Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Bacharelado em Ciência da Computação

Implementação de otimizações de performance no GraphPlan

Fernando Fuentes Giroleti e Ramon Fraga Pereira

Orientador: Prof. Dr. Felipe Meneguzzi

Trabalho de Conclusão

Porto Alegre, 10 de julho de 2013

Sumário

Lista de Figuras

Lista de Algoritmos

Lista de Símbolos e Abreviaturas

IA	Inteligência Artificial	7
ICAPS	International Conference on Automated Planning and Scheduling	8
API	Application Programming Interface	8
LPO	Lógica de Primeira Ordem	12
STRIPS	Stanford Research Institute Problem Solver	14
ADL	Action Definition Language	15
PDDL	Planning Domain Definition Language	15
DWR	*Dock Worker Robots*	19
UML	Unified Modeling Language	39

1 Introdução

1.1 Justificativa

Inteligência Artificial (IA) é uma área da Ciência da Computação que tem como objetivo estudar e simular comportamentos de agentes[1] inteligentes, e possui diversas áreas de aplicação. Por exemplo:

- Aprendizagem;
- Automação industrial;
- Banco de dados dedutivos e mineração de dados;
- Escalonamento de tarefas;
- Jogos;
- Otimização e controle de processos;
- Planejamento;
- Sistemas especialistas.

Sistemas de IA [17] funcionam de forma organizada e inteligente, ou seja, são capazes de lidar com problemas complexos buscando aprender e aplicar o conhecimento adquirido reagindo rápido e adequadamente às novas situações. A subárea de IA que será explorada neste trabalho é a de planejamento automatizado em ambientes determinísticos e completamente observáveis.

Planejamento está diretamente ligado ao raciocínio: é uma forma de escolher e organizar ações perante situações, ou seja, escolhas que seguem condições para atingir objetivos. O ser humano possui uma importante habilidade, que é a capacidade de planejar suas ações a fim de levar a um determinado objetivo. Em IA, planejamento também é de suma importância, o qual está relacionado a problemas de logística, agentes inteligentes/deliberativos, planejamento de processos industriais, escalonamento, jogos de estratégia, e etc.

Ao longo dos anos, a área de IA passou por transformações, onde muitas delas abrangem novas técnicas para resolução de determinados tipos de problema. Em 1995, Avrim Blum e Merrick Furst desenvolveram o GraphPlan: um algoritmo de planejamento automático que realiza refinamentos sucessivos em uma estrutura de dados em forma de grafo. O GraphPlan vem influenciando no desenvolvimento de diversos algoritmos de planejamento até hoje. Desde então, muitos pesquisadores estudaram a ideia aplicando novos conceitos sobre ela (otimizações, alterações pontuais, técnicas diversas, etc.) surgindo, assim, novos aprimoramentos a partir do GraphPlan original.

[1] No que diz respeito a IA, um agente é algo capaz de perceber seu ambiente por meio de sensores e agir sobre esse ambiente por intermédio de atuadores.

1.2 Motivação

Um dos principais desafios da área de planejamento é desenvolver algoritmos eficientes otimizando o tempo de geração dos planos. A resolução destes desafios de eficiência é tão importante para a área de IA que existem campeonatos onde algoritmos competem sob vários critérios (ICAPS) a fim de resolver diversos tipos de problemas utilizando formalismos de planejamento. Algoritmos de planejamento podem ser usados na resolução de alguns problemas do mundo real (de logística, por exemplo) e na implementação de mecanismos de raciocínio para agentes autônomos. Neste contexto, o foco deste trabalho será o estudo, aplicação e avaliação de um subconjunto de técnicas de otimização do algoritmo GraphPlan estudadas nos últimos anos.

Problemas de planejamento que utilizam um formalismo de planejamento clássico, como o STRIPS, são comprovadamente de complexidade EXPSPACE[2]. Desta forma, no pior caso o GraphPlan, assim como todos os outros algoritmos de planejamento, terá performance limitada. Porém, em diversos domínios, é possível obter planos com excelente desempenho. Ao longo do tempo surgiram vários trabalhos sobre o GraphPlan buscando otimizá-lo. Implementando um subconjunto destas otimizações, visamos melhorar sua performance na média e para alguns domínios específicos. O objetivo de otimizá-lo é o mesmo que em qualquer algoritmo: redução do tempo de execução e do uso de recursos do computador.

1.3 Organização do Volume

No Capítulo 2 são descritos os objetivos deste trabalho. No Capítulo 3 revisamos a bibliografia referente a planejamento, modelos, representações e linguagens. No Capítulo 4 é apresentada a linguagem de planejamento PDDL utilizada neste trabalho. No Capítulo 5 é explorado o funcionamento do GraphPlan. No Capítulo 6 são descritas possíveis otimizações para o GraphPlan. No Capítulo 7 são apresentadas as implementações do GraphPlan analisadas. No Capítulo 8 é descrito o *Adapter* desenvolvido juntamente com uma API do PDDL4J. No Capítulo 9 são descritas as correções aplicadas na implementação original do planejador utilizado neste trabalho. No Capítulo 10 são descritas otimizações implementadas no planejador. Finalmente, no Capítulo 11, é apresentada uma avaliação do desempenho do planejador implementado neste trabalho frente a outras implementações de planejadores modernos.

[2]EXPSPACE é o conjunto de todos os problemas de decisão resolvíveis por uma Máquina de Turing determinística em espaço $O(2^{p(n)})$, onde $p(n)$ é uma função polinimial de n.

2 Objetivos

Na primeira parte do trabalho de conclusão realizamos as seguintes tarefas:

- Estudamos diversas técnicas de otimizações desenvolvidas para o GraphPlan;
- Avaliamos a viabilidade de implementação de um subconjunto destas otimizações;
- Definimos a API para realização do *parsing* em PDDL: o PDDL4J;
- Definimos a implementação do GraphPlan na qual aplicaremos as otimizações avaliadas: o JavaGP;
- Modelamos um domínio de planejamento em PDDL: o *Dock Worker Robots*.

