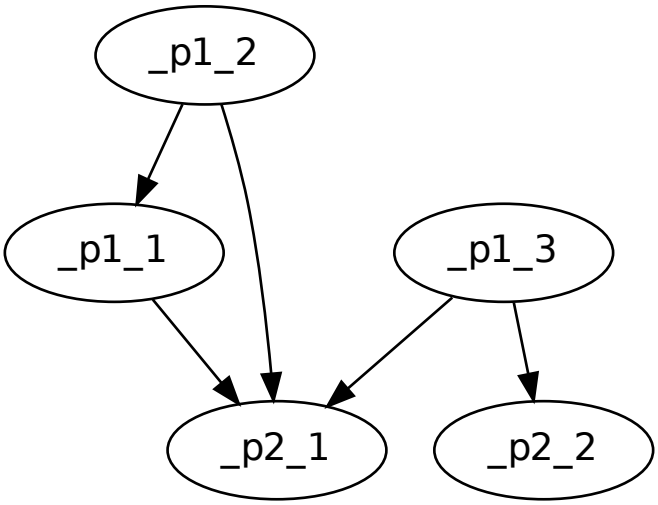

Figura 3.12: Exemplo de um grafo orientado para o *SER*

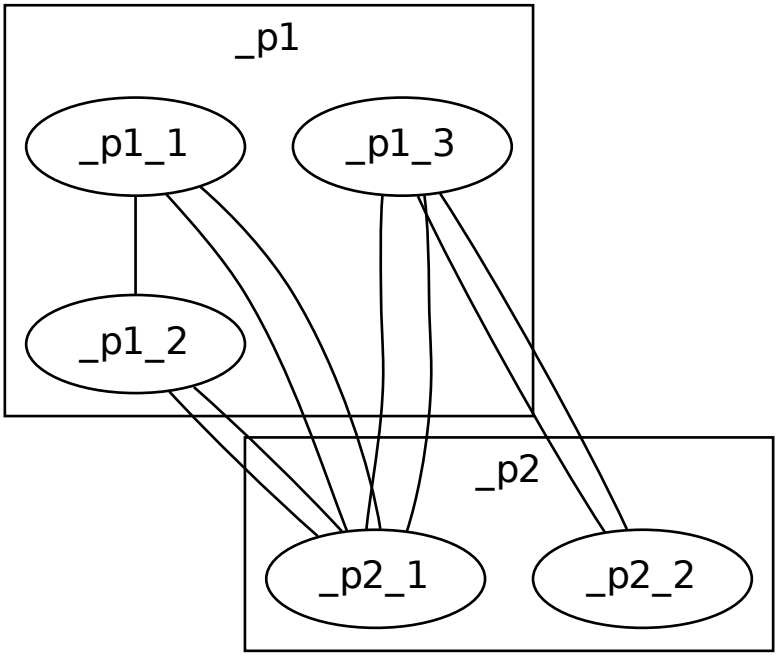

Figura 3.13: Exemplo do multigrafo de restrições

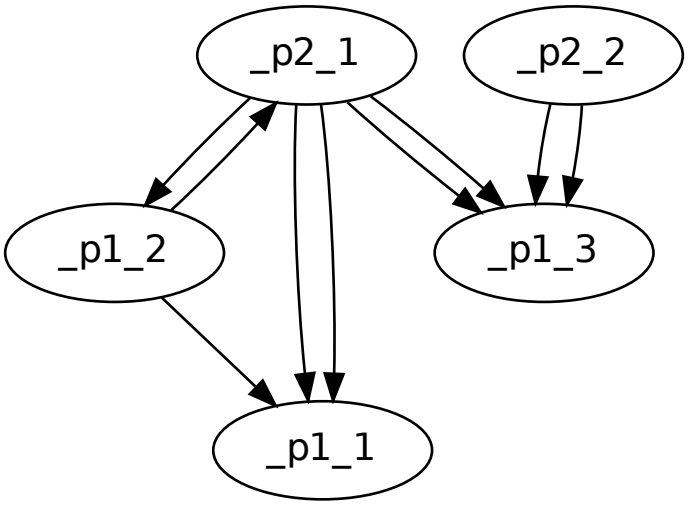

Figura 3.14: Exemplo do multigrafo orientado para o *SMER*

nos aos retângulos dos processos, contendo dentro de si elipses correspondendo aos vértices multiplicados.

grafo orientado de múltiplos nós Orientado de forma que apenas um nó por subprocesso possa estar ativo de cada vez, e subprocessos concorrentes não possam estar ativos simultaneamente [3]. A Figura 3.16 mostra o grafo de múltiplos nós orientado de modo a preservar as propriedades requisitadas.

3.6 Exemplos de dinâmicas

As interações criadas por diferentes orientações *SER* podem ser difíceis de prever. É interessante então ter exemplos de aplicação das dinâmicas capazes de receber grafos já orientados e modelar os deslocamentos de agentes resultantes. A execução de diferentes modelagens pode revelar consequências não previstas e auxiliar no aprimoramento dos modelos.

Os primeiros exemplos foram criados em NetLogo aproveitando as propriedades oferecidas pela linguagem. Com um modelo definido foi possível criar um exemplo mais complexo permitindo alterações dos grafos. Este foi desenvolvido integrado ao sistema de compilação do ReSATyrus para controle das interações *SER* envolvidas.

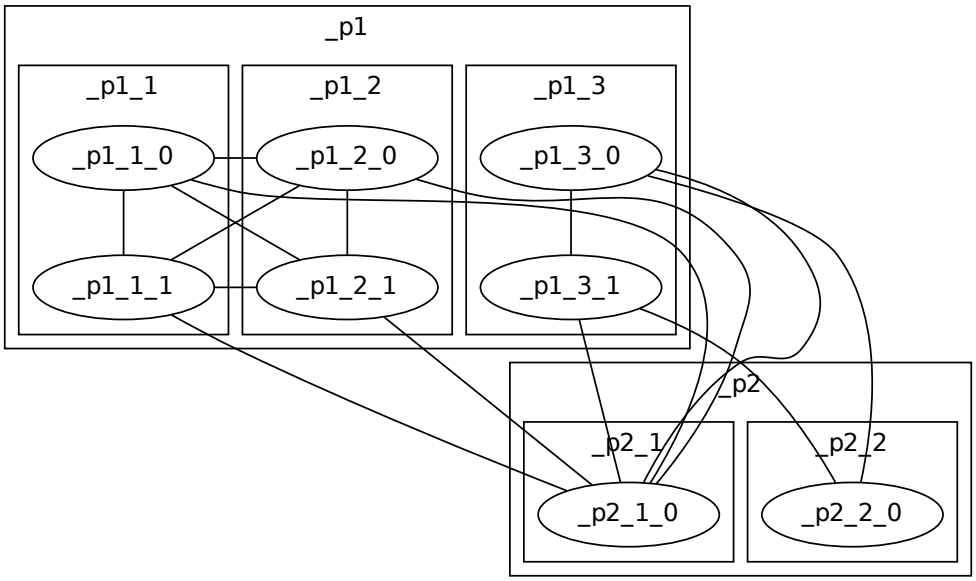

Figura 3.15: Exemplo do grafo de restrições com vértices multiplicados

3.6.1 Exemplos em NetLogo

Como já foi dito, exemplos de dinâmicas mais simples foram desenvolvido em NetLogo. Inicialmente, o grafo era montado manualmente para o estudo. O trabalho associado a montar o grafo na orientação inicial foi o principal motivador da criação do ReSATyrus, especialmente para evitar erros possíveis de surgir, por exemplo, ao esquecer de incluir uma aresta na representação.

Uma das primeiras decisões importantes foi escolher entre relacionar cada processo a um agente móvel ou ao planejamento de um grupo de agentes. Para tal foram criados então um programa em NetLogo que associa processos com agentes individuais (Figura 3.17) e outro que planeja o deslocamento de um grupo (Figura 3.18).

Em ambos os modelos os recursos disputados são posições (coordenadas na região delimitada) definidas na região de modo que uma sequência de posições defina um caminho. Essas posições são escolhidas de forma que, contanto que um veículo mantenha o controle sobre uma posição, nenhum outro veículo possa colidir com ele. Elas também são posicionados em intervalos para dividir trechos a serem percorridos em frações menores que podem ser compartilhadas por veículos com o mesmo sentido.

Cada modelagem possui suas vantagens: o modelo de "planejamento", por exemplo, resulta em grafos menores e permite criar escalas que podem ser utilizadas depois sem necessidade de manter o grafo. Por outro lado, equiparar cada processo a um

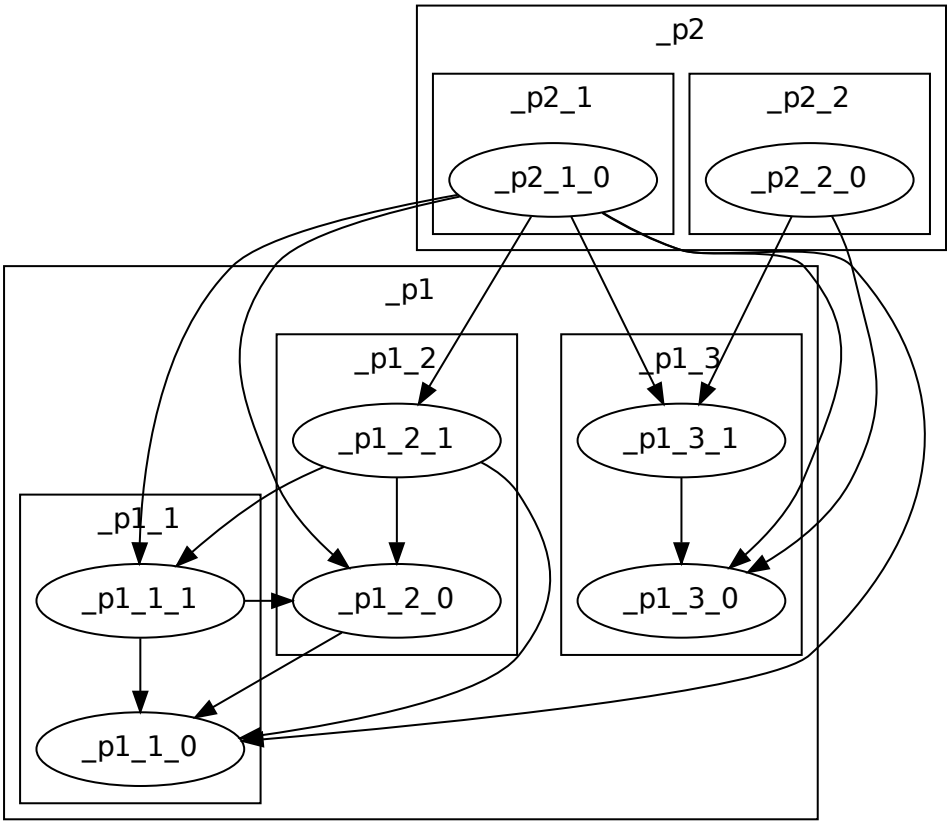

Figura 3.16: Exemplo do grafo de múltiplos nós orientado

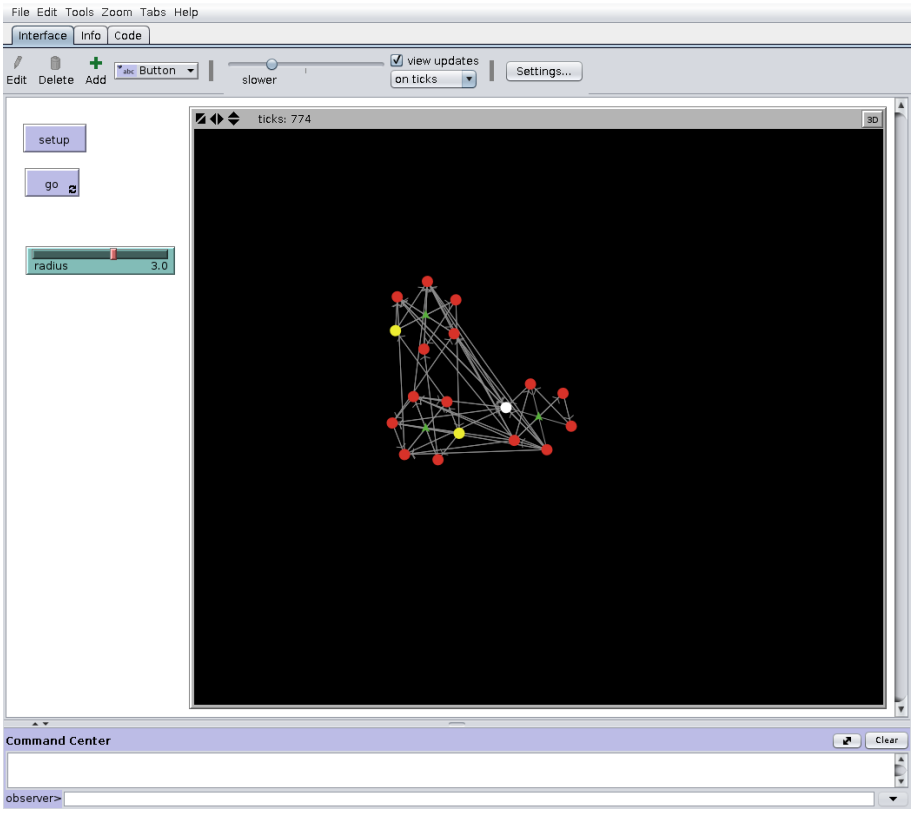

Figura 3.17: Sistema com um processo por robô.

agente permitiu modelagens mais simples, mesmo que mais extensas. A equivalência entre processos e subgrafos do grafo utilizado pelo *SER* ainda permite dividir o processamento entre os agentes, cada um responsável pelo subgrafo correspondente a seu processo.

No primeiro caso, a expressão de recursos de um processo é representada pela conjunção dos recursos envolvidos, com o uso de negação para impedir que recursos correspondentes a pontos adjacentes estejam ativos ao mesmo tempo. Como exemplo, uma rota que passe por pontos identificados como `a`, `b`, `c` e `d` ficaria `(not d and a and not d) or (not a and b and not c) or (not b and c and not d) or (not c and d and not a)`. Já no segundo, é utilizada a associação com ou-exclusivo dos recursos envolvidos. Utilizando o mesmo exemplo base ficaria `xor(a, b, c, d)`. Em ambos a orientação é importante para garantir que os recursos serão acessados na ordem correta. Inclusive, com as orientações apropriadas, ambos poderiam utilizar a mesma representação do primeiro exemplo. O ReSATyrus possui a restrição `optgroup` para auxiliar nessa etapa.

Assim, pode-se afirmar que cada modelagem adequa-se melhor a diferentes tipos de problemas. O planejamento permite gerar escalas de tarefas para grupos de agentes e já foi utilizado para modelar sistemas de manufatura [7]. Já representar os agentes móveis por processos individualmente adapta-se melhor para desenvolver controles para agentes autônomos, dispensando um controle centralizado. O exemplo

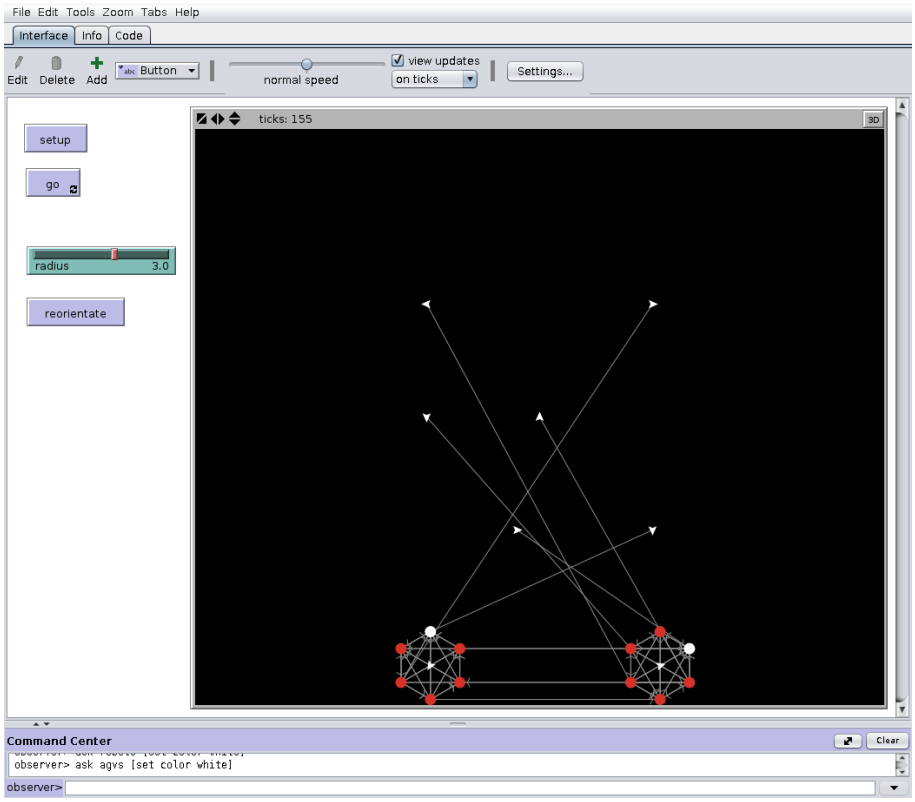

Figura 3.18: Sistema com planejamento de rotas.

da Subseção 3.6.2 continuou o estudo desse estilo de modelagem para explorar seu potencial como mecanismo para coordenação entre veículos.

3.6.2 Controle de trens

Para estudo, foi criado um exemplo tendo como foco o seguinte problema: dado um conjunto de trilhos e rotas sobre esses trilhos, como pode um conjunto de trens trafegar sem colisões? Como restrição adicional, dois trens não podem compartilhar um trecho de trilhos.

O programa para implementar a dinâmica usa o ReSATyrus para controlar a dinâmica *SER*. Dessa forma, tanto os conceitos utilizados na coordenação dos agentes quanto o próprio funcionamento do ReSATyrus são testados. Embora o programa com uso do ReSATyrus tenha uma natureza centralizada, o mecanismo utilizado mantém seus elementos que permitem uma implementação distribuída. A Figura 3.19 retrata a aparência da aplicação, enquanto a Figura 3.20 mostra o código correspondente usado pelo ReSATyrus internamente.

O código demonstra a modelagem dos trens como processos. Cada identificador de processo `t`<*número*> corresponde a um único trem. Os processos com identificadores terminados por `_temp` relacionam-se com o processo com identificar semelhante, indicando que o trem correspondente mudou de rota. A implementação

do sistema de controle usa esses processos temporários para gerenciar os recursos que estavam em uso pelo trem quando houve a mudança de rota. Conforme um subprocesso temporário libera seus recursos, os subprocessos correspondentes deixam de existir, até que o processo inteiro termine e reste apenas o processo principal do trem.

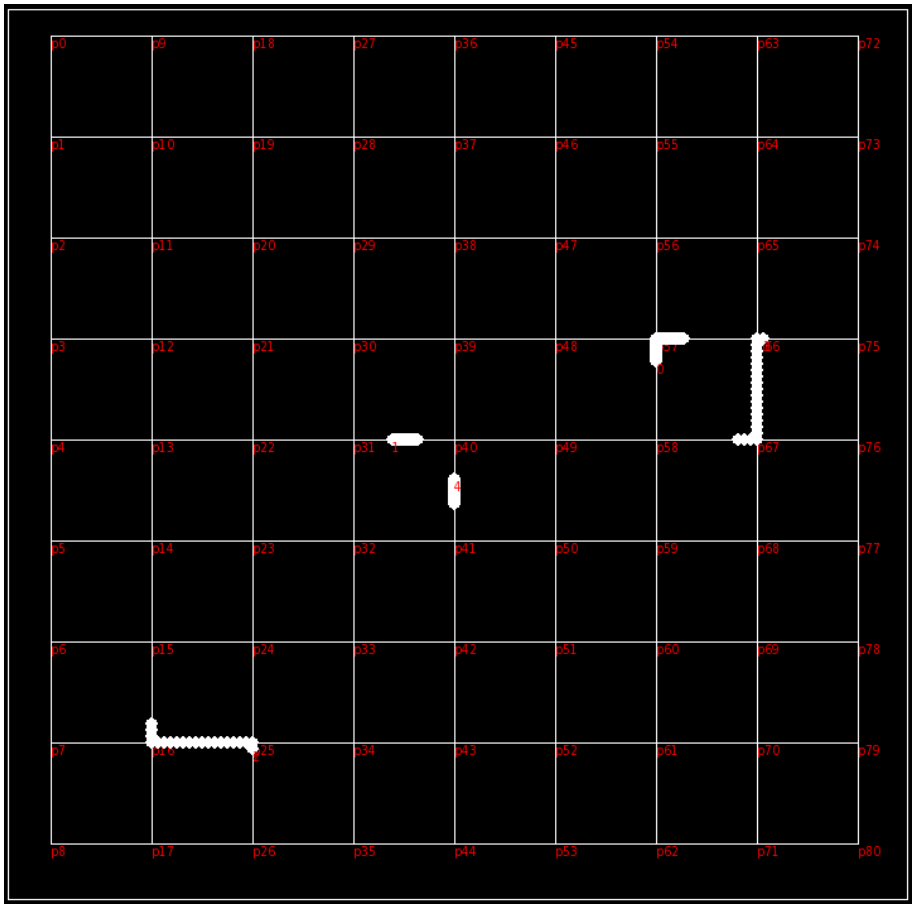

Figura 3.19: Exemplo de Controle de Trens.

As primeiras iterações deste programa seguiram nos moldes definidos anteriormente: cada processo corresponde a um veículo individual, e a expressão é construída de forma a haver no máximo um subprocesso ativo por processo, o qual corresponde ao próximo ponto de deslocamento do agente. O comprimento dos trens adiciona um grau de complexidade à expressão.

No caso apresentado na Subseção 3.6.1, os veículos eram curtos e não ocupariam mais que uma posição do mapa (recurso a ser disputado) a cada momento. Para eles bastavam então representações que garantissem o uso de um recurso, o que pode ser verificado na presença de subgrafos completos nas Figuras 3.17 e 3.18. Um trem ou qualquer outro veículo comprido utilizado em situação semelhante pode ocupar mais de um ponto ao mesmo tempo, e é importante garantir que outro agente não tente deslocar-se para esse ponto enquanto há presença do trem anterior. Uma solução possível seria desenvolver expressões que possuíssem nós correspondentes para cada

```
t0 = 1;
t1 = 1;
t2 = 1;
t2_temp = 1;
t3 = 1;
t3_temp = 1;
t4 = 1;
intgroup t0: p57 or p58 or p67 or p66;
intgroup t1: p31 or p32 or p41 or p40;
intgroup t2: p25 or p26 or p17 or p16;
intgroup t2_temp: p16;
intgroup t3: p75 or p76 or p67 or p66;
intgroup t3_temp: p67 or p66;
intgroup t4: p40 or p49 or p50 or p41;
```

Figura 3.20: Descrição do compartilhamento de recursos pelos trens modelada como um sistema de processos.

possível combinação de uso de recursos por cada trem. Tal abordagem iria requerer capacidade de representação quadrática em relação ao número de "pontos" no mapa. Entretanteo, outra solução alternativa foi implementada para o problema-exemplo: tratar a ocupação de pontos como subprocessos que podem ocorrer simultaneamente.

Nessa modelagem, cada processo possui apenas uma conjunção dos recursos utilizados. Tendo, outra vez, o mesmo trajeto de exemplo, ele ficaria `a or b or c or d`. Cada veículo possui um controlador interno que define qual o próximo ponto que ele tem que alcançar. Uma vez que ele tenha prioridade para utilizar o recurso correspondente, o recurso permanece em seu poder até o corpo do trem passar totalmente por cima dele.

Por vezes isso pode resultar em pequenos contratempos: por exemplo, se dois trens com velocidades muito distintas compartilham um ponto, é possível que o mais rápido tenha que ficar parado esperando o outro completar sua volta, embora pudesse prosseguir com seu caminho. Uma modificação que resolve este contratempo é realizar a reversão de arestas em qualquer subprocesso que não esteja sendo utilizado pelo agente no momento, isto é, correspondente a um ponto que não está sendo percorrido nem seja o alvo mais próximo. Considerando que não há colisões pré-existentes e as propriedades de organização do modelo, as propriedades do *SER* garantem que não haverá colisões contanto que nenhum recurso em uso seja liberado e a operação do programa demonstra isso.

Outra modificação que este programa apresenta é a capacidade de modificar rotas. Cada vez que um veículo atinge uns dos pontos de sua rota, ele pode decidir permanecer, ou mudar para outra rota que compartilhe aquele ponto. Para realizar essa operação, o trem modifica sua representação de uso de recursos para adequar-

se à nova rota e invoca o ReSATyrus. Este gera um novo grafo considerando as relações criadas pela nova representação. O programa cuida de garantir que relações de prioridades já existentes sejam preservadas conforme a necessidade, por exemplo, um trem que esteja sobre um nó precisa garantir a prioridade dele.

A execução do programa revela dois possíveis problema: (i) travamentos por rotas conflitantes, (ii) travamento por excesso de veículos. A primeira situação ocorre quando dois veículos tentam percorrer o mesmo trecho em direções opostas, logo um mantém inacessível o recurso de que o outro precisa para continuar. O segundo ocorre quando há tantos trens em um trajeto que não restam recursos livres para um trem deslocar-se, logo ficam todos parados.

Deve ser observado ainda que a quantidade de veículos simulados pode causar lentidão no programa, visto que o grafo inteiro precisa ser recriado. A modificação apenas de processos alterados permitiria uma possibilidade para acelerar esse processo, porém o ReSATyrus ainda não apresenta essa funcionalidade. Essa limitação é relativa ao tamanho do grafo gerado. Testes iniciais com o uso do `xor` resultavam em grafos muito grandes e densos, com tempos de desenvolvimento proibitivos.

3.6.2.1 Rotas conflitantes

Um trecho que não pode ser compartilhado entre dois veículos em sentidos contrários precisa ser modelado de modo que esses veículos não o possam acessar simultaneamente. Contudo, ao construir a descrição de um sistema complexo com múltiplos veículos é possível que um trecho sujeito a veículos seja partido em secções, por exemplo, para permitir cruzamentos em diferentes alturas. Nesse caso, uma solução é utilizar um recurso extra correspondente ao conjunto inteiro, que será utilizado pelos veículos que trafeguem pelo trecho disputado para assegurar que não venha outro pela direção oposta.

A Figura 3.21 servirá como base para demonstrar essa ideia. Considerando um veículo que passe por A, B, C e D, enquanto outro passe por C, B, A e E, ambos na ordem apresentada, haverá ameaça de colisão no trecho delimitado por A, B e C. Para ambos a restrição relativa a esses recursos é `A or B or C`, por exemplo. A adição de um recurso para controle consistiria de mudar a restrição para `(A and R) or (B and R) or (C and R)`. Alternativamente, pode-se modelar o trecho inteiro como um só recurso, mas isso pode afetar negativamente a modelagem, por exemplo, numa grade semelhante à utilizada na Figura 3.19.

Quando esta situação não for evitada, as consequências do erro dependem do restante da descrição dos processos existentes. Nas abordagens com uso de `xor`, cada veículo libera o recurso que ocupa antes de deslocar-se para o próximo. A consequência então seria uma colisão frontal entre veículos. Por outro lado, com o uso do `or` cada veículo manteria o uso dos trechos que ocupa, o resultado então seria

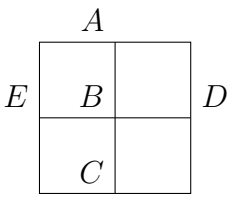

Figura 3.21: Exemplo onde pode haver colisão por rotas conflitantes.

um travamento do sistema, como foi observado em experimentos.

3.6.2.2 Excesso de veículos

Esta situação pode ocorrer quando há uma série de trechos conectados de modo a formar um ciclo. Considerando cada trecho como correspondente a um recurso, é natural que se houver tantos veículos quanto trechos, eles não possam mover-se pois nenhum deles vai abandonar sua posição para outro entrar. Havendo uma rota que percorra esse ciclo nessa situação, pode-se dizer que a rota estará sobrecarregada.

Embora seja simples não sobrecarregar uma rota, bastando colocar menos veículo do que a quantidade máxima, o mesmo problema pode ocorrer em outras condições. Um exemplo é um sistema com duas rotas que se cruzam, gerando um ciclo no meio, como na Figura 3.22. Embora cada uma das rotas separadamente suporte até cinco veículos sem sobrecarregar-se, a sobreposição delas forma um círculo com os nós B, C, E e F. Com dois veículos em cada rota então é possível que ocorra um travamento.

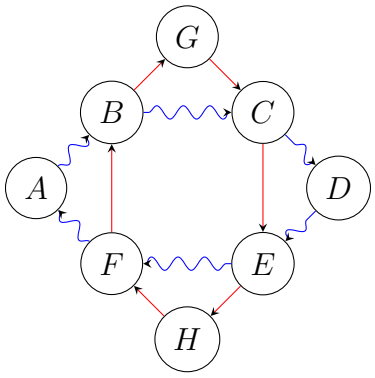

Figura 3.22: Exemplo de cruzamento de rotas que pode causar um problema com excesso de veículos.

O problema relatado sobre o grafo da Figura 3.22 pode ser visto como uma generalização do problema anterior, se dois veículos indo em direções opostas forem considerados como percorrendo um ciclo composto por dois trechos. A Figura 3.23 exemplifica a ideia. Naturalmente, então, a mesma solução aplica-se, utilizando um recurso correspondente ao ciclo para restringir a quantidade de veículos. Quanto mais trechos possui o ciclo, contudo, menos eficiente a solução será. Um modo de

desenvolver essa solução seria usar um recurso com contador, de modo que ele possa ser utilizado por uma certa quantidade de agentes ao mesmo tempo. O desenvolvimento de uma solução ficará para trabalhos futuros.

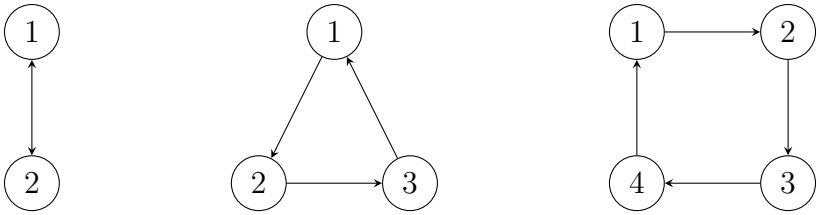

Figura 3.23: Ciclos de comprimento 2, 3, 4.

Capítulo 4

Orientação do *SMER*– multi-Alg

Na Seção 2.3.2 for apresentada uma heurística simples para gerar multidigrafos próprios para uso com *SMER*. Essa heurística é contudo limitada, em particular em situações com vários nós com a mesma reversibilidade. Nesta seção apresenta-se uma nova alternativa baseada na ideia de adaptar as heurísticas de orientação do *SER*, que será chamada de multi-Alg. Como o multi-Alg depende de uma orientação acíclica obtida previamente, uma heurística utilizada como base pode ser indicada ao final do nome, por exemplo, multi-Alg-Colors ou multi-Alg-Edges.

Para todo multigrafo G^M existe um grafo G equivalente, condensando cada conjunto de múltiplas arestas entre x e y em uma única aresta. A partir de G é possível gerar um digrafo D para uso no *SER*. D^M é gerado então com a seguinte regra: para cada arco de y para x em D, orientam-se os arcos de acordo com a seguinte equação:

$$\begin{cases} a_{xy} = r_x \\ a_{yx} = r_y - \text{mdc}(r_x, r_y) \end{cases} \tag{4.1}$$

A Figura 4.1 mostra um exemplo da conversão.

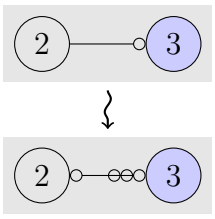

Figura 4.1: Conversão realizada pelo multi-Alg.

Algoritmo 4.1: multi-Alg

Variáveis:

V^+, inicializado pela orientação

V^-, inicializado pela orientação

$V \leftarrow V^+ \cup V^-$

$e_j, \forall j \in V$
$r_j, \forall j \in V$, já inicializado
$status \leftarrow SER$

Entrada:
$msg_i = \emptyset$ ou $msg_i = smer$ e $status = SER$
Ação:
$status \leftarrow SMER$
Para cada $v \in V$**:**
Se $v \in V^+$**:**
$e_v \leftarrow r$
Fim Se
Se $v \in V^-$**:**
$e_v \leftarrow r - \text{mdc}(r, r_v)$
Fim Se
Enviar smer **para** v

4.1 Corretude do multi-Alg

É possível provar que todo D^M gerado por esse método respeitará as condições apresentadas na Equação 2.4 para formação de período. Segue-se a prova por absurdo.

Teorema 4. *multi-Alg resulta em multidigrafos com período SMER*

Demonstração. Prova por absurdo. Considere o grafo G, com D obtido de G e D^M obtido de D, D acíclico. Assuma que D^M não permite formação de período *SMER*. Pela Equação 2.4, existe em D^M pelo menos um ciclo C^M tal que:

$$\max(\sum_{(x,y)\in C} a_{xy}, \sum_{(x,y)\in C} a_{yx}) \geq \sum_{x\in C} r_x \tag{4.2}$$

Considerando o ciclo C em G correspondente a C^M, define-se uma variável γ_{xy} tal que:

$$\gamma_{xy} \begin{cases} 1 & \text{se } (y \to x) \text{ em } D \\ 0 & \text{caso contrário} \end{cases} \tag{4.3}$$

e $\gamma_{yx} = 1 - \gamma_{xy}$.

A Figura 4.2 mostra um exemplo de uso do γ_{xy}. Observa-se que, sendo y anterior a x e z anterior a y ao percorrer o ciclo C, então $\gamma_{xy} = \gamma_{yz}$ se e somente se $(x \leftarrow y), (y \leftarrow z) \in D$ ou $(x \to y), (y \to z) \in D$. Em outras palavras, percorrendo-se o

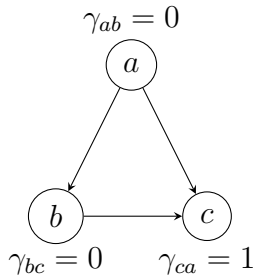

Figura 4.2: Exemplo de disposição de valores do γ_{xy}.

ciclo C em um sentido, o valor de γ_{xy} para um nó x e seu nó anterior y permanece constante apenas se o sentido dos arcos percorridos for constante.

Conforme declarado na Equação 4.1, a orientação resultante para cada par (x, y) de D^M possui $a_{xy} = r_x$ e $a_{yx} = r_y - \text{mdc}(r_x, r_y)$, se $\gamma_{xy} = 1$, com valores correspondentes no sentido contrário se não. Assim, o total de arestas orientadas em um determinado sentido em um ciclo C é de:

$$\sum_{(x,y)\in C} r_x - \gamma_{yx} \,\text{mdc}(x, y) \tag{4.4}$$

Como foi assumido que essa orientação não permite formação de período:

$$\sum_{(x,y)\in C} r_x - \gamma_{yx} \,\text{mdc}(x, y) \geq \sum_{x\in C} r_x \tag{4.5}$$

Como o máximo divisor comum de dois inteiros positivos é também um inteiro positivo, a Equação 4.5 se e somente se $\gamma_{yx} = 0$ para todo $x \in C$ em um sentindo, o que corresponde a um ciclo em D, conforme ilustrado pela Figura 4.2. Tal ocorrência contradiz o que foi definido ao início da demonstração, logo ocorre um absurdo. Portanto D^M permite formação de período sempre que for orientado conforme definido no Algoritmo 4.1. □

O ReSATyrus utiliza este método para gerar as orientações *SMER*. As mesmas heurísticas disponíveis para orientar o *SER* podem ser escolhidas para gerar a orientação *SMER*. Uma comparação da concorrência obtida por cada heurística está disponível na seção seguinte.

4.2 Testando multi-Alg

O multi-Alg converte uma orientação acíclica obtida por um digrafo para uma orientação utilizável pelo *SMER*. As diferentes heurísticas existentes para criar uma orientação acíclica de forma distribuída impactam a concorrência e o tamanho do

período de formas diferentes [5]. É interessante então verificar se essas heurísticas, quando convertidas pelo multi-Alg, possuem efeitos semelhantes para o *SMER*.

Para testar o multi-Alg foi desenvolvido um programa de testes capaz de executar o *SMER* e o *SER*. A interface pode ser vista na Figura 4.3. Primeiramente o programa gera um grafo aleatório distribuindo uma quantidade definida de nós em um espaço limitado. A distância entre nós é usada como medida para definir criação de arestas e pode ser regulada para gerar grafos de diferentes densidades. Se a criação do grafo resultar em componentes desconexas o programa cria arestas adicionais até que o grafo seja conexo. Durante a criação do grafo, cada nó recebe ainda uma reversibilidade definida de forma aleatória, dentro de um valor máximo previamente estabelecido como parâmetro do teste.

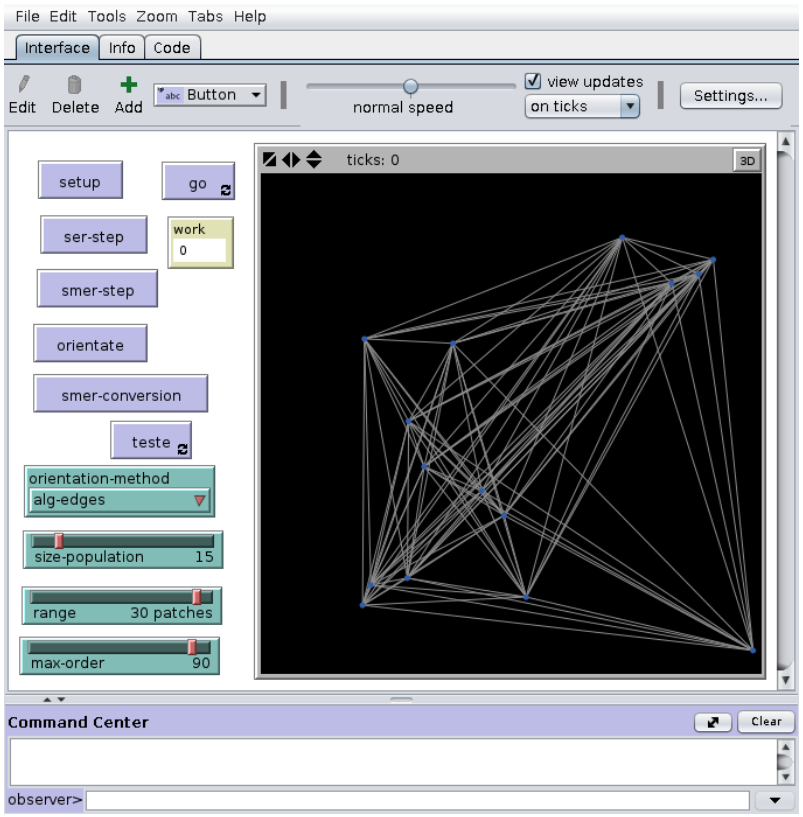

Figura 4.3: Interface do programa para testes do multi-Alg

O programa segue então com o uso de uma das heurísticas anteriormente discutidas para gerar uma orientação acíclica para iniciar o *SER*. Foram testados o Alg-Neighbors, o Alg-Colors e o Alg-Edges. O programa executa por um número definido de iterações. A cada iteração todos os sorvedouros formados anteriormente revertem suas arestas. Depois de um número de reversões definido pelo usuário, a orientação é convertida para um modelo utilizável pelo *SMER* utilizando o multi-Alg ou a alternativa presente em [4]. Repete-se então a mesma quantidade de iterações, revertendo-se agora as arestas segundo as regras do *SMER*.

Para os testes foram usadas as seguintes medidas:

distância máxima variada em 1, 10, 20 e 30 unidades (a área do teste tem lateral de 33 unidades);

tamanho da população fixado em 100 nós;

reversibilidade variável de 10 a 100 em incrementos de 10 unidades, isso significa que na menor reversibilidade um nó pode agir no máximo dez vezes comparado a outro, enquanto na máxima pode agir até cem vezes.

heurística do *SER* variável entre as três heurísticas apresentadas: Alg-Neighbors, Alg-Colors, Alg-Edges.

Para a medir a concorrência, como todos os testes rodaram pelo mesmo número de iterações e com o mesmo número de vértices, foi medido apenas o total de vezes que cada nó tornou-se um sorvedouro. Os testes foram repetidos 100 vezes por grafo para gerar médias e variâncias e esses testes foram repetidos 84 vezes ao todo. Os dados obtidos foram agrupados pelos parâmetros para permitir comparação. Para comparar os efeitos de diferentes valores para reversibilidade máxima do *SMER*, também foi preservado o grafo, alterando-se as reversibilidades de cada nó para cada teste.

4.2.1 Comparação entre variantes

Os dados foram organizados em gráficos para análise dos testes. Todos os gráficos colocam no eixo vertical as medidas de concorrências. O eixo horizontal pode mostrar ou grafos criados com diferentes parâmetros (alcance de arestas ou reversibilidade máxima) ou grafos criados sob os mesmos parâmetros, para ilustrar a variação da concorrência. O alcance das arestas foi utilizado durante a criação dos grafos para definir quais nós estavam conectados, de forma que um maior alcance corresponde a um grafo mais denso. A reversibilidade dos nós também foi definida aleatoriamente em um intervalo que vai de 1 até o valor máximo definido.

Primeiramente, verificou-se que as medidas observadas no *SER* correspondem aos mesmos resultados encontrados em [5]. As Figuras 4.4 e 4.5 mostram os dados encontrados. Também pode-se observar que as heurísticas não oferecem tanta diferença nas menores densidades, possivelmente por ser mais provável alcançar a orientação ótima, já que o grafo aproxima-se de uma árvore. A influência de diferentes heurísticas também diminui um pouco com o aumento da densidade conforme o grafo aproxima-se de ser um grafo completo. A diferença entre o Alg-Neighbors e o Alg-Edges tende também a ser menor do que entre o Alg-Neighbors e o Alg-Colors.

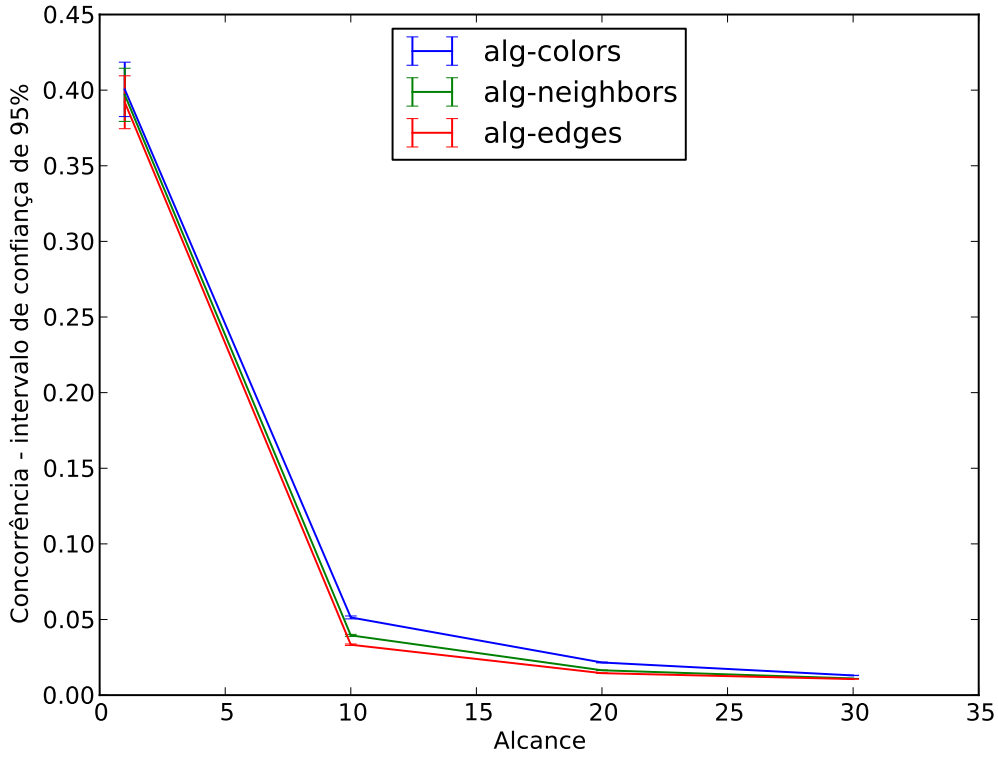

Figura 4.4: Influência do alcance na concorrência.

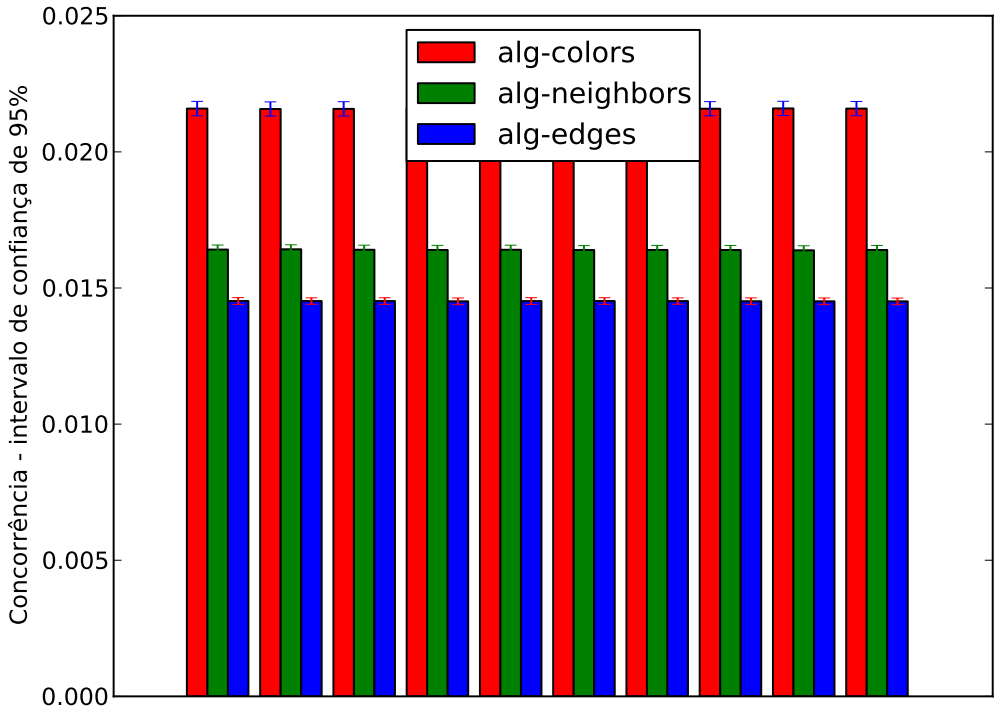

Figura 4.5: Comparação entre grafos de mesma densidade.

Para a aplicação no *SMER* a diferença entre as heurísticas foi menor. Em alguns casos, quanto maior a diferença possível entre reversibilidades, menor a diferença de influência entre as heurísticas. As Figuras 4.6, 4.7 e 4.8 ilustram os pontos apresentados. Quanto maior a diferença de reversibilidade entre os nós, menor a diferença na concorrência entre grafos orientados a partir das diferentes heurísticas. Isso pode ser mais pronunciado, como na Figura 4.6, ou menos, como nas Figuras 4.7 e 4.8. Nota-se ainda que essa aproximação acontece tanto para o multi-Alg quando para o método de orietanção presente em Santos [4], identificado nos gráficos pelo sufixo `-alex`.

Comparado ao algoritmo de orientação presente em Santos [4], o multi-alg apresenta concorrência menores em grafos pouco densos, conforme aumentam as diferenças de reversibilidade, porém em grafos mais densos ambos são comparáveis. Na Figura 4.6 nota-se claramente que o multi-Alg apresenta um desempenho pior para diferenças de reversibilidade maior, mas essas diferenças desaparecem nas Figuras 4.7 e 4.8.

Ambos os gráficos também demonstram uma relação aparentemente caótica entre maior valor possível de reversibilidade e concorrência alcançada, embora seja possível notar uma certa coerência nos nas variações de cada algoritmo. Essa relação caótica é muito mais visível examinando os resultados para um grafo individual, como presente nas Figuras 4.9, 4.10 e 4.11. Isto também pode ser visto como as variações sendo as mesmas para cada algoritmo, independente do método utilizado para gerar a orientação *SER* utilizada como base. Tal comportamento ainda não possui uma causa definida nem relações observáveis com os parâmetros envolvidos.

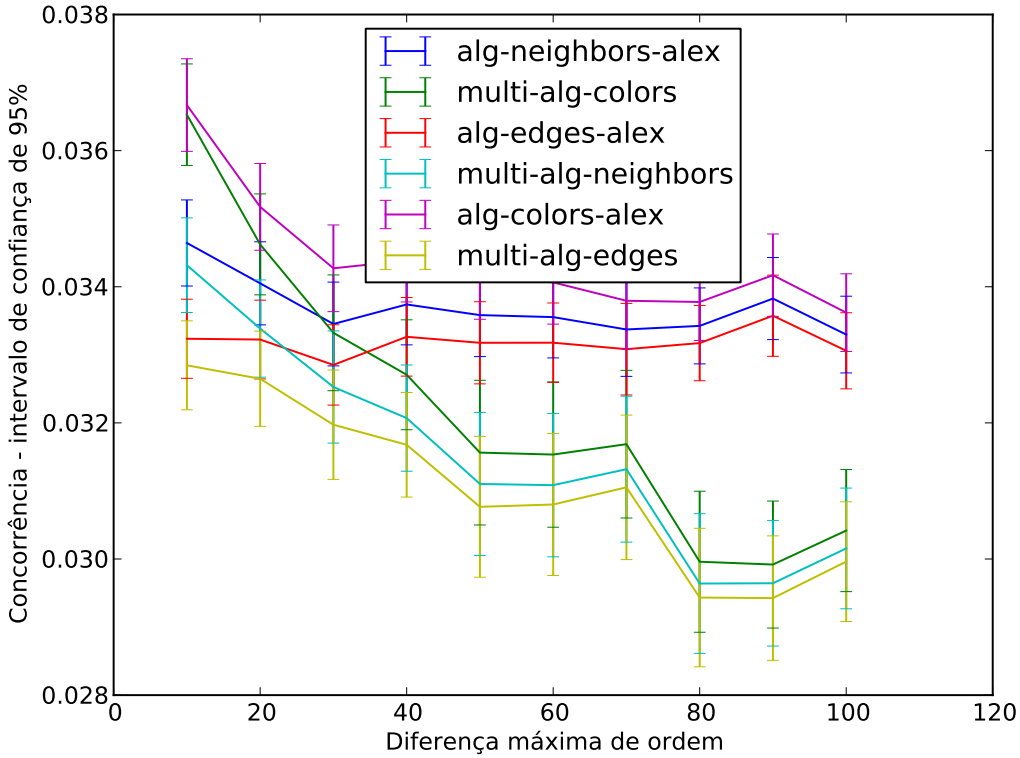

Figura 4.6: Concorrência para *SMER* em grafos criados com alcance 10.

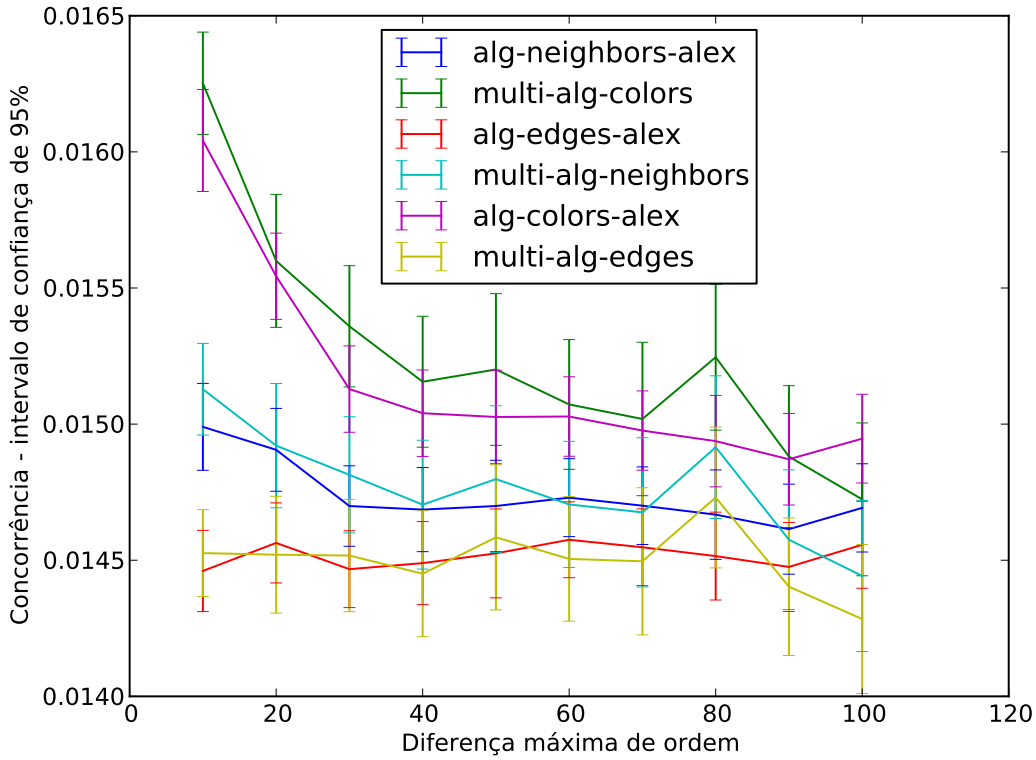

Figura 4.7: Concorrência para *SMER* em grafos criados com alcance 20.

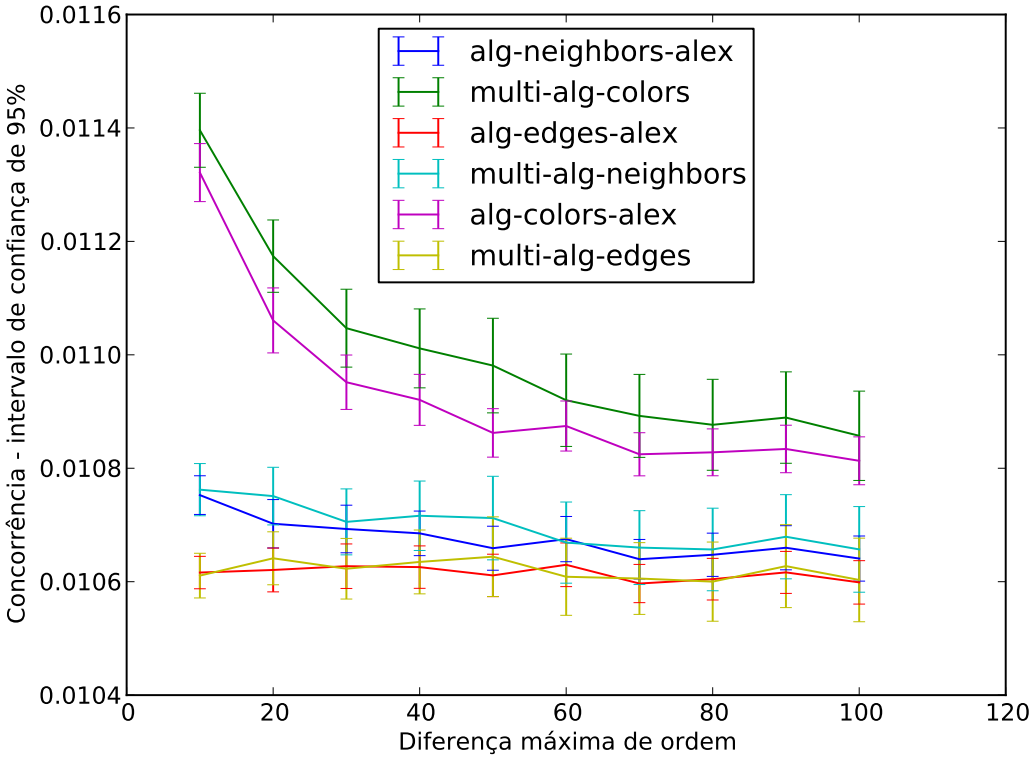

Figura 4.8: Concorrência para *SMER* em grafos criados com alcance 30.

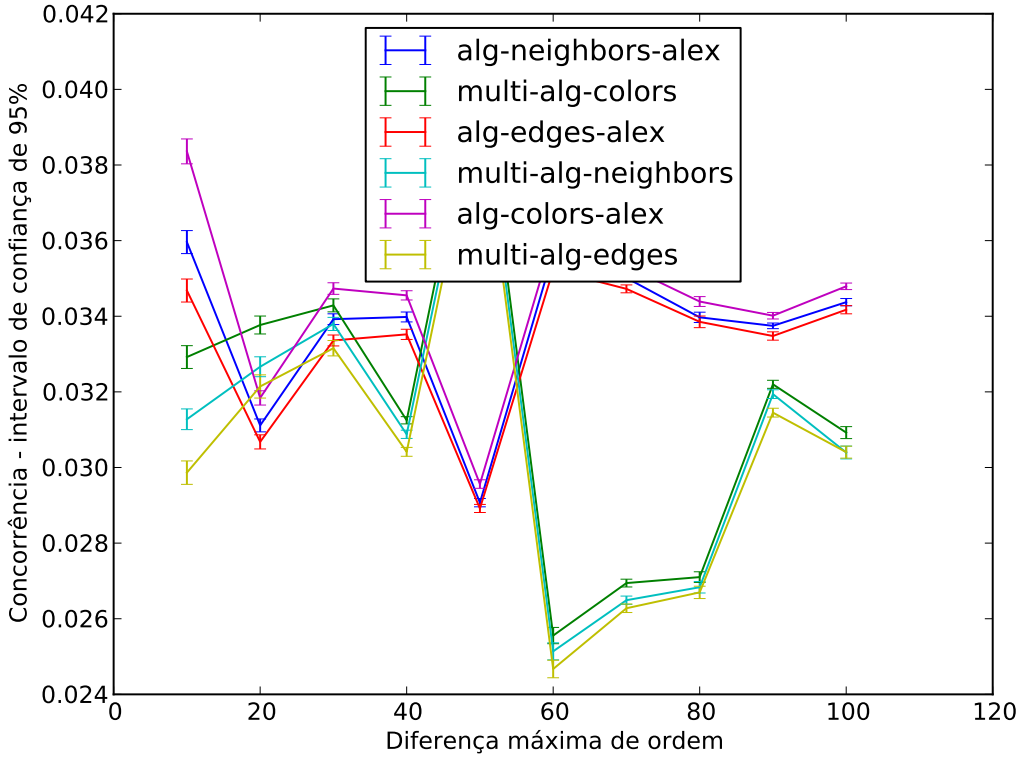

Figura 4.9: Concorrência para *SMER* um grafo criado com alcance 10.

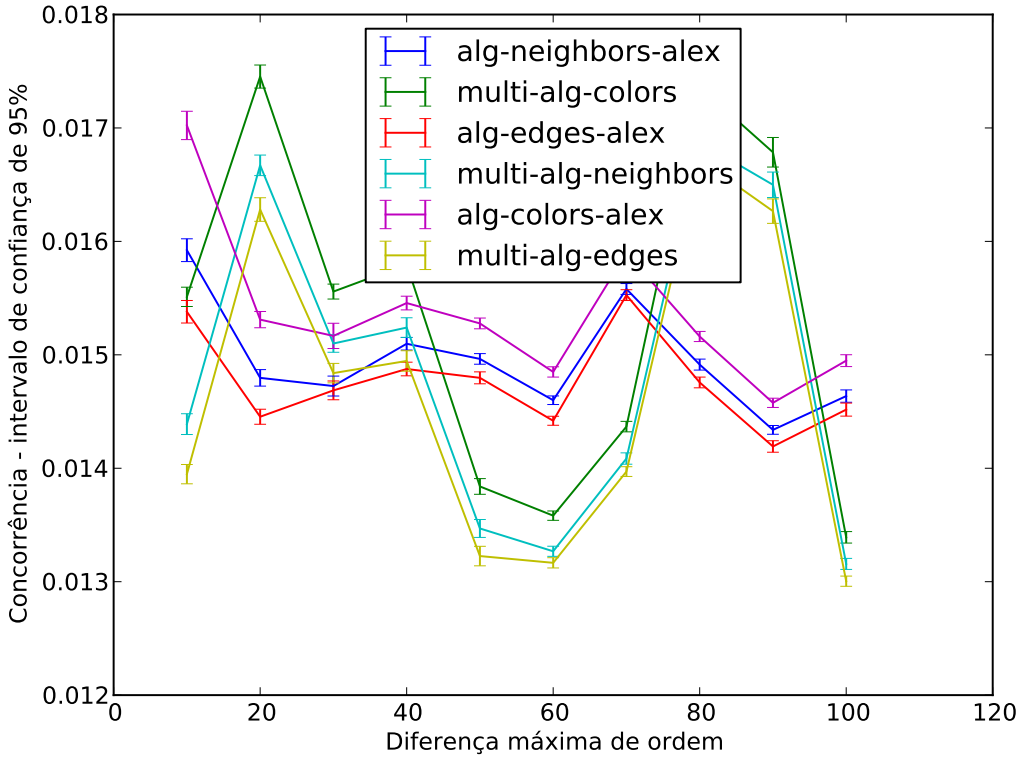

Figura 4.10: Concorrência para *SMER* um grafo criado com alcance 20.

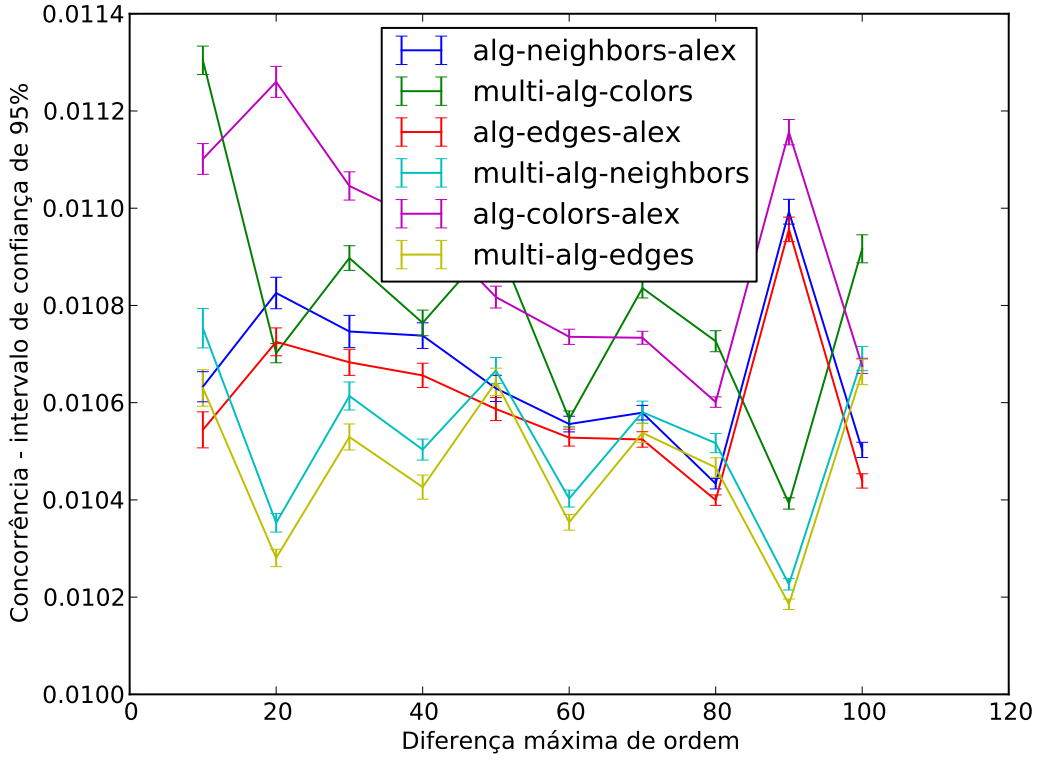

Figura 4.11: Concorrência para *SMER* um grafo criado com alcance 30.

Capítulo 5

Conclusão

De modo geral, o ReSATyrus conseguiu atingir a meta de gerar grafos para os mecanismos *SER* e *SMER*, o que é melhor demonstrado nos exemplos de dinâmicas. O método para gerar orientações utilizáveis pelo *SMER* também foi demonstrado como funcional, com uma variação dependendente da heurística utilizada como base e da reversibiliade dos nós. Há, contudo, ainda vias que podem ser exploradas para continuar o desenvolvimento do ReSATyrus.

5.1 Sumário

O *SER* é um mecanismo de compartilhamento de recursos com implementação distribuída. Uma implementação por um sistema de troca de mensagens permitiu que agentes distribuídos compartilhem um conjunto limitado de recursos. Representando então o ambiente de movimento nesses recursos, o escalonamento por reversão de arestas ofereceu um sistema para evitar colisões. A existência de algoritmos distribuídos para gerar a orientação acíclica foi um aspecto importante para a aplicação do mecanismo. A modelagem inicial com uso de processos concorrentes gerados por uso de `xor` revelou-se um pouco limitada, sendo substuída por processos paralelos gerados por `or`.

SMER ofereceu uma solução para situações que necessitem de uso desigual. Embora considerado para melhor lidar com situações em que os veículos alcancem cruzamentos em intervalos desiguais, a modelagem alternativa do *SER* foi mais simples e apresentou resultados satisfatórios.

O multi-Alg ofereceu uma nova forma para gerar orientações apropriadas para uso com o *SMER* por agentes anônimos. Como existem múltiplas heurísticas capazes de gerar orientações acíclicas nestas condições, foi possível então adaptar o *SMER* para os mesmos ambientes. As diferenças entre heurísticas diminuíram consideravelmente, contudo, quando há diferença significativa de reversibilidades.

O ReSATyrus facilitou a análise de modelagens diferentes tanto para o *SER* como

para o *SMER*. Sua criação foi fundamental para comparar diferentes formas para modelar o problema, permitindo a comparação rápida com risco menor de erros. Embora seja uma ferramente centralizada, a análise dos modelos permitiu desenvolver aplicações distribuídas mais eficientes. Trabalhos futuros podem desenvolver versões paralelas da ReSATyrus, de modo que diferentes agentes possam ter seu gerador de grafos, que interage com os geradores de grafos de outros agentes para criar as interações entre processos.

A aplicação no controle de veículos ofereceu um teste interessante para o *SER* e o ReSATyrus. A necessidade de gerar novos grafos conforme os veículos alterem suas rotas utilizou as capacidades do ReSATyrus com sucesso. Foi importante ainda considerar o impacto da geração contínua dos grafos no sistema. Em um programa para exemplo das dinâmicas, o ReSATyrus representou um custo considerável em tempo conforme aumenta a quantidade de veículos. Uma abordagem mais otimizada, que não precise gerar o grafo inteiro a cada alteração, ofereceria uma resposta mais rápida.

5.2 Trabalhos futuros

Apesar do trabalho já realizado no ReSATyrus, ainda restam pontos que podem ser futuramente investigados. A linguagem apresentada constitui um primeiro esboço, que apresenta apenas as funcionalidades essenciais. O programa de exemplo mesmo, como já indicado, pode ser otimizado. Além disso, uma implementação física do problema apresentado nos exemplos seria desejável. Seria interessante interligar o ReSATyrus e o SATyrus, tentando relacionar os problemas tratados por ambos. O travamento por veículos em sentidos opostos ou sobrecarregando um ciclo necessita de um método para ser evitado. O multi-alg também apresenta pontos a serem expandidos.

A linguagem utilizada pelo ReSATyrus contém atualmente apenas as estruturas necessárias para expressar os problemas. Para desenvolvimento e teste, este conjunto foi suficiente, contudo ele apresenta-se limitado para expressão de problemas maiores. Não há, por exemplo, funcionalidades para evitar repetição de código, que seriam úteis para estruturas semelhantes, mas com pequenas variações, como um conjunto de subprocessos que que precisam de dois recursos, um comum e o outro único para aquele subprocesso.

Também já foi apontado que o programa usado para testar o ReSATyrus apresenta baixo desempenho quando lida com múltiplos veículos. Essa implementação pode ser melhorada, evitando a geração do grafo completo a cada modificação. Idealmente, o ReSATyrus pode ainda ser modificado para poder trabalhar de forma distribuída, de modo que cada fração seja responsável por um processo e possa

interagir com as outras para manter o grafo completo.

Essas modificações poderiam ainda colaborar para uma implementação prática dos exemplos. Alternativamente, essa implementação poderia usar as ideias encontradas nos testes com a ReSATyrus aplicadas diretamente no controle dos veículos. Tal demonstração reforçaria a possibilidade de usar o *SER* para controlar veículos em ambientes com rotas previamente mapeadas.

Seria ainda uma tarefa promissora reunir o ReSATyrus e o SATyrus. Apesar de realizarem tarefas distintas, seria possível talvez utilizar o SATyrus para encontrar a orientação ótima de um grafo gerado pelo ReSATyrus, por exemplo. Este é um problema NP-completo e semelhante ao problema da coloração, que o SATyrus consegue tratar. Unir as duas ferramentas então ofereceria uma solução ainda mais completa, mesmo que as heurísticas distribuídas sejam mais realistas, em sistemas em que o tempo de resposta seja crítico ou com implementação distribuída.

A Subseção 3.6.2 concluiu com um problema encontrado durante o desenvolvimento do sistema de controle de trens. Algumas condições podem resultar em veículos seguindo em direções opostas ou sobrecarregando um ciclo de modo que nenhum deles pode deslocar-se. Apesar de uma posível solução ser mencionada ao fim da subseção, a mesma não foi implementada. Uma solução que permita evitar esses travamentos seria vital para uso em ambiente de produção desse método para coordenação de veículos.

Finalmente, o estudo do multi-Alg pode ser aprofundado. O multi-Alg precisa ser comparado a outras heurísticas de inicialização do *SMER* para ter uma melhor compreensão de sua utilidade. O comportamento caótico observado na medida da concorrência nas Figuras 4.9 e 4.10 também requer investigação mais profunda.

Referências Bibliográficas

[1] DIJKSTRA, E. "Hierarchical ordering of sequential processes", *Acta Informatica*, v. 1, n. 2, pp. 115–138, 1971. ISSN: 0001-5903. doi: 10.1007/BF00289519. Disponível em: <http://dx.doi.org/10.1007/BF00289519>.

[2] BARBOSA, V. C., GAFNI, E. "Concurrency in heavily loaded neighborhood-constrained systems", *ACM Trans. Program. Lang. Syst.*, v. 11, pp. 562–584, October 1989. ISSN: 0164-0925. doi: http://doi.acm.org/10.1145/69558.69560. Disponível em: <http://doi.acm.org/10.1145/69558.69560>.

[3] BARBOSA, V. C., BENEVIDES, M., FRANÇA, F. M. G. "Sharing Resources at Nonuniform Access Rates", *MST: Mathematical Systems Theory*, v. 34, 2001.

[4] DOS SANTOS, A. A. *Escalonamento Distribuído Livre de Colisões para Redes de Sensores Sem Fio com Múltiplas Fontes e Múltiplos Sumidouros*. Doutorado, PESC, Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia, UFRJ, 2012.

[5] ARANTES JR., G. M., FRANÇA, F. M. G., MARTINHON, C. A. "Randomized generation of acyclic orientations upon anonymous distributed systems", *Journal of Parallel and Distributed Computing*, v. 69, n. 3, pp. 239–246, mar. 2009. ISSN: 0743-7315 (print), 1096-0848 (electronic).

[6] CASSIA, R. F., ALVES, V. C., BESNARD, F. G.-D., et al. "Synchronous-to-Asynchronous Conversion of Cryptographic Circuits", *Journal of Circuits, Systems, and Computers*, v. 18, n. 2, pp. 271–282, 2009. Disponível em: <http://dx.doi.org/10.1142/S0218126609005058>.

[7] ACUNA, H. G., DUTRA, M. S., FRANÇA, F. M. G. "Distributed Control of Job-shop Systems via Edge Reversal Dynamics for Automated Guided Vehicles". In: *INTELLI 2012*, pp. 25–30, Chamonix, France, 2012.

XPS. Disponível em: <http://www.thinkmind.org/index.php?view=article&articleid=intelli_2012_2_10_80020>.

[8] LENGERKE, O., CARVALHO, D., LIMA, P. M. V., et al. "Controle distribuído de Sistemas Job Shop usando Escalonamento por Reversão de Arestas". In: *Procs. of The XIV Latin Ibero-American Congress on Operations Research (CLAIO 2008)*, pp. 1–3, September 2008.

[9] GONÇALVES, V. C. F., LIMA, P. M. V., MACULAN, N., et al. "A Distributed Dynamics for Webgraph Decontamination". In: *4th international conference on Leveraging applications of formal methods, verification, and validation*, ISoLA 2010, pp. 462–472, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN: 3-642-16557-5, 978-3-642-16557-3.

[10] ALVES, D. S. F., GONÇALVES, V. C. F., LIMA, P. M. V., et al. "G-DI: a Graph Decontamination Iterator for the Web". In: *Proceedings of the ACM WebSci'11*, pp. 1–4, Koblenz, Germany, jun. 2011.

[11] ALVES, D. S. F., SOARES, E. E. D. M., STRACHAN, G. C., et al. "Mobile Ad Hoc Robots and Wireless Robotic Systems: Design and Implementation". cap. A Swarm Robotics Approach To Decontamination (in preparation), IGI Global, 2011.

[12] ALVES, D., FRANÇA, F., DE MACEDO MOURELLE, L., et al. "The Effect of Intelligent Escape on Distributed SER-Based Search". In: Murgante, B., Gervasi, O., Misra, S., et al. (Eds.), *Computational Science and Its Applications – ICCSA 2012*, v. 7333, *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 101–112, 2012. ISBN: 978-3-642-31124-6. Disponível em: <http://dx.doi.org/10.1007/978-3-642-31125-3_8>. 10.1007/978-3-642-31125-3_8.

[13] SZWARCFITER, J. L., MOSCARINI, M., PETRESCHI, R. "On Node Searching and Starlike Graphs". In: *Congressus Numerantium*, v. 131, pp. 5–84, 1998.

[14] CENTENO, C. C., DOURADO, M. C., PENSO, L. D., et al. "Irreversible conversion of graphs", *Theor. Comput. Sci*, v. 412, n. 29, pp. 3693–3700, 2011. Disponível em: <http://dx.doi.org/10.1016/j.tcs.2011.03.029>.

[15] ALVES, D. S. F., FRANÇA, F. M. G., DE MACEDO MOURELLE, L., et al. "The Effect of Intelligent Escape on Distributed SER-Based Search". In: Murgante, B., Gervasi, O., Misra, S., et al. (Eds.), *ICCSA*

(1), v. 7333, *Lecture Notes in Computer Science*, pp. 101–112. Springer, 2012. ISBN: 978-3-642-31124-6. Disponível em: <http://dx.doi.org/10.1007/978-3-642-31125-3>.

[16] LIMA, P., MORVELI-ESPINOZA, M., PEREIRA, G., et al. "SATyrus: a SAT-based neuro-symbolic architecture for constraint processing". In: *Hybrid Intelligent Systems, 2005. HIS '05. Fifth International Conference on*, pp. 6 pp.–, 2005. doi: 10.1109/ICHIS.2005.97.

[17] MONTEIRO, B. F. *SATyrus2: Compilando Especificações de Raciocínio Lógico*. Mestrado, PESC, Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia, UFRJ, 2010.

[18] KELB, P., MARGARIA, T., MENDLER, M., et al. "Mosel: A flexible toolset for monadic second-order logic". In: Brinksma, E. (Ed.), *Tools and Algorithms for the Construction and Analysis of Systems*, v. 1217, *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 183–202, 1997. ISBN: 978-3-540-62790-6. doi: 10.1007/BFb0035388. Disponível em: <http://dx.doi.org/10.1007/BFb0035388>.

[19] FOURER, R., GAY, D., KERNIGHAN, B. "AMPL: A Mathematical Programing Language". In: Wallace, S. (Ed.), *Algorithms and Model Formulations in Mathematical Programming*, v. 51, *NATO ASI Series*, Springer Berlin Heidelberg, pp. 150–151, 1989. ISBN: 978-3-642-83726-5. doi: 10.1007/978-3-642-83724-1_12. Disponível em: <http://dx.doi.org/10.1007/978-3-642-83724-1_12>.

[20] GANSNER, E. R., NORTH, S. C. "An open graph visualization system and its applications to software engineering", *SOFTWARE - PRACTICE AND EXPERIENCE*, v. 30, n. 11, pp. 1203–1233, 2000.

[21] WILENSKY, U. "NetLogo". http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL, 1999.

[22] PAPERT, S. A. *Mindstorms: Children, Computers, And Powerful Ideas*. 1993. ISBN: 0465046746.

Apêndice A

Trabalhos Publicados

Os seguintes trabalhos foram publicados durante os estudos de mestrado correspondentes a esta dissertação:

- Daniel S. F. Alves, Douglas O. Cardoso, Hugo C. C. Carneiro, Felipe M. G. França, and Priscila M. G. Lima. An empirical study of the influence of data structures on the performance of vg-ram classifiers. In *1st BRICS Countries Congress (BRICS-CCI) and 11th Brazilian Congress (CBIC) on Computational Intelligence*, 1st BRICS Countries Congress (BRICS-CCI) on Computational Intelligence, September 2013.

- Daniel Alves, Felipe França, Luiza de Macedo Mourelle, Nadia Nedjah, and Priscila Lima. The effect of intelligent escape on distributed ser-based search. In Beniamino Murgante, Osvaldo Gervasi, Sanjay Misra, Nadia Nedjah, Ana Rocha, David Taniar, and Bernady Apduhan, editors, *Computational Science and Its Applications – ICCSA 2012*, volume 7333 of *Lecture Notes in Computer Science*, pages 101–112. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-31125-3_8.

- Daniel S. F. Alves, Danilo S. Carvalho, Diego F. P. Souza, Douglas O. Cardoso, and Hugo C. C. Carneiro. Credit analysis classification with wisard. In *1st BRICS Countries Congress (BRICS-CCI) and 11th Brazilian Congress (CBIC) on Computational Intelligence*, 1st BRICS Countries Congress (BRICS-CCI) on Computational Intelligence – CI Algorithm Competition – Third place, September 2013.

Apêndice B

Gramática do ReSATyrus

⟨*program*⟩ ::= ⟨*list_definition*⟩ ⟨*list_constraint*⟩

⟨*list_definition*⟩ ::= ⟨*empty*⟩
| ⟨*list_definition*⟩ ⟨*definition*⟩ ;

⟨*definition*⟩ ::= ⟨*ident*⟩ '`=`' ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*value*⟩ '`+`' ⟨*value*⟩
| ⟨*value*⟩ '`-`' ⟨*value*⟩
| ⟨*value*⟩ '`*`' ⟨*value*⟩
| ⟨*value*⟩ '`/`' ⟨*value*⟩
| ⟨*value*⟩ '`%`' ⟨*value*⟩
| ⟨*value*⟩

⟨*value*⟩ ::= '`(`' ⟨*expr*⟩ '`)`'
| ⟨*ident*⟩
| ⟨*integer*⟩

⟨*list_constraint*⟩ ::= ⟨*empty*⟩
| ⟨*list_constraint*⟩ ⟨*constraint*⟩ ;

⟨*constraint*⟩ ::= '`intgroup`' ⟨*ident*⟩ '`:`' ⟨*formula*⟩
| '`optgroup`' ⟨*ident*⟩ '`:`' ⟨*orientation*⟩

⟨*orientation*⟩ ::= ⟨*list_resources*⟩

⟨*list_resources*⟩ ::= ⟨*list_resources*⟩ '`->`' ⟨*resource*⟩
| ⟨*resource*⟩

⟨*resource*⟩ ::= ⟨*ident*⟩

⟨*formula*⟩ ::= ⟨*wwf*⟩

⟨*wwf*⟩ ::= ⟨*wwf*⟩ ‘`or`’ ⟨*wwf2*⟩
| ⟨*wwf*⟩ ‘`and`’ ⟨*wwf2*⟩
| ⟨*wwf2*⟩

⟨*wwf2*⟩ ::= ‘`not`’ ⟨*wwf3*⟩
| ‘`xor`’ ‘`(`’ ⟨*wwf_list*⟩ ‘`,`’ ⟨*wwf*⟩ ‘`)`’
| ⟨*wwf3*⟩

⟨*wwf_list*⟩ ::= ⟨*wwf_list*⟩ ‘`,`’ ⟨*wwf*⟩
| ⟨*wwf*⟩

⟨*wwf3*⟩ ::= ⟨*ident*⟩
| ‘`(`’ ⟨*wwf*⟩ ‘`)`’

⟨*ident*⟩ ::= ⟨*letter*⟩ (⟨*letter*⟩ | ⟨*digit*⟩)*

⟨*integer*⟩ ::= ⟨*digit*⟩+

⟨*letter*⟩ ::= [_a−zA−Z]

⟨*digit*⟩ ::= $[0-9]$

Apêndice C

Uso do ReSATyrus

Neste apêndice são detalhados os requisitos necessários para a instalação e uso de ReSATyrus, assim as bases para sua utilização. Até o momento, o ReSATyrus foi testado apenas em Linux (Ubuntu 12.04), mas, a princípio, é tão portável quanto seus requisitos.

C.1 Requisitos para uso

Por herdar o código base do SATyrus, o ReSATyrus também foi desenvolvido em Python, logo este deve estar instalado no sistema. O ReSATyrus foi testado com Python versão 2.7 e possivelmente funcione com versões mais antigas, mas não há garantias. Também não funciona com versões seguintes ao Python 3, devido às mudanças introduzidas nesta nova versão.

Para as capacidades envolvidas na interpretação do código é utilizado o módulo PLY (Python Lex-Yacc). Esta biblioteca de funções para Python busca implementar funcionalidades semelhantes aos programas clássicos Lex e Yacc usados para a análise lexical e sintática utilizada na criação de compiladores. O ReSATyrus foi testado com a versão 3.4 disponível nos repositórios do Ubuntu.

Os grafos são exportados na linguagem DOT utilizada pelo pacote de programas Graphviz. Para gerar as estruturas intermediárias e fazer a conversão final é utilizada a biblioteca pygraphviz. Este módulo precisa de algumas alterações para poder ser utilizado pelo ReSATyrus, mas o código modificado está disponível junto com o código do ReSATyrus.

C.2 Utilização e opções

O ReSATyrus foi desenvolvido para uso em linha de comando, de modo a ser mais fácil de integrar com outras ferramentas. Para uso, o ReSATyrus deve ser

chamado com o nome do arquivo contendo a descrição dos processos e opções para execução. As opções são descritas a seguir, a maioria consiste em especificar quais grafos devem ser gerados e para quais arquivos serão exportados.

C.2.1 Opções de execução

-h, -help Exibe uma mensagem de ajuda explicando as opções disponíveis e termina a execução.

-V, -version Exibe a versão do ReSATyrus e termina a execução.

-v, -verbose Faz o ReSATyrus exibir mais mensagens, para ajudar a examinar a execução.

-alg-colors-out ALG_COLORS_NAME Define criação do digrafo para *SER* orientado inicialmente pelo Alg-Colors, o qual será exportado para o arquivo `ALG_COLORS_NAME`.

-comp-out COMP_NAME Define a criação de um grafo dito grafo completo, que mostra as relações entre processos e recursos, o qual será exportado para o arquivo `COMP_NAME`

-alg-neighbors-smer-out ALG_NEIGHBORS_SMER_NAME Define criação do multidigrafo para *SMER* orientado inicialmente pelo multi-Alg utilizando como base o Alg-Neighbors, o qual será exportado para o arquivo `ALG_NEIGHBORS_SMER_NAME`.

-conc-out CONC_NAME Define a criação do grafo base do *SER*, o qual será exportado para o arquivo `CONC_NAME`.

-alg-colors-smer-out ALG_COLORS_SMER_NAME Define a criação do multigrafo para *SMER* orientado inicialmente pelo multi-Alg utilizando como base o Alg-Colors, o qual será exportado para o arquivo `ALG_COLORS_SMER_NAME`.

-alg-edges-out ALG_EDGES_NAME Define a criação do digrafo para *SER* orientado inicialmente pelo Alg-Edges, o qual será exportado para o arquivo `ALG_EDGES_NAME`.

-smer-out SMER_NAME Define a criação do grafo base do *SMER*, o qual será exportado para o arquivo `SMER_NAME`.

-multiser-out MULTISER_NAME Define a criação de um grafo base para o *SER* que utilize múltiplos nós para implementar a dinâmica *SMER*, o qual será exportado para o arquivo `MULTISER_NAME`.

-multiser-oriented-out MULTISER_NAME Define a criação de um grafo orientado para o *SER* que utilize múltiplos nós para implementar a dinâmica *SMER*, o qual será exportado para o arquivo `MULTISER_ORIENTED_NAME`.

-alg-neighbors-out ALG_NEIGHBORS_NAME Define a criação do digrafo para *SER* orientado inicialmente pelo Alg-Neighbors, o qual será exportado para o arquivo `ALG_NEIGHBORS_NAME`.

-hint-out HINT_NAME Define a criação de um grafo contendo apenas as arestas orientadas que podem ser inferidas a partir das definições declaradas pelo `optgroup`, o qual será exportado para o arquivo `HINT_NAME`.

-res-out RES_NAME Define a criação do dito grafo de recursos, que relaciona recursos e processos que os utilizam e que será exportado para o arquivo `RES_NAME`.

-alg-edges-smer-out ALG_EDGES_SMER_NAME Define a criação do multidigrafo para *SMER* orientado inicialmente pelo multi-Alg utilizando como base o Alg-Edges, o qual será exportado para o arquivo `ALG_EDGES_SMER_NAME`.

-default Ativa a geração de todos os grafos, com nomes definidos por um padrão. O nome padrão é criado a partir das opções com a extensão `.dot`. Assim, o nome padrão para o grafo base do *SER* seria `ser-out.dot`.