Implementação e Análise de algoritmos para revisão de crença

Heitor Reis Ribeiro

Monografia apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Bacharel em Ciência da Computação

Orientadora: Prof[a]. Dra. Renata Wassermann

São Paulo, novembro de 2015

Resumo

Revisão de crenças é uma área de pesquisa que estuda as mudanças aplicáveis ao estado epistêmico de um agente racional. Este estado pode mudar de acordo com a introdução de novos conhecimentos ou com a remoção destes e, para representar estas mudanças, existem três operações principais: expansão, contração e revisão. Este trabalho compara a eficiência de algoritmos para as construções mais conhecidas da operação de contração através de testes utilizando um novo programa que implementa alguns dos algoritmos presentes no *framework Bcontractor* com a abordagem para revisão de crenças baseada em consistência descrita por Delgrande e Schaub, concluindo com uma comparação entre os resultados, relacionando as fórmulas contidas neles.
Palavras-chave: Revisão de crenças, Lógica, Representação de conhecimento.

Sumário

1 Introdução **1**

2 Fundamentos **3**

2.1 Linguagem Lógica . . . 3

2.1.1 Sintaxe . . . 3

2.1.2 Operador de Consequência . . . 4

2.2 Revisão de Crenças . . . 5

2.2.1 Representação do Estado Epistêmico ∼ Conjuntos de Crença . . . 5

2.2.2 Mudança de Crenças . . . 6

2.2.3 Construções *partial meet* . . . 7

2.3 O conceito de compatibilidade AGM . . . 8

2.4 Representação alternativa: Bases de Crenças . . . 9

2.4.1 Construções *kernel* . . . 10

2.5 Definições para o Algoritmo de Delgrande e Schaub . . . 10

3 Implementação **13**

3.1 Bcontractor: aspectos gerais . . . 13

3.2 Representação no programa desenvolvido . . . 14

3.3 Os algoritmos implementados . . . 16

3.3.1 Suposições iniciais . . . 16

3.3.2 Kernel e Remainder Sets . . . 17

3.3.3 Belief Change Extension . . . 22

4 Testes **25**

4.1 Aspectos Gerais . . . 25

4.2 Resultados: Tempo e Chamadas ao Reasoner . . . 25

4.3 Resultados: Inclusão . . . 27

5 Conclusões e Trabalhos Futuros **29**

Referências Bibliográficas **31**

Lista de Algoritmos

1 Kernel-Black-Box 17
2 Remainder-Black-Box 18
3 Algoritmo de Reiter 19
4 Kernel 20
5 Partial Meet 20
6 Belief Change Extension 23

Capítulo 1

Introdução

Na área de Inteligência Artificial, existe o conceito de agente, que é descrito em [RN03] como algo que pode perceber o seu ambiente e atuar nele. Com isso, considera-se um agente racional um subtipo de agente que age para obter o melhor resultado, mesmo com incertezas sobre o ambiente em que está situado. Tal definição permite a caracterização de vários tipos de agente, sendo que este trabalho irá se utilizar dela referindo-se a um agente baseado em conhecimento.

O componente principal de um agente baseado em conhecimento é a representação deste, o que pode ser obtido com um conjunto logicamente aberto (base de crenças / *Knowledge Base*) ou fechado (conjunto de crenças / *Knowledge Set*) de sentenças. Este conjunto caracteriza o estado epistêmico do agente, algo que pode ser alterado através das interações dele com o ambiente. Tais alterações podem incorporar novas crenças (*Expansão*), removê-las (*Contração*) ou adicioná-las mantendo a consistência lógica do conjunto (*Revisão*). A área de Representação de Conhecimento que estuda estas operações sobre o estado epistêmico de um agente racional é denominada Revisão de Crenças.

Nesta área, uma das formas mais difundidas de se caracterizar tais operações é através de uma representação formal destas, sendo que a forma mais comum de realizar isso é através de postulados de racionalidade, que são o principal resultado do artigo publicado por Alchourrón, Gärdenfors e Makinson (AGM) [AGM85]. Nele são formalmente definidas as consequências formais para as descritas operações epistêmicas através dos chamados *postulados AGM*.

Pelo fato de as *suposições AGM* não se aplicarem a todas as lógicas (especificamente algumas lógicas de descrição), foram feitos trabalhos focados em prover uma caracterização formal alternativa ou verificar a aplicabilidade dos postulados a estes tipos de lógica. Um destes trabalhos originou um *framework* capaz de suportar diferentes construções para as operações citadas de forma a facilitar a comparação entre as diferentes implementações delas: *Bcontractor* [Lun13].

A partir deste *framework*, pôde-se comparar diferentes implementações dos algoritmos mais difundidos de contração e revisão de crenças e, a partir disso, obter uma base de testes para comparação de novas implementações destas operações. Com isso, foi criado um novo programa para realizar uma comparação entre os métodos anteriores e a abordagem de mudança de crenças baseada em consistência[1] apresentada por Delgrande e Schaub [DS03].

Este trabalho se iniciará com uma breve introdução dos fundamentos da teoria de Revisão de Crenças, seguindo a abordagem AGM e explicando seus postulados (Capítulo 2), para depois mostrar a construção e implementação resultante dos algoritmos (capítulo 3) e os resultados dos testes e comparações (Capítulo 4), concluindo com uma análise sobre os resultados obtidos e possíveis extensões da implementação (Capítulo 5).

[1] Tradução livre de *consistency-based approach for belief change*

Capítulo 2

Fundamentos

Para ser iniciada a explicação das implementações e construções sobre revisão de crenças, é necessário explicar os termos e regras que serão utilizados nas próximas seções.

2.1 Linguagem Lógica

Uma linguagem que pode ser usada para a representação de conhecimento é a lógica proposicional clássica (LPC), formada por sentenças que representam as próprias crenças (ou conhecimentos). O conjunto de todas essas sentenças é chamado de *Linguagem* da lógica, que será representado por L.

2.1.1 Sintaxe

A sintaxe da lógica define as fórmulas bem formadas ou sentenças que pertencem a ela [RN03]. Para isso, existem dois tipos de sentenças:

- **Sentenças atômicas** consistem de um único símbolo proposicional representando uma proposição que pode ser *verdadeira* ou *falsa*. Tais símbolos serão representados como letras minúsculas. Além disso, dois símbolos especiais serão separados pois seu significado é fixo: Um indicando uma proposição sempre verdadeira: $\top$; e outro para uma proposição sempre falsa: $\bot$;
- **Sentenças complexas** são construídas a partir das Sentenças atômicas utilizando conectivos lógicos.

Os conectivos lógicos são descritos a seguir em sua ordem de precedência:

- $\neg$: Símbolo representando a negação de uma sentença. Com isso, define-se um *literal* como uma sentença atômica (por exemplo, p, denominando-se um literal positivo), ou uma sentença atômica negada (por exemplo, $\neg q$, denominando-se um literal negativo);
- $\wedge$: Símbolo representando a conjunção de sentenças;
- $\vee$: Símbolo representando a disjunção de sentenças;
- $\rightarrow$: Símbolo representando a implicação de sentenças;
- $\leftrightarrow$: Símbolo representando a equivalência entre sentenças.

Com isso, obtém-se uma definição formal da Linguagem:

> Dado um conjunto de símbolos proposicionais $\mathbb{P} = \{a, b, c, \ldots\}$, a linguagem L será o menor conjunto que satisfaz:

1. $\mathbb{P} \cup \{\top, \bot\} \subseteq L$
2. $\forall \alpha, \beta \ : \ \alpha, \beta \in L \Rightarrow \alpha \wedge \beta$, $\alpha \vee \beta$, $\alpha \rightarrow \beta$, $\alpha \leftrightarrow \beta$, $\neg\alpha \in L$

2.1.2 Operador de Consequência

Uma vez definida a linguagem, é possível definir uma função importante para as futuras definições em conjuntos de crença: o operador de consequência. Para a definição deste, primeiramente são definidos os conceitos de valoração e modelo:

Dado um conjunto de símbolos proposicionais $\mathbb{P} = \{a, b, c, \ldots\}$, uma valoração é uma função do tipo $f \ : \ \mathbb{P} \rightarrow \{0, 1\}$.
A classe de modelos associada à lógica é formada por todas as valorações.

Com esta definição, obtemos o conceito de satisfazibilidade:

Dada uma valoração f, dizemos que f satisfaz uma fórmula atômica p se e somente se $f(p) = 1$, o que será representado por $\models_f p$

Através da definição acima, temos a relação de satisfazibilidade para qualquer tipo de sentença complexa em L:

Dada uma valoração f e $\forall \alpha, \beta \ : \ \alpha, \beta \in L$:

- $\models_f \top$;
- $\not\models_f \bot$;
- $\models_f \neg\alpha$ se e somente se $\not\models_f \alpha$;
- $\models_f \alpha \wedge \beta$ se e somente se $\models_f \alpha$ e $\models_f \beta$;
- $\models_f \alpha \vee \beta$ se e somente se $\models_f \alpha$ ou $\models_f \beta$;
- $\models_f \alpha \rightarrow \beta$ se e somente se $\models_f \neg\alpha$ ou $\models_f \beta$.

Utilizando as relações de satisfazibilidade, percebe-se que cada sentença poderá ter diferentes conjuntos de valorações para os quais a sentença será satisfazível.

Seja $\mathbb{M}$ o conjunto dos modelos em L:

- Dada $\alpha \in L$ uma sentença, se temos um modelo $m \in \mathbb{M}$ fazendo com que $\models_m \alpha$, então diz-se que m é um modelo para α;
- Dado $S = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, um conjunto de sentenças com $\alpha_i \in L \ \forall i$, define-se $G \in \mathbb{M}$ como um modelo para S se e somente se G é modelo para cada $\alpha_i \in S$;
- Uma sentença ou conjunto de sentenças é dita (o) satisfazível em $\mathbb{M}$ se existe ao menos um modelo para ela (e) em $\mathbb{M}$;

- Uma sentença ou conjunto de sentenças é dita (o) válida (o) em $\mathbb{M}$ se todos os modelos em $\mathbb{M}$ são modelos para ela (e).

Finalmente, é dada a definição do operador de consequência para LPC:

Dado $A = \{\alpha_1, \alpha_2, \ldots, \alpha_n \ : \ \alpha_i \in L \ \forall i\}$, F o conjunto de modelos para A, $G(x)$ um operador que retorna os modelos para a sentença x:
$Cn(A) = \{\alpha \in L \ : \ F \subseteq G(\alpha)\}$

Um operador de consequência definido desta maneira satisfaz as seguintes propriedades:

Dados A e B dois conjuntos de sentenças em L:

- *monotonicidade*: se $A \subseteq B \ \rightarrow \ Cn(A) \subseteq Cn(B)$
- *inclusão*: $A \subseteq Cn(A)$
- *idempotência*: $Cn(A) = Cn(Cn(A))$

Uma lógica cujo operador de consequência satisfaz as três propriedades acima é denominada *tarskiana*.

2.2 Revisão de Crenças

Dada a definição de linguagem lógica na seção anterior, é necessário definir como será representado o estado epistêmico do agente para as definições que virão a seguir. Tal representação inicial será baseada na abordagem AGM [AGM85], inclusive para atingir uma maior clareza nas definições das operações e dos postulados que as restringe.

2.2.1 Representação do Estado Epistêmico ∼ Conjuntos de Crença

A abordagem AGM utiliza o conceito de conjuntos de crenças (*belief sets*) para representar exatamente as sentenças que são aceitas pelo agente no estado atual [Gä88]. As sentenças devem ser representadas de acordo com uma lógica específica:

Define-se uma lógica como um par $<L, Cn>$ em que L é uma linguagem lógica e Cn um operador de consequência em L. Para $<L, Cn>$ ser utilizada na abordagem AGM, ela deve satisfazer as *suposições AGM*:

- L deve ser fechada pelos conectivos lógicos convencionais ($\wedge, \vee, \Rightarrow$ e $\neg$);
- (tarskianicidade) $<L, Cn>$ deve ser tarskiana;
- (compacidade) Se $\alpha \in Cn(K)$ então $\exists K' \subseteq K$ finito, tal que $\alpha \in Cn(K')$
- (dedução) $\alpha \in Cn(K \cup \{\beta\})$ se e somente se $\beta \rightarrow \alpha \in Cn(K)$
- (supra-classicalidade) Sendo C_{lpc} o operador de consequência da LPC (definido anteriormente), vale que $C_{lpc}(A) \subseteq Cn(A)$

Dadas as suposições, é utilizada esta lógica $<L, Cn>$ para se definir os conjuntos de crença:

Um conjunto de crenças $K = \{\alpha_1,\ \alpha_2,\ \ldots : \alpha_i \in L\ \forall i\}$ é um conjunto logicamente fechado de sentenças em L, ou seja, $K = Cn(K)$;

Através desta definição é possível classificar uma atitude epistêmica para cada sentença $\alpha \in L$:

- α é aceita: Significando que $\alpha \in K$;
- α é rejeitada: Significando que α é inconsistente com K, se $\neg\alpha \in K$;
- α é indeterminada: Significando que $\alpha \notin K$ e $\neg\alpha \notin K$.

Uma mudança de crença atuando sobre o conjunto K em relação a uma sentença α mudará a atitude epistêmica do agente sobre esta sentença de uma das três listadas acima para outra.

2.2.2 Mudança de Crenças

Expansão:

Para um conjunto de crenças K, a expansão por uma sentença α será denotada por $K + \alpha$. Os postulados para expansão são:

1. $K + \alpha$ é um conjunto de crenças, particularmente: $K + \alpha = Cn(K + \alpha)$;
2. $\alpha \in K + \alpha$;
3. $K \subseteq K + \alpha$;
4. Se $\alpha \in K$ então $K = K + \alpha$;
5. Dado H um conjunto de crenças, se $K \subseteq H$ então $K + \alpha \subseteq H + \alpha$;
6. Para qualquer conjunto de crenças K e $\forall\alpha \in L$, $K + \alpha$ é o menor conjunto de crenças que satisfaz os postulados de 1 a 6.

Através dos postulados descritos acima, é possível obter como resultado o *teorema de representação* para a expansão, pelo fato de poder-se obter uma definição explícita desse processo [Gä88].

Teorema de representação para a Expansão: A função de expansão + satisfaz os postulados de 1 a 6 para a expansão se e somente se $K + \alpha = Cn(K \cup \{\alpha\})$

Contração:

Para um conjunto de crenças K, e contração por uma sentença α será denotada por $K - \alpha$. O resultado da contração tornará α indeterminada em K. Os postulados para contração são:

1. (*fecho*): $K - \alpha = Cn(K - \alpha)$;
2. (*sucesso*): Se $\alpha \notin Cn(\varnothing)$ então $\alpha \notin Cn(K - \alpha)$;
3. (*extensionalidade*): Se $Cn(\alpha) = Cn(\beta)$ então $K - \alpha = K - \beta$;
4. (*inclusão*): $K - \alpha \subseteq K$;
5. (*vacuidade*): Se $\alpha \notin K$ então $K - \alpha \subseteq K$;
6. (*recuperação*): $(K - \alpha) + \alpha = K$.

Revisão:

Para um conjunto de crenças K, a revisão por uma sentença α será denotada por $K * \alpha$. Esta operação irá adicionar α ao conjunto K e manter a consistência deste no processo. Os postulados para revisão, utilizando o exemplo descrito, são:

1. (*fecho*): $K * \alpha = Cn(K * \alpha)$;
2. (*sucesso*): $\alpha \in K * \alpha$;
3. (*inclusão*): $K * \alpha \subseteq K + \alpha$;
4. (*vacuidade*): Se $K + \alpha$ é consistente, então $K * \alpha = K + \alpha$;
5. (*consistência*): Se α é consistente, então $K * \alpha$ é consistente;
6. (*extensionalidade*): $Cn(\alpha) = Cn(\beta) \rightarrow K * \alpha = K * \beta$.

2.2.3 Construções *partial meet*

Uma construção para a Contração foi definida em [AGM85], satisfazendo todos os postulados para esta. Antes desta operação ser definida, são necessários dois conceitos: *Conjunto Resíduo* (*Remainder Set*) e função de seleção.

Seja K um conjunto de crenças e α uma sentença. Então $K \perp \alpha$ é o conjunto resíduo de K em relação a α e $X \in K \perp \alpha$ se e somente se:

- $X \subseteq K$;
- $\alpha \notin X$ e para qualquer X', se $X \subset X'$ então $\alpha \in Cn(X')$.

Com essa definição temos os conjuntos resíduos, representando o conjunto dos subconjuntos de um conjunto de crenças que não implicam uma fórmula. Para ter a construção completa para a contração, é feita outra definição:

Dado um conjunto de crenças K, uma função de seleção para K é uma função γ satisfazendo:

- Se $K \perp \alpha \neq \varnothing$ então $\gamma(K \perp \alpha) \neq \varnothing$ e $\gamma(K \perp \alpha) \subseteq K \perp \alpha$;
- Se $K \perp \alpha = \varnothing$ então $\gamma(K \perp \alpha) = K$.

Com essas definições, é possível explicitar o resultado de [AGM85], referenciado em [Rib10]: "A contração *partial meet* é a intersecção dos subconjuntos maximais de K que não implicam α que o agente acredita de forma mais arraigada". A decisão de quais conjuntos escolher será consequência da operação de seleção utilizada na seguinte construção:

Dada uma Lógica $\langle L, Cn \rangle$, seja $K \subseteq L$ um conjunto de crenças, $\alpha \in L$ uma crença e γ uma função de seleção para K. A contração *partial meet* utilizando a função de seleção γ, simbolizada por $-_\gamma$ é definida por:

$$K -_\gamma \alpha = \bigcap \gamma(K \perp \alpha)$$

2.3 O conceito de compatibilidade AGM

A restrição sobre a lógica utilizada no processo de revisão de crenças já foi estudada antes por Flouris em [FPA04]. Nesse trabalho foram definidos os postulados *generalizados* para contração:

Dada uma lógica $\langle L, Cn\rangle$, um conjunto de proposições A tal que $A \subseteq L$ (ao invés de apenas uma sentença $\alpha \in L$) e um conjunto de crenças $K \subseteq L$:

K-1: Fecho : $Cn(K - A) = K - A$

K-2: Inclusão : $K - A \subseteq K$

K-3: Vacuidade : Se $A \notin Cn(K)$ então $K - A = K$

K-4: Sucesso : Se $A \notin Cn(\varnothing)$ então $A \notin Cn(K - A)$

K-5: Preservação/Extensionalidade [Rib10] : $A \notin Cn(\varnothing) \rightarrow A \notin Cn(K - A)$

K-6: Recuperação : $K \subseteq Cn((K - A) \cup A)$

Aplicados a uma lógica que satisfaz todas as suposições AGM, os postulados acima são equivalentes aos postulados originais para contração.

Flouris utilizou os postulados generalizados em seu trabalho sobre revisão de crenças em ontologias [Flo06], onde definiu o conceito de uma lógica AGM-compatível[1]:

Dado um conjunto de crenças K e um operador de contração $-$ em K, $-$ é dito um *operador de contração AGM-compatível* $\leftrightarrow$ $-$ satisfaz os postulados AGM básicos para contração (K-1) — (K-6)

Uma lógica $\langle L, Cn\rangle$ é dita AGM-compatível se é possível definir ao menos um operador de contração AGM-compatível nela.

Além disso, foi mostrado também que o postulado que gera problemas para a definição de tal operação em lógicas não clássicas é a Recuperação (K-6):

Para qualquer lógica (tarskiana [Rib10]) $\langle L, Cn\rangle$ existe ao menos um operador que satisfaz (K-1) — (K-5).

O que pode ser visto através da definição de um operador simples de contração:

$$K - X = \begin{cases} Cn(K) & \text{se } X \notin Cn(K) \\ Cn(\varnothing) & \text{se } X \subseteq Cn(K) \end{cases}$$

Este operador pode ser definido para qualquer lógica (tarskiana [Rib10]) e satisfaz (K-1) — (K-5) [Flo06]. No mesmo trabalho, Flouris descreveu a estrutura de *conjuntos complementares* que estão associados ao postulado de recuperação:

Seja $\langle L, Cn\rangle$ uma lógica tarskiana e dois conjuntos de crença $K, X \subseteq L$. O conjunto de crenças complementares de X com respeito a K, representado por $X^{-}(K)$ é definido por:

[1] Tradução livre de *AGM-compliant logic*

- Se $Cn(\varnothing) \subset Cn(X) \subseteq Cn(K)$, então $X^-(K) = \{Y \subseteq L \mid Cn(Y) \subset Cn(K),\ Cn(X \cup Y) = Cn(K)\}$
- Senão, $X^-(K) = \{Y \subseteq L \mid Cn(Y) = Cn(K)\}$

Com os conjuntos complementares, então, Flouris definiu o conceito de *decomponibilidade* (*decomposability*):

Dada uma lógica tarskiana $\langle L, Cn\rangle$ e um conjunto de crenças $K \subseteq L$:

- K é *decomponível* $\leftrightarrow X^-(K) \neq \varnothing\ \forall X \subseteq L$;
- $\langle L, Cn\rangle$ é *decomponível* $\leftrightarrow \forall K, X \subseteq L, X^-(K) \neq \varnothing$, ou seja, se $\forall K \subseteq L, K$ é *decomponível*.

Para então finalmente enunciar um dos resultados mais importantes do seu trabalho:

[Flo06] Uma lógica $\langle L, Cn\rangle$ é AGM-compatível $\leftrightarrow \langle L, Cn\rangle$ é *decomponível*.

2.4 Representação alternativa: Bases de Crenças

Bases de crenças são uma forma de representação finita para o estado epistêmico do agente. Ao invés de representar o fecho lógico de um conjunto de sentenças, como no caso de conjuntos de crença, as bases de crença são conjuntos finitos de sentenças que, quando aplicado o operador de consequência, obtém-se a representação completa do conjunto de crenças do agente.

Dada uma lógica $\langle L, Cn\rangle$, uma Base de Crenças é definida como $B = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$: $\alpha_i \in L\ \forall i$; Além disso, $K = Cn(B)$ é o conjunto de crenças associado à base B.

Dada esta definição, existem quatro classificações para uma crença $\alpha \in L$ em relação a essa representação [Rib10]:

- α é rejeitada se não é consistente com B;
- α é aceita explicitamente se $\alpha \in B$;
- α é aceita implicitamente se $\alpha \in Cn(B) \setminus B$;
- α é indeterminada se $\alpha \notin Cn(B)$ e α é consistente com B.

A contração *partial meet* funciona de forma similar para Bases de crença, com a exceção dos postulados que a caracterizam, já que alguns deles (como a recuperação) não podem ser respeitados utilizando a estrutura de bases de crença. Com isso, em [Han14] é descrito um teorema de representação para a contração *partial meet* para Bases de Crença:

Um operador $-$ é um operador de contração *partial meet* para um conjunto A se e somente se ele satisfaz os seguintes quatro postulados:

- **Sucesso**: Se $p \notin Cn(\varnothing)$ então $p \notin Cn(A - p)$;
- **Inclusão**: $A - p \subseteq A$;

- **Relevância**: Se $q \in A$ e $q \notin A - p$, então existe um conjunto A' tal que $A - p \subseteq A' \subseteq A$ e $p \notin Cn(A')$ mas $p \in Cn(A' \cup \{q\})$;
- **Uniformidade**: Se $\forall A' \subseteq A,\ p \in Cn(A') \leftrightarrow q \in Cn(A')$, então $A - p = A - q$.

2.4.1 Construções *kernel*

Uma contração kernel (*kernel contraction*) é uma generalização da contração *partial meet* que utiliza uma definição similar à de *remainder sets* para selecionar os menores subconjuntos de um conjunto de crenças que implicam uma fórmula [Han94].

Dada uma lógica $\langle L, Cn\rangle$, $K \subseteq L$ uma base de crenças e $\alpha \in L$ uma sentença. Define-se $K \perp\!\!\!\perp \alpha$ como um conjunto Kernel e $\forall X \subseteq K \perp\!\!\!\perp \alpha$ como α-kernel de K. Além disso, tem-se $\forall X \subseteq K \perp\!\!\!\perp \alpha$:

- $X \subseteq K$;
- $\alpha \in Cn(X)$;
- $\forall Y \subset X, \alpha \notin Cn(Y)$.

Para uma contração por uma sentença α utilizando os conjuntos kernel, é removido da base de crenças ao menos um elemento de cada α-kernel, o que pode ser comparado às funções de seleção para os conjuntos resíduos. Neste caso, são definidas as funções de incisão [Han94]:

Uma função de incisão (*incision function*) σ para uma base de crenças K é uma função tal que $\forall \alpha$:

- $\sigma(K \perp\!\!\!\perp \alpha) \subseteq \bigcup(K \perp\!\!\!\perp \alpha)$;
- $\varnothing \neq X \in K \perp\!\!\!\perp \alpha \rightarrow X \cap \sigma(K \perp\!\!\!\perp \alpha) \neq \varnothing$.

Ao contrário das funções de seleção, as funções de incisão selecionam conjuntos de sentenças para serem descartados após a operação de contração. Com essas definições, é dada a construção para *kernel contraction*[2]:

Dada uma Lógica $\langle L, Cn\rangle$, seja $K \subseteq L$ uma base de crenças, $\alpha \in L$ uma crença e σ uma função de incisão para K. A contração kernel é definida como:

$$K -_\sigma \alpha = K \setminus \sigma(K \perp\!\!\!\perp \alpha).$$

2.5 Definições para o Algoritmo de Delgrande e Schaub

Em [DS03], foi definida uma diferente abordagem ao processo de revisão de crenças denominado *consistency-based belief change* que segue alguns requerimentos diferentes daqueles descritos até

[2] A definição para a contração kernel segue as descrições encontradas em [Han94], mas também se utiliza das simplificações para o operador $-_\sigma$ encontradas em [Rib10] e [Han14], o que não deve alterar a validade ou corretude da definição.

agora. Considere uma lógica L descrita sobre um alfabeto P de letras representando proposições atômicas e sua linguagem associada L_P. Além disso, dado um alfabeto P, define-se um alfabeto disjunto $P' = \{p' \mid p \in P\}$ e, para $\alpha \in L_P$, α' é o resultado de substituir em α cada proposição atômica $p \in P$ por $p' \in P'$. Esta definição é estendida para conjuntos de fórmulas:

Dado um conjunto de fórmulas K definidas sobre a linguagem L_P, $K' = \{\alpha' \mid \alpha \in K\}$

Um cenário de mudança (*belief change scenario*) em L_P é definido como uma tripla $B = (K, R, C)$ em que K, R e C são conjuntos de crença (conjuntos de fórmulas) descrevendo um cenário em que a base de crenças K será modificada para que as crenças em R estejam contidas no resultado e as crenças em C não. Portanto, para uma contração a exemplo das definidas anteriormente, temos $|C| = 1$ e $R = \varnothing$. Para a definição de uma contração utilizando esta construção, é necessária a definição de *Belief Change Extension*:

Dado um cenário de mudança $B = (K, R, C)$ em L_P, defina EQ como um conjunto maximal de equivalências $EQ \subseteq \{p \equiv p' \mid p \in P\}$ tal que:

$$Cn(K' \cup R \cup EQ) \cap (C \cup \{\bot\}) = \varnothing$$

Com isso, $Cn(K' \cup R \cup EQ) \cap L_P$ é chamada de (*consistent*) *belief change extension* de B

Para melhor entendimento, é citado um exemplo de obtenção de uma *belief change extension*:

Por exemplo, seja um cenário de mudança $B = (\{p \wedge q\}, \{\neg q\}, \varnothing)$. Temos o seguinte:

- $K = \{p \wedge q\}$ e $K' = \{p' \wedge q'\}$
- $R = \{\neg q\}$ e $C = \varnothing$, consequentemente, $C \cup \{\bot\} = \{\bot\}$
- $EQ_1 = \{p \equiv p'\}$, sendo que $Cn((K' \cup R \cup EQ_1) \cap \{\bot\}) = \varnothing$
- $EQ_2 = \{q \equiv q'\}$, sendo que $Cn((K' \cup R \cup EQ_2) \cap \{\bot\}) \neq \varnothing$
- $EQ_3 = \{p \equiv p', q \equiv q'\}$, sendo que $Cn((K' \cup R \cup EQ_3) \cap \{\bot\}) \neq \varnothing$
- $EQ_4 = \varnothing$, sendo que $Cn((K' \cup R \cup EQ_4) \cap \{\bot\}) = \varnothing$

Neste caso, apenas o EQ_1 consegue ser maximal e satisfazer a restrição definida acima, portanto $Cn(K' \cup R \cup EQ_1) \cap L_P$ é a única *belief change extension* de B.

O seguinte teorema enumera algumas propriedades cujo efeito será visível ao realizar as implementações:

Teorema [DS03]: Seja K uma base de crenças e $\alpha \in L_P$. Seja $EQ, EQ* \subseteq \{p \equiv p' \mid p \in P\}$.

1. Se EQ determina uma *belief change extension* de (K, R, C), então para $(p \equiv p') \in \overline{EQ}, \exists \phi \in C \cup \{\bot\} \mid K' \cup R \cup \{\neg\phi\} \cup EQ \vdash p \equiv \neg p'$;
2. Se EQ determina uma dada *belief change extension* de $(K, R, \{\alpha\})$, então $K' \cup R \cup \{\neg\alpha\} \cup EQ \vdash p \equiv \neg p'$ $\forall (p \equiv p') \in \overline{EQ}$;
3. Se E_1 e E_2 são duas *belief change extensions* distintas de $(K, R, \{\alpha\})$, então $E_1 \cup E_2 \vdash \bot$;

4. Se $K \nvdash \neg\alpha$, então $\{p \equiv p' \mid p \in P\}$ determina a única *consistent belief change extension* de $(K, \{\alpha\}, \varnothing)$;
5. Se EQ determina uma *belief change extension* de $(K, \varnothing, \{\alpha \wedge \beta\})$, então EQ determina uma *belief change extension* de $(K, \varnothing, \{\alpha\})$ ou de $(K, \varnothing, \{\beta\})$;
6. Se EQ determina uma *belief change extension* de $(K, \varnothing, \{\alpha\})$, então existe um conjunto de equivalências $EQ*$ que determina uma *belief change extension* de $(K, \varnothing, \{\alpha \wedge \beta\})$ tal que $EQ \subseteq EQ*$;
7. EQ determina uma *belief change extension* E_1 de $(K, \{\alpha\}, \varnothing)$ se e somente se EQ determina uma *belief change extension* E_2 de $(K, \varnothing, \{\neg\alpha\})$. Além disso, $E_1 = Cn(E_2 \cup \{\alpha\})$.

Através das definições acima e das propriedades das *belief change extensions* enumeradas acima, são listadas novas construções para revisão e contração:

Seja K uma base de crenças, α uma fórmula e definindo $(E_i)_{i \in I}$ como a família de todas as *belief change extensions* de $(K, \{\alpha\}, \varnothing)$. Então são definidas:

choice revision: $K \dot{+}_c \alpha = E_i$ A revisão de K por α utilizando uma função de seleção c tal que $c(I) = i$

skeptical revision: $K \dot{+}_c \alpha = \bigcap_{i \in I} E_i$ A revisão de K por α

Seja K uma base de crenças, α uma fórmula e definindo $(E_i)_{i \in I}$ como a família de todas as *belief change extensions* de $(K, \varnothing, \{\alpha\})$. Então são definidas:

choice contraction $K -_c \alpha = E_i$ A contração de K por α utilizando uma função de seleção c tal que $c(I) = i$

skeptical contraction $K -_c \alpha = \bigcap_{i \in I} E_i$ A contração de K por α

Capítulo 3

Implementação

3.1 Bcontractor: aspectos gerais

O *Bcontractor* é um *framework* desenvolvido por Renato Lundberg [Lun13] cujo objetivo é servir como base para implementação e comparação de construções para revisão e contração de crenças em lógicas proposicionais e de descrição. A partir deste *framework*, foi criada uma extensão com o propósito de testar os algoritmos já desenvolvidos e compará-los com uma nova implementação, mais direcionada à facilidade de uso e modificação de fórmulas, porém menos genérica em termos das possíveis construções relativas às operações sobre as bases de crença, pois não permite que construções novas sejam feitas a partir das operações já definidas sobre os conjuntos de crenças, como faz o *Bcontractor*. Por exemplo, o *Bcontractor* permite a definição de diferentes funções de seleção através das interfaces criadas nele, já o programa implementado utiliza apenas as funções de seleção relevantes para os testes e não possui uma interface para a criação de novas funções desse tipo pelo usuário.

Um exemplo da facilidade de uso do programa desenvolvido:

```
1 # Bcontractor
2 PropositionalSentence sentence_1 = new PropositionalSentence(new int[][] {{1, 2,
      -3}, {4, 5}});
3 PropositionalSentence sentence_2 = new PropositionalSentence(new int[][] {{2,
     3}});
4 ISet<PropositionalSentence> sentence_set = ISets.asISet(new ArrayList<
     PropositionalSentence>() {{ add(sentence_1); add(sentence_2); }});
5
6 # Programa desenvolvido
7 sentence_1 = Formula.new('a v b v ¬c ^ d v e')
8 sentence_2 = Formula.new('b v c')
9 sentence_set = FormulaSet[sentence_1, sentence_2]
```

Das classes existentes no *framework*, são listadas as mais importantes para o desenvolvimento do novo programa:

PropositionalSentence: Cuja implementação serviu como base para a classe Formula, especificamente o método de negação de sentenças (*negate*, com sua implementação recursiva). Diferente do programa desenvolvido, esta classe utiliza números inteiros para representar os átomos da sentença (sendo que os negativos correspondem a átomos negados) e, com isso, possui os métodos necessários para ordenar as cláusulas (*sortClauses*), literais (*sortLiterals*) e calcular o máximo destes (*computeMaxVar*).

Ainda assim, é possível obter uma versão mais compreensível de uma sentença ou conjunto destas através do método *toUnparsedRepresentation*, que transforma a sentença seguindo a notação de inteiros em uma *string* representando a mesma fórmula, mas em CNF. Inspirado neste método, foi implementado um método da mesma classe que, dada a entrada de uma fórmula em CNF, retorna uma nova sentença proposicional representando a entrada na notação em inteiros: *fromUnparsedRepresentation*.

MinisatReasoner: Uma abstração para chamadas ao programa *minisat* utilizando-se de arquivos temporários que guardam as sentenças a serem analisadas. A nova implementação não possui a maior parte das funcionalidades relacionadas ao resultado do processamento, exceto a validação final (satisfazível ou não), além de delegar a responsabilidade das chamadas a ele para as classes representando as sentenças e conjuntos destas.

ISet: Representação dos conjuntos/bases de crença (além de outras estruturas como Kernel e Remainder Sets). As características obtidas desta classe também são encontradas na classe *Set* do Ruby, mas restringindo a definição dos métodos utilizados para operações entre conjuntos (união, intersecção, etc.), ao invés de manter uma interface permitindo uma implementação qualquer destes métodos.

Implementação de Algoritmos para as operações: Em que se enquadram classes como *Kernel*, *BlackboxKernelOperator* e *RemainderSet*, juntamente com os algoritmos cuja construção foi descrita em [Lun13] e [Rib10].

3.2 Representação no programa desenvolvido

O programa implementado para a realização dos experimentos teve como base a implementação do pacote *propositional* do *Bcontractor*, mas consiste em uma reimplementação de algumas funcionalidades deste. Ele é uma aplicação feita na linguagem *Ruby*, que permite a criação de fórmulas e conjuntos destas (usados para representar bases de crença), além de incluir os métodos necessários para implementar as construções de contração definidas anteriormente. A escolha da linguagem *Ruby* se deve ao fato de tornar o desenvolvimento e manipulação (principalmente de tipos enumeráveis e strings) mais fácil e rápido. Em contrapartida, ela possui um desempenho menor e uso maior de memória. Esse poder de manipulação pode ser visto no trecho de código abaixo:

```
1 a = Formula.new('a v b v ¬c ^ g v ¬h')
2 p a.string_form
3 "a OR b OR -c AND g OR -h"
4 a.clauses.each { |clause| p clause.literals }
5 ["a", "b", "-c"]
6 ["g", "-h"]
7 b = FormulaSet.new a.clauses.map { |clause| Formula.new clause.literals.join(' ^
      ') }
8 p b.first.string_form
9 "a AND b AND -c"
10 p b.last.string_form
11 "g AND -h"
12 c = Formula.new b.map(&:string_form).join(' v ')
13 p c.string_form
14 "a AND b AND -c OR g AND -h"
```

Estas modificações que podem ser feitas em strings e enumeráveis são rápidas para programar e não precisam de muitas linhas, parâmetros ou adequações de tipos, apenas de conhecimento da linguagem.

A aplicação se utiliza do programa *minisat* [ES15] para a realização dos raciocínios necessários para garantir a consistência das bases de crenças. São listadas a seguir as principais classes do programa:

Formula: Representação de uma fórmula ou sentença construída a partir de uma entrada que deve estar representada na *forma normal conjuntiva*.

Para a criação de objetos do tipo Formula, foi criado um *parser* que torna necessária uma entrada representando uma sentença na forma normal conjuntiva, sendo que as cláusulas serão separadas pelos símbolos de disjunção ($\wedge$), tornando os parênteses desnecessários. Qualquer conjunto de letras minúsculas será interpretado como um átomo da sentença, desde que não conflite com os símbolos utilizados para disjunção ($\vee$ representado pela string 'v'), conjunção

(∧ representado pela string '^') ou negação (¬ representado pela string '¬', com o símbolo U+00AC em Unicode). Após a criação da fórmula, sua representação em string mudará para uma forma independente destes caracteres, utilizando as strings 'OR' para disjunção, 'AND' para conjunção e '-' para a negação. Esta última forma de representação também é aceita como entrada para construção de fórmulas.

```
1    formula = Formula.new "a v b ^ ¬c v bc"
2    p formula.string_form
3    => "a OR b AND -c OR bc"
```

Após a inicialização, os métodos permitem que a fórmula seja negada (*negated*) ou comparada a outra fórmula (==). Utilizando o programa *minisat*, é possível saber se uma formula é satisfazível (*satisfiable?*) ou se é uma tautologia (*tautology?*).

```
1    formula = Formula.new "a v b ^ ¬c"
2    p formula.negated.string_form
3    => "-a OR c AND -b OR c"
```

```
1    formula = Formula.new "a v ¬a"
2    p formula.tautology?
3    => true
```

```
1    formula = Formula.new "¬a v c ^ a ^ ¬c"
2    p formula.satisfiable?
3    => false
```

Além disso, existem os métodos criados para a implementação do algoritmo de Delgrande e Schaub [DS03]: *prime_var* para representar a troca de uma variável por uma versão alternativa (como p para p'), *replace* que generaliza o método anterior para qualquer versão alternativa (inclusive outra variável, existente ou não na fórmula) e *conjunct* que concatena uma cláusula ou fórmula à fórmula original.

```
1    formula = Formula.new "¬a v b"
2    p formula.string_form
3    => "-a OR b"
4    p formula.prime_var("a").string_form
5    => "-a' OR b"
6    p formula.conjunct(formula.prime_var("b")).string_form
7    => "-a OR b AND -a OR b'"
8    p formula.replace("a", "c").string_form
9    => "-c OR b"
```

Clause: Representação de uma cláusula em CNF, ou seja, a disjunção de vários literais.

Uma cláusula pode ser negada (*negate*) e comparada com outra cláusula (==). Para uso no programa *minisat*, a cláusula pode ser convertida ao formato DIMACS CNF (*minisat_format*) [Sat93].

FormulaSet: Representação de um conjunto de fórmulas que foi utilizado para realizar as operações com bases de crença.

Um conjunto de fórmulas também possui os métodos necessários para a implementação do algoritmo de Delgrande e Schaub (*satisfiable?*, *prime* e *replace*), cujo comportamento é similar ao encontrado na classe Formula.

Além disso, possui o método "*entails?*" cujo argumento é uma fórmula, representando a seguinte equivalência:

> Dado K uma base de crenças ou *FormulaSet* e α uma sentença ou *formula*, então são equivalentes:

- $K.entails?(\alpha) = true$
- $\alpha \in Cn(K)$

É importante salientar que cada fórmula adicionada a um conjunto de fórmulas pode alterar o mapeamento dos átomos feito para uso na classe MinisatReasoner. O código abaixo explicita um exemplo de tal situação:

```
1     formula_1 = Formula.new "a v b"
2     formula_2 = Formula.new "c v d"
3     p formula_1.atoms_map
4     => {"a"=>1, "b"=>2}
5     p formula_2.atoms_map
6     => {"c"=>1, "d"=>2}
7     formula_set = FormulaSet[formula_1, formula_2]
8     p formula_1.atoms_map
9     => {"a"=>1, "b"=>2, "c"=>3, "d"=>4}
10    p formula_2.atoms_map
11    => {"a"=>1, "b"=>2, "c"=>3, "d"=>4}
```

Os métodos para realização de revisões e contrações estão encapsulados em módulos separados pelo algoritmo utilizado (Kernel, PartialMeet ou BeliefChange) que são incluídos por esta classe, garantindo a qualquer objeto instanciado acesso às operações implementadas.

MinisatReasoner: Abstração para as chamadas ao minisat utilizado na verificação de satisfazibilidade de Formulas e FormulaSets. O programa a ser chamado precisa do caminho completo do executável (que normalmente está em */usr/bin/minisat*), o que pode ser especificado no arquivo *constants.yml* dentro do diretório *config*.

Um requisito importante para o funcionamento das chamadas ao minisat é o fato de o programa possuir as permissões para leitura e escrita dentro do diretório onde está sendo executado, para criação de arquivos temporários .cnf que serão lidos pelo minisat.

3.3 Os algoritmos implementados

3.3.1 Suposições iniciais

No capítulo 2 foram mostradas as definições para lógicas que são compatíveis com a abordagem AGM para revisão de crenças considerando o uso de conjuntos de crença para a representação do estado epistêmico e construção das operações. Para implementações computacionais, tal abordagem torna-se inviável por vários motivos (espaço necessário, tempo de execução, etc.). A representação utilizada baseia-se em bases de crença, que são mais simples de serem descritas e modificadas, além de possuírem um maior poder de representação do que os conjuntos [Han14].

Além disso, é necessário que a lógica na qual as fórmulas serão definidas seja compatível com as operações realizadas pelo *reasoner*, com as operações realizadas sobre os conjuntos e com a representação escolhida. A lógica proposicional clássica é a escolha mais óbvia para isso, mas também é possível utilizar especializações desta (como a Lógica de Horn).

Para cada operação sobre conjuntos definida nos algoritmos, será mostrado um método (majoritariamente) equivalente definido na classe *Set* da linguagem Ruby [BN15], o que pode gerar uma diferença entre a interpretação dos algoritmos referenciados e a implementação mostrada junto destes devido à notação das operações (& para intersecção, + para união, etc.). Adicionalmente, para funções que recebem como parâmetro inicial um conjunto representando a base de crenças, a implementação em Ruby utiliza métodos para tais construções, encapsulando esse argumento assim como abreviando chamadas a outros métodos deste mesmo objeto, como por exemplo o método *each*, que, quando chamado sem referência a um objeto enumerável, significa que este é o conjunto sendo utilizado na função (Algo que, em outras linguagens, seria referenciado como *self* ou *this*).

3.3.2 Kernel e Remainder Sets

Para obtenção dos α-kernels de uma base de crenças foi utilizado o algoritmo descrito em [Kal06] e referenciado em [Rib10], mas sem a utilização das otimizações relacionadas à base de crenças utilizada como entrada para o algoritmo. Esta abordagem é baseada na estratégia expande-encolhe (*expand-shrink*) que divide a operação em duas partes: dada uma base de crenças K e uma fórmula α, na primeira parte (*expand*), é criado um novo conjunto de crenças K' e são adicionadas a ele fórmulas $\beta \in K$ até que a fórmula α seja consequência de K'. Na segunda parte (*shrink*), são retiradas do conjunto K' fórmulas β e, caso α deixe de ser consequência de $K' \setminus \{\beta\}$, β é reintroduzida a K'.

Algoritmo 1 Kernel-Black-Box

```
1:  function KERNEL-BLACK-BOX(K, α)
2:      ▷ Expand
3:      K' ← ∅
4:      for β ∈ K do
5:          K' ← K' ∪ {β}
6:          if α ∈ Cn(K') then
7:              break
8:          end if
9:      end for
10:     ▷ Shrink
11:     for β ∈ K' do
12:         if α ∈ Cn(K' \ {β}) then
13:             K' ← K' \ {β}
14:         end if
15:     end for
16:     return K'
17: end function
```

Implementação 3.1: ***Implementação de Kernel-Black-Box***

```
1   def kernel_black_box(formula)
2     new_set = FormulaSet.new
3     each do |f|
4       new_set += [f]
5       break if new_set.entails? formula
6     end
7     shrunk = new_set
8     new_set.each do |f|
9       without_formula = shrunk - [f]
10      shrunk = without_formula if without_formula.entails? formula
11    end
12    shrunk
13  end
```

Esta abordagem foi chamada de black-box por sua independência do *reasoner* utilizado, considerando o fato de que ele só precisa possuir a funcionalidade de decidir a satisfazibilidade de um conjunto de fórmulas para execução da função.

Existe uma abordagem similar para encontrar um elemento do *remainder set* utilizado na contração *partial meet* [Res14]. Ela consiste em, a partir de um conjunto de fórmulas (que pode ser vazio), adicionar as fórmulas pertencentes a K até que que o conjunto satisfaça a fórmula a ser contraída α. Similarmente, esta abordagem é chamada de *black-box* por não depender de funcionalidades adicionais do *reasoner* utilizado.

Implementação 3.2: ***Implementação de Remainder-Black-Box***

Algoritmo 2 Remainder-Black-Box

```
1: function Remainder-Black-Box(K, α, X)
2:     K' ← X
3:     for β ∈ K do
4:         if α ∉ Cn(K' ∪ {β}) then
5:             K' ← K' ∪ {β}
6:         end if
7:     end for
8:     return K'
9: end function
```

```
1  def remainder_black_box(formula, x)
2    x_prime = x.clone
3    each do |beta|
4      new_set = x_prime + [beta]
5      x_prime = new_set unless new_set.entails? formula
6    end
7    x_prime
8  end
```

A partir destas funções, foram implementados os algoritmos para *kernel* e *partial meet* utilizando adaptações do algoritmo de Reiter para encontrar esses conjuntos [Rib10]. Este algoritmo encontra os cortes mínimos de uma classe de conjuntos (que, nestes casos, assim como nas implementações encontradas no *Bcontractor*, pode ser considerado como um conjunto de conjuntos). Os cortes mínimos são subconjuntos minimais tais que sua intersecção com quaisquer elementos de B não será vazia, ou seja:

Dado um conjunto de conjuntos (classe de conjuntos) B, um corte minimal X de B é tal que:

- $\forall B' \in B, X \cap B' \neq \varnothing$
- $\nexists X' \subset X \mid \forall B' \in B, X' \cap B' \neq \varnothing$

Algoritmo 3 Algoritmo de Reiter

function REITER(B)
  $Corte \leftarrow \varnothing$
  $fila \leftarrow$ uma fila vazia
  $S \leftarrow \{x\} \mid x \in B$
  for $s \in S$ **do**
    insira $\{s\}$ no começo da $fila$
  end for
  while $fila$ não está vazia **do**
    $Hn \leftarrow$ último elemento da $fila$
    remove o último elemento da $fila$
    if $\exists C \in Corte \mid C \subseteq Hn$ **then**
      continue
    else if $\exists S \in B \mid Hn \cap S = \varnothing$ **then**
      for $s \in S$ **do**
        insira $Hn \cup \{s\}$ na $fila$
      end for
    else
      $Corte \leftarrow Corte \cup \{Hn\}$
    end if
  end while
  return $Corte$
end function

Implementação 3.3: ***Implementação da adaptação do Algoritmo de Reiter***

```
1   def reiter(class_of_sets)
2     cut = Set.new
3     queue = []
4     s = class_of_sets.first
5     s.each { |c| queue.unshift FormulaSet[c] }
6     until queue.empty?
7       hn = queue.pop
8       next if cut.any? { |x| hn.include? x }
9       s = class_of_sets.find { |c| (hn & c).empty? }
10      if !s.nil?
11        s.each { |x| queue.unshift(hn + [x]) }
12      else
13        cut += [hn]
14      end
15    end
16    cut
17  end
```

As adaptações deste algoritmo utilizadas para encontrar todos os α-*kernels* e *remainder sets* foram descritas em [Res14], assim como sua prova de funcionamento. Com isso, são listados os seguintes algoritmos:

Algoritmo 4 Kernel

```
1: function KERNEL(B, α)
2:     Corte ← ∅
3:     fila ← uma fila vazia
4:     S ← Kernel-Black-Box(B, α)
5:     Kernel ← {S}
6:     for s ∈ S do
7:         insira {s} no começo da fila
8:     end for
9:     while fila não está vazia do
10:        Hn ← último elemento da fila
11:        remove o último elemento da fila
12:        if ∃C ∈ Corte | C ⊆ Hn then
13:            continue
14:        else if α ∈ Cn(B \ Hn) then
15:            S ← Kernel-Black-Box(B \ Hn, α)
16:            Kernel ← Kernel ∪ {S}
17:            for s ∈ B \ S do
18:                insira Hn ∪ {s} na fila
19:            end for
20:        else
21:            Corte ← Corte ∪ {Hn}
22:        end if
23:    end while
24:    return Kernel
25: end function
```

Algoritmo 5 Partial Meet

```
1: function PARTIALMEET(B, α)
2:     fila ← uma fila vazia
3:     S ← Remainder-Black-Box(B, α, ∅)
4:     Residuo ← {S}
5:     for s ∈ S do
6:         insira {s} no começo da fila
7:     end for
8:     while fila não está vazia do
9:         Hn ← primeiro elemento da fila
10:        remove o primeiro elemento da fila
11:        if Hn ⊢ α then
12:            continue
13:        end if
14:        S ← Remainder-Black-Box(B, α, Hn)
15:        Residuo ← Residuo ∪ {S}
16:        for s ∈ B \ S do
17:            insira Hn ∪ {s} na fila
18:        end for
19:    end while
20:    return Residuo
21: end function
```

Implementação 3.4: ***Implementação da adaptação do Algoritmo de Reiter para Kernel***

```
1  def kernel(formula)
2    cut = Set[]
3    queue = []
4    s = kernel_black_box(formula)
5    the_kernel = Set[s]
6    s.each { |f| queue.unshift FormulaSet.new [f] }
7    until queue.empty?
8      hn = queue.pop
9      next if cut.any? { |c| hn.include? c }
10     new_set = clone - hn
11     if new_set.entails? formula
12       s = new_set.kernel_black_box formula
13       the_kernel += [s]
14       s.each { |f| queue.unshift(hn + [f]) }
15     else
16       cut += [hn]
17     end
18   end
19   the_kernel
20 end
```

Implementação 3.5: ***Implementação da adaptação do Algoritmo de Reiter para Partial Meet***

```
1  def partial_meet(formula)
2    queue = []
3    s = remainder_black_box(formula, FormulaSet.new)
4    remainder = Set[s]
5    (clone - s).each { |f| queue.unshift FormulaSet.new [f] }
6    until queue.empty?
7      hn = queue.pop
8      next if hn.entails? formula
9      s = remainder_black_box(formula, hn)
10     remainder += [s]
11     (clone - s).each { |f| queue.unshift(hn + [f]) }
12   end
13   remainder
14 end
```

Existe ainda a abordagem de se obter um *Remainder Set* a partir do *Kernel* apresentada em [FFKI06] e referenciada em [Rib10] com a seguinte formalização:

> Seja L uma lógica tarskiana compacta, $B \subseteq L$ um conjunto de crenças e $\alpha \in L$ uma fórmula:
>
> 1. $X \in \{B \setminus Y \mid Y \in B \perp \alpha\}$ se somente se X é um corte mínimo de $B \perp\!\!\!\perp \alpha$
> 2. $X \in B \perp\!\!\!\perp \alpha$ se e somente se X é um corte mínimo de $\{B \setminus Y \mid Y \in B \perp \alpha\}$

Tal abordagem torna possível a obtenção dos *Remainder Sets* nos casos em que o conjunto Kernel é mais rápido ou fácil de ser encontrado (o que foi algo comumente observado nos testes). A seguinte implementação foi feita:

Implementação 3.6: ***Implementação de uma forma de obtenção do*** **Remainer Set** ***a partir do*** **Kernel**

```
1  def remainder_set(formula)
2    set = reiter(duplicate.kernel(formula))
3    remainder = Set.new
4    set.each do |s|
5      copy = duplicate
6      s.each { |x| copy.delete_if { |f| f == x } }
7      remainder << copy
8    end
```

```
9       remainder
10    end
```

3.3.3 Belief Change Extension

Em [DS03] é descrito um algoritmo polinomial que, dado um cenário de mudança, retorna uma *belief change extension* de acordo com a forma de execução decidida para revisão. Para a implementação deste algoritmo, existem algumas mudanças executadas na base de crenças utilizada no cenário resultando em uma fórmula (não um conjunto destas) que representa uma *belief change extension*. Mais detalhadamente, algumas das mudanças são citadas:

> Seja $B = (K, R, C)$ um cenário de mudança em L_P, e $E = \{EQ_1, EQ_2, \ldots EQ_n\}$ um conjunto de conjuntos maximais de equivalências tais que $\forall EQ \in E,\ EQ \subseteq \{p \equiv p' \mid p \in P\}$ tal que se $Cn(K' \cup R \cup EQ) \cap (C \cup \{\bot\}) = \varnothing$, então $Cn(K' \cup R \cup EQ) \cap L_P$ é uma *consistent belief change extension* de B. Para cada conjunto $EQ \in E$, são feitas as substituições:
>
> - para $p \equiv p' \in EQ$, substitua p uniformemente por p' em K';
> - para $p \equiv p' \notin EQ$, substitua $\neg p$ uniformemente por p' em K'.
>
> O resultado de tais substituições em $K \wedge \alpha$ será uma fórmula equivalente a alguma *choice revision* para K e α. A disjunção de todas as sentenças obtidas desta maneira, ou seja, utilizando todos os conjuntos $EQ \in E$, será equivalente a $K \dot{+} \alpha$.

Estas definições supõem uma pequena alteração nos parâmetros de uma *belief change extension*, tratando a base de crenças K como uma fórmula para alcançar uma representação satisfatória do resultado. Para melhorar o entendimento destas mudanças, é dado o exemplo simples de revisão citado em [DS03]:

> Seja $B = (K = \{p \wedge q\}, R = \{\neg p \vee \neg q\}, C = \varnothing)$ um cenário de mudança.
> Temos quatro conjuntos de equivalência para considerar:
>
> 1. $EQ_1 = \{p', q'\}$. Temos que $K' = \{p' \wedge q'\}$ e $K' \cup R \cup EQ_1 = \{p \wedge q, \neg p \vee \neg q\}$, que não é consistente, portanto EQ_1 é descartado;
> 2. $EQ_2 = \{p'\}$. Temos que $K' = \{p' \wedge q\}$ e $K' \cup R \cup EQ_2 = \{p \wedge \neg q, \neg p \vee q\}$, que é consistente, gerando a *belief change extension* $\{p \equiv q\}$;
> 3. $EQ_3 = \{q'\}$. Temos que $K' = \{p \wedge q'\}$ e $K' \cup R \cup EQ_3 = \{\neg p \wedge q, p \vee \neg q\}$, que é consistente, gerando a *belief change extension* $\{p \equiv q\}$;
> 4. $EQ_4 = \varnothing$. Temos que $K' = K = \{p \wedge q\}$ e $K' \cup R \cup EQ_4 = \{\neg p \wedge \neg q, p \vee q\}$, que não é consistente, portanto EQ_4 é descartado.
>
> Ao final, temos duas *consistent belief change extensions* que são unidas por disjunção, gerando o resultado final: $K \dot{+} \{\neg p \vee \neg q\} = \{(p \equiv q) \vee (p \equiv q)\} \equiv \{p \equiv q\}$.

A abordagem descrita acima é a utilizada no algoritmo mostrado a seguir:

Algoritmo 6 Belief Change Extension

function BELIEFCHANGE($B = (K, R, C)$)
  if $K \vdash \bot$ ou $R \vdash \bot$ **then**
    return $\bot$
  end if
  $In \leftarrow Out \leftarrow \varnothing$
  ▷ $Atoms(X)$ retorna um conjunto com os átomos da fórmula ou conjunto de fórmulas X
  $At \leftarrow Atoms(K) \cap (Atoms(R) \cup Atoms(C))$
  ▷ $Prime(S, A)$ retorna S com cada átomo $a \in A$ trocado por a'
  $K' \leftarrow Prime(K, At)$
  for all $\alpha \in At$ **do**
    if $\forall \phi \in C \cup \{\bot\}$ temos $K' \cup R \cup \{p \equiv p' \mid p \in In \cup \{\alpha\}\} \nvdash \phi$ **then**
      $In \leftarrow In \cup \{\alpha\}$
    else
      $Out \leftarrow Out \cup \{\alpha\}$
    end if
  end for
  for all $p \in In$ **do**
    ▷ $Replace(S, At_1, At_2)$ retorna S com cada ocorrência de At_1 substituída por A_2
    $K' \leftarrow Replace(K', p', p)$
  end for
  for all $p \in Out$ **do**
    $K' \leftarrow Replace(K', p', \neg p)$
  end for
  return $((\bigwedge_{\alpha \in K'} \alpha) \wedge (\bigwedge_{\beta \in R} \beta))$
end function

Implementação 3.7: ***Implementação do algoritmo de Belief Change Extension***

```
1 def belief_change
2   return Formula.new('⊥') if @k.contradiction? || @r.contradiction?
3   in_set = Set.new
4   out_set = Set.new
5   at = @k.atoms & (@r.atoms + @c.atoms)
6   k_prime = k.prime(at)
7   at.each do |a|
8     equivalence_set = create_equivalence_set(in_set, k_prime, a)
9     if equivalence_set.satisfiable? && @c.all? { |phi| !equivalence_set.entails
          ?(phi) }
10      in_set += [a]
11    else
12      out_set += [a]
13    end
14  end
15  in_set.each { |p| k_prime = k_prime.replace("#{p}'", "#{p}") }
16  out_set.each { |p| k_prime = k_prime.replace("#{p}'", "#{FormulaParser::
        REPLACED_NEGATION_SYMBOL}#{p}") }
17  @r.empty? ? k_prime.single_formula_format : k_prime.single_formula_format.
        conjunct(@r.single_formula_format)
18 end
```

Existe uma abordagem diferente para *belief change extensions* em relação à contração executada na parte de implementação, que claramente não está representada no algoritmo descrito acima:

> Seja $B = (K, R, C)$ um cenário de mudança em L_P, e $E = \{EQ_1, EQ_2, \ldots EQ_n\}$ um conjunto de conjuntos maximais de equivalências tais que $\forall EQ \in E,\ EQ \subseteq \{p \equiv p' \mid p \in P\}$

tal que se $Cn(K' \cup R \cup EQ) \cap (C \cup \{\bot\}) = \varnothing$, então $Cn(K' \cup R \cup EQ) \cap L_P$ é uma *consistent belief change extension* de B. Para cada conjunto $EQ \in E$, são feitas as substituições:

- para $p \equiv p' \in EQ$, substitua p uniformemente por p' em K';
- para $p \equiv p' \notin EQ$, substitua p' uniformemente por $\bot$ e por $\top$, adicionando a disjunção dos resultados em K'. Ou seja, para cada fórmula $\alpha \in K'$, se α possuir o átomo p', serão criadas duas novas fórmulas e estas serão adicionadas ao K', substituindo a antiga fórmula α

O exemplo simples citado em [DS03] consiste em: Dado $K = \{p \wedge q\}$, $R = \varnothing$ e $C = \{q\}$, a *belief change extension* obtida neste caso será $(p \wedge \bot) \vee (p \wedge \top) \equiv p$.

Capítulo 4

Testes

4.1 Aspectos Gerais

Para as funções de seleção/incisão, foram utilizadas reimplementações dos algoritmos desenvolvidos em [Lun13], que estão descritos abaixo:

[Kernel] *Smallest First Incision*: Seleciona a menor fórmula, utilizando um contador pelo número de cláusulas, de cada elemento do Kernel. Caso haja empate, a fórmula vista primeiro (*first*) é selecionada. Note que esta função não seleciona a menor incisão;

[Kernel] *Full Incision*: Seleciona a união de todos os α-kernels;

[Remainder Set] *Single Selection*: Seleciona um único elemento do *remainder set* que, nesta implementação, será um elemento aleatório do conjunto obtido;

[Remainder Set] *Full Selection*: Seleciona todos os elementos do *remainder set*;

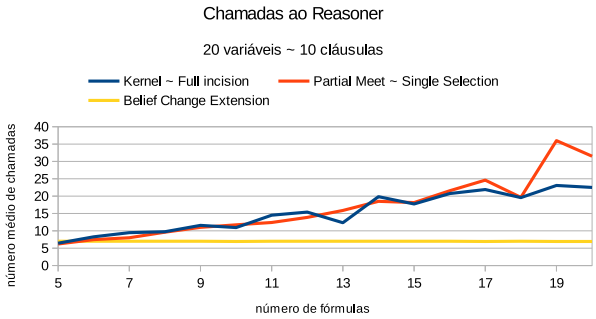
[Remainder Set] *Maxichoice*: Seleciona a intersecção dos maiores elementos do *remainder set*.

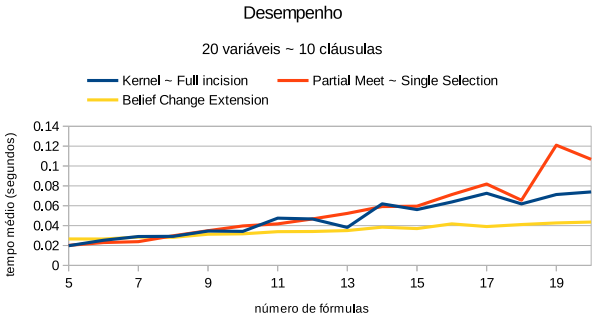
Os testes foram realizados em uma máquina configurada com Ruby versão 2.2.3, sendo executado no sistema operacional Fedora versão 22 e utilizando a versão 2.2.0 do programa minisat, num processador AMD Phenom II X4 965 com acesso a 8GB de memória RAM.

Cada cenário foi gerado aleatoriamente e cada valor apontado nos gráficos foi obtido com a média de 30 execuções de cada algoritmo listado. Cada base de crenças foi composta de fórmulas 3SAT[1] cujo tamanho gradativamente aumentava até ocorrer falta de memória.

Para se obter uma maior precisão, a cada teste foi desativado o *Garbage Collector* do Ruby e foi utilizado o módulo *Benchmark* para medição de tempos de execução para blocos de código.

4.2 Resultados: Tempo e Chamadas ao Reasoner





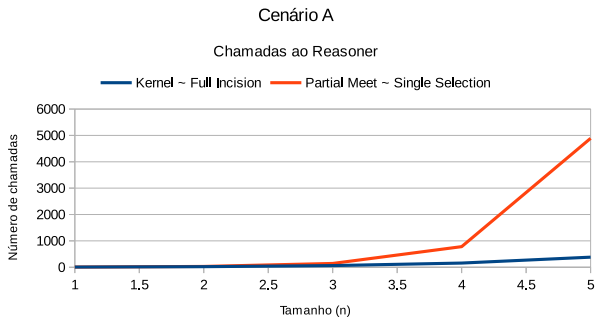
[1] Abordagem apresentada em [Lun13]

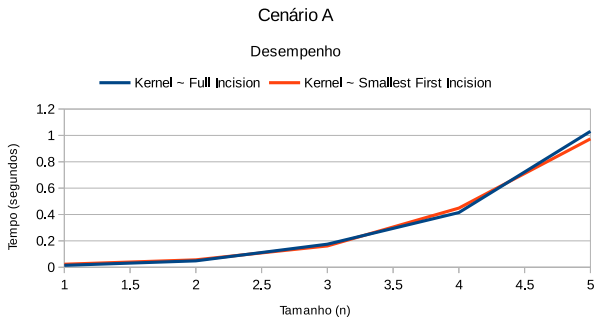
É possível notar a superioridade no desempenho para os casos aleatórios do algoritmo *Kernel*, mas tal vantagem se dá pelo fato de esses cenários estarem, em média, mais propícios ao cálculo do conjunto *Kernel* devido ao gerador aleatório possuir uma chance ligeiramente maior de resultar em uma situação mais adequada à obtenção deste conjunto. Tal resultado também foi visto em [Lun13], cujo algoritmo de geração de cenários é similar ao construído nesta aplicação. Em [Rib10] foi feita esta análise e uma comparação de cenários que podem ser gerados para que algum dos dois algoritmos tenha maior desempenho, criando as seguintes construções:

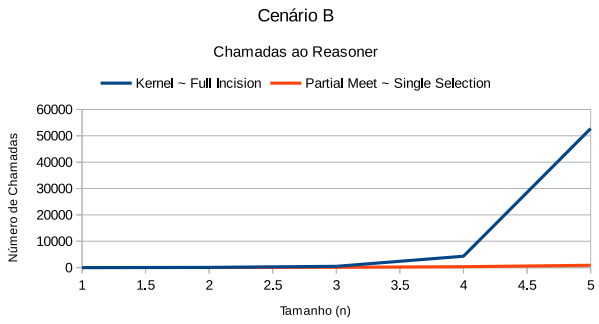
Para uma Base de crenças $B = \{b_1, b_1 \to a, b_2, b_2 \to a, \ldots, b_n, b_n \to a\}$ [Cenário A], tem-se que $B \perp a$ possui um número de elementos proporcional a 2^n e $B \perp\!\!\!\perp a$, proporcional a n.

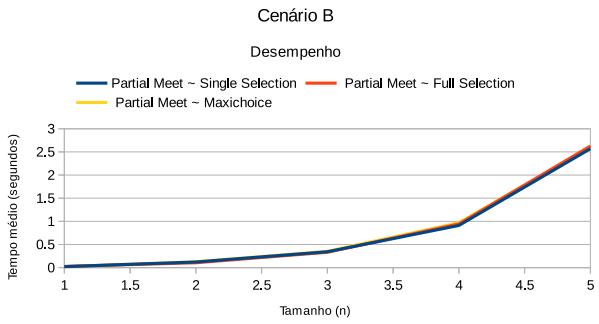
Em contrapartida, para uma base de crenças $B = \{b_1, b'_1, (b_1 \vee b'_1) \to b_2, (b_1 \vee b'_1) \to b'_2, \ldots, (b_n \vee b'_n) \to a\}$ [Cenário B], o contrário ocorre.

Foram implementados casos de teste comparando o tempo médio para estas construções e seus resultados são apresentados abaixo (foram omitidas as referências aos algoritmos de tempo exponencial para manter a escala do gráfico):

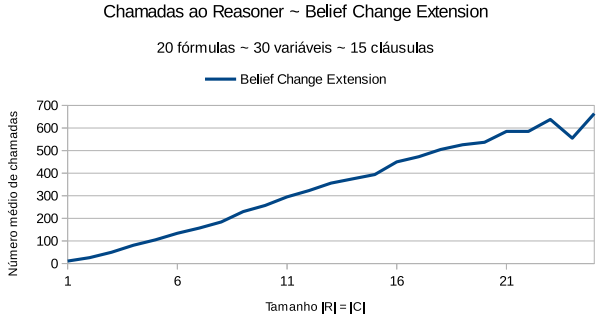


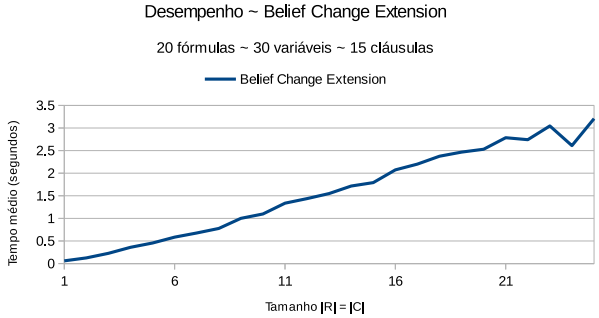






Além disso, foi feita uma análise do desempenho do algoritmo de *belief change extension* utilizando conjuntos de revisão R e contração C tais que $|R| = |C| > 1$. Os resultados indicam o crescimento polinomial do tempo de execução em decorrência do tamanho dos conjuntos e das chamadas ao *reasoner*:





4.3 Resultados: Inclusão

Sobre as diferentes funções de incisão e seleção utilizadas, foi possível obter uma relação entre os resultados considerando uma relação de inclusão. Sejam K_1 e K_2 dois resultados obtidos de qualquer um dos algoritmos, $K_1 \subseteq K_2 \iff \forall \alpha \in K_1, \alpha \in Cn(K_2)$, ou seja, testando se um conjunto resultado (K_2) satisfaz cada fórmula de outro conjunto resultado (K_1). Com isso, foi gerada a seguinte tabela sobre os resultados obtidos a partir de 100 execuções dos algoritmos utilizando o mesmo método de geração, com cláusulas de tamanho 3 (3SAT), 3 cláusulas por fórmula, 20 fórmulas e 30 variáveis (Cada valor corresponde ao número de vezes que o resultado de cada linha (A) esteve contido no resultado de cada coluna (B) nos testes executados):

$A \subseteq B$	Full Incision	Smallest First Incision	Single Selection	Full Meet	Maxichoice Selection	Belief Change Extension
Full Incision	100%	100%	100%	100%	100%	36%
Smallest First Incision	85%	100%	94%	85%	100%	36%
Single Selection	85%	94%	100%	85%	85%	36%
Full Meet	100%	100%	100%	100%	100%	36%
Maxichoice Selection	100%	100%	100%	100%	100%	36%
Belief Change Extension	0%	0%	0%	0%	0%	100%

Capítulo 5

Conclusões e Trabalhos Futuros

Feitas e comparadas as implementações de cada algoritmo, embora limitando-se a apenas alguns dos mais conhecidos na literatura e uma implementação adaptada de um algoritmo de revisão múltipla, vê-se que os resultados obtidos foram condizentes com o que foi visto anteriormente, principalmente em [Lun13]. Além disso, nota-se também a falta de algo assemelhando-se a um banco de dados para casos interessantes, comuns ou problemáticos de revisão ou contração, o que criou a dependência de um gerador aleatório de casos de teste, algo que nitidamente pode enviesar os resultados.

Os algoritmos que se utilizaram dos conjuntos *Kernel* provaram-se mais rápidos na maioria dos casos gerados, o que traz à tona a seguinte pergunta: "Qual algoritmo é o melhor nesta situação?", se for considerado um caso mais específico no qual nenhum dos cenários especiais para *Kernel* ou *Remainder Set* descritos no capítulo anterior possa ser identificado. Um trabalho que se dedicasse à identificação destes casos seria interessante em relação à aplicação destes algoritmos.

O algoritmo de *belief change extension* provou-se rápido e eficaz ao satisfazer o postulado de *Sucesso* nos casos em que foi executado, mas sua construção leva em conta a definição para cenários de revisão em que algumas fórmulas podem não estar presentes no fecho lógico do resultado final, gerando um resultado não esperado para uma contração como descrita em [DS03], principalmente pela diferenciação do comportamento do algoritmo para os dois casos. Sua aplicação para cenários de revisão também deverá ser guiada pelo resultado final, no qual algumas proposições ou fórmulas de presença indesejável podem ser adicionadas. Uma abordagem mais adequada seria uma extensão do algoritmo contendo as construções definidas no programa COBA 2.0 [DHS02].

O programa desenvolvido possui alguns dos problemas esperados pela escolha da linguagem de programação Ruby, principalmente a velocidade de execução e o uso de memória que, em alguns casos de teste, onde o coletor de lixo embutido na linguagem teve que ser interrompido para conseguir uma medida de tempo mais precisa, alcançou mais de 16GB para conjuntos de crença de tamanho pequeno (10 fórmulas e 15 variáveis, por exemplo). Apesar disso, a linguagem permitiu um desenvolvimento mais rápido e intuitivo, garantindo também a possibilidade de futuras extensões como o desenvolvimento de uma interface (talvez web, facilmente alcançável com o *framework* Rails) mais próxima aos usuários não acostumados com a linguagem, assim como foi feito com COBA 2.0.

O programa pode ser encontrado no repositório *git* anexado [1] e possui a licença GNU GPLv3 [Lic07], podendo ser modificado e redistribuído livremente.

[1] http://bitbucket.org/marcheing/belief_change_rb

Referências Bibliográficas

[AGM85] Carlos E Alchourrón, Peter Gärdenfors e David Makinson. On the logic of theory change: Partial meet contraction and revision functions. *The journal of symbolic logic*, 50(02):510–530, 1985. 1, 5, 7

[BN15] James Britt e Neurogami. Ruby 2.2.3 - documentation for set, 2015. http://ruby-doc.org/stdlib-2.2.3/libdoc/set/rdoc/Set.html. 16

[DHS02] James P Delgrande, Aaron Hunter e Torsten Schaub. Coba: A consistency-based belief revision system. Em *Logics in Artificial Intelligence*, páginas 509–512. Springer, 2002. 29

[DS03] James P Delgrande e Torsten Schaub. A consistency-based approach for belief change. *Artificial Intelligence*, 151(1-2):1 – 41, 2003. 1, 10, 11, 15, 22, 24, 29

[ES15] Niklas Eén e Niklas Sörensson. The minisat project, 2015. http://minisat.se/. 14

[FFKI06] Marcelo A Falappa, Eduardo L Fermé e Gabriele Kern-Isberner. On the logic of theory change: Relations between incision and selection functions. Em *ECAI*, volume 6, páginas 402–406, 2006. 21

[Flo06] Giorgos Flouris. On belief change in ontology evolution. *Artificial Intelligence Communications*, 19(4):395–398, 2006. 8, 9

[FPA04] Giorgos Flouris, Dimitris Plexousakis e Grigoris Antoniou. Generalizing the agm postulates: preliminary results and applications. Em *NMR*, páginas 171–179, 2004. 8

[Gä88] Peter Gärdenfors. *Knowledge in flux: Modeling the dynamics of epistemic states*. The MIT press, 1988. 5, 6

[Han94] Sven Ove Hansson. Kernel contraction. *The Journal of Symbolic Logic*, 59(03):845–859, 1994. 10

[Han14] Sven Ove Hansson. Logic of belief revision. Em Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2014 edição, 2014. 9, 10, 16

[Kal06] Aditya Anand Kalyanpur. Debugging and repair of owl ontologies. 2006. 17

[Lic07] GNU General Public License. version 3. *Free Software Foundation. http://www. gnu. org/copyleft/gpl. html. Last checked May*, 13:2010, 2007. 29

[Lun13] Renato Urquiza Lundberg. Análise empírica de algoritmos de revisão sobre bases de crenças. Dissertação (mestrado em ciência da computação), Instituto de Matemática e Estatística, Universidade de São Paulo, 2013. 1, 13, 14, 25, 26, 29

[Res14] Fillipe Manoel Xavier Resina. Revisão de crenças em lógicas de descrição - um plug-in para o protégé. Dissertação (mestrado em ciência da computação), Instituto de Matemática e Estatística, Universidade de São Paulo, 2014. 17, 19

[Rib10] Márcio Moretto Ribeiro. *Revisão de crenças em lógicas de descrição e em outras lógicas não clássicas*. Tese de Doutorado, Universidade de São Paulo, 2010. 7, 8, 9, 10, 14, 17, 18, 21, 26

[RN03] Stuart J. Russell e Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edição, 2003. 1, 3

[Sat93] Satlib. Satisfiability suggested format, 1993. http://www.satlib.org/Benchmarks/SAT/satformat.ps. 15