Para a segunda parte do trabalho de conclusão estabeleceu-se os seguintes objetivos:

- Refatorar o JavaGP para implementação das otimizações de maneira modular;
  - Integrar um parser PDDL no JavaGP.
- As implementações das otimizações foram classificadas em três categorias, de acordo com as suas expectativas e complexidade teórica de implementação: *mínimo*, correspondente ao conjunto mínimo de otimizações que planejamos implementar; *desejável*, correspondente ao conjunto que esperamos conseguir implementar até o final do trabalho; e *máximo*, correspondente às otimizações altamente complexas cuja implementação ocorrerá no caso de termos superestimado o tempo de implementação das categorias anteriores.
  - Mínimo: Memoização, Armazenamento de Mutexes e Hipótese do Mundo Fechado;
  - Desejável: Inferência de Tipos, Predicados Estáticos e In-Place Graph Expansion; e
  - Máximo: Simetria, Foco na Regressão e Invariantes Deduzidas Automaticamente. Essas otimizações não são triviais de serem implementadas, sendo portanto um acréscimo ao trabalho.
- Avaliação de performance utilizando outros planejadores através de *benchmarks*, juntamente com as implementações das otimizações desenvolvidas.

3 Planejamento

A palavra planejamento normalmente implica em elaborar ou propôr uma ou mais ações a fim de tentar realizar um objetivo. Em outras palavras, pode-se dizer que planejamento tem a finalidade de determinar (antecipar) as ações necessárias para atingir um determinado objetivo. Por exemplo, suponha que há uma sala escura e deseja-se iluminá-la. Quais ações podem ser executadas para iluminá-la (objetivo)? Algumas das alternativas seriam acender a luz ou abrir uma janela, por exemplo.

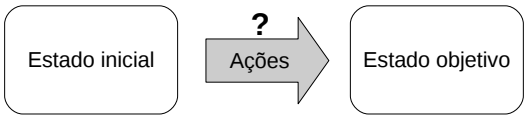

Figura 3.1: Visão geral de Planejamento.

No que diz respeito à IA, planejamento clássico é uma subárea de estudo que busca encontrar uma sequência de ações, chamada de *plano*, que resolva um determinado problema proposto. A complexidade de se construir planos depende de uma variedade de propriedades relacionadas ao ambiente onde opera um agente. Um ambiente pode ser determinístico ou não-determinístico, e pode conter um único agente ou vários agindo concorrentemente. A interação entre o agente e o ambiente ocorre a partir de percepções (que o ambiente proporciona ao agente) e ações (que o agente aplica ao ambiente). Essas ações podem ser instantâneas ou com um determinado tempo de duração, e as percepções podem ser parcialmente ou totalmente observáveis. Esta variedade de propriedades implica em uma alta complexidade para se construir um plano em uma grande variedade de algoritmos de planejamento.

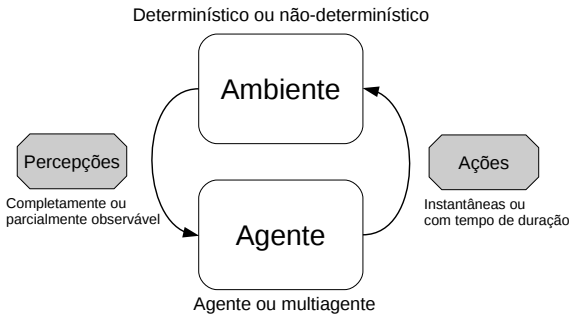

Figura 3.2: Visão geral da complexidade para se construir planos.

Como citado anteriormente, existem diversas formas de planejamento. Para elaboração deste trabalho será explorado o planejamento automatizado.

3.1 Planejamento Automatizado

Planejamento automatizado [10] é a análise de como um sistema computacional pode elaborar uma sequência de ações até chegar ao seu objetivo. Desta forma, dada uma situação inicial, um conjunto de ações e uma situação final desejada, o objetivo do planejamento é determinar uma sequência de ações que soluciona um determinado problema.

Essa sequência de ações que soluciona um determinado problema de planejamento é chamada de *plano* e o algoritmo que o constrói é chamado de *planejador*. O *planejador* tem como entrada um problema de planejamento (em uma descrição formal) e, utilizando essa entrada, busca solucionar o problema baseando-se em algoritmos de busca e heurísticas.

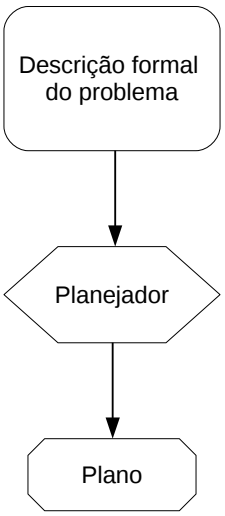

Figura 3.3: Visão geral de Planejamento automatizado.

As principais questões que envolvem planejamento são:

- Como representar o problema?
- Como representar estados e ações?
- Como realizar eficientemente a busca por uma solução?
- Como criar ou usar heurísticas?

3.1.1 Modelo de Planejamento Independente do Domínio

O modelo de planejamento clássico independente do domínio [10] prevê o conhecimento completo sobre o ambiente, o qual deve ser determinístico e completamente observável. Com isso, esse modelo é caracterizado pela descrição de um domínio finito e um conjunto de ações aplicáveis a este domínio. Esse modelo prevê que todas as alterações no domínio são consequências diretas das ações de um agente, sendo estas consequências determinísticas.

3.1.2 Representação de Planejamento Automatizado

Para representar um problema de planejamento deve haver uma forma de formalizá-lo, e uma das principais formas é utilizando a lógica matemática. Baseado no exemplo da sala da seção anterior, veremos uma possível formalização utilizando a Lógica Proposicional[1], identificando os três principais elementos que compõe um problema de planejamento. Supõe-se que há uma sala escura e deseja-se iluminá-la:

- **Estado inicial:** *Sala escura.*
- **Estado objetivo:** *Sala iluminada.*
- **Ação:** *Acender a luz.*
  - **Pré-condição:** *Sala escura.*
  - **Efeito:** $\neg$*Sala escura* $\wedge$ *Sala iluminada.*

O primeiro elemento é o estado inicial, que é representado pela proposição *Sala escura.* O segundo elemento é o estado objetivo, que é a proposição *Sala iluminada.* O terceiro elemento é o conjunto de ações, que neste exemplo é *Acender a luz.* Essas ações possuem pré-condições e efeitos. As pré-condições devem ser verdadeiras para que a ação possa ser executada. Os efeitos são as consequências dessas ações.

A representação utilizando Lógica Proposicional tem expressividade limitada. Para resolver esta limitação, as principais linguagens de planejamento utilizam como base um fragmento finito de Lógica de Primeira Ordem (LPO). Com isso, é possível expressar uma modelagem mais complexa de domínios.

A Lógica de Primeira Ordem baseia-se nos seguintes elementos [10]:

- Um ***termo*** t pode ser definido como ***constante***, ***função*** ou ***variável.***
  - Uma ***constante***, por exemplo, é denotada com símbolos *"a, b, c, ..."*, representando objetos conhecidos (definidos).
  - Uma ***variável***, por exemplo, é denotada com símbolos *x, y, z* e etc, representando objetos não conhecidos (indefinidos).
  - Uma ***função*** é denotada com um símbolo de função f aplicado a n termos, ou seja, $f(t_0, \ldots, t_n)$. Em planejamento, para manter o domínio finito, geralmente limitam-se os termos de uma função.
- Um ***predicado*** é denotado com um símbolo de predicado p aplicado a n termos, ou seja, $p(t_0, \ldots, t_n)$.
- Os ***conetivos*** (operadores lógicos) são representados pelos símbolos $\rightarrow, \leftrightarrow, \wedge$ e $\vee$, utilizados para conectar duas ou mais sentenças.
- Os ***quantificadores*** são representados pelos símbolos $\exists$ (existencial) e $\forall$ (universal), utilizados para expressar o "Existe ..." e "Para todo ..." respectivamente. Em domínios de planejamento, normalmente limitam-se o uso de quantificadores.

[1] Lógica Proposicional é um formalismo representado por símbolos proposicionais (verdadeiro ou falso) que permitem o uso de conectivos lógicos.

O exemplo utilizado na Lógica Proposicional será modificado a fim de dar mais expressividade ao mesmo (Lógica de Primeira Ordem).

Sala1 está escura.
Sala2 está iluminada.
Toda sala escura não está iluminada.
Toda sala iluminada não está escura.

Esse conjunto de frases em Lógica de Primeira Ordem pode ser representado da seguinte forma:

Escura(*Sala1*).
Iluminada(*Sala2*).
$\forall x$ (Escura(x) $\wedge$ $\neg$Iluminada(x)).
$\forall x$ (Iluminada(x) $\wedge$ $\neg$Escura(x)).

As constantes *Sala1* e *Sala2* representam uma sala. Escura(x) e Iluminada(x) são predicados que representam uma relação ou propriedade conforme o termo x. As frases que contêm o quantificador $\forall$ significam que, para toda sala x que está escura, a mesma não está iluminada; e respectivamente, para toda sala x que está iluminada a mesma não está escura.

3.1.2.1 Descrição Formal de um Problema de Planejamento

Para se entender planejamento automatizado, precisamos primeiro definir a terminologia a ser utilizada, em particular os termos-chave abaixo: [15]:

- **Estados:** é um conjunto finito de *átomos ground*[2] os quais possuem valores lógicos (verdadeiro ou falso) a partir de uma interpretação.
- **Operadores:** podem ser definidos por uma 4-tupla (*name(O), pre(O), add(O), del(O)*), onde *name(O)* é a descrição do operador e *pre(O)* define-se como as pré-condições do operador. Os operadores *add(O)* e *del(O)* também podem ser denominados como efeitos, onde *add-list* e *delete-list* são respectivamente conjuntos de átomos resultantes da operação. Um operador também pode ser uma *3*-tupla (*name(O), pre(O), effects(O)*), onde *effects(O)* é a consequência da operação (*add-list* e *delete-list*). Esses *Operadores* instanciados são denominados *ações*.
- **Domínio:** em planejamento, é definido a partir da enumeração dos predicados possíveis e do conjunto de *ações* em um determinado ambiente.
- **Problema:** é definido pela descrição do estado inicial e do estado objetivo associado a um determinado domínio.
- **Plano:** é uma sequência ordenada de *Operadores* instanciados (ações) gerada a partir de um *Domínio* e um *Problema* que, quando executados a partir do estado inicial, resulta em um estado contendo o estado objetivo.

[2] Um átomo *ground* é uma fórmula que não pode ser dividida em sub-fórmulas, e não possui variáveis.

3.1.3 Linguagens de Planejamento

Em 1971, Fikes e Nilsson [5] desenvolveram um robô chamado Shakey capaz de planejar suas tarefas. Para realizar o planejamento dessas tarefas, foram desenvolvidos uma linguagem e um planejador, chamado de STRIPS (Stanford Research Institute Problem Solver). Essa implementação tinha o objetivo de determinar as ações e os movimentos do robô.

O STRIPS ganhou mais repercussão pela sua linguagem do que pelo planejador. Com isso, teve muita influência na área de planejamento devido à sintaxe e semântica de sua linguagem, a qual baseia-se em Lógica de Primeira Ordem.

A representação de um problema de planejamento em STRIPS possui as seguintes características:

- Um conjunto de predicados que definem os estados inicial e final do problema; e
- Um conjunto de ações, cada uma com pré-condições e efeitos. Uma ação contém uma identificação e representa um predicado que pode ser aplicado a n-termos (variáveis ou constantes). Essa ação pode ser executada desde que satisfaça a pré-condição definida para a mesma, onde o resultado dessa execução são denominados como efeitos. Esses efeitos possuem a característica de eliminar e/ou adicionar predicados a um determinado estado do problema;

Para mostrar a representação da linguagem STRIPS, utilizamos o exemplo do Jantar Surpresa (por Daniel Weld [18]). Supõe-se que um indivíduo deseja preparar uma janta surpresa para sua esposa, que está dormindo. O marido pode realizar quatro ações:

- *cozinhar*: o marido precisa estar com as mãos limpas, e resulta no *jantar*.
  - pré-condição: $maosLimpas$
  - efeito: $jantar$
- *embrulhar*: a casa precisa estar em silêncio antes de embalar o *presente* para a esposa, e resulta no *presente*.
  - pré-condição: $silencio$
  - efeito: $presente$
- *carregarLixo*: o marido leva o *lixo* pra fora da casa. Resulta em casa limpa, porém mãos sujas.
  - pré-condição: nenhuma
  - efeito: $\neg lixo \wedge \neg maosLimpas$
- *reciclarLixo*: elimina o *lixo* dentro de casa através de uma máquina de reciclagem. Resulta na casa limpa, porém com barulho (que acarreta no despertar da esposa).
  - pré-condição: nenhuma
  - efeito: $\neg lixo \wedge \neg silencio$

```
Initial Conditions: (and (lixo) (maosLimpas) (silencio))
Goal: (and (janta) (presente) (not (lixo)))

Actions:
    cozinhar      : precondition (maosLimpas)
                  : effect (jantar)

    embrulhar     : precondition (silencio)
                  : effect (presente))

    carregarLixo  : precondition
                  : effect (and (not (lixo)) (not (maosLimpas)))

    reciclarLixo  : precondition
                  : effect (and (not (lixo)) (not (silencio)))
```

A linguagem ADL (Action Definition Language) [10] foi criada em 1987 devido ao fato do STRIPS não ser suficiente para representar alguns problemas de planejamento. Esta linguagem pode ser vista como uma extensão do STRIPS, pois suporta tipos, efeitos, e permite negações explicitas e também desigualdade de termos nas pré-condições. Abaixo segue a representação do ADL referente ao exemplo proposicional do Jantar Surpresa.

```
Init(lixo AND maosLimpas AND silencio)
Goal(janta AND presente AND NOT(lixo))

Action(
       Cozinhar,
              PRECOND: maosLimpas
              EFFECT : jantar
)

Action(
       Embrulhar,
              PRECOND: silencio
              EFFECT : presente
)

Action(
       carregarLixo,
              PRECOND:
              EFFECT : NOT(lixo) AND NOT(maosLimpas)
)

Action(
       reciclarLixo,
              PRECOND:
              EFFECT : NOT(lixo) AND NOT(silencio)
)
```

O PDDL (Planning Domain Definition Language) surgiu com o intuito de padronizar a formalização de algoritmos de planejamento substituindo STRIPS, ADL e outras linguagens. O PDDL será a linguagem de planejamento utilizada para elaboração deste trabalho e será descrita

em detalhes no próximo capítulo.

4 Linguagem de Planejamento PDDL

Conforme visto no capítulo anterior, o PDDL [4] é a linguagem padrão utilizada em competições de planejamento, tendo este surgido a partir da consolidação das várias extensões desenvolvidas sobre a linguagem STRIPS. Estas extensões incluem, entre outras, definição de tipos, operadores com quantificação, desigualdade de termos (nas pré-condições e efeitos) e pré-condições podem ser negadas.

O PDDL possui sua definição dividida em duas partes: domínio e problema, onde cada parte é representada por um arquivo de descrição. A definição do domínio em PDDL descreve o conhecimento a respeito do problema a ser resolvido traduzindo essa definição em tipos de objeto, predicados, funções e ações. Já a definição do problema descreve um problema de planejamento a ser resolvido, composta pelos estados inicial e final.

4.1 Domínio em PDDL

A definição do domínio descreve o conhecimento a respeito do problema, possuindo as seguintes declarações:

- Nome do domínio (**domain**): é um identificador do domínio para ser usado na validação de problemas;
- Requisitos (**requirements**): informa ao interpretador que características de PDDL serão aceitas na descrição do domínio, ou seja, que tipos de domínio e qual a expressividade deste. Por exemplo: utilização de quantificadores, negação de pré-condições, tipagem de termos e etc;
- Tipos (**types**): se os objetos do domínio possuírem um tipo, informa-se quais são os tipos disponíveis;
- Constantes (**constants**): opcional, objetos comuns a todos os problemas neste domínio;
- Predicados (**predicates**): lista de predicados. Cada um possui um nome e uma tupla de argumentos representando a aridade do predicado, e se for o caso, os tipos dos termos;
- Funções (**functions**): lista de funções. Possui um nome e uma tupla de argumentos que representam um valor numérico; e
- Ações (**action(s)**): lista de ações (possivelmente tipadas). Possui um nome, uma lista de parâmetros, pré-condições e efeitos.

4.1.1 Exemplo de Domínio em PDDL

Abaixo segue representação do domínio em PDDL referente ao exemplo proposicional do Jantar Surpresa.

```
(define
    (domain jantarSurpresa)
    (:requirements :strips)
    (:predicates
              (lixo)
              (maosLimpas)
              (silencio)
              (presente)
              (jantar))

       (:action cozinhar
              :precondition(maosLimpas)
              :effect(jantar))

       (:action embrulhar
              :precondition(silencio)
              :effect(presente)
       )

       (:action carregarLixo
              :precondition(lixo)
              :effect(and (not(lixo)) (not(maosLimpas)))
       )

       (:action reciclarLixo
              :precondition(lixo)
              :effect(and (not(lixo)) (not(silencio)))
       )
)
```

4.2 Problema em PDDL

A definição do problema representa uma ou mais instâncias do problema a ser resolvido, possuindo as seguintes declarações:

- Nome do problema (**problem**): é um identificador do problema;
- Nome do domínio (**domain**): nome do domínio que deve coincidir com o nome declarado no domínio;
- Objetos (**objects**): lista de termos que serão usadas para instanciar os elementos do domínio, tais como predicados e ações;
- Estado inicial (**init**): estado inicial do problema, especificando valores iniciais para predicados e funções declarados no domínio;
- Objetivo (**goal**): estado final objetivo do problema, especificando valores finais para predicados e funções declarados no domínio; e

- Métrica(s) (**metric**) (opcional): é utilizada para especificar quais funções do domínio deve-se tentar minimizar ou maximizar.

4.2.1 Exemplo de Problema em PDDL

Abaixo segue um problema descrito em PDDL para o exemplo proposicional do Jantar Surpresa.

```
(define
    (problem jantar)
    (:domain jantarSurpresa)
    (:init ( and (lixo) (maosLimpas) (silencio)))
    (:goal ( and (jantar) (presente) (not (lixo))))
)
```

4.3 Domínio a Ser Utilizado Para Testes do GraphPlan: *Dock Worker Robots*

O domínio que utilizaremos como modelo de teste ao longo do trabalho será o *Dock Worker Robots* (DWR) [10]. Este domínio representa um porto com áreas que correspondem a docas, guindastes, áreas de armazenamento para containers, e áreas de estacionamento para carros de transporte. Este porto é servido por alguns carros de transporte e guindastes que movem os containers com a finalidade de carregar e descarregar navios. Pelo fato de ser um domínio expandível (é possível ser modelado e testado em variados números de determinados objetos) optou-se por seu uso para experimentação neste trabalho.

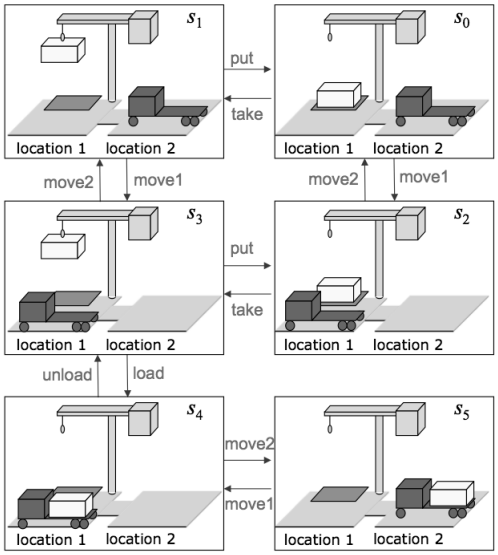

Figura 4.1: Visão geral do domínio DWR.

Abaixo segue uma descrição dos objetos existentes no domínio DWR, juntamente com suas restrições. A formalização completa deste domínio em PDDL está incluída no Apêndice A deste volume.

- Docas: são áreas/locais de armazenamento.
- Carros Robôs: são objetos que podem se mover para docas/localizações adjacentes. Os carros robôs podem carregar no máximo um container por vez.
- Containers: são objetos que podem ser empilhados, carregados sobre os carros robôs, ou carregados pelos guindastes.
- Guindaste: cada um pertence a uma única doca/localização. Podem carregar um container de uma pilha para um carro robô e vice-versa.
- Pilhas: são áreas fixas onde os containers são empilhados.

5 GraphPlan

Em 1995, Blum e Furst [1] propuseram um novo algoritmo de planejamento chamado GraphPlan. Nesta abordagem, o algoritmo cria uma estrutura compacta em forma de grafo (dirigido) estruturado em níveis que armazena informações referentes ao problema de planejamento a ser solucionado. Essa estrutura, chamada de Grafo de Planejamento, é construída progressivamente buscando a cada passo eliminar inconsistências relacionadas ao problema, a fim de reduzir o espaço de estados considerado na busca por um plano. A grande vantagem do GraphPlan é que quanto mais informações o grafo contiver mais rápido será a busca por um plano. Como o Grafo de Planejamento pode ser construído em tempo polinomial e a busca por planos é exponencial, o investimento de tempo no refinamento do Grafo de Planejamento resulta em ganhos significativos de desempenho [1].

A partir do Grafo de Planejamento, o GraphPlan orienta sua busca por um plano, combinando características de dois algoritmos de planejamento [14]: Planejamento de Ordem Parcial[1] e Planejamento de Ordem Total[2]. Essa representação do algoritmo chamou a atenção da comunidade de IA por ser um algoritmo simples e robusto, retornando sempre a sequência de ações mais curta possível para se gerar um plano, ou provando que não há um plano válido. A eficiência deste algoritmo foi comprovada em vários domínios, apresentando uma velocidade de planejamento melhor que seus antecessores Prodigy [12] e UCPOP [16].

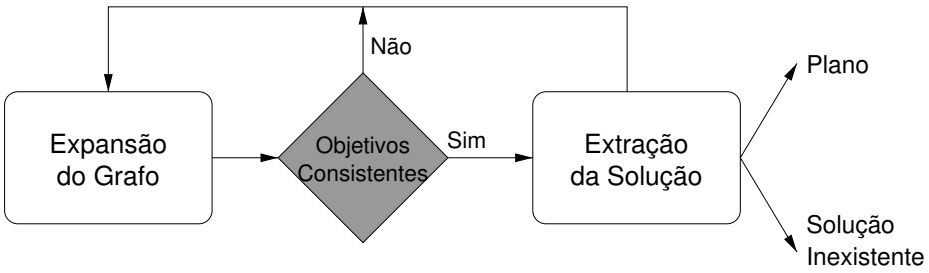

Figura 5.1: Visão geral do GraphPlan [13].

[1] *Planejamento de Ordem Parcial*: representa planos onde somente alguns passos são ordenados;
[2] *Planejamento de Ordem Total*: representa planos onde cada passo do plano é ordenado com relação a todos os passos restantes;

O algoritmo do GraphPlan é divido em duas fases:

- **Expansão do Grafo:** extende o grafo progressivamente (*forward construction*) até atingir uma condição necessária (mas não suficiente) para um plano existir; e
- **Extração da Solução:** realiza uma busca regressiva (*backward path extraction*) no grafo a procura de uma solução. Caso não encontre, o algoritmo volta a expandir o grafo.

Nas próximas seções, serão descritos detalhes referentes às duas fases do GraphPlan.

5.1 Expansão do Grafo

O Grafo de Planejamento gerado pelo GraphPlan é estruturado em níveis, onde cada nível possui nodos de um determinado tipo. Estes nodos são de dois tipos: os que representam proposições e os que representam ações. Um nodo proposição representa uma parte do estado e um nodo ação representa modificações possíveis nas proposições. Os níveis do grafo estão divididos em níveis de proposições e de ações alternando de tipo a cada nível. Níveis pares são proposições, enquanto ímpares são as ações, sendo que o nível *0* é sempre composto pelas proposições que representam o estado inicial do problema[3].

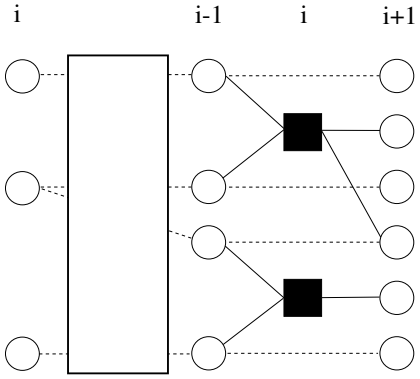

Figure 5.2: Grafo de Planejamento. Círculos brancos representam proposições e quadrados pretos ações.

A ligação entre os níveis é representada por dois tipos de arestas: arestas de pré-condição e arestas de efeito. Arestas de pré-condição efetuam a ligação entre os níveis de proposição e ação, e arestas de efeito fazem a ligação entre os níveis de ação e proposição. A cada novo passo no processo de expansão do grafo, dois novos níveis são adicionados:

1. um nível de ações, que representa todas as ações que poderiam ser tomadas (com pré-condições satisfeitas) de acordo com os estados possíveis no nível anterior; e
2. um nível de proposições que é gerado em consequência dos efeitos das ações;

[3]Esta abordagem pode mudar dependendo da implementação. Algumas implementações consideram um nível de ações e seus efeitos como um único nível.

Todas as ações em um dado nível do grafo são, a princípio, executáveis em paralelo. Não obstante, muitas dessas podem ser mutuamente exclusivas (também chamadas de *mutex*). Portanto, deve-se ter cuidado com essas relações de exclusão mútua após a geração dos níveis com o intuito de gerar um grafo consistente no que diz respeito à extração de um plano que atinja o objetivo especificado.

Algoritmo 1 Expansão do Grafo [10] .

Expand($\langle P_0, A_1, \mu A_1, P_1, \mu P1, \ldots, A_{i-1}, \mu A_{i-1}, P_{i-1}, \mu P_{i-1}\rangle$)

begin

$A_i \leftarrow \{a \in A \mid \text{precond}(a) \subseteq P_{i-1} \text{ and } \text{precond}^2(a) \cap \mu P_{i-1} = \emptyset\}$

$P_i \leftarrow \{p \mid \exists\ a \in A_i : p \in \text{effects}^+(a)\}$

$\mu A_i \leftarrow \{(a, b) \in A_i^2,\ a \neq b \mid \text{effects}^-(a) \cap [\text{precond}(b) \cup \text{effects}+(b)] \neq \emptyset$
or $\text{effects}^-(b) \cap [\text{precond}(a) \cup \text{effects}^+(a)] \neq \emptyset$
or $\exists\ (p, q) \in \mu P_{i-1} : p \in \text{precond}(a),\ q \in \text{precond}(b)\}$

$\mu P_i \leftarrow \{(p, q) \in P_i^2,\ p \neq q \mid \forall\ a, b \in A_i,\ a \neq b :$
$p \in \text{effects}^+(a),\ q \in \text{effects}^+(b) \implies (a, b) \in \mu A_i\}$

foreach $a \in A_i$ **do:**

link a with precondition arcs to precond(a) in P_{i-1}
positive arcs to $\text{effects}^+(a)$ and negative arcs to $\text{effects}^-(a)$ in P_i

return($\langle P_0, A_1, \mu A_1, \ldots, P_{i-1}, \mu P_{i-1}, A_i, \mu A_i, P_i, \mu P_i\rangle$)

end

O primeiro passo é identificar A_i e P_i. A_i são todas as ações cujas precondições estejam incluídas no nível de proposições P_{i-1}, onde uma análise par a par das pré-condições das ações presentes em A_i é feita para verificar se existem os *mutexes* contidos em μP_{i-1}. P_i representa todas as proposições que são efeitos das ações em A_i. Os *mutexes* de ações são adicionados em μA_i e os *mutexes* de proposições são adicionados em μP_i. Para cada ação a em A_i, as ligações do nível P_i são criadas em duas etapas: seja a uma ação que possua arestas de pré-condição ligando com a(s) pré-condição(ões) de a nas arestas positivas de P_{i-1}; liga-se a com efeitos positivos de a, e liga-se as arestas negativas com os efeitos negativos de a em P_i. Após isso, retorna-se a nova tupla gerada com a expansão do grafo.

5.1.1 Execução do GraphPlan

Tomando como exemplo o domínio da Seção 3.1.3, o objetivo do marido é preparar o *jantar* e o *presente*, e ao mesmo tempo deixar a casa limpa, ou seja, *jantar* $\wedge$ *presente* $\wedge$ $\neg$*lixo*. Como estado inicial, define-se que o marido está em silêncio, de mãos limpas, e com *lixo* dentro de casa. Para começar a execução do GraphPlan, o primeiro estado contém o estado inicial do problema:

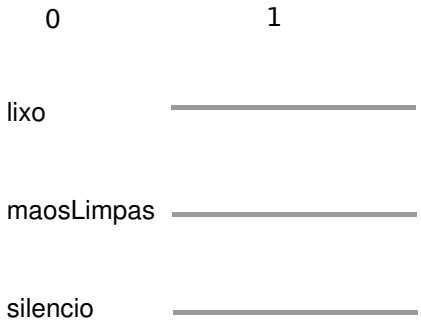

Figura 5.3: Estado inicial do GraphPlan para o problema do Jantar Surpresa.

A partir disso, verifica-se quais ações podem ser executadas no dado momento. As ações com relação ao *lixo* (*carregarLixo* e *reciclarLixo*) sempre serão executadas, já que não possuem pré-condições. Pode-se também executar *cozinhar* e *embrulhar*, já que *maosLimpas* e *silêncio* estão contidos neste nível. Logo, todas as ações são executadas.

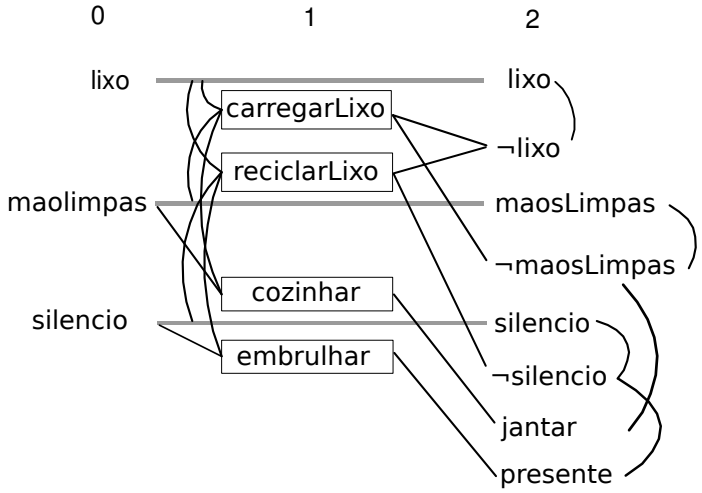

Figura 5.4: Ações executadas (com efeitos) a partir do nível 1.

A Figura 5.4 mostra o problema expandido até o nível *2*, com um nível de ação e outro de proposição. Em cada nível de ações, o algoritmo sempre tenta executar todas as ações naquele ponto no tempo, mas isso não é possível em todos os casos. Uma das informações mais importantes para o aumento da eficiência na busca por uma solução é a descoberta do maior número possível de relações binárias de exclusão mútua, também chamadas de *mutex*. Essa verificação é feita entre pares de ações e entre pares de proposições, com o intuito de descartar possíveis planos sem sequer gerá-los, caracterizando uma otimização no processamento.

5.1.2 Exclusão Mútua Entre Pares de Ações

Duas ações no mesmo nível são *mutex* entre elas por alguma das seguintes regras:

- **Efeitos Inconsistentes:** ações que anulam efeitos de outras.

- **Interferência:** ações que anulam pré-condições de outras.
- **Necessidades Concorrentes:** condições geradas por ações mutuamente exclusivas.

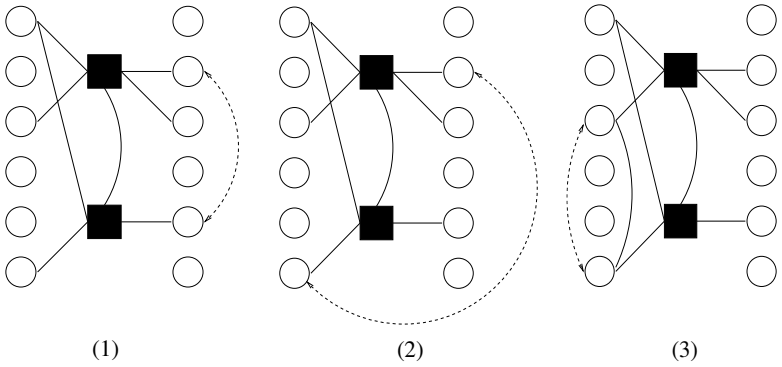

Figura 5.5: Exclusão mútua entre ações. (1) Efeitos Inconsistentes; (2) Interferência; (3) Necessidades Concorrentes.

Caso não haja interferência entre as ações, significa que ambas podem ser executadas ao mesmo tempo em um determinado momento.

5.1.3 Exclusão Mútua Entre Pares de Proposições

Duas proposições no nível *i* são *mutex* se uma é a negação da outra, ou se todas as formas de alcançar as proposições (ou seja, a partir de ações no nível *i-1*) tornam, as duas proposições em questão, *mutex* entre elas. Essa relação é chamada de *Suporte Inconsistente*.

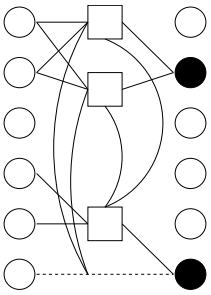

Figura 5.6: Exclusão mútua entre proposições.

Conhecidas as regras de descoberta de *mutexes*, pode-se notar a presença de algumas delas no exemplo do Jantar Surpresa utilizado até então. No nível *2*, ¬*silencio* é *mutex* com *presente* por causa de *Suporte Inconsistente*. A ação *carregarLixo* é *mutex* com a proposição *lixo* devido a *Efeitos Inconsistentes*. Já *reciclarLixo* é *mutex* com *embrulhar* por *Interferência*, pois *reciclarLixo* elimina *silencio*.

5.1.4 Condição Mínima Para Executar a Extração da Solução

Sempre que o GraphPlan gera um nível de proposições, a seguinte verificação é feita: se todas ou parte das proposições presentes no último nível podem em conjunto satisfazer o objetivo especificado no começo da execução, e nenhuma delas são *mutex*, então existe a chance de um plano existir. No exemplo do Jantar Surpresa, o nível *2* contém todas as proposições presentes no objetivo: $jantar \wedge presente \wedge \neg lixo$. A partir dessa verificação, a segunda parte do algoritmo é executada: a extração da solução.

5.2 Extração da Solução

Embora necessária, a existência de todos os literais do objetivo (subobjetivos) no último nível gerado ainda não é suficiente para garantir a existência de um plano. Neste momento, o GraphPlan executa uma tentativa de extração da solução: uma procura regressiva encadeada de nível em nível, ou seja, respeitando as arestas do grafo em busca de um plano que talvez exista no Grafo de Planejamento gerado até então. Para realizar a busca de um plano válido, o algoritmo efetua uma busca a partir do último nível gerado, ou seja, nível i. Após isso, busca-se um conjunto de ações e proposições consistentes no nível anterior (i-*1*) e que não sejam *mutex*. Esse processo repete-se recursivamente até o nível *0*, e então o plano resultante será uma sequência de níveis, onde cada nível é composto por ações que podem ser executadas em qualquer ordem.

Basicamente, a extração da solução ocorre em três passos:

- Inicia com as proposições finais, ou seja, no último nível (nível i);
- Busca um conjunto de ações consistentes (não *mutex*) no nível anterior (nível i-*1*) que gere as condições finais. Como resultado, gera-se um conjunto de proposições necessárias para as ações escolhidas; e
- Repete-se o processo recursivamente até o nível inicial (nível *0*).

A extração da solução considera cada um dos subobjetivos na busca por um plano, um após o outro. Subobjetivos são proposições selecionadas a partir de um nível de ação k do Grafo de Planejamento. No caso do *jantar*, os subobjetivos são *jantar, presente* e $\neg lixo$. Para cada um desses subobjetivos no nível i (último nível), o GraphPlan escolhe uma ação a no nível anterior (i-*1*) que resulte no subobjetivo. Se mais de uma ação produz como efeito essa proposição, então o GraphPlan precisa considerar todas essas ações para garantir a existência do plano, ou seja, é necessário avaliar a escolha que leva ao caminho consistente até aquele subobjetivo. Se a é não *mutex* (consistente) com todas as ações que foram escolhidas até então no nível atual, então o GraphPlan avança para o próximo subobjetivo. Se não há disponível nenhuma escolha que satisfaça todas essas condições, o algoritmo retrocede para uma escolha selecionada anteriormente. Após encontrar um conjunto de ações consistentes no nível i-*1*, o GraphPlan tenta recursivamente achar um plano para esse conjunto unindo todas as pré-condições dessas ações no nível anterior (i-*2*), e assim por diante. Se as proposições estiverem presentes no nível zero, então o GraphPlan encontrou uma solução para o objetivo. Caso contrário, o GraphPlan expande o Grafo de Planejamento executando todas as ações possíveis nas proposições do nível i, para então tentar novamente a extração da solução.

Algoritmo 2 Extração da Solução [10].

Extract(G, g, i)

begin

if i = 0 **then return** ($\langle\rangle$)
if g $\in \nabla$(i) **then return** (failure)
$\pi_i \leftarrow$**GP-Search**(G, g, $\emptyset$, i)
if $\pi_i \neq$ failure **then return** (πi)
∇(i)$\leftarrow\nabla$(i) $\cup$ {g}
return (failure)

end

GP-Search(G, g, π_i, i)

begin

if g = $\emptyset$ **then**
$\prod \leftarrow$ **Extract**(G, $\cup${precond(a) | $\forall$a $\in \pi_i$}, i - 1)
if $\prod$ = failure **then return** (failure)
return ($\prod . \langle\pi_i\rangle$)
else
select any p $\in$ g
resolvers$\leftarrow${a $\in$ A_i | p $\in$ effects$^+$(a) and $\forall$b $\in \pi_i$: (a, b) $\notin \mu A_i$}
if *resolvers* = $\emptyset$ **then return** (failure)
nondeterministically choose a $\in$ *resolvers*
return (**GP-Search**(G, g $-$ effects$^+$(a), $\pi_i \cup$ {a}, i))

end

O algoritmo da extração possui três parâmetros: o grafo G, os subobjetivos g e o nível i corrente. Se o nível for zero, retorna um conjunto vazio denotando que o planejamento teve sucesso, sendo que as ações encontradas nos níveis visitados nas recursões anteriores irão compor o plano resposta. Se os subobjetivos estão contidos na tabela de memoização não existirá plano, pois a tabela de memoização armazena tuplas de erros das tentativas que não deram certo. A extração é centrada quase que em sua totalidade em uma subfunção chamada *GP-Search*, que será responsável por encontrar a sequência $\prod$ de ações nível a nível, que juntas levam à solução. As ações escolhidas em um nível i serão armazenadas em π_i. Caso passe pelas primeiras verificações, o método GP-Search é chamado e realiza a verificação de *mutexes* e subobjetivos, para então tentar a extração. A ideia da função *GP-Search* é ir eliminando os subobjetivos de g a partir do momento em que eles não sejam *mutex*. Se g estiver vazio, