Construção Automática de Teoria em Grafos

Hemerson Pistori

Dissertação de Mestrado

Construção Automática de Teoria em Grafos

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Hemerson Pistori e aprovada pela Banca Examinadora.

Campinas, 11 de maio de 1998.

Jacques Wainer (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Instituto de Computação
Universidade Estadual de Campinas

Construção Automática de Teoria em Grafos

Hemerson Pistori

Maio de 1998

Banca Examinadora:

- Jacques Wainer (Orientador)
- Guilherme Bittencourt
  Departamento de Automação e Sistemas
  Universidade Federal de Santa Catarina
  Florianópolis SC - Brasil
- Heloisa Vieira de Souza
  Instituto de Computação – Unicamp
- Ariadne Maria Brito Rizzoni Carvalho (Suplente)
  Instituto de Computação – Unicamp

Prefácio

Este trabalho apresenta SCOT, um sistema de construção automática de teoria inspirado no programa AM de Douglas Lenat. O AM é conhecido por ter "redescoberto" uma série de conceitos e conjecturas famosos em teoria dos números, aritmética e geometria [Len82]. Apesar do grande interesse despertado por este programa, este linha de pesquisa continua sendo muito pouco explorada. Um dos grandes problemas com o AM é a complexidade do seu conjunto de heurísticas, que é representado através de um sistema de produção contendo 243 regras. Com o SCOT, nos buscamos uma melhor estruturação e organização na representação das heurísticas, facilitando assim a análise e a manipulação das mesmas. A construção automática de teoria é também conhecida como *aprendizagem por descoberta* ou *aprendizagem por exploração*.

Abstract

In this work we present SCOT, an automatic theory construction system inspired on Lenat's program AM. AM "rediscovered" some well-known concepts and conjectures from number theory, arithmetic and geometry [Len82]. Despite the great interest surrounding that program, further contributions to this research line are scarce. One of the main problem with AM is the great complexity of the heuristic set, which is represented as a production system with 243 rules. With SCOT we propose a revival of the "AM's research", emphasizing the clarity and "manipulability" of the heuristic set. Automatic theory construction is also known as *learning by discovery* or *learning by exploration*.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Aprendizagem de máquina é a área da Inteligência Artificial que estuda teorias e mecanismos para a automatização do processo de refinamento e aquisição de novos conhecimentos e habilidades, por parte de um sistema de computação. O início desta linha de pesquisa se confunde com o próprio início da Inteligência Artificial, uma vez que a aprendizagem automática sempre foi um dos principais requisitos de um sistema inteligente. Com o sucesso comercial dos chamados sistemas especialistas, nos anos 80 e 90, esta linha de pesquisa recebeu um novo impulso. Os sistemas especialistas dependem normalmente de uma base de conhecimentos muito densa e complexa. Extrair este conhecimento de um especialista humano, para implanta-lo num computador, é sem dúvida um problema extremamente complicado. A solução deste problema depende muito das tecnologias que começam a ser geradas pelas pesquisas em aprendizagem de máquina.

Existem atualmente inúmeras pesquisas em andamento que abordam o problema da aquisição de conhecimento das mais variadas formas, nos mais variados domínios. Estas linhas de pesquisa diferem principalmente pela maneira de representar o conhecimento adquirido, pela representação do conjunto de treinamento, pelo domínio de atuação e pela estratégia de inferência. Vários autores ([Fir88],[Mic86]) preferem a classificação por estratégia de inferência, que se baseia em comparações com processos humanos de aprendizagem, como *memorização*, *operacionalização* (learning by taking advice), *dedução*, *analogia* e *indução*.

A memorização é a forma mais básica de aprendizagem, envolvendo apenas o armazenamento de novas informações. Vários programas utilizam a memorização de resultados de cálculos complexos para evitar reprocessamentos dispendiosos. O próprio *jogador de damas* de Samuel [Sam63] se utilizava de uma forma de memorização para otimizar o cálculo de sua função de avaliação heurística quando algumas situações de tabuleiro se repetiam em diferentes jogos. É claro que o programa de Samuel utilizava outros formas mais inteligentes para melhorar seu desempenho durante e depois de vários jogos. Já

na operacionalização o programa recebe informações (também chamadas de conselhos) que devem ser "traduzidas" antes de serem utilizadas em processamentos posteriores. O módulo de tradução funciona como uma espécie de compilador ou interpretador, e deve ser fornecido diretamente por um programador. Tanto a operacionalização, quanto a memorização, embora satisfaçam grande parte das definições de aprendizagem, não são totalmente aceitas, principalmente por pessoas externas ao campo da IA, como uma forma legítima de aprendizagem.

Na aprendizagem por dedução o sistema deve possuir um mecanismo de inferência dedutiva que permita que um conjunto de informações possa ser compactado em uma regra ou um grupo de regras gerais. Com isto, podemos ter no sistema algumas informações não diretamente disponíveis, mas que podem ser obtidas através de um processo de dedução. Como o processo de dedução pode ser muito custoso, torna-se importante a existência de algum mecanismo que tente minimizar o tempo de acesso às informações acessadas com maior frequência (como a memorização, por exemplo).

A aprendizagem por analogia é talvez o mais complexo dos processos de aprendizagem utilizados pelo ser humano. Ela envolve basicamente o mapeamento de soluções de um problema para outro. Acontece que este mapeamento envolve normalmente informações muito sutis sobre diversos domínios diferentes, o que dificulta sensivelmente o trabalho de compreensão dos processos envolvidos na criação de uma analogia.

A aprendizagem por indução é a que tem recebido maior atenção por parte dos pesquisadores. Na sua forma mais conhecida temos um sistema que recebe uma lista de exemplos positivos e negativos (conjunto de treinamento) de um determinado conceito e devolve uma "definição" para o conceito. Este definição pode então ser utilizada, pelo programa, na classificação de novos exemplos. Os sistemas atuais diferem-se principalmente pelos seguintes fatores:

- Linguagem de representação, tanto do conjunto de treinamento, quanto da definição obtida.
- Não disponibilidade de exemplos negativos.
- Maneira com que os exemplos são inseridos no sistema: todos de uma só vez ou incrementalmente.
- Possibilidade de se induzir definições para mais de um conceito ao mesmo tempo.
- Existência de erros no conjunto de treinamento.
- Possibilidade de se utilizar algum conhecimento prévio sobre o domínio de onde foram tirados os exemplos.

Como a grande maioria das pesquisas atuais giram em torno da aprendizagem por indução, a classificação por estratégia de inferência acaba não sendo muito útil. Uma outra classificação, mais atual, leva em consideração o tipo do conjunto de treinamento e a "quantidade" de conhecimento precodificado no sistema. Nesta classificação, as linhas atuais de pesquisa podem ser divididas em *pouco conhecimento e conjunto de treinamento preclassificado*, *muito conhecimento e conjunto de treinamento preclassificado*, *muito conhecimento e conjunto de treinamento não classificado* e *muito conhecimento e conjunto de treinamento indefinido*. O conjunto de treinamento indefinido ocorre quando o próprio sistema é capaz de modificar seu conjunto de treinamento, como por exemplo, no programa AM de Lenat[Len82]. Já no caso do conjunto de treinamento não classificado, temos um conjunto de exemplos definidos, mas não temos uma classificação prévia imposta sobre estes exemplos.

O problema da aprendizagem com pouco conhecimento e conjunto de treinamento preclassificado foi o primeiro a ser extensivamente estudado. Um dos mais bem sucedidos sistemas deste gênero é o ID3 [Qui82], que infere árvores de decisão a partir de vetores de atributos representando o conjunto de exemplos. Estas árvores de decisão são posteriormente convertidas em regras de produção, podendo então servir de conhecimento para sistema especialistas, como no caso dos sistemas comerciais Super-Expert e RuleMaster [DMZ84]. Uma evolução importante do ID3 é o C4.5 [Qui93], programa criado 1994 que vem se tornando um benchmark para os sistemas de indução de árvores de decisão. Como as árvores de decisão precisam normalmente ser convertidas em regras de produção, sistemas como CN2 [CN89], GREEDY3 [PH90] e o BEXA [TC96], buscam otimizar este procedimento através da inferência direta das regras.

Outra estratégia bastante popular de aprendizagem com pouco conhecimento e conjunto de treinamento preclassificado é conhecida como *lazy learning*. Neste caso, o sistema posterga a inferência de uma descrição completa do conceito, armazenando alguns dos exemplos do conjunto de treinamento para posteriores comparações. IBn [DWA91], MBRTalk [Sta87] e NGE [Sal91] são exemplos de sistemas que se utilizam deste estratégia, também conhecida por *case-based learning*, *memory based*, *instance based* e *edited k-nearest neighbor*.

Na aprendizagem com muito conhecimento e conjunto de treinamento preclassificado temos, normalmente, um conjunto de treinamento com poucos exemplos e precisamos compensar este fato com um maior conhecimento inicial sobre o domínio ou problema. Em 1986, Mitchel [TMKC86] e DeJong e Mooney [DM86] propuseram a EBL (explanation-Based learning), um mecanismo de aprendizagem capaz de induzir conceitos a partir de um único exemplo positivo. Um sistema EBL depende, no entanto, de um conhecimento prévio bem estruturado sobre o domínio em questão, o que costuma dificultar bastante a sua utilização prática. Estender a EBL para trabalhar com domínios mal-estruturados,

como acontece nos humanos, tem sido uma área ativa de pesquisas (ver Tadepalli [Tad89]).

Uma outra linha de aprendizagem que trabalha com muito conhecimento e conjunto de treinamento preclassificado e que conta com uma forte fundamentação teórica é a *programação lógica indutiva*. A PLI busca fundamentações teóricas firmes para o problema da aprendizagem, utilizando as representações usadas em programação em lógica (Logic Programming). Entre os algoritmos mais conhecidos de PLI estão o FOIL [Qui90], o FOCL [PB93], o GOLEM [MF92] e o CIGOL [MB88].

No caso da aprendizagem com muito conhecimento e conjunto de treinamento não classificado temos um conjunto de treinamento de exemplos não classificados e queremos encontrar maneiras "interessantes" ou úteis de agrupar (cluster) estes exemplos. As principais abordagens são a *conceptual clustering*, com o sistema CLUSTER [SM86] e a *bayesian clustering*, com o sistema AUTOCLASS [CKS+88]. Note que neste tipo de sistema pode acontecer do conhecimento inicial sobre o domínio ser pequeno, no entanto, o sistema precisa compensar este fato com algum tipo de conhecimento geral (estética, funcionalidade, ...) que permita a escolha de alguns agrupamentos ao invés de outros. É por isto que classificamos estas linhas de pesquisa como aprendizagem com *muito conhecimento*.

A aprendizagem com muito conhecimento e conjunto de treinamento indefinido é também conhecida como *aprendizagem por descoberta*, *aprendizagem por exploração* ou *construção automática de teoria*. Neste tipo de aprendizagem não existe um conjunto de treinamento fixo e o sistema pode, e deve, gerar novos conceitos e exemplos destes novos conceitos. Cada novo conceito, junto com seus exemplos, passa a fazer parte do domínio de exploração do sistema, que pode crescer indefinidamente. Um conjunto de heurísticas tentam fornecer ao sistema uma forma de "distinguir" conceitos intrinsecamente interessantes, com base em algum tipo de regularidade ou peculiaridade. O mais conhecido exemplo de um sistema deste tipo é o AM (Automated Mathematician), que ficou famoso por suas descobertas na teoria dos números e aritmética.

O AM foi um dos primeiros experimentos em construção automática de teoria já realizados. Ele foi desenvolvido por volta de 1980 por Douglas Lenat que relata que seu programa foi capaz de "redescobrir" conceitos importantes como os números primos, as leis de Morgan, o teorema fundamental da aritmética e a conjectura de Goldback [Len82]. Na base de conhecimentos inicial do AM haviam apenas conceitos elementares da teoria dos conjuntos e um grupo de 243 heurísticas representadas como regras de produção.

O significado de cada conceito era representado através de um código em LISP capaz de reconhecer exemplos de um conceito ou de gerar exemplos do conceito. A partir da análise destes exemplos o AM poderia fazer pequenas alterações (mutações sintáticas) neste código, criando assim novos conceitos. Por exemplo, se o código que dava significado a um determinado conceito contivesse o teste "Se A e B então C" e pela análise de exemplos

o AM decidisse criar um conceito mais genérico que este, ele simplesmente retiraria uma das condições (A por exemplo) criando assim um novo conceito onde o teste passaria a ser apenas "Se B então C".

Susan Epstein, propõe uma novo esquema para a representação de conhecimento em programas de construção automática de teoria [Eps83, Eps88]. Para chegar ao seu esquema Epstein buscou inspiração numa série de experimentos mentais (thought experiments) com seres humanos. Neste experimentos um grupo de pessoas era requisitado a explorar um determinado subdomínio da teoria dos grafos na busca de novas descobertas. A linha de raciocínio seguida por cada elemento era anotada passo a passo. Para demonstrar a aplicabilidade deste esquema ela implementou um programa chamado GT (Graph Theorist) que, a partir de um conjunto inicial de conceitos, gera exemplos, propõe novos conceitos, sugere novas conjecturas e prova teoremas na área da teoria dos grafos.

A exploração no GT se restringe a classes de grafos e a relações entre estas classes. Classes de grafos são representadas por expressões algébricas que fornecem um algoritmo para a geração "ordenada" de exemplos de grafos. Estas expressões algébricas são triplas ordenadas $< f, S, \sigma >$, onde S é um conjunto de grafos "minimos" que obedecem a propriedade que define a classe (seed set), f é a descrição de uma operação que dado um grafo pertencente a classe gera um novo grafo que também pertence a classe, e σ são restrições a serem respeitadas pela operação. Por exemplo, a classe GRAFOS ACÍCLICOS é descrita pela tripla $< A_x + A_{yz}A_z, \{K_1\}, \{y \in V, x, z \notin V\} >$. Nesta tripla, a operação deve ser lida como "ou adicione um vértice x ou adicione um vértice z e uma aresta ligando y e z". A restrição $\{y \in V, x, z \notin V\}$ impede que sejam introduzidos ciclos no grafo, garantindo assim que o novo grafo seja acíclico. Exemplos de grafos acíclicos são gerados a partir da aplicação recorrente da operação sobre K_1 e sobre os grafos subsequentes. Novas classes (conceitos) são gerados a partir de alterações no seed set, na operação ou nas restrições. Estas alterações são escolhidas de forma a garantir que a nova classe seja uma especialização ou uma generalização da classe anterior. Novas classes também podem ser geradas a partir da "junção" (merge) das definições de duas outras classes.

Neste documento apresentaremos o SCOT, um sistema do construção automática de teoria cujo domínio também é a teoria dos grafos. Criamos um mecanismo para representação do domínio que facilita a caracterização do processo de descoberta como uma busca num espaço de conceitos. Com isto, um grupo importante de heurísticas que lidam com características estruturais dos conceitos puderam ser representadas como macro-operadores. Detectamos outros 6 grupos diferentes de heurísticas. Criamos mecanismos de representação específicos para cada grupo de heurística, facilitando assim todo o processo de análise e construção de novas heurísticas. Ao contrário de Epstein, não abordamos diretamente a questão do processo de descoberta em seres humanos nem a prova de teoremas. No entanto, a exploração no SCOT não se limita apenas a classes de grafo. Nosso

esquema de representação trata de maneira uniforme todos os conceitos relacionados a teoria dos grafos. Isto permite a ele a construção de conceitos como ARESTA DE CORTE, VÉRTICE SIMPLICIAL e CAMINHO MÍNIMO, que estão totalmente fora do alcançe do GT.

O SCOT se constitui de 5 módulos que compartilham informações através de uma lista de conceitos, uma lista de conjecturas e um mecanismo simplificado de agenda. Os módulos tratam, respectivamente, da geração de conceitos, da análise de exemplos de conceitos, da geração de conjecturas, da análise de conjecturas e do refinamento de conjecturas. Alguns grupos de heurísticas estão ligados diretamente à um determinado módulo. Outros grupos, no entanto, são utilizados por mais de um módulo.

O restante deste documento está organizado da seguinte forma. No capítulo 2 fornecemos uma breve introdução ao programa AM. No capítulo 3 descrevemos detalhadamente o funcionamento do SCOT. O próximo capítulo ilustra o funcionamento do SCOT com algumas de suas descobertas. Os seis apêndices contêm respectivamente: descrição das classes que definem os principais objetos do SCOT, a lista de conceitos da base inicial, a lista de operadores, a lista de macro-operadores, resumo das classes de heurísticas usados pelo SCOT e o algoritmo usada na geração de grafos aleatórios.

Capítulo 2

AM - Automated Mathmatician

George Polya , ainda nos anos 50, constatou a existência de alguns padrões de raciocínio que pareciam se repetir na história de diversas descobertas analisadas por ele, principalmente em Matemática [Pol45]. A partir desta análise e da sua experiência como professor, Polya conseguiu isolar algumas heurísticas que foram chamadas de *heurísticas de descoberta*. Estas heurísticas chegaram a ser publicadas num pequeno livreto que se propunha a ajudar as pessoas a resolverem qualquer tipo de problema matemático. No final dos anos 70, Douglas Lenat implementou um programa chamado AM que se utilizava de *heurísticas de descoberta* para fazer explorações num determinando domínio matemático [Len77]. Durante o processo de exploração o programa poderia sugerir a criação de novos conceitos e de conjecturas envolvendo estes conceitos. Num trabalho posterior, com um programa chamado EURISKO, Lenat passou a incluir as próprias heurísticas no domínio de exploração, para que o programa pudesse também criar novas heurísticas quando as iniciais começassem a perder sua "aplicabilidade". O sucesso do AM e do EURISKO serviu para demonstrar a viabilidade da utilização de heurísticas em programas de aprendizagem de máquina.

O conhecimento inicial do AM é representando através de 115 frames [Min81], representando conceitos elementares da teoria dos conjuntos. Cada frame possui em média 20 slots, sendo que nem todos começam preenchidos. A principal tarefa do AM é preencher slots vazios e criar novos frames.

Os principais slots de um frame são:

Nome Um identificador do conceito que a princípio é um simples símbolo gerado sequencialmente pelo sistema mas que pode ser alterado pelo usuário para conter alguma palavra mais significativa.

Generalizações Lista de conceitos que representam generalizações deste conceito.

Especializações Lista de conceitos que representam especializações deste conceito.

Código Recursivo Implementa a função característica do conceito. Serve para gerar exemplos do conceito e para ser manipulado sintaticamente na geração de novos conceitos.

Código Rápido Equivalente otimizado do código recursivo.

Valor Valor numérico manipulado por diversas heurísticas que tentam captar o grau de "interessabilidade" deste conceito em relação aos outros.

Exemplos Exemplos do conceito.

Conjecturas Conjecturas envolvendo o conceito.

A principal estrutura de controle do AM é uma agenda de tarefas ordenada por prioridade. A prioridade de uma tarefa é determinada por uma lista de justificativas simbólicas associadas a cada tarefa. Estas justificativas são fornecidas pela própria heurística que insere a tarefa na agenda. Uma mesma tarefa pode ser sugerida por várias heurísticas diferentes. Neste caso, pode-se ter na agenda uma única tarefa com várias justificativas. O formato de uma tarefa é basicamente o seguinte:

> Execute a operação X no slot Y do conceito Z utilizando no máximo T segundos e M blocos de memória.

O scheduler do AM sempre pega a tarefa com maior prioridade e para executa-la procura por heurísticas (regras de produção) compatíveis com a operação, o slot, e o conceito indicado na tarefa. Todas as regras compatíveis são executadas sequencialmente até que o limite de tempo e espaço associado a cada tarefa seja atingido. Toda regra está associada diretamente a um conceito, um slot e uma operação. Em alguns casos, uma regra pode ainda ser "herdada" pelos conceitos que são especializações do conceito ao qual a regra está diretamente ligada. O exemplo abaixo mostra como as heurísticas do AM colaboram entre sí na descoberta de novos conceitos:

Uma das heurísticas do AM pedia para que fossem encontrados exemplos de um conceito. Ao tentar encontrar exemplos do conceito IGUALDADE ENTRE CONJUNTOS, através da geração aleatória de pares de listas, o AM encontra muito poucos exemplos de listas iguais. Uma outra heurística sugere então que se encontre generalizações deste conceito. Através de mutações sintáticas no código que caracteriza este conceito o AM cria 3 novos conceitos. Um destes novos conceitos equivale ao conceito IGUALDADE ENTRE A CARDINALIDADE DE DOIS CONJUNTOS. Deste conceito o AM deriva primeiramente o conceito TAMANHO. Depois ele usa este conceito para criar um novo conceito que consiste de um conjunto ordenado de listas com "todos" os tamanhos possíveis: () , (T) , (TT) , (TTT)

Tabela 2.1: Frame Representando Números Primos no AM

Name	Primes
Statement.Lisp	Lambda (n) (Apply* (Lisp-Statement Doubleton) (Apply* (Compiled-Coded-if-Then Divisors-of) n))
Specializations	Odd-primes, Small-primes, Pair-primes
Generalizations	Positive Numbers
Is-a	Class-of-number
Extreme-Exs	2,3
Extreme-non-exs	0,1
Typical-exs	5,7,11,13,17,19
Typical-non-exs	34,100
Conjectures Good-conj-units	Unique-factorization, Formula-for-d(n) Times, Divisors-of, Exponentiate, Numbers-with-3-divis, Squaring
Worth	800
Origin	Application of H2 to divisors-of Defined-Using: Divisors-of
Creation-Date	3/19/76 18:45
History	NGoodExamples: 840 NBadExamples: 5000 NGoodConjectures: 3 NBadConjectures: 7

, (TTTT) , etc. Este novo conceito equivale ao conjunto dos NÚMEROS INTEIROS. Alterando os conceitos que trabalhavam com o conceito primitivo LISTA, para trabalharem com este novo conceito (números inteiros), o AM consegue chegar a vários conceitos aritméticos interessantes. Por exemplo, fazendo com que a função APPEND, que consta da base inicial do AM, passe a trabalhar apenas com exemplos de NÚMEROS INTEIROS (ao invés de qualquer tipo de lista), obtêm-se o equivalente ao conceito ADIÇÃO. É a partir daqui que começam as explorações do AM em teoria elementar dos números [Len84]. A tabela 2 mostra o frame que representa o conceito de números primos no AM.

Apresentamos abaixo algumas das heurísticas do AM.

- Se alguns (mas não todos) exemplos de algum conceito X são também exemplos de outro conceito Y, crie um novo conceito representando a intersecção de X e Y.
- Se f é uma função de A em B, então considere os elementos de A que são mapeados em elementos extremos de B. Crie um novo conceito representando este subconjunto de A.
- Se existem muito poucos exemplos do conceito X, então adicione na agenda a tarefa de encontrar generalizações de X.

- Se existe uma função $f : AxA \rightarrow B$ então defina a função $g : A \rightarrow B$ onde $g(x) = f(x, x)$.

Depois de algumas horas de execução a taxa de conceitos interessantes descobertos pelo AM caia abruptamente. Lenat conclui que as heurísticas tinham um limite de atuação que as tornava inútil quando os conceitos descobertos começavam a sair de um certo domínio. Lenat passa então a considerar a questão de como as heurísticas, em si, são descobertas. Ele propõe um novo programa, o Eurisko, onde as próprias heurísticas faziam parte do domínio de exploração do sistema. Este novo domínio, no entanto, se mostrou muito mais difícil que o anterior. O desempenho do Eurisko, embora surpreendente em algumas situações específicas, era bem inferior ao do AM, de modo geral.

2.1 Problemas com o AM

Num artigo sobre os problemas com metodologia nas pesquisas em IA na década de 80 [RH84], Ritchie e Hanna usaram o trabalho de Lenat como exemplo. Eles apontam sérias discrepâncias entre o texto escrito e o programa, e sugerem que estes problemas obscureciam a real contribuição dos trabalhos de Lenat. No mesmo jornal onde este artigo foi publicado, aparece uma resposta de Lenat a estas críticas[Len84]. Neste artigo, Lenat admite algumas falhas na exposição do seu trabalho e tenta responder a cada uma das questões levantadas por Ritchie e Hanna.

O que mais nos chamou a atenção no artigo de Lenat foi a constatação de que a alta taxa de acerto do AM se deve principalmente ao grande relacionamento entre LISP e matemática. O problema é que um dos grandes propósitos de Lenat, com o AM, era demonstrar que o processo de descoberta pode ser guiado por um conjunto bem definido[1] de heurísticas. Ora, se o LISP é fator essencial no sucesso do AM, nada podemos concluir sobre a afirmação inicial de Lenat de que o processo de descoberta pode ser guiado por um conjunto de heurísticas. A verdade é que o artigo de Lenat, embora tenha respondido algumas das questões, serviu também para acrescentar outras questões mais fundamentais sobre o significado do AM.

Embora o AM continue sendo citado em diversas publicações atuais, as questões fundamentais levantadas a 20 anos atrás, não receberam grandes contribuições. Acreditamos que os fatores que mais desestimularam a continuidade do trabalho de Lenat foram a questão da "dependência" com o LISP e a complexidade do conjunto de heurísticas do AM.

Para resolver a questão da "dependência", criamos um mecanismo que favorece a distinção entre o alvo da exploração (conhecimento do domínio), as heurísticas de descoberta

[1] A ponto de ser implementável num sistema de computação

e as estruturas de controle. Optamos também por não utilizar os mecanismos de mutação sintática de código, que apesar de poderosos, dificultam a interpretação dos conceitos gerados por este metódo.

O problema da complexidade do conjunto de heurísticas se deve, principalmente, a uma falta de estrutura para sua representação. As regras de produção usadas pelo AM não possuiam uma estrutura bem definida para as ações e condições, que poderiam conter os mais variados códigos em LISP. Com isto, Lenat conseguia agrupar, num mesmo sistema de produção, heurísticas que não tinham praticamente nada em comum. Dividindo o conjunto de heurísticas do SCOT, em 7 grupos, nós conseguimos determinar mecanismos de representação mais estruturados para cada grupo. Estes grupos não são arbitrários, eles correspondem a uma divisão do processo de descoberta em módulos funcionais. Esta divisão em módulos é outra vantagem do SCOT em relação ao AM, que trabalhava apenas com uma única estrutura de controle explícita, a agenda.

Capítulo 3

SCOT - Sistema de Construção de Teoria

Lenat abandonou o AM em nome do EURISKO por achar que o limite do AM estava no fato dele não ser capaz de descobrir novas heurísticas. Este limite, no entanto, não impediu a sua excepcional performance nos domínios em que ele foi executado. E se pudessemos repetir esta performance em domínios mais densos e menos explorados que a teoria dos números e a geometria. Será que isto já não seria bastante interessante ? Acreditamos que a resposta para esta pergunta seja positiva e que muita coisa ainda precisa ser feita na linha do AM, antes de partimos para as complicações adicionais da descoberta automática de heurísticas.

Ao contrário de Lenat, no entanto, optamos por uma forma menos "ad-hoc" de trabalho. Nosso principal objetivo não é provar que um programa de computador possa fazer descobertas interessantes em um determinado domínio. O que buscamos é uma maior compreensão dos processos essênciais envolvidos em programas de construção automática de teoria. Buscamos também mecanismos mais estruturados para a representação das heurísticas de descoberta e do domínio a ser explorado por estas heurísticas. A teoria dos grafos foi escolhida tanto por ser uma área com grande potencial para novas descobertas quanto por garantir a distinção entre o domínio de exploração e a linguagem de implementação. Além disto, o fato de trabalharmos num dominio totalmente diferente do AM, pode ajudar na separação entre heurísticas gerais de descoberta, heurísticas de descoberta específicas para o domíno e o domínio de exploração em sí.

O SCOT é composto de 5 módulos: (I) *geração de conceitos*, (II) *análise de exemplos*, (III) *geração de conjecturas*, (IV) *análise de conjecturas* e (V) *refinamento de conjecturas*. Estes módulos são executados ciclicamente até que um limite máximo de tempo, fornecido pelo usuário, seja atingido. O domínio de exploração é representado através de *conceitos*, *exemplos*, *conjecturas* e *operadores*. As heurísticas são separadas em 7 grupos distintos,

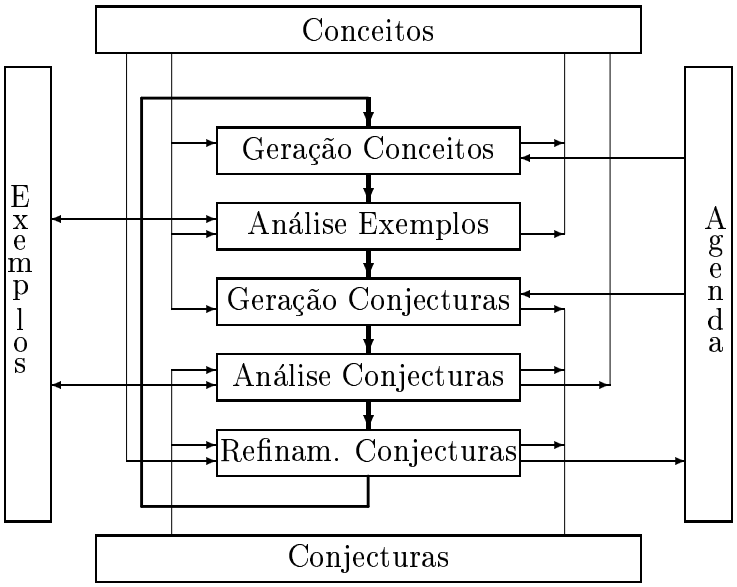

Figura 3.1: Os 5 módulos do SCOT

cada qual com um mecanismo próprio de representação e aplicação: *macro-operadores*, *heurísticas de restrição de aplicabilidade de operadores*, *heurística de cálculo do valor do conceito*, *heurísticas de análise de exemplos*, *heurística de indução de conjectura*, *heurística para ordenação de conjecturas* e *heurísticas estritamente numéricas*. Uma agenda é usada para facilitar a comunicação entre alguns módulos. Esta agenda, no entanto, possui apenas um papel secundário no funcionamento global do SCOT.

A figura 3.1 mostra um diagrama de fluxo de dados simplificado representando o funcionamento do SCOT (sem as heurísticas). No módulo de geração de conceitos temos a criação de novos conceitos a partir da manipulação de conceitos anteriores. O módulo de análise de exemplos tenta avaliar o "quão interessante" é um conceito, com base em exemplos do mesmo. Conjecturas são gerados no módulo de geração de conjectura enquanto o módulo de análise de conjectura tenta determinar "empiricamente" a relação entre os conceitos envolvidos numa conjectura. O último módulo, o de refinamento de conjecturas, executa alterações nas conjecturas na tentativa de obter outras mais interessantes.

Os módulos de análise (de exemplos e de conjectura) terminam apenas quando todos os conceitos e conjecturas gerados no ciclo são analisados. Os outros módulos respeitam um limite máximo de tempo de execução e um limite no total de conceitos e conjecturas que podem ser gerados a cada ciclo. Os módulos de análise, que dependem da geração maciça de exemplos de conceitos, são executados distribuidamente (ver seção 3.7.1).

3.1 Estruturas de Dados do SCOT

O SCOT foi implementado em um Lisp com extensões para programação orientada à objetos. As principais estruturas de dados do SCOT são representadas como instâncias de uma determinada classe. A representação do domínio de exploração se utiliza de 4 classes: conceito, exemplo, operador e conjectura. Aproveitaremos o alto nível de abstração proporcionado pela OOP para explicar a estrutura destas classes, nas próximas seções.

3.1.1 Conceitos

As principais propriedades de um conceito são a *origem*, a *entrada* e o *tipo*. O principal método é o GeraExemplo, que gera um exemplo aleatório do conceito, sempre que requisitado. O funcionamento do método GeraExemplo depende principalmente da *origem* do conceito. A *origem* do conceito indica se ele foi descoberto pelo sistema ou se ele é um conceito *primitivo*. O método GeraExemplo para conceitos primitivos executa um código predefinido. No caso de conceitos descobertos, a origem é o operador e os operandos (outros conceitos) usados na geração do conceito. Todo conceito descoberto é criado pela aplicação de operadores à conceitos anteriores (operandos). O método GeraExemplo para conceitos descobertos, combina métodos dos operadores e os métodos GeraExemplo de conceitos primitivos, para construir exemplos.

A *entrada* de um conceito representa uma dependência dos exemplos de alguns conceitos com exemplos de outros conceitos. Quando requisitamos um exemplo de um conceito X cuja *entrada* é Y precisamos fornecer como parâmetro um exemplo de Y. No entanto, se o exemplo de Y não for fornecido, o sistema automaticamente requisitará um exemplo de Y. Por exemplo, o conceito primitivo UM GRAFO não depende de qualquer outro conceito. O conceito UM VÉRTICE DE UM GRAFO depende do conceito UM GRAFO, ou seja, para obtermos um exemplo de UM VÉRTICE DE UM GRAFO precisamos fornecer um exemplo de UM GRAFO. O conceito O GRAU DE UM VÉRTICE, por sua vez, depende do conceito UM VÉRTICE DE UM GRAFO. No entanto, se um exemplo de O GRAU DE UM VÉRTICE for requisitado sem a entrada (um exemplo de UM VÉRTICE DE UM GRAFO), o SCOT automaticamente requisita um exemplo de um UM VÉRTICE DE UM GRAFO. Como UM VÉRTICE DE UM GRAFO também é um conceito dependente, um exemplo de UM GRAFO é automaticamente requisitado. A lista de entradas de qualquer conceito deve terminar sempre em um conceito com entrada nula (sem entrada), para que o SCOT possa sempre ser capaz de gerar exemplos de qualquer conceito sem interferência externa.

O *tipo* determina como os exemplos de um conceito serão representados internamente. O *tipo* de um conceito é restringido por uma gramática e está intimamente ligado ao domínio de exploração do sistema. Esta gramática será mostrada na próxima seção,

quando falaremos mais sobre os exemplos.

Finalmente, conceitos podem ser classificados como *determinísticos* e *não determinísticos*. Um conceito é dito *determinístico* quando sua entrada é não nula, e para um mesmo exemplo de entrada o código GeraExemplo do conceito fornece sempre o mesmo exemplo. Conceitos *não-determinísticos* podem fornecer exemplos diferentes partindo de um mesmo exemplo de entrada. O conceito VÉRTICE (de um GRAFO), por exemplo, é um conceito não-determinístico, pois, para um mesmo exemplo de grafo o código GeraExemplo pode devolver diferentes exemplos de vértice (Desde que o grafo tenha mais de um vértice). Já o conceito GRAU DE UM VÉRTICE é determinístico, pois, para um mesmo exemplo de vértice o código GeraExemplo devolve sempre um mesmo exemplo. Vizinhança é outro conceito determinístico.

3.1.2 Exemplos

Um exemplo é composto de apenas 3 propriedades: *conceito*, *entrada* e *conteúdo*. A propriedade *conceito* simplesmente aponta para o conceito do qual ele é um exemplo. A *entrada* de um exemplo está relacionada com a *entrada* do conceito. Ela aponta justamente para um exemplo do conceito apontado pela *entrada* do conceito. A propriedade mais significativa de um exemplo é o *conteúdo*, que pode conter deste um simples valor inteiro à uma lista representando um grafo, dependendo do *tipo* do conceito. No caso específico do SCOT, o *tipo* está restrito à gramática mostrada na tabela 3.1. Vale ressaltar que o SCOT trabalha atualmente apenas com grafos simples não-direcionados.

Tabela 3.1: Gramática de Tipos no SCOT

S	→	{S}	; Um conjunto de elementos do tipo S.
		$\{S\}^n$	; Um conjunto com cardinalidade definida,
			; ou seja, o conteúdo de um exemplo deste
			; conceito terá sempre a mesma cardinalidade n.
		[S]	; Uma sequência de elementos do tipo S.
		B	; Um booleano.
		I	; Um número inteiro.
		V	; Um inteiro indicando um dos vértices de um grafo.
		G	; Uma lista onde o primeiro elemento é o conjunto
			; de vértices e o segundo elemento é um conjunto de
			; pares de vértices indicando as adjacências.
			; Por exemplo, ((123)((12)(13)(23))) representa um K_3.
n	→		; Um número inteiro.

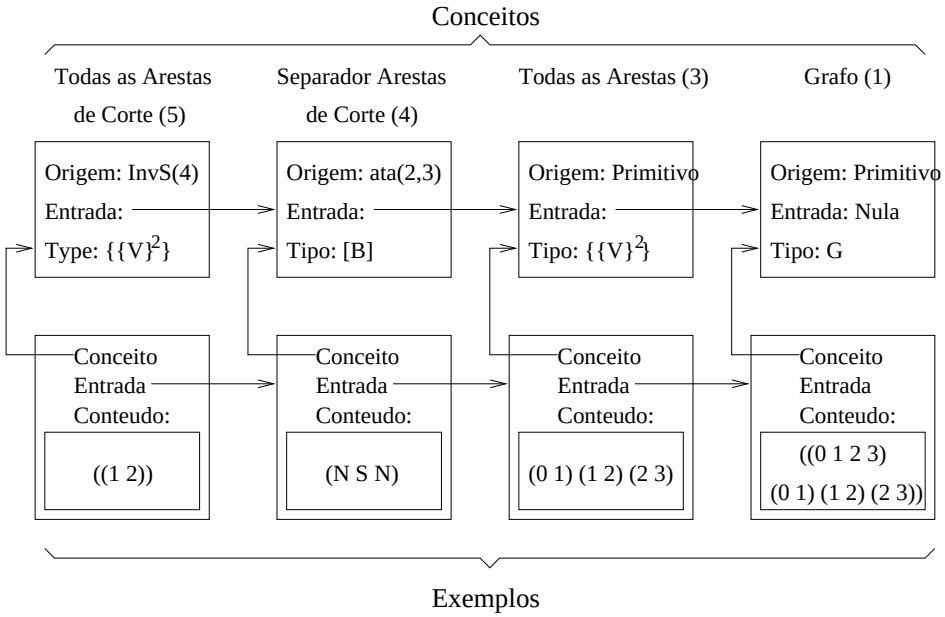

Figura 3.2: Exemplos e Conceitos

A figura 3.2 mostra exemplos de 4 conceitos relacionados entre si (da direita para esquerda): UM GRAFO, TODAS AS ARESTAS (de um grafo), SEPARADOR DE ARESTAS DE CORTE e TODAS AS ARESTAS DE CORTE (de um grafo). O conceito UM GRAFO gera (usando seu método GeraExemplo) exemplos aleatórios de grafos. O conceito TODAS AS ARESTAS pega este grafo como sua entrada e gera um exemplo contendo todas as aresta deste grafo. O SEPARADOR DE ARESTAS DE CORTE apenas mapeia cada aresta do grafo em um booleano que indica se aquela aresta é ou não uma aresta de corte (naquele grafo). A origem deste conceito indica que ele foi criado pela aplicação de um operador ATA (apply to all, ver apêndice C.13) sobre os conceitos 2 (não mostrado nesta figura) e 3 (TODAS AS ARESTAS). O conceito 2 é um RECONHECEDOR DE ARESTAS DE CORTE[1], um conceito booleano (tipo=B) que possui como entrada o conceito UMA ARESTA e que é usado para classificar arestas de corte. O operador ATA é inspirado no operador de mesmo nome definido por Backus[Bac87]. Ele é usado para "aplicar" um determinado conceito com entrada de tipo X (Ex. RECONHECEDOR DE ARESTAS DE CORTE) a cada um dos elementos de um conjunto ou sequência de X's(Ex. O conteúdo do exemplo de TODAS AS ARESTAS.

O conceito TODAS AS ARESTAS DE CORTE seleciona, a partir dos exemplos dos conceitos 3 e 4, todas as arestas que são mapeadas em "sim" pelo exemplo do conceito 4. A origem de TODAS AS ARESTAS DE CORTE indica que ele foi criado pela aplicação do

[1]Um reconhecedor é qualquer conceito do tipo booleano. Ele é equivalente ao que Epstein chama de *tester* [ES91]

operador INVS (Inverte valores Sim, ver apêndice C.12) sobre o conceito 4. Este operador usa sempre a entrada e a entrada da entrada para criar um conjunto formado apenas por elementos que mapeiam em "sim".

3.1.3 Operadores

Para criar novos conceitos o SCOT utiliza um mecanismo inspirado no sistema de programação funcional (SPF) de Backus[Bac87]. A base deste mecanismo são os operadores que equivalem aos functional forms de Backus. Os operadores do SCOT (Ver apêndice C.2) nascem de uma "extensão" dos operadores funcionais matemáticos, como por exemplo, a composição de funções e a inversa de uma função. Além disto, ao contrário do que ocorre no sistema de Backus, os conceitos no SCOT possuem *tipo*. A principal consequência disto é que enquanto no SPF os operadores podiam ser aplicados a qualquer conceito, no SCOT, a aplicação de alguns operadores pode ser restrita a conceitos de um determinado tipo. Por exemplo, o operador CARDINALIDADE é usado para transformar um conceito cujos exemplos são conjuntos (ou sequências) em um outro conceito cujos exemplos são inteiros representando a cardinalidade dos conjuntos. Portanto, ele só pode ser aplicado a conceitos do tipo conjunto ou sequência. Dizemos então que a *aplicabilidade* do operador CARDINALIDADE se restringe a conceitos do tipo conjunto ($\{S\}$) ou sequência ($[S]$).

Cada operador possui um código TESTAAPLICABILIDADE que testa a sua aplicabilidade sobre um conceito, e um código GERACONCEITO que cria um novo conceito a partir de um operando. Além disto, cada operador possui um código GERAEXEMPLO que é usado na geração de exemplos. Usando uma analogia com funções matemáticas e o operador composição, estes três códigos poderiam ser descritos da seguinte forma:

TestaAplicabilidade Dado duas funções $f(x) : A \to B$ e $g(x) : C \to D$ devolve um valor booleano indicando se f e g podem ser compostas, ou seja, que a imagem de g é um subconjunto do domínio de f.

GeraConceito Cria uma nova função h correspondendo a função $fog(x) : C \to B$.

GeraExemplo Dado f, g e $x \in C$, devolve $f(g(x))$.

Operadores podem ser unários, como o INVS, ou binários, como o ATA. Na analogia acima a composição seria um operador binário no SCOT, e as funções f, g e h seriam conceitos. Os domínios A e C também seriam conceitos, enquanto B e D seriam tipos. A origem do conceito h seria um operador chamado composição aplicado aos operandos f e g. Sua entrada seria o conceito C, uma vez que para gerar um exemplo de h é necessário

antes um exemplo de C. O tipo de h seria o mesmo que o de f. A figura C.14 mostra um operador do SCOT inspirado justamente na composição de funções.

A geração de exemplos no SCOT é um processo recursivo que termina sempre nos geradores de exemplos de conceitos primitivos. Geradores de exemplos de conceitos descobertos (não primitivos) chamam códigos GeraExemplo dos operadores que os originaram. Estes códigos, por sua vez, podem requerer exemplos dos operandos. Os operandos podem não ser primitivos, o que gerará novas chamadas ao código GeraExemplo de outros operadores (ou dos mesmos operadores com diferentes operandos). O código GeraExemplo de cada um dos operadores do SCOT é mostrado no apêndice C.2. O código TestaAplicabilidade apenas garante que o sistema não aplicará um operador em conceitos que tornem o código GeraExemplo inconsistente. O código GeraConceito é quem cria uma nova instância da classe conceito e ajusta os valores das propriedades deste novo conceito.

3.1.4 Conjecturas

A última classe de objetos utilizadas pelo SCOT é a classe conjectura. Definimos conjectura como sendo uma relação entre dois conceitos distintos, ambos de tipo B (booleano), e com a mesma entrada, não nula. Inicialmente, no momento em que estes objetos são criados no módulo II, a relação entre os dois conceito é indefinida. É apenas no módulo IV, quando a relação é empiricamente "descoberta", que um objeto conjectura pode passar realmente a representar uma conjectura real. Esta "análise empírica" é guiada pela *heurística de indução de conjectura* (explicada detalhadamente na seção 3.6.4) que compara uma quantidade predeterminada de exemplos de cada conceito. Estes exemplos são sempre "obtidos" para um mesmo exemplo da entrada, ou seja, o SCOT primeiro gera um exemplo da entrada (que é comum aos dois conceitos), e depois passa este exemplo para os geradores de exemplos dos dois conceitos. As relações possíveis entre os conceitos E (Esquerdo) e D (Direito) de uma conjectura são:

Equivalencia O conteúdo de um exemplo de E é Sim se e somente se o conteúdo de um exemplo de D é Sim.

E **Implica em** D O conteúdo de um exemplo de D é Sim sempre que o conteúdo de um exemplo de E é Sim.

D **Implica em** E O conteúdo de um exemplo de E é Sim sempre que o conteúdo de um exemplo de D é Sim.

Indeterminada Nenhuma das relações anteriores foi detectada mas existem algumas entradas onde tanto E quanto D devolvem exemplos com conteúdo igual a Sim.

Sem Relação Nenhuma das relações anteriores foi detectada.

3.2 Representação Gráfica de Conceitos

Introduziremos agora uma notação para representar graficamente a estrutura de um conceito. Nesta notação conceitos são representados por círculos (figura 3.3) e o texto que aparece dentro do círculo é o tipo do conceito. As linhas grossas ligam o conceito a sua entrada, com a seta saindo da entrada e entrando no conceito. Setas grossas que não partem de qualquer círculo indicam uma entrada nula (conceito não tem entrada). As linhas finas ligam o operando, ou operandos, ao conceito gerado. A legenda nas linhas finas é o nome do operador. Linhas pontilhadas representam uma ligação de especialização/generalização entre dois conceitos (seta vai do mais genérico para o mais específico). Estas ligações serão explicadas na próxima seção.

A figura 3.3 mostra os mesmos conceitos da figura 3.2 usando agora a notação gráfica. Para facilitar a visualização utilizamos os mesmo números para identificar os mesmos conceitos em ambas as figuras. Deixamos também de incluir alguns detalhes como a entrada e a origem do conceito 2.

Embora os conceitos 5, 6 e 7 que aparecem na figura 3.3 sejam de certa forma equivalentes, eles diferem na entrada. Todos estes três conceitos geram como exemplo um conjunto formado por todas as arestas de corte de um grafo, no entanto, apenas o conceito 7 parte diretamente de um exemplo de grafo.

3.3 Especializações, Generalizações e Duais

Assim como no AM, o SCOT mantém também uma hierarquia de especializações e generalizações entre os conceitos de sua base. Estas abstrações [2] são ainda classificadas entre lógicas ou empíricas. Abstrações lógicas são consequências da aplicação de determinados operadores à determinados conceitos e são "marcadas" no momento em que o conceito é criado. Por exemplo, o código GeraConceito do operador INVS sempre marca a entrada do seu operando como generalização do conceito gerado pela sua aplicação. Na figura 3.3, a seta pontilhada mostra justamente este fato, com o conceito 3 (TODAS AS ARESTAS) sendo marcado como uma generalização do conceito 5 (TODAS AS ARESTAS DE CORTE), pela aplicação do operador INVS.

Especializações empíricas surgem quando o SCOT "descobre" uma conjectura do tipo *E implica em D*. Ele então marca *E* como sendo uma especialização empírica de *D* (e,

[2] Usaremos o termo abstração para nos refenciarmos tanto à uma generalização quanto à uma especialização

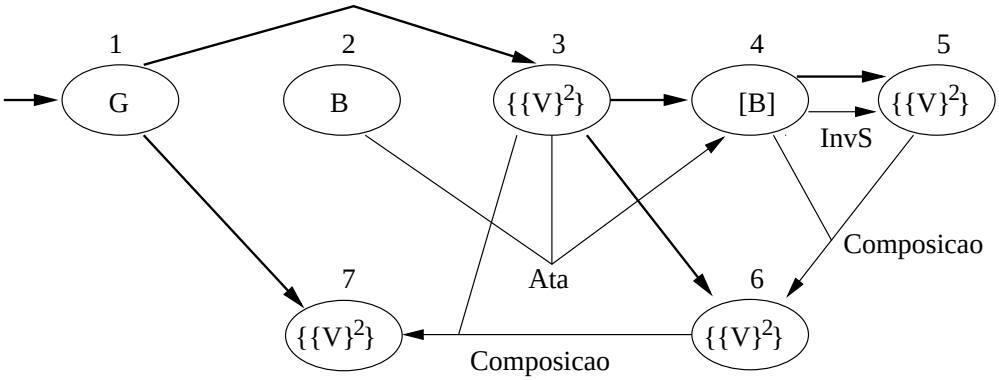

Figura 3.3: Conceito: Todas as Arestas de Corte

é claro, D como sendo um generalização de E). Conjecturas do tipo D *implica em* E são tratadas de maneira simétrica. Além disto, o SCOT respeita a transitividade destas relações, ou seja, se A é especialização de B e B é especialização de C, então, A é marcado como especialização de C.

As abstrações empíricas envolvem apenas conceitos de tipo B (Booleano). No entanto, conceitos com outros tipos podem também ser ligados por uma abstração. Listamos a seguir o significado de uma ligação de especialização para cada tipo onde pode haver uma ligação. Vamos assumir que temos dois conceitos α e β ambos do mesmo tipo, e que α é especialização de β:

Booleano (B) O conteúdo de um exemplo de β é Sim sempre que conteúdo de um exemplo de α é Sim. Por exemplo, α poderia representar o conceito RECONHECEDOR DE GRAFO ÁRVORE e β poderia ser um RECONHECEDOR DE GRAFO CONEXO.

Conjunto sem cardinalidade definida ({S}) Para um mesmo exemplo de entrada, o conteúdo de um exemplo de α é sempre um subconjunto do conteúdo de um exemplo de β. Por exemplo, α = TODAS AS CLIQUES MAXIMAIS DE UM GRAFO e β = TODAS AS CLIQUES DE UM GRAFO.

Vértice (V), grafo (G), conjunto definido ($\{S\}^n$) O conjunto formado por todos os exemplos de α é um subconjunto do conjunto formado por todos os exemplos de β. Por exemplo, ÁRVORE e GRAFO CONEXO, ARESTA DE CORTE e ARESTA.

Observe que na descrição acima distinguimos o conceito GRAFO CONEXO e RECONHECEDOR DE GRAFO CONEXO. No SCOT, reconhecedores e geradores são tratados como conceitos distintos. Reconhecedores são sempre do tipo B, enquanto geradores podem ser de qualquer outro tipo, exceto B ou I. Existe, no entanto, uma relação importante entre um reconhecedor e um gerador que não pode ser perdida. Chamamos esta relação de *dualidade* e um RECONHECEDOR DE GRAFO CONEXO, por exemplo, será chamado de dual do conceito GRAFO CONEXO, e vice-versa. Abstrações são sempre repassadas para seus duais, por exemplo, se o SCOT descobrir que o conceito RECONHECEDOR DE ÁRVORE é uma especialização do conceito RECONHECEDOR DE GRAFO CONEXO, ele vai automaticamente marcar o conceito ÁRVORE como sendo uma especialização do conceito GRAFO CONEXO.

3.4 Macro-operadores

Um macro-operador é uma espécie de molde (template) onde vários operadores são aplicados em sequência sobre um grupo de conceitos que funcionam como parâmetros. A figura 3.4 mostra um dos macro-operadores usados pelo SCOT (ver outros macro-operadores no apêndice D). Os dois círculos com contorno mais escuro representam os parâmetros, ou variáveis, do macro-operador. O gráfico de um macro-operador fornece também as restrições que devem ser aplicadas aos parâmetros. Por exemplo, a figura 3.4 mostra que o tipo do conceito 3 deve ser uma sequência de X, onde X é o tipo da entrada do conceito 2.

Os macro-operadores também ajudam no processo manual de "busca por heurísticas". Primeiro desenhamos o gráfico de diversos conceitos reconhecidamente interessantes da teoria dos grafos. Depois analisamos estes gráficos à procura de padrões na sequência de aplicação de operadores. Assinalamos nestes padrões os conceitos que devem tornar-se parâmetros e obtemos assim novos macro-operadores. Todos os macro-operadores do SCOT foram criados através deste procedimento. A automatização deste procedimento pode ser um primeiro passo na transformação do SCOT em um programa que possa também descobrir novas heurísticas.

3.5 Pontos de Substituição

Um ponto de substituição é um conceito P que participa direta ou indiretamente na geração de um conceito T. Se substituirmos, na estrutura de T, o conceito P por uma abstração P' de P, criaremos um conceito T' que será uma abstração de T. Chamamos o conceito T de alvo do ponto de substituição P. Pontos de substituição podem ser diretos ou inversos. Pontos diretos são aqueles que se substituídos por uma especialização

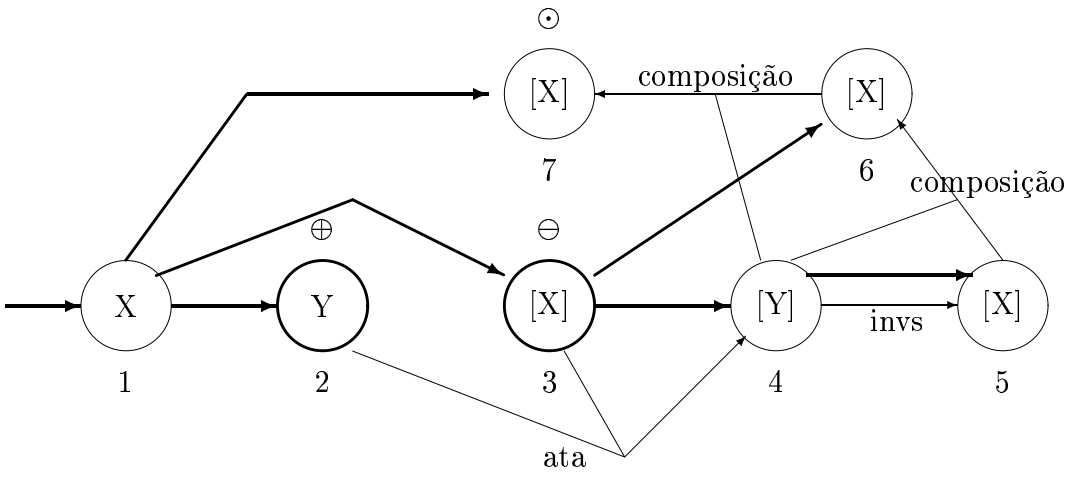

Figura 3.4: Um macro-operador

criam uma especialização do alvo e se substituídos por uma generalização criam uma generalização do alvo. Pontos inversos "invertem" o tipo de abstração: especializações do ponto geram generalizações do alvo e generalizações do ponto geram especializações do alvo. Pontos de substituição são usados para se obter abstrações de um conceito a partir da substituição de alguns dos conceitos que participam da sua estrutura. Os sinais $\oplus$ e $\ominus$ que aparecem sobre os parâmetros 2 e 3 na figura 3.4 indicam pontos de substituição diretos e inversos, respectivamente. O conceito alvo é marcado por um $\odot$. O mecanismo de aplicação de macro-operadores e de operadores se encarrega de marcar os pontos de substituição nos conceitos gerados.

3.6 Módulos

3.6.1 Módulo I - Geração de Conceitos

A lista inicial de conceitos do SCOT começa com cerca de 20 conceitos básicos da teoria dos grafos e 30 conceitos diversos envolvendo valores booleanos e números inteiros (ver apêndice B para lista completa de conceitos do SCOT). O SCOT utiliza apenas 3 mecanismos para criar novos conceitos: os pontos de substituição, os operadores e os macro-operadores. A geração de conceitos divide-se em duas fases, a primeira utiliza os pontos de substituição, a segunda a aplicação de operadores e macro-operadores.

Fase 1 : Substituições

A agenda do SCOT contém apenas um tipo de tarefa: crie especializações (ou generalizações) para o conceito X. Estas tarefas são colocadas na agenda apenas no módulo V e é apenas no módulo V que elas podem ser retiradas da agenda. Durante a fase 1 do módulo I o SCOT tenta executar cada uma das tarefas da agenda utilizando os pontos de substituição dos conceitos que aparecem na tarefa. O procedimento para executar a tarefa *crie especializações para o conceito* X é o seguinte:

Para cada ponto de substituição p de X *faça*
    Se p é direto *então*
        Para cada especialização e de p *faça*
            Crie um "clone" de X mas substitua, neste clone,
            o conceito p pelo conceito e.
    Senão ; p é inverso
        Para cada generalização g de p *faça*
            Crie um "clone" de X mas substitua, neste clone,
            o conceito p pelo conceito g.

O procedimento é simétrico para tarefas que pedem generalizações.

As tarefas podem permanecer na agenda por vários ciclos. Por isto os pontos substituídos são marcados para que no próximo ciclo o sistema não repita as substituições já efetuadas. A execução das tarefas da agenda só pára quando não existem mais quaisquer pontos a serem substituídos ou quando o tempo máximo de execução do módulo termina.

Fase 2 : Operadores e Macro-operadores

Cada operador e cada macro-operador possui um valor numérico associado. Na fase 2 da geração de conceitos operadores e macro-operadores são colocados numa mesma lista e um deles é escolhido através de um processo semi-aleatório, onde a probabilidade de escolha de um elemento é proporcional ao seu valor numérico. Com o operador ou macro-operador definido inicia-se o processo de escolha dos conceitos que deverão servir de operandos (operador) ou parâmetros (macro-operador). Depois de escolher os conceitos o SCOT aplica o operador ou o macro-operador sobre eles, gerando assim novos conceitos. Este processo é repetido até que o limite de tempo ou de total de conceitos do ciclo seja atingido.

Tanto o mecanismo de escolha de operandos quanto o de parâmetros levam em consideração o VALOR de um conceito. Este VALOR é um número calculado dinâmicamente pela *heurística de cálculo do valor*, a partir de 6 propriedades numéricas ligadas a cada conceito:

Desempenho É o tempo de execução médio que o conceito demora para gerar um exemplo.

Profundidade Conceitos primitivos tem profundidade 1. Conceitos não primitivos derivados da aplicação de operadores unários tem profundidade igual a profundidade do operando mais 1. Conceitos não primitivos derivados da aplicação de operadores binários tem profundidade igual a profundidade do operando com maior profundidade mais 1.

ValorExterno Valor fornecido diretamento pelo usuário.

ValorExemplos Valor calculado no módulo de análise de exemplos.

TotalEquivalentes Total de conjecturas do tipo *equivalência* envolvendo o conceito.

TotalAbstrações Total de especializações e generalizações lógicas ou empíricas do conceito.

A fórmula ligando estas 6 variáveis é a seguinte:

$$(\alpha * Desem)^{i_1} + (\beta * Prof)^{i_2} + (\gamma * ValExt.)^{i_3} + (\Delta * ValExe.)^{i_4} + (\delta * TotEq.)^{i_5} + (\theta * TotAbs.)^{i_6}$$

Os coeficientes rotulados com letras gregas são números reais associados a cada variável. Eles podem ser alterados entre duas execuções do SCOT, para favorecer uma ou outra propriedade. Os coeficientes $i_1, i_2, ..., i_6$ podem possuir os valores 1 ou -1, e são usados para inverter a contribuição da propriedade no VALOR do conceito. Por exemplo, fazendo $i_2 = 1$ poderemos favorecer uma expansão em profundidade dos conceito. Já com $i_2 = -1$, a expansão em largura é favorecida. Os valores atuais destes coeficientes foram obtidos empiricamente, a partir dos resultados de diversas execuções com valores diferentes.

Para escolher um operando para um operador unário o sistema primeiramente usa o código TestaAplicabilidade do operador e constrói uma lista com todos os conceitos sobre os quais o operador é aplicável. Um dos conceitos da lista é então escolhido através de um processo semi-aleatório, similar ao usado na escolha dos operadores, que favorece conceitos com maiores valores.

No caso de operadores binários o sistema monta uma lista de pares de conceitos que passam pelo teste de aplicabilidade do operador. O valor de cada elemento da lista é a media aritmética entre os VALORES dos dois conceitos que fazem parte do par.

No caso de um macro-operador o sistema escolhe cada um dos n parâmetros individualmente, levando em consideração o VALOR de cada conceito. Se os conceitos escolhidos não satisfizerem as condições de aplicabilidade do macro-operador o sistema retorna ao início da fase II para escolher um novo operador ou macro-operador.

As *heurísticas de restrição de aplicabilidade de operadores* são testes que ficam embutidos no código TestaAplicabilidade de alguns operadores. Elas restringem ainda mais a aplicabilidade de alguns operadores, diminuindo assim o número de conceitos "candidatos a operandos". Por exemplo, existe no SCOT um operador chamado POTÊNCIA que é usado para gerar conceitos cujos exemplos são conjuntos potência dos conceitos de entrada[3]. Embora o operador POTÊNCIA possa ser aplicado indefinidamente sobre os conceitos gerados por ele, dificilmente um conceito com mais de duas aplicações de POTÊNCIA será interessante (A geração de exemplos para tal conceito seria um processo altamente "explosivo"). Por isto, o SCOT incorpora uma heurística de restrição de aplicabilidade dentro do código TestaAplicabilidade do operador POTÊNCIA para eliminar candidatos a operando que já possuam um operador POTÊNCIA em sua estrutura.

3.6.2 Módulo II - Análise de Exemplos

Neste módulo são gerados 40 exemplos para cada um dos novos conceitos criados no ciclo. Este exemplos são então testados pelas *heurísticas de análise de exemplos*. Estas heurísticas buscam normalmente características não interessantes que possam desvalorizar o conceito, como por exemplo:

- Todos os exemplos tem exatamente o mesmo conteúdo.
- Os conteúdos de todos os exemplos são sequências de elementos iguais entre si.
- Os conteúdos de todos os exemplos são grafos triviais.
- Para todos os exemplos o conteúdo do exemplo é igual ao conteúdo do exemplo de entrada. Esta propriedade é análoga ao conceito de *identidade* em funções matemáticas.
- Todos os exemplos são indefinidos (sistema não conseguiu gerar exemplos do conceito).

Quando qualquer característica não interessante é encontrada a propriedade VALOREXEMPLOS do conceito é zerada. Todas as heurísticas de análise de exemplos da versão atual do SCOT ou desvalorizam o VALOREXEMPLOS ou não fazem nada. Ainda não encontramos nenhuma heurística que deva aumentar o VALOREXEMPLOS com base na análise de exemplos.

No último exemplo de características não interessantes falamos de exemplos indefinidos. Um exemplo indefinido acontece quando o sistema não consegue gerar exemplos do conceito. Tomemos, por exemplo, o conceito aresta, cuja entrada é o conceito grafo e cujo

[3]Eg. Dado o conjunto $C = (123)$ a potência de C é o conjunto $(\phi(1)(2)(3)(12)(13)(23)(123))$

tipo é $\{V\}^2$. Quando um exemplo de aresta é requisitado o sistema automaticamente requere um exemplo de grafo. Pode acontecer do exemplo de grafo ser um grafo trivial (sem arestas). Neste caso, ao invés de devolver um exemplo de aresta, o sistema devolve um símbolo indicando que não foi possível gerar o exemplo requisitado. Chamamos este símbolo de exemplo indefinido. Cabe a rotina chamadora requisitar um novo exemplo ou desistir, sempre que receber um exemplo indefinido.

3.6.3 Módulo III - Geração de Conjecturas

Um *candidato a conjectura* é qualquer par de conceitos booleanos (com tipo B), ambos com a mesma entrada, não nula. No módulo de geração de conjecturas alguns destes candidatos são escolhidos e passam a fazer parte de um objeto conjectura. Candidatos podem ser especiais ou normais. A diferença entre um candidato especial e um candidato normal é que o primeiro possui um dos seus conceitos sendo apontado por pelo menos uma tarefa da agenda e o segundo não. O valor de um candidato é calculado pela média aritmética do VALOR de cada um dos seus dois conceitos. A cada ciclo, o módulo de geração de conjecturas escolhe os 15 candidatos especiais e os 2 candidatos normais com os maiores valores. Caso não existam candidatos suficientes para completar o número requisitado (15 especiais e 2 normais) o SCOT pega todos os disponíveis e passa para o próximo módulo.

3.6.4 Módulo IV - Análise de Conjecturas

Definimos conjectura como sendo uma relação entre dois conceitos. A ordem destes conceitos é importante, pois quando a relação entre os conceitos for de implicação temos que saber quem implica em quem. Denominaremos os dois conceitos da conjectura por *lado esquerdo* e *lado direito*, ou E e D, por economia. As relações possíveis entre os lados da conjectura são cinco, como citado na seção 3.1.4: equivalência, E implica em D, D implica em E, Indeterminada e Sem relação.

Os conceitos E e D são sempre reconhecedores (tipo B), em outras palavras, eles respondem se um determinado exemplo passado como parâmetro (entrada) é ou não um exemplo do dual do reconhecedor (ver capítulo 3 seção 3.3). Tomemos, como exemplo, o conceito RECONHECEDOR DE GRAFO CONEXO. Este conceito tem entrada igual a grafo e dual igual a GRAFO CONEXO. Quando o gerador de exemplos deste conceito é chamado passando-se como parâmetro um exemplo de grafo, o conteúdo do exemplo devolvido será um booleano indicando se o grafo passado é conexo ou não. Ou seja, que o grafo passado como parâmetro é um exemplo de um grafo conexo.

Para descobrir a relação entre E e D o sistema gera primeiramente 100 exemplos do conceito de entrada. Depois ele gera exemplos de E e D para cada um destes 100

exemplos. Os conteúdos dos exemplos assim gerados são agrupados em uma lista de pares ordenados. Cada par ordenado é formado por dois valores booleanos: o conteúdo do exemplo de E e o conteúdo do exemplo de D, respectivamente. O significado de cada uma das 4 combinações possíveis para cada um destes pares ordenados é a seguinte (Usando S para *sim* e N para *não*):

(S S) O exemplo de entrada é reconhecido tanto pelo lado esquerdo quanto pelo lado direito.

(N N) O exemplo de entrada não é reconhecido por nenhum dos lados.

(S N) O exemplo de entrada é reconhecido apenas pelo lado esquerdo.

(N S) O exemplo de entrada é reconhecido apenas pelo lado direito.

Se tomarmos E como um RECONHECEDOR DE GRAFO CONEXO e D como um RECONHECEDOR DE ÁRVORE o significado de cada par seria o seguinte:

(S S) O grafo em questão é conexo e é árvore.

(N N) O grafo em questão nem é conexo nem é árvore.

(S N) O grafo em questão é conexo mas não é árvore.

(N S) O grafo em questão é árvore mas não é conexo[4].

A partir desta lista de pares o sistema cálcula 4 valores percentuais, que chamaremos de *percentuais de indução*:

PercSS Percentual de (S S) sobre o total de pares.

PercNN Percentual de (N N) sobre o total de pares.

PercSN Percentual de (S N) sobre o total de pares.

PercNS Percentual de (N S) sobre o total de pares.

A *heurística de indução de conjectura* simplesmente usa a tabela 3.2 para definir qual o tipo da relação envolvendo os dois conceitos da conjectura.

Otimizamos a implementação desta heurística para que um número mínimo de exemplos sejam necessários (o máximo é 100). O teste para determinar a relação é executado depois de cada exemplo gerado. O sistema pára de gerar exemplos assim que uma das relações é totalmente determinada.

[4]Note que o par (N S) nunca apareceria na lista dos 100 pares da conjectura envolvendo os reconhecedores de grafo conexo e árvore.

Tabela 3.2: Heurística de Indução de Conjectura

Relação	PercSS	PercNN	PercSN	PercNS
Equivalencia	$> 30\%$	—	$= 0\%$	$= 0\%$
E Implica em D	$> 5\%$	—	$> 10\%$	$= 0\%$
D Implica em E	$> 5\%$	—	$> 0\%$	$= 10\%$
Indeterminada	$> 5\%$	—	$> 0\%$	$> 0\%$
Sem Relação	$= 0\%$	—	—	—

3.6.5 Módulo V - Refinamento de Conjecturas

O módulo de refinamento de conjecturas se apoia em duas premissas:

- A equivalência é a mais interessante entre as cinco relações classificadas no módulo IV.
- Podemos definir uma heurística de avaliação que dado duas conjecturas responda qual das duas está "mais próxima" de uma equivalência.

Nossa heurística de avaliação utiliza apenas o tipo da relação e os percentuais de indução para responder qual entra duas conjecturas está "mais próxima" de uma equivalência. Quando as duas conjecturas estão igualmente próximas, nossa heurística escolhe uma das duas arbitrariamente. As relações do tipo *E implica em D* e *D implica em E* não precisam ser diferenciadas nesta avaliação e portanto usaremos o termo *implicação* para descrever estes dois tipos. Apresentamos a seguir o funcionamento da heurística de avaliação, dado duas conjecturas X e Y:

Se o tipo da relação de X é diferente do tipo de Y *então*
  Considera a seguinte ordem entre os tipos de relação:
  Equivalencia > Implicação > Indeterminada > Sem Relação
Senão ; os tipos de X e Y são iguais
  SomaX = PercSN (de X) + PercNS (de X)
  SomaY = PercSN (de Y) + PercNS (de Y)
  Se SomaX < SomaY *então*
    X está mais próximo
  Senão
    Y está mais próximo

A figura 3.5 usa diagramas de Venn para mostrar a ordenação imposta sobre as conjecturas, a partir dos totais proporcionais de SS, SN, NS e NN que aparecem nos exemplos

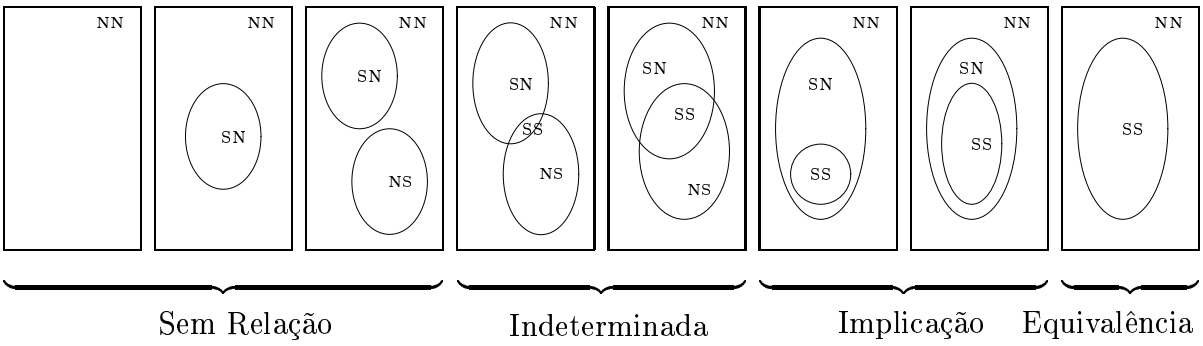

Figura 3.5: Ordenação de Conjecturas

dos 2 conceitos da conjectura. Observe como duas conjecturas do mesmo tipo implicação (ou indeterminada) são ordenadas com base nas proporções de SN e NS.

Para melhorar uma conjectura (aproxima-la da equivalência) o sistema tenta primeiramente substituir um dos conceitos por alguma abstração deste conceito, respeitando a seguinte regra:

Se Relação é do tipo E *implica em* D *então*
  Substitua E por uma generalização de E ou
  Substitua D por uma especialização de E ou
Se Relação é do tipo D *implica em* E *então*
  Substitua D por uma generalização de D ou
  Substitua E por uma especialização de E ou
Se Relação é do tipo *indeterminada então*
  Substitua D por uma generalização de D ou
  Substitua E por uma especialização de E ou
  Substitua E por uma generalização de E ou
  Substitua D por uma especialização de E ou
Se Relação é do tipo *equivalencia* ou *sem relação então*
  Não precisa tentar melhorar

Para cada substituição possível o sistema cria uma nova conjectura. A relação destas novas conjecturas só será definida no próximo ciclo, quando estas conjecturas puderem ser analisadas pelo módulo IV. Depois que a relação é definida e os percentuais de indução calculados, o módulo V pode então determinar se as substituições melhoraram ou não a conjectura.

Quando o sistema não consegue melhorar uma conjectura, usando as abstrações existentes, ele insere tarefas na agenda pedindo a criação de novas abstrações para os conceitos da conjectura. Ele também marca as abstrações testadas para evitar reprocessamento. Sempre que o sistema encontra uma conjectura melhor ele retira da agenda o pedido para novas abstrações. As tarefas também podem ser retiradas da agenda quando uma quantidade predeterminada de ciclos é atingida e o sistema não conseguiu melhorar a conjectura.

3.7 Otimizações

3.7.1 Processamento Distribuído

Tanto o módulo de análise de exemplos quanto o de análise de conjectura podem ser executados distribuidamente. Para isto, o sistema particiona o conjunto de conceitos (na análise de exemplos) ou de conjecturas (na análise de conjecturas), levando em consideração a quantidade de máquinas disponíveis, e atribui um subconjunto diferente para cada máquina. Depois que todos as máquinas terminam de processar a sua parte o sistema "recolhe" todos os resultados e continua a execução dos módulos centralizados (geração de conceitos, geração de conjecturas e refinamento de conjecturas).

3.7.2 Arquivo Permanente de Exemplos

Normalmente, os exemplos de conceitos são descartados depois de serem analisados nos módulos II e IV. No entanto, como os exemplos dos novos conceitos sempre dependem de exemplos de conceitos mais antigos, existe sempre a necessidade de se estar gerando exemplos de conceitos que já passaram pelos módulos II e IV.

Depois de algum tempo de execução os conceitos gerados tendem a se tornar cada vez mais complexos. Esta complexidade se reflete no desempenho do conceito, que passa a demorar mais para gerar exemplos. Além disto, alguns conceitos possuem geradores de exemplos inerentemente exponenciais. A geração de exemplos destes conceitos pode demorar horas. Para evitar o desperdício de processamento os exemplos destes conceitos são armazenados em arquivos permanentes sempre que o tempo médio para gerar um exemplo atinge um certo valor máximo.

Depois que o arquivo permanente atinge um certa quantidade de exemplos o gerador de exemplos pode parar de gerar novos exemplos e passar a reaproveitar os exemplos do arquivo. Existe porém uma situação onde o gerador de exemplos é obrigado a gerar um novo exemplo, mesmo tendo atingido a quantidade máxima de exemplos permanentes: é quando um exemplo de entrada é passado como parâmetro e não existe nenhum exemplo

no arquivo permanente com a entrada requisitada.

O problema com o reaproveitamento de exemplos é que a "aleatoriedade" na geração de exemplos é prejudicada. Depois que o arquivo atinge a quantidade suficiente de exemplos o sistema fica "restrito" aos exemplos gerados até aquele ponto. Por isto é necessário que o limite máximo de exemplos seja suficientemente alto para minimizar este problema. No futuro pretendemos fazer com que o próprio sistema calcule dinamicamente a quantidade suficiente de exemplos permanentes.

3.7.3 Reaproveitamento de Execuções

O SCOT pode ser executado durante horas ou dias dependendo apenas da quantidade de memória disponível. Para evitar que um processamento muito longo seja desperdiçado, o sistema permite que os conceitos e conjecturas "descobertos" numa execução sejam gravados em disco para que numa próxima execução eles não precisem ser "redescobertos". A quantidade de conceitos e conjecturas gerados durante uma execução pode ser muito grande, e muitos destes conceitos e conjecturas podem não ser "interessantes". Como a performance global do sistema é afetada pela quantidade de conceitos e conjecturas atuais, o sistema permite que apenas uma parte dos conceitos e conjecturas sejam preservados. Atualmente é o usuário quem determina como devem ser ordenados os conceitos (valor, desempenho, profundidade, ...) e qual o percentual de conceitos que devem ser preservados, obedecendo-se a ordem. Dentre as conjecturas, apenas aquelas envolvendo algum dos conceitos preservados são mantidas.

Capítulo 4

Resultados

Um dos principais problemas com um sistema de construção automática de teoria é a análise de sua eficiência. Em sistemas que aprendem classificadores a partir de um conjunto de treinamento predefinido esta tarefa consiste basicamente em apresentar novos exemplos ao sistema e analisar a taxa de acerto do sistema frente a estes novos exemplos. Esta mesma ideia não pode ser aplicada aos sistemas de construção automática de teoria, uma vez que os próprios conceitos gerados por estes sistemas são os responsáveis pela geração de exemplos.

Mesmo quando não estamos buscando a descoberta de conceitos originais, é muito difícil saber se o conceito proposto pelo sistema está ao menos, entre aqueles já "consagrados" como interessantes pelos estudiosos da área. No caso particular da teoria dos grafos, e do SCOT, podemos ter conceitos estruturalmente muito complexos, com centenas de operadores e operandos, o que torne a análise manual destes conceitos impraticável. Mesmo quando recorremos aos exemplos do conceito a tarefa continua sendo muito difícil, visto que estes exemplos não são apenas números e listas de números, como era o caso da maioria dos conceitos do AM.

A principal forma que encontramos para analisar os resultados do SCOT foi a criação de um arquivo contendo vários conceitos conhecidos da teoria dos grafos, codificados segundo a representação usada pelo SCOT. Criamos também um programa auxiliar que, após cada execução, procura por estes conceitos e quando algum deles é encontrado, o renomeia e aumenta o valor da sua propriedade ValorExterno. Este mecanismo, no entanto, não nos permite dizer nada sobre os conceitos descobertos pelo SCOT e que não estão codificados no arquivo de teste.

Começando com o conjunto de 45 conceitos, 15 operadores e 5 macro-operadores (descritos nos anexos), o SCOT é executado normalmente por cerca de 15 horas, usando de 5 a 25 máquinas, dependendo da disponibilidade. Depois de cada execução o sistema fornece, em média, cerca de 600 novos conceitos, dos quais, cerca de 60% são marcados

pelo SCOT como não interessantes. O sistema sugere também uma média de 300 conjecturas (entre igualdades e implicações). O sistema já "descobriu" 22 dos 30 conceitos precodificados na base de teste. Entre este conceitos estão, por exemplo: grau de um vértice, clique, aresta de corte e reconhecedores para grafos regulares, conexos, árvores, ciclos e estrelas.

Entre os conceitos que ele não conseguiu "descobrir" estão um classificador de grafos cordais e um classificador de ciclos hamiltonianos. Um primeiro estudo do motivo que impediu com que o SCOT descobrisse estes conceitos (pelo menos na exata representação que propomos no arquivo de teste) nos levou a duas hipóteses: (I) Não existe nenhum macro-operador, na lista do SCOT, que possa se usado para "agilizar" a aplicação dos muitos operadores necessários para criação destes conceitos. Como os macro-operadores tem prefência sobre os operadores, a descoberta destes conceitos acaba sendo prejudicada. (II) As heurísticas de restrição de aplicabilidade, de alguma forma impedem que estes conceitos sejam gerados. As outras heurísticas não podem estar causando este problema uma vez que elas não evitam que o conceito seja gerado, apenas diminuem o valor de uma propriedade numérica, depois que o conceito já foi gerado.

Apesar das descobertas do SCOT ainda se encontrarem num nível bastante elementar, o que realmente nos chamou a atenção foi o modo como algumas das descobertas foram feitas. Vejamos, por exemplo, como o SCOT chegou a conjectura TODO GRAFO CICLO É REGULAR.

- Aplicando o macro-operador TOTAL sobre os conceitos primitivos TODOS OS CAMINHOS SAINDO DE UM VÉRTICE e UM PAR DE VÉRTICES o sistema cria o conceito TOTAL DE CAMINHOS LIGANDO DOIS VÉRTICES. (Módulo I - Fase 2)

- Aplicando o operador unário CARDINALIDADE sobre o conceito primitivo VIZINHANÇA seguido de uma COMPOSIÇÃO ele cria o conceito GRAU DE UM VÉRTICE. Este novo conceito é do tipo inteiro, e tem como entrada o conceito vértice (entrada de vizinhança). (Módulo I - Fase 2)

- Aplicando o macro-operador CLI sobre os conceitos TOTAL DE CAMINHOS LIGANDO DOIS VÉRTICES, TODOS OS PARES DE VÉRTICES DO GRAFO e TODOS OS INTEIROS MAIORES QUE 0, o sistema cria o conceito RECONHECEDOR DE GRAFOS CONEXOS. (Módulo I - Fase 2)

- Substituindo o conceito TODOS OS INTEIROS MAIORES QUE 0 por TODOS OS INTEIROS SÃO IGUAIS A 2 no conceito RECONHECEDOR DE GRAFOS CONEXOS o sistema cria o conceito RECONHECEDOR DE GRAFOS CICLO. Como o conceito TODOS OS INTEIROS IGUAIS A 2 é uma especialização do conceito TODOS OS INTEIROS SÃO MAIORES QUE 0, o sistema marca o RECONHECEDOR DE GRAFO CICLO como sendo

uma especialização lógica do conceito RECONHECEDOR DE GRAFO CONEXO(Módulo I - Fase 1)

- Aplicando novamente o macro-operador CLI, agora sobre os conceitos GRAU DE UM VÉRTICE, TODOS OS VÉRTICES DO GRAFO e TODOS OS INTEIROS IGUAIS ENTRE SI, o sistema cria o conceito RECONHECEDOR DE GRAFOS REGULARES. (Módulo I - Fase 2)

- Eventualmente o sistema cria um objeto conjectura C_1 ligando os conceitos reconhecedores de grafos conexos e regulares, nesta ordem. (Módulo III).

- Analisando C_1 o sistema chega aos seguintes *percentuais de indução*, $PercSS = 2\%$, $PercNN = 52\%$, $PercSN = 37\%$, $PercNS = 9\%$. Ele marca então o tipo da relação como sendo *Indeterminada*. (Módulo IV)

- Como C_1 é *indeterminada* o sistema tenta "refinar" a conjectura substituindo um dos conceitos da conjectura por alguma abstração. Eventualmente, o sistema substitui o RECONHECEDOR DE GRAFO CONEXO pelo RECONHECEDOR DE GRAFO CICLO, criando um novo objeto conjectura C_2. (Módulo V)

- O sistema analisa C_2 obtendo os percentuais $PercSS = 4\%$, $PercNN = 80\%$, $PercSN = 0\%$, $PercNS = 16\%$. O sistema marca então esta conjectura como sendo do tipo "E implica em D". (Módulo IV)

- Finalmente, o sistema compara C_1 e C_2 e opta por manter C_2, uma vez que sua heurística diz que implicações são melhores que relações indeterminadas. (Módulo V).

É importante ressaltar que a sequência de operações acima listadas, não ocorreram necessariamente em uma única execucação do SCOT. Alterando o VALOREXTERNO dos conceitos iniciais e analisando os resultados intermediários entre execuções, nós pudemos "guiar" o SCOT mais rapidamente para uma ou outra conjectura interessante. No entanto, mesmo quando o SCOT é executado sem interações com usuário e com os valores iniciais equilibrados, podemos notar diversos conceitos "razoavelmente" interessantes. Boa parte destes conceitos envolvem a noção de grafos vazios e grafos completos aplicados em contextos diferentes. Ele foi capaz, por exemplo, de fornecer 7 reconhecedores diferentes para grafo vazio e 5 para grafos completos.

Um gerador de grafos cliques foi obtido a partir de um RECONHECEDOR DE GRAFOS COMPLETOS, de um conceito que reconhece conjuntos cuja intersecção é não vazia, do conceito primitivo TODOS OS SUBGRAFOS DE UM GRAFO e dos operadores ATA, INVS,

MAXIMAL, MAPEAMENTO e COMPOSIÇÃO. Basicamente, este gerador obtem primeiramente todos os subgrafos do grafo e os classifica quanto a "completude", aplicando o reconhecedor de grafos completos em cada um destes subgrafos (ATA). O operador INVS é usado para isolar apenas os subgrafos completos (cliques) dos outros subgrafos. O operador MAXIMAL, junto com o conceito primitivo TODOS OS VÉRTICES DE UM GRAFO são usados para se obter todas as cliques maximais[1] do grafo. O operador de MAPEAMENTO, aplicado aos conceitos CLIQUES MAXIMAIS e RECONHECEDOR DE INTERSEÇÃO NÃO VAZIA mapeia as cliques maximais em um conjunto de vértices e cria arestas ligando apenas as cliques maximais com pelo menos um vértice em comum (intersecção não vazia). Aplicando ainda algumas composições o SCOT chega a um conceito que a partir de um grafo (entrada) devolve o grafo clique deste grafo. É importante notar que o SCOT gerou, de maneira análoga, um conceito equivalente ao grafo complementar de um grafo clique utilizando o RECONHECEDOR DE INTERSECÇÕES VAZIAS ao invés do RECONHECEDOR DE INTERSECÇÕES NÃO VAZIAS.

[1] Conjuntos de vértices que no grafo original formam subgrafos completos e que não está contido em qualquer outro conjunto de vértices com esta mesma propriedade

Capítulo 5

Conclusões

Além do problema da análise dos conceitos gerados, acreditamos que ainda falta um pouco mais de foco nos mecanismos de busca do SCOT. Ou seja, a taxa de conceitos interessantes dado o número total de conceitos criados ainda é muito baixa. É claro que existem alguns limites na habilidade do SCOT em determinar o quanto um conceito é interessante. Alguns conceitos na teoria dos grafos são interessantes por servirem de modelos para problemas em outros domínios, não acessíveis ao SCOT. Por exemplo, planaridade é um conceito interessante devido a uma propriedade geométrica de uma das possíveis representações do grafo. SCOT não tem acesso a informações deste tipo e nunca poderia, dado o conjunto atual de conceitos iniciais e operadores, chegar ao conceito de planaridade.

Nós tentamos diferentes formas de busca, alterando principalmente os coeficientes da heurística que calcula o VALOR dos conceitos, e não encontramos qualquer evidência de que o SCOT pare por não poder gerar mais conceitos.

Um dos pontos fortes do SCOT é a representação de conceitos e heurísticas. Grande parte dos conceitos mais conhecidos da teoria dos grafos podem ser representados diretamente a partir do conjunto atual de operadores e conceitos iniciais. A expressividade pode ainda ser melhorada com o acréscímo de novos operadores e conceitos iniciais sem que a estrutura geral e as heurísticas existentes sofram qualquer alteração.

Alem disto, o mapeamento da geração de novos conceitos como uma busca num espaço de conceitos possibilitará o reaproveitamento de diversas ideias já consolidadas nesta área. A descoberta automática de macro-operadores, por exemplo, poderá ser usada pelo SCOT na descoberta de novas heurísticas, ou mesmo como uma ferramenta para auxiliar na inserção de heurísticas que não possam ser facilmente definidas. Neste último caso, um pesquisador poderia fornecer alguns conceitos comprovadamente interessantes para que o sistema pudesse inferir um ou mais macro-operadores. Estes macro-operadores seriam usados posteriormente na geração de conceitos análogos, de alguma forma, aos que deram origem aos macro-operadores.

A expressividade do nosso sistema está diretamente ligada ao conjunto de operadores, a lista de conceitos iniciais e a gramática de tipos. Cada um destes itens pode ser classificado como genérico ou específico para o domínio de exploração do sistema. Para trocar o domínio do sistema (de grafos para aritmética, por exemplo), são necessárias substituições apenas na parte específica da lista de operadores, da lista inicial de conceitos e da gramática.

Acreditamos que o SCOT poderá ser aperfeiçoado para se tornar uma boa ferramenta para estudo de heurísticas. As heurísticas atuais são poucas, mas bem estruturadas, de forma que novas heurísticas podem ser facilmente inseridas. Com isto, o pesquisador poderia inserir manualmente novas heurísticas, por exemplo, para testar uma mesma idéia em grupos de conceitos distintos.

Apêndice A

Classes do SCOT

A.1 Conceito

Origem Aponta para o operador e para os operandos usados na geração do conceito. Se o conceito for primitivo contém Null .

Entrada Conceito do qual ele é dependente ou Null , caso o conceito não dependa de outro.

Tipo Determina a forma do conteúdo dos exemplos do conceito.

Desempenho É o tempo de execução médio que o conceito demora para gerar um exemplo.

Profundidade Definimos a profundidade de um conceitos recursivamente: Conceitos primitivos tem profundidade 1. Conceitos não primitivos derivados da aplicação de operadores unários tem profundidade igual a profundidade do operando mais 1. Conceitos não primitivos derivados da aplicação de operadores binários tem profundidade igual a profundidade do operando com maior profundidade mais 1.

ValorExterno Valor fornecido diretamento pelo usuário.

ValorExemplos Valor calculado no módulo de análise de exemplos.

TotalEquivalentes Total de conjecturas do tipo "equivalencia" envolvendo o conceito.

TotalAbstrações Total de especializações e generalizações lógicas ou empíricas do conceito.

Determinístico Indica se o conceito é determinístico.

GeraExemplo No caso de conceitos primitivos contém o código para geração de exemplos do conceito. Para conceitos não primitivos contém apenas uma chamada para o código GeraExemplo do operador que deu origem ao conceito.

Externo Indica se o conceito é externo à teoria dos grafos. Para limitar seu domínio à teoria dos grafos o SCOT nunca combina conceitos externos entre si. Nenhum dos conceitos gerados pelo SCOT é externo.

A.2 Exemplo

Conceito Ponteiro para o conceito que gerou o exemplo.

Entrada Ponteiro para um exemplo do conceito apontado pela propriedade Entrada do objeto conceito.

Conteúdo Conteúdo do exemplo.

A.3 Operador

Nome Nome do operador. Começa sempre com $.

TestaAplicabilidade Código que recebe um ou dois conceitos como parâmetro e devolve um booleano indicando se o operador pode ser aplicado a estes conceitos.

GeraConceito Código que recebe um ou dois conceitos como parâmetro e cria um novo conceito.

GeraExemplo Código usado para gerar exemplos de conceitos.

QtdOperandos Quantidade de operandos. Indica se o operador é unário ou binário.

A.4 Macro-operador

Nome Nome do macro-operador.

TestaAplicabilidade Código que recebe alguns conceitos como parâmetro e devolve um booleano indicando se o macro-operador pode ser aplicado a estes conceitos.

GeraConceito Código que executa o macro-operador.

QtdParâmetros Quantidade de parâmetros do macro-operador.

A.5 Conjectura

LadoEsquerdo Aponta para um conceito.

LadoDireito Aponta para um conceito.

Relação Tipo da relação entre os dois conceitos da conjectura.

Apêndice B

Lista de Conceitos

B.1 Conceitos da Teoria dos Grafos

Grafo	**Vértice**
Um grafo simples Entrada: Null Tipo: G Dual: Reconhecedor Grafo	Um dos vértices do Grafo Entrada: Grafo Tipo: V Dual: Reconhecedor Vértice
Par de Vértices*	**Adjacência**
Dois vértices de um Grafo. Serve de entrada para o conceito "Adjacência". Entrada: Grafo Tipo: $\{V\}^2$ Dual: Reconhecedor Par de Vértices	Booleano indicando se dois vértices de um Grafo são adjacentes Entrada: Par de Vértices Tipo: B
TodosV	**Todos2V**
Todos os vértices do Grafo Entrada: Grafo Tipo: {V}	Todos os pares de vértices do Grafo Entrada: Grafo Tipo: $\{\{V\}^2\}$
TodosCamSaiV	**AlgunsVértices**
Todos os caminhos que saem de um vértice Entrada: Vértice Tipo: {[V]}	Conjunto de Vértices de um Grafo. Serve de entrada para o conceito "Induzido" Entrada: Grafo Tipo: {V}

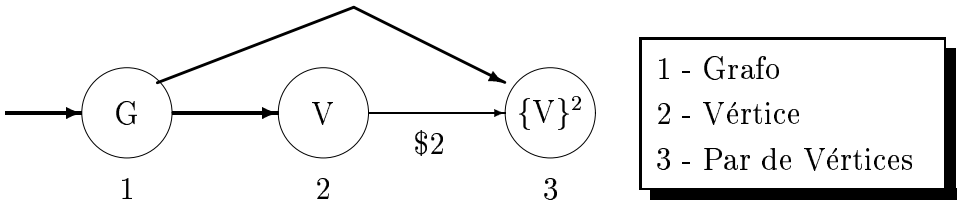

Figura B.1: Par de Vértices

InduzidoV	**Vizinhança**
Subgrafo induzido por um conjunto de vértices. Busca o primeiro grafo "caminhando" pelas Entradas. Entrada: AlgunsVértices Tipo: G	Conjuntos dos vértices que são adjacentes a um determinado vértice Entrada: Vértice Tipo: {V}
Todas as Arestas*	**Aresta***
Todas as arestas do Grafo Entrada: Grafo Tipo: $\{\{V\}^2\}$	Uma aresta do grafo Entrada: Grafo Tipo: $\{V\}^2$
Reconhecedor Vértice	**Reconhecedor Par de Vértices**
Devolve sempre Sim Entrada: Vértice Tipo: B Dual: Vértice	Devolve sempre Sim Entrada: Par de Vértices Tipo: B Dual: Par de Vértices
Reconhecedor Grafo	
Devolve sempre Sim Entrada: Grafo Tipo: B Dual: Grafo	

* Estes conceitos não são primitivos. Os gráficos destes conceitos são mostrados nas figuras B.1, B.2 e B.3. Observe que na figura B.3 o conceito "todas as arestas" aparece "resumido" em um único círculo. Para se obter a árvore de geração completa do conceito "aresta" é necessário "expandir" este círculo usando o gráfico da figura B.2

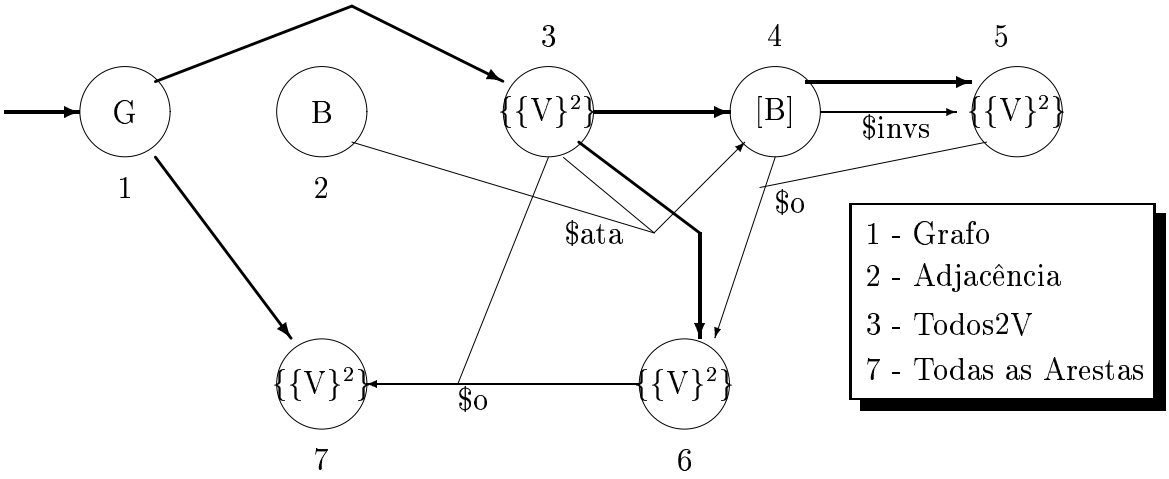

Figura B.2: Todas as Arestas

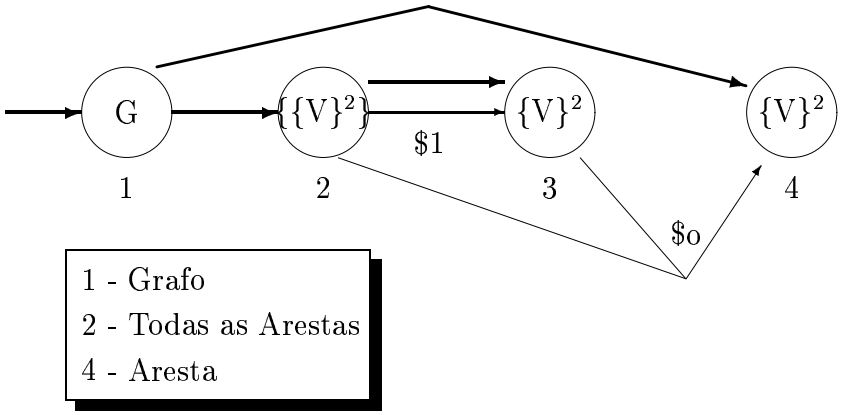

Figura B.3: Aresta

B.2 Conceitos Externos

Bool	**Inteiro**
Um booleano Entrada: Null Tipo: B	Um inteiro Entrada: Null Tipo: I
SBool	**SInteiro**
Sequência de Booleanos Entrada: Null Tipo: [B]	Sequência de Inteiros Entrada: Null Tipo: [I]
NenhumSim	**TodosSim**
Testa se nenhum dos elementos é igual a Sim Entrada: SBool Tipo: B	Testa se todos os elementos são iguais a Sim Entrada: SBool Tipo: B
AlgumSim	**Par**
Testa se pelo menos um dos elementos é igual a Sim Entrada: SBool Tipo: B	Testa se o inteiro é par Entrada: Inteiro Tipo: B
Menor	**Maior**
Menor inteiro do conjunto Entrada: SInteiro Tipo: I	Maior inteiro do conjunto Entrada: SInteiro Tipo: I
Iguais	**Iguais1**
Testa se todos os elementos são iguais entre si Entrada: SInteiro Tipo: B	Testa se todos os elementos são iguais a 1 Entrada: SInteiro Tipo: B
Iguais2 Teste se todos os elementos são iguais a 2 Entrada: SInteiro Tipo: B	**Maior0** Testa se todos os elementos são maiores que 0 Entrada: SInteiro Tipo: B

MenorIgual1	**MenorIgual2**
Testa se todos os elementos são menores ou iguais a 1 Entrada: SInteiro Tipo: B	Testa se todos os elementos são menores ou iguais a 2 Entrada: SInteiro Tipo: B
Crescente	
Testa se um sequência de inteiros é crescente. Entrada: SInteiro Tipo: B	

É importante esclarecer que na base inicial nenhum dos conceitos externos com entrada SInteiro e tipo B faz uso de quantificadores existenciais ("existe algum ..."). Com isto, garantimos a existência do ponto de substituição inverso no macro-operador LG (Apêndice D.1). Se a base inicial contivesse, por exemplo, o conceito "existe algum 0 entre os inteiros", nós já não poderiamos garantir que este ponto de substituição funcionasse sempre como um ponto inverso.

Apêndice C

Lista de Operadores

Como descrito no apêndice A.5, cada operador possui 3 algoritmos associados a ele: TestaAplicabilidade, GeraConceito e GeraExemplo. Neste apêndice o algoritmo GeraExemplo é descrito textualmente enquanto os outros dois são mostrados graficamente, num único gráfico. Estes gráficos seguem a "semântica" definida para os gráficos de conceitos, descrita na seção 3.2, com os seguintes acréscimos:

- Rótulos op, op1 e op2 denotam operandos.
- Rótulos cg denotam conceitos gerados depois que o operador é aplicado.
- As letras X, Y e Z, quando usadas numa descrição de tipo (texto dentro do círculo), representam variáveis que podem ser instanciadas por qualquer tipo. Dentro de um mesmo gráfico, variáveis com o mesmo nome representam sempre o mesmo tipo.

O conceito gerado pela aplicação do operador será denominado ConceitoG. As funções Length, First, Second e Append são equivalentes às usadas para manipulação de Listas em Lisp. A função MakeInstance aciona construtores de objetos.

C.1 Operadores Unários

Cardinalidade

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Se ConceitoG.Entrada $\neq$ Null e ExemploEntrada = Null *então*
  ExemploEntrada $\leftarrow$ ConceitoG.Entrada.GeraExemplo()
$n \leftarrow$Length(ExemploEntrada.Conteúdo)

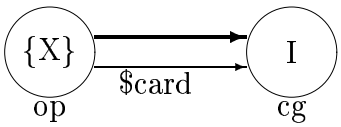

Figura C.1: Operador Cardinalidade

S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←n
Retorna(S)

O algoritmo GeraExemplo do operador cardinalidade pode ser interpretado da seguinte forma: GeraExemplo possui dois parâmetros de entrada, o primeiro é um objeto da classe conceito e o segundo é um objeto da classe exemplo. A primeira coisa que GeraExemplo faz é verificar se ConceitoG não depende de outro conceito. Caso ConceitoG dependa de um exemplo de outro conceito (ConceitoG.Entrada $\neq$ Null) e este exemplo não foi passado como parâmetro (ExemploEntrada = Null), GeraExemplo requisita a geração de um exemplo do conceito "ConceitoG.Entrada". Com ExemploEntrada definido GeraExemplo calcula então a cardinalidade do conjunto armazenado na propriedade conteúdo deste objeto (n ←Length(ExemploEntrada.Conteúdo)). A última parte do algoritmo trata da criação de um objeto da classe Exemplo (S), do ajuste nos valores das propriedades deste objeto e do retorno deste objeto.

Do gráfico do operador cardinalidade podemos tirar as seguintes informações: o operando deve ser do tipo conjunto, o conceito gerado terá tipo I e o operando será a entrada do conceito gerado.

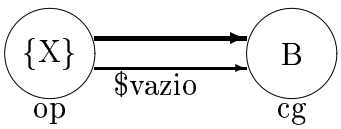

Figura C.2: Operador Vazio

Vazio

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
    ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
Se ExemploEntrada.Conteúdo = Null *então*
    Res ←Sim
Senão
    Res ←Não
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←Res
Retorna(S)

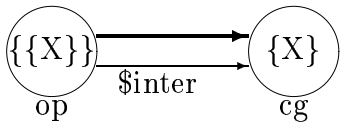

Figura C.3: Operador Interseção

Interseção

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
    ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
α ←ExemploEntrada.Conteúdo
β ←First(α)
Para cada elemento x de α *faça*
    $\beta \leftarrow \beta \cap x$
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG

S.Entrada ←ExemploEntrada
S.Conteúdo ←β
Retorna(S)

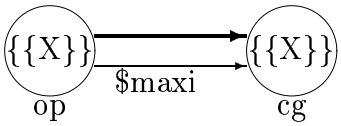

Figura C.4: Operador Maximal

Maximal

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
α ←ExemploEntrada.Conteúdo
β ←Null
Para cada elemento x de α *faça*
Se x não está contido em nenhum
outro elemento de α (x é maximal em α) *faça*
β ←Append(β, x)
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←β
Retorna(S)

Negacao

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*

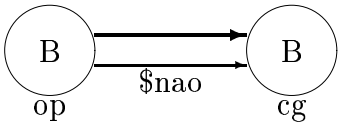

Figura C.5: Operador Negação

ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
Se ExemploEntrada.Conteúdo = Sim *então*
Res ←Não
Senão
Res ←Sim
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←Res
Retorna(S)

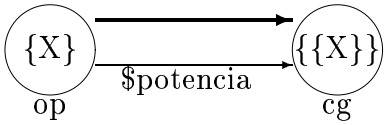

Figura C.6: Operador Potência

Potencia

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
X ←MontaPotencia(ExemploEntrada.Conteúdo)
Ex: Potência de {123} é $\{\emptyset, \{1\}, \{2\}, \{3\}, \{12\}, \{13\}, \{23\}, \{123\}\}$

X ←RetireConjuntoVazioeUnário(X)
; Alterei a definição de potência para evitar a geração de grafos triviais.
; Desta forma a potência de {123} passa a ser {{12}, {13}, {23}, {123}}
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←X
Retorna(S)

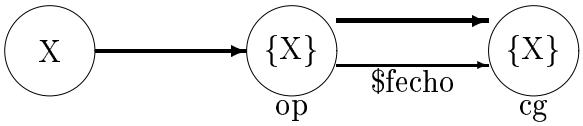

Figura C.7: Operador Fecho

Fecho

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
E ←ExemploEntrada
Operando ←ConceitoG.Origem.Operando ; Operando usado na criação de ConceitoG
α ←E.Conteúdo
γ ←α ∪ E.Entrada.Conteúdo
TamanhoAntes ←length(γ)
Tentativas ←EvitaLoop ←0
Enquanto Tentativas < 10 e EvitaLoop < 200 *faça*
α' ←Null
Para cada elemento x de α *faça*
X ←MakeInstance(Exemplo)
X.Conceito ←Operando.Entrada
X.Entrada ←E.Entrada.Entrada
X.Conteúdo ←x

G ←Operando.GeraExemplo(X)
Para cada elemento g de G.Conteúdo *faça*
$\gamma \leftarrow \gamma \cup g$
$\alpha' \leftarrow \alpha' \cup g$
Se TamanhoAnterior = Length(γ) *então* ++Tentativas
Senão Tentativas ←0
TamanhoAnterior ←Length(γ)
$\alpha \leftarrow \alpha'$
++EvitaLoop
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←γ
Retorna(S)

O operador $fecho é inspirado no conceito de "fecho transitivo reflexivo" da matemática (funções). Por exemplo, aplicando este operador sobre o conceito primitivo vizinhanca (cf. apêndice B) teriamos o conceito de "todos os vértices alcançaveis por um determinado vértice". Exemplos deste conceito seriam conjuntos formados pelo vértice, pelos vizinhos do vértice, pelos vizinhos dos vizinhos dos vértice, pelos vizinhos dos vizinhos dos vizinhos do vértice e assim por diante.

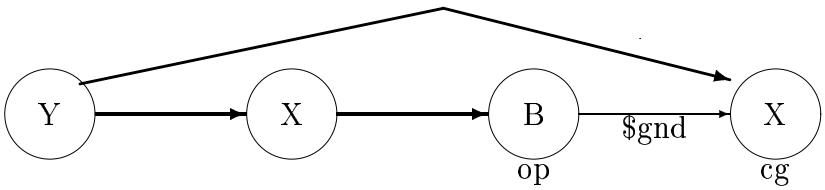

Figura C.8: Operador Gerador Não Determinístico

Gerador Não Determinístico

CódigoExecuta(ConceitoG , ExemploEntrada)

Se ExemploEntrada = Null e ConceitoG.Entrada ≠ Null *então*

```
    ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
Operando ←ConceitoG.Origem.Operando
EvitaLoop ←0
Enquanto EvitaLoop < 200 faça
      EntOp ← Operando.Entrada.GeraExemplo(ExemploEntrada)
      E ←Operando.GeraExemplo(EntOp)
      Se E.Conteúdo = Não então
         ++EvitaLoop
      Senão break  ; Encontrou um exemplo
Se EvitaLoop >= 200 então
    Retorna(Indefinido)
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.ExemploEntrada ←EntOp.Entrada
S.Conteúdo ←EntOp.Conteúdo
Retorna(S)
```

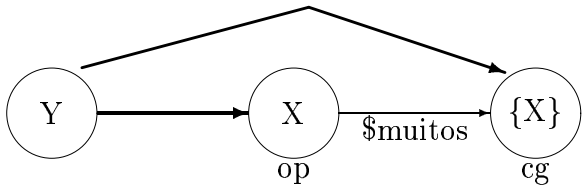

Figura C.9: Operador Muitos

Muitos

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

```
Se ConceitoG.Entrada ≠ Null  e ExemploEntrada = Null  então
    ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
Operando ←ConceitoG.Origem.Operando
α ←Null
Tentativas ←EvitaLoop ←0
```

TamanhoAnterior = Length(α)
Enquanto Tentativas < 10 e EvitaLoop < 200 *faça*
    E ←Operando.GeraExemplo(ExemploEntrada)
    α ←$\alpha \cup$ E.Conteúdo
    Se TamanhoAnterior = Length(α) *então* ++Tentativas
    Senão Tentativas ←0
    TamanhoAnterior ←Length(α)
    ++EvitaLoop
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←α
Retorna(S)

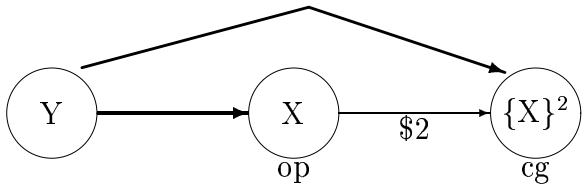

Figura C.10: Operador Dois

Dois

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
    ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
Operando ←ConceitoG.Origem.Operando
α ←Null
EvitaLoop ←0
Enquanto EvitaLoop < 15 *faça*
    E ←Operando.GeraExemplo(ExemploEntrada)
    α ←$\alpha \cup$ E.Conteúdo

Se Length(α) = 2 *então*
break
++ EvitaLoop
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←α
Retorna(S)

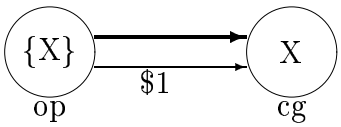

Figura C.11: Operador Um

Um

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
um ←RandomMember(ExemploEntrada.Conteúdo)
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←*um*
Retorna(S)

Inversa de Sim

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*

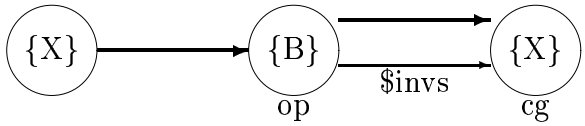

Figura C.12: Operador Inversa de Sim

ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
α ←ExemploEntrada.Entrada.Conteúdo ; Conjunto de X's
β ←ExemploEntrada.Conteúdo ; Conjunto de B's
γ ←Null
Para i = 1 até Length(α) *faça*
Se $\beta[i]$ = Sim *então*
γ ←Append($\gamma, \alpha[i]$)
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←γ
Retorna(S)

C.2 Operadores Binários

Apply to All

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Operando1 ←ConceitoG.Origem.Operando1
Operando2 ←ConceitoG.Origem.Operando2
Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
α ←ExemploEntrada.Conteúdo
β ←Null
Para cada elemento x de α *faça*
X ←MakeInstance(Exemplo)

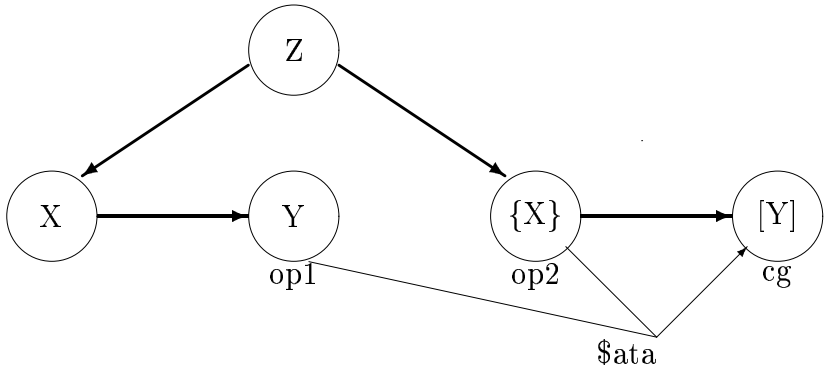

Figura C.13: Operador Apply to All

X.Conceito ←Operando1.Entrada
X.Entrada ←ExemploEntrada.Entrada
X.Conteúdo ←x
Y ←Operando1.GeraExemplo(X)
β ←Append(β,Y.Conteúdo)
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←β
Retorna(S)

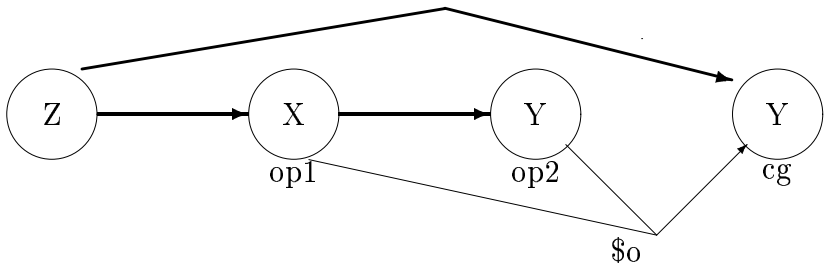

Figura C.14: Operador Composição

Composição

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Operando1 ←ConceitoG.Origem.Operando1
Operando2 ←ConceitoG.Origem.Operando2
Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
  ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
E_1 ←Operando1.GeraExemplo(ExemploEntrada)
E_2 ←Operando2.GeraExemplo(E_1)
E_2.Conceito ←ConceitoG
E_2.Entrada ←ExemploEntrada
Retorna(E_2)

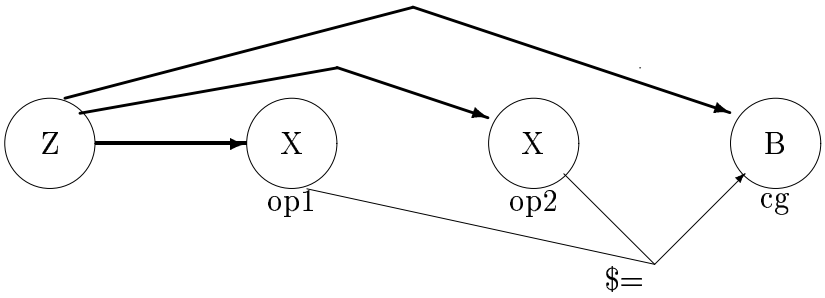

Figura C.15: Operador Testa Igualdade

Testa Igualdade

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Operando1 ←ConceitoG.Origem.Operando1
Operando2 ←ConceitoG.Origem.Operando2
Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
  ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
E_1 ←Operando1.GeraExemplo(ExemploEntrada)

E_2 ←Operando2.GeraExemplo(ExemploEntrada)
Se E_1.Conteúdo = E_2.Conteúdo *então*
    Res ←Sim
Senão
    Res ←Não
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←Res
Retorna(S)

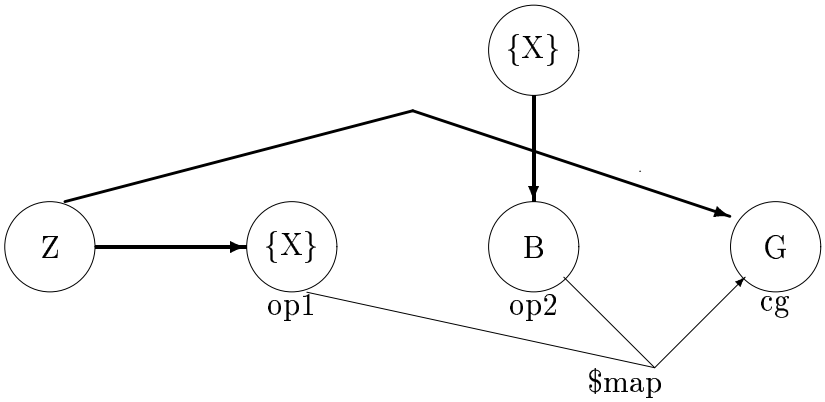

Figura C.16: Operador Mapeamento

Mapeamento

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Operando1 ←ConceitoG.Origem.Operando1
Operando2 ←ConceitoG.Origem.Operando2
Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*
    ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
E ←Operando1.GeraExemplo(ExemploEntrada)
α ←E.Conteúdo

Vértices ←{ 0 1 2 ... Length(α) }
Arestas ←Null
Para x de 0 até Length(α)-1 *faça*
    Para y de $x+1$ até Length(α) *faça*
        X ←MakeInstance(Exemplo)
        X.Conceito ←Operando2.Entrada
        X.Entrada ←ExemploEntrada
        X.Conteúdo ←{$\alpha[x]$,$\alpha[y]$}
        Y ←Operando2.GeraExemplo(X)
        Se Y.Conteúdo = Sim *então*
            Arestas ←Arestas ∪{$\{x, y\}$}
β ←{ Vértices , Arestas } ;
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←β
Retorna(S)

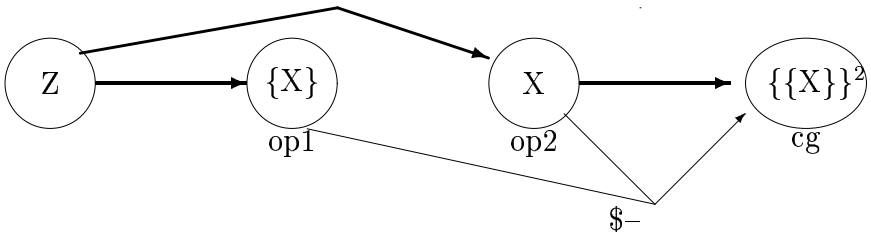

Figura C.17: Operador Remove Um

Remove Um

GeraExemplo(Conceito: ConceitoG , Exemplo: ExemploEntrada)

Operando1 ←ConceitoG.Origem.Operando1
Operando2 ←ConceitoG.Origem.Operando2
Se ConceitoG.Entrada ≠ Null e ExemploEntrada = Null *então*

```
    ExemploEntrada ← ConceitoG.Entrada.GeraExemplo()
E ←Operando1.GeraExemplo(ExemploEntrada.Entrada)
Elemento ←ExemploEntrada.Conteudo
ConjuntoAntes ←E.Conteudo
ConjuntoDepois ←ConjuntoAntes - Elemento
res ←{ ConjuntoAntes , ConjuntoDepois }
S ←MakeInstance(Exemplo)
S.Conceito ←ConceitoG
S.Entrada ←ExemploEntrada
S.Conteúdo ←res
Retorna(S)
```

Apêndice D

Lista de macro-operadores

Mostramos neste apêndice os gráficos de 4 macro-operadores utilizados pelo SCOT. Junto com cada gráfico forneceremos um exemplo de como o macro-operador pode ser usado para gerar alguns conceitos importantes da teoria dos grafos.

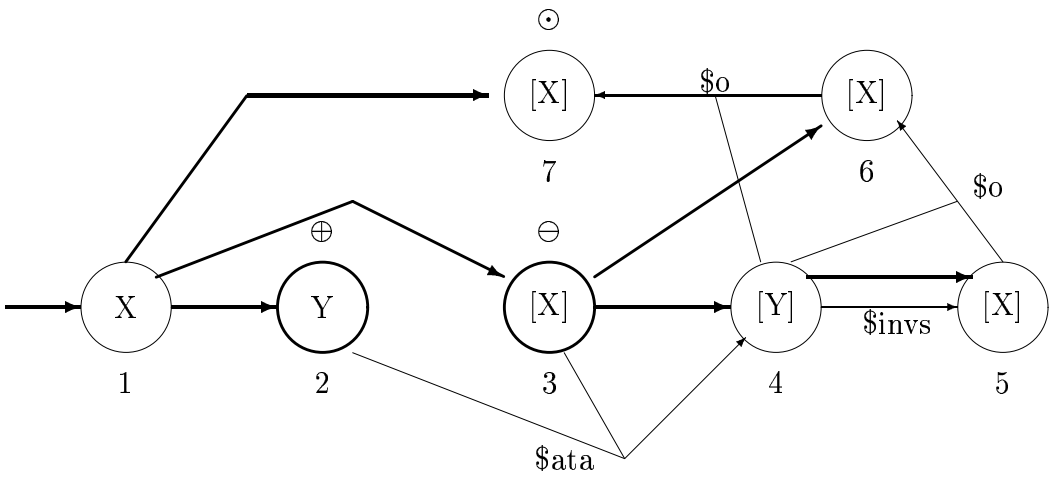

Figura D.1: Macro-operador LG

O macro-operador LG (figura D.1) pode ser usado, por exemplo, na construção de conceitos com a seguinte forma: "todos os subgrafos Z de um grafo", onde Z é uma classe de grafos (eg. conexos, ciclos, árvores, cordais, etc). Para isto basta substituir o parâmetro 2 pelo conceito reconhecedor da classe e o parâmetro 3 pelo conceito "todos os subgrafos de um grafo". O conceito 4 passaria a ser um conceito cujos exemplos "devolvem" uma sequência de booleanos, cada booleano correspondendo à um dos subgrafos do grafo e

indicando se o subgrafo pertence ou não a classe definida pelo parâmetro 2. O operador $invs aplicado ao conceito 4 gera então um conceito cujos exemplos fornecem uma lista dos subgrafos que foram reconhecidos pela classe, ou seja, que mapeiam em Sim nos exemplos do conceito 4. Os operadores composição, neste macro-operador, servem apenas para ajustar as entradas, de forma que a entrada do conceito 7 seja o conceito grafo (entrada do parâmetro 3, neste exemplo). Por exemplo, substituindo o parâmetro 2 pelo conceito "reconhecedor de grafos ciclo" e o parâmetro 3 pelo conceito "todos os subgrafos de um grafo", obtemos o conceito "todos os subgrafos ciclo de um grafo".

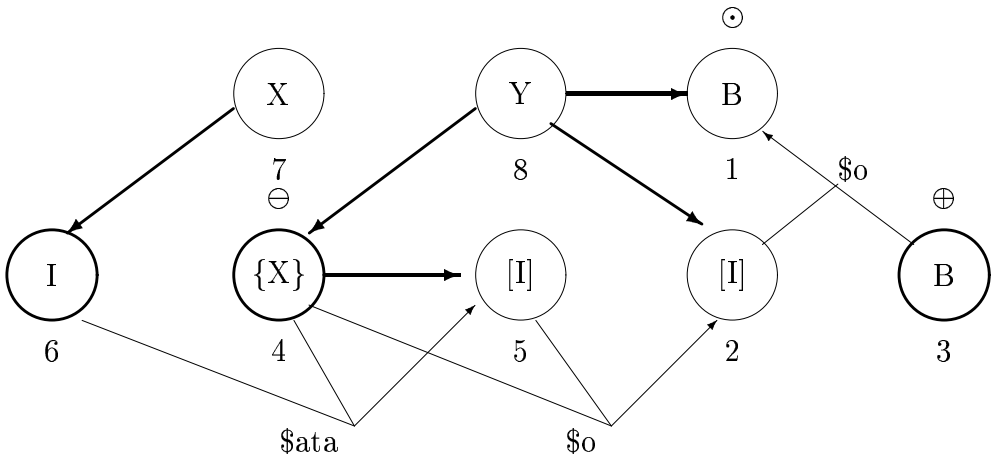

Figura D.2: Macro-operador CLI

O macro-operador CLI (figura D.2) pode ser usado, por exemplo, na construção do conceito reconhecedor de grafos conexos. Para isto temos que substituir o parâmetro 2 pelo conceito "total de caminhos ligando dois vértices" , o parâmetro 3 pelo conceito "todos os pares de vértices do grafo" e o parâmetro 6 pelo conceito "todos os inteiros >= 0[1]". O operador $ata gera um conceito cujos exemplos devolvem, para cada par de vértices do grafo, o total de caminhos ligando estes vértices. A primeira composição ajusta a entrada do conceito 5 para que esta seja um grafo. A segunda composição gera o conceito 7 que reconhece grafos onde todos os pares de vértices estão ligados por pelo menos um caminho, o que corresponde a uma das definições "populares" de grafos conexos. Substituindo-se o parâmetro 6 pelos conceitos "todos os inteiros iguais a 1", ou "todos os inteiros iguais a 2", obteríamos, respectivamente, os reconhecedores para árvores e para grafos ciclo.

[1]Conceito primitivo e externo que consta da base inicial do SCOT (cf. apêndice B.2)

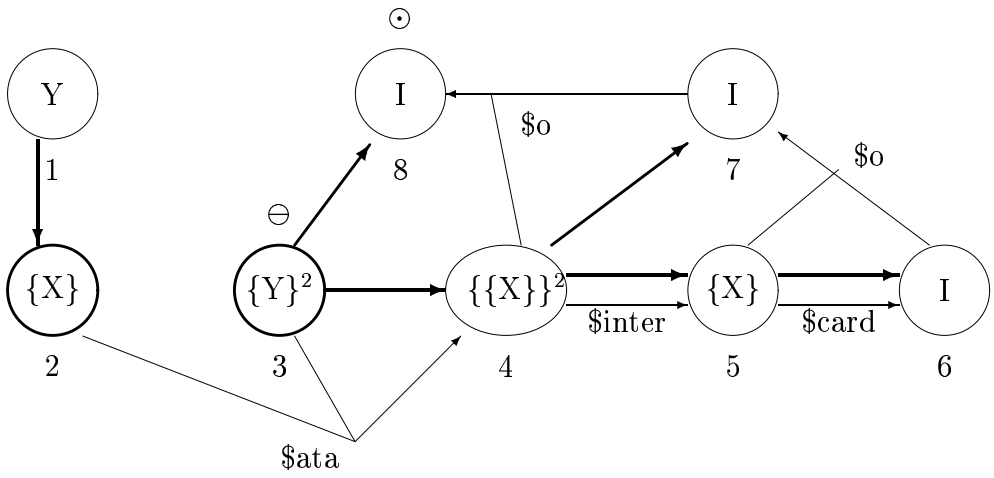

Figura D.3: Macro-operador Total

O conceito "total de caminhos ligando dois vértices" citado no exemplo de utilização do macro-operador CLI pode ser criado apartir da aplicação do macro-operador total (figura D.3). Para isto temos que substituir o parâmetro 2 pelo conceito "todos os caminhos que saem do vértice" e o parâmetro 3 pelo conceito "par de vértices". O operador $ata cria um conceito cujos exemplos representam um par de conjuntos: o conjunto α de todos os caminhos saindo de um dos vértices e o conjunto β de todos os caminhos saindo do outro vértice. O operador $inter cria um conceito cujos exemplos devolvem a interseção entre os conjuntos α e β. Os exemplos do conceito 5 representam justamente todos os caminhos ligando um par de vértices[2]. O operador $card gera então um conceito cujos exemplos fornecem a cardinalidade da interseção de α e β. Os operadores de composição apenas ajustam as entradas para que a entrada do conceito 8 seja exatamente um par de vértices. O conceito 8 é justamente o "total de caminhos ligando dois vértices".

O macro-operador completo (figura D.4) pode ser usado, por exemplo, na construção do conceito "reconhecedor de grafo completo". Para isto devemos substituir o parâmetro 2 pelo conceito "todas as arestas do grafo" e o parâmetro 5 pelo conceito "todos os pares de vértices do grafo". Com a aplicação dos operadores $card e $o os conceitos 4 e 7 passarão a representar, respectivamente, o "total de pares de vértices do grafo[3]" e o "total de arestas do grafo". O operador $= será usado então para criar um conceito cujos exemplos devolvem Sim apenas quando o total de pares de vértices do grafo for igual ao total de

[2]Caminhos são tratados como conjuntos de inteiros no momento da aplicação da interseção
[3]Em outras palavras, o total máximo de arestas que o grafo pode conter

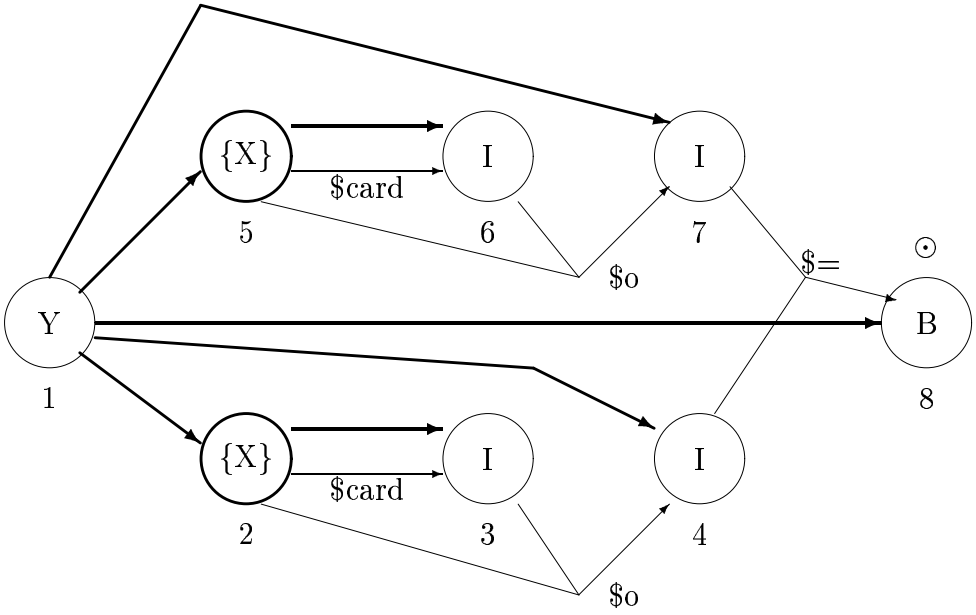

Figura D.4: Macro-operador Completo

arestas do grafo. Este conceito é justamente o reconhecedor de grafos completos.

Apêndice E

Heurísticas

As heurísticas usadas pelo SCOT podem ser agrupadas em 7 classes. Cada classe é diferenciada pelo seu campo de atuação dentro do sistema e pela maneira como são representadas. Descreveremos agora cada uma destas classes:

Heurísticas de restrição de aplicabilidade do operador Heurísticas que diminuem a aplicabilidade de um operador, acrescentando restrições ao conteúdo da árvore de geração dos candidatos a operando.

Macro-operadores Esquema de aplicação de operadores. Cada macro-operador gera vários conceitos intermediários, utilizando vários operadores, para construir o conceito alvo. Estes esquemas de aplicação nascem da análise manual da árvore de geração de conceitos comprovadamente interessantes.

Heurística de cálculo do valor do conceito Fórmula que utiliza algumas propriedades numéricas dos conceitos para determinar um único número para cada conceito. Esta número é usado para "favorecer" ou "prejudicar" a escolha do conceito durante os módulos de geração de conceitos e de conjecturas.

Heurística de análise de exemplos Usadas no módulo II para definir se um conceito deve ou não ter os seus votos zerados. São testes envolvendo o conteúdo de vários exemplos de um mesmo conceito.

Heurística de indução de conjecturas Heurística que utiliza os percentuais de indução para definir qual a relação entre os conceitos da conjectura (ver seção 3.6.4).

Heurística para ordenação de conjecturas Heurística que utiliza os percentuais de indução e a relação entre os conceitos da conjectura para ordenar todas as conjecturas analisadas (ver seção 3.6.5).

Heurísticas estritamente numéricas São valores que aparecem em vários pontos do sistema e foram escolhidos empiricamente: experimentamos diversos valores e observamos como o sistema se comportava com cada um destes valores. Listamos abaixo cada um destes valores:

Tempo Máximo do módulo Valor em milisegundos que especifica qual deve ser o tempo máximo de execução dos módulos de geração de exemplos, de geração de conjecturas e de refinamento de conjecturas.

Total Exemplos Análise Determina o total de exemplos que devem ser gerados, para cada conceito, no módulo de análise de exemplos.

Total Exemplos Relação Determina o total de exemplos que podem ser gerados até que a relação de uma conjectura seja determinada.

Idade Máxima Tarefa Tempo máximo, em ciclos de execução dos módulos, que uma tarefa pode permanecer na agenda sem que a conjectura que determinou sua inserção seja refinada.

Máximo de Conceitos Número máximo de conceitos que podem ser gerados a cada ciclo.

Máximo de Conjecturas Número máximo de conjecturas (objetos) que podem ser geradas a cada ciclo.

Votos Iniciais Valor da propriedade votos de cada um dos conceitos da base inicial, dos operadores e dos macro-operadores.

Quebra Loop Valores que evitam situações de loop infinito na geração de exemplos. Aparecem dentro de métodos GeraExemplo de operadores ou conceitos primitivos. Na descrição do operador $fecho (ver apêndice C.1), por exemplo, temos dois "Quebra Loops" representados pelas variáveis Tentativas e EvitaLoop.

Mesmo com todo o esforço para separar as heurísticas do restante do sistema, ainda restaram alguns pontos de decisão que devem ser aceitos como heurísticas que não se encaixam em nenhuma das classes anteriores:

- Quais os conceitos que devem fazer parte da base inicial ?
- Quais os operadores e macro-operadores que devem fazer parte do sistema ?
- O gerador de grafos deve gerar grafos triviais ? (Acreditamos que não).
- O gerador de exemplos de subgrafos, que é um conceito primitivo no SCOT, deve gerar subgrafos com apenas um vértice ? (também optamos pelo não, por entender

que grafos triviais apenas atrapalham na descoberta de relações e conceitos interessantes).

Apêndice F

Gerador de Grafos

Procedimento: GeraExemploGrafo
Descrição: Retorna, através de duas listas, um grafo simples não direcionado. A primeira lista é o conjunto de vértices do grafo. Estes vértices são representados como inteiros de 0 até o total de vértices do grafo menos 1. A segunda lista é o conjunto de arestas. Cada aresta é representada por uma lista de dois inteiros, cada inteiro correspondendo a uma das extremidades da aresta.
Entrada: MáximoDeVértices

```
QtdVértices ←SorteiaQtdVértices(MáximoDeVértices)
Probabilidade ←SorteiaRealEntre(0.2,0.7)
Vértices ←Null
Arestas ←Null
Para v1 de 0 até (QtdVértices - 1) faça
    Para v2 de (v1 + 1) até (QtdVértices - 1) faça
        Se SorteiaRealEntre(0,1) <= Probabilidade então
            Aresta ←Aresta + [v1,v2]
    Vértices ←Vértices + v1
Retorna( [Vértices,Arestas] )
```

Procedimento: SorteiaQtdVértices
Descrição: Devolve a quantidade de vértices que o grafo deve conter. Devolve sempre um número maior que 1, para evitar grafos com apenas um vértice.
Entrada: Máximo
Retorna(um valor inteiro entre 2 (inclusive) e Máximo (inclusive))

Bibliografia

[Bac87] John F. Backus. *ACM Turing Award Lectures, the first twenty years*, chapter Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, pages 63–130. Addison-Wesley, 1987.

[CKS+88] P. Cheeseman, J. Kelly, M. Self, J. Stutz, W. Taylor, and D. Freeman. Autoclass: A bayesian classification system. *Proceedings of the Fifth International Conference on Machine Learning*, 1988.

[CN89] P. E. Clark and T. Niblett. The cn2 induction algorithm. *Machine Learning*, 3:261–288, 1989.

[DM86] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1:154–176, 1986.

[DMZ84] C. E. Riese D. Michie, S. H. Muggleton and S. M. Zubrick. Rulemaster: A second generation knowledge engineering facility. *IEEE and AAAI First Conference on Artificial Intelligence Applications*, pages 591–597, 1984.

[DWA91] M. K. Albert D. W. Aha, D. Kibler. Instance-based learning algorithm. *Machine Learning Journal*, 6:37–66, 1991.

[Eps83] S. L. Epstein. Knowledge representation in mathematics: A case study in graph theory. *Ph. D. dissertation, Department of Computer Science, Rutgers University*, 1983.

[Eps88] S. L. Epstein. Learning and discovery: One system's search for mathematical knowledge. *Computational Intelligence*, 4-1, 1988.

[ES91] S. L. Epstein and N. S. Sridharan. Knowledge representation for mathematical discovery: Three experiments in graph theory. *Applied Intelligence*, 1, 1991.

[Fir88] Morris W. Firebaugh. *Artificial Intelligence - A knowledge-based approach.* Boyd and Fraser, 1988.

[Len77] D. B. Lenat. Automated theory formation in mathmatics. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

[Len82] Douglas B. Lenat. The nature of heuristics. *Journal of Artificial Intelligence*, 19, November 1982.

[Len84] Douglas B. Lenat. Why am and eurisko appear to work. *Journal of Artificial Intelligence*, 23, 1984.

[MB88] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. *Proceedings of the Fifth International Workshop on Machine Learning*, pages 339–352, 1988.

[MF92] Stephen Muggleton and Cao Feng. *Inductive Logic Programming*, chapter Efficient Induction of Logic Programs, pages 281–297. Academic Press, 1992.

[Mic86] Ryszard S. Michalsk. *Machine Learning - An Artificial Intelligence Approach, Vol. II*, chapter Understanding the Nature of Learning: Issues and Reserach Directions, pages 3–25. Addison-Wesley, 1986.

[Min81] M. L. Minsky. *A framework for representing knowledge*, pages 95–128. MIT Press, 1981.

[PB93] Michael J. Pazzani and Clifford A. Brunk. Finding accurate frontiers: A knowledge-intensive approach to relational learning. *The National Conference on Artificial Intelligence*, pages 328–334, 1993.

[PH90] G. Pagallo and D. Haussler. Boolean feature discovery in empirical learning. *Machine Learning*, 5:71–100, 1990.

[Pol45] G. Polya. *Mathmatics and Plausible Reasoning.* Princeton University Press, 1945.

[Qui82] J. R. Quinlan. *Introductory Readings in Expert Systems*, chapter Semi-autonomous acquisiton of pattern-based knowledge. Gordon and Breach, 1982.

[Qui90] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Qui93] J. R. Quinlan. *C4.5: Programs for machine Learning.* San Mateo, CA: Morgan Kaufmann, 1993.

[RH84] G. D. Ritchie and F. K. Hanna. Am: A case study in ai methodology. *Journal of Artificial Intelligence*, 23, 1984.

[Sal91] S. L. Salzberg. A nearest hyperrectangle learning method. *Machine Learning*, 6, 251-276 1991.

[Sam63] Arthur L. Samuel. *Some Studies in Machine Learning Using the Game of Checkers*, pages 71–105. McGraw-Hill Book Company, 1963.

[SM86] R. E. Stepp and R. S. Michalski. Conceptual clustering of structured objects: A goal-oriented approach. *Artificial Intelligence*, 28:43–69, 1986.

[Sta87] C. W. Stanfill. Memory-based reasoning applied to english pronunciation. In *Proc. of AAAI-87*, pages 577–581, Seattle, WA, 1987.

[Tad89] Prasad Tadepalli. Laze explanation-based learning: A solution to the intractable theory problem. *Proceedings of International Joint Conference on Artificial Intelligence*, pages 694–700, 1989.

[TC96] H. Theron and I. Cloete. Bexa: A covering algorith for learning propositional concept despcriptions. *Machine Learning Journal*, 24:5–40, 1996.

[TMKC86] R. Keller T. Mitchell and S. Kedar-Cabelli. Explanation based learning: A unifying view. *Machine Learning*, 1:47–80, 1986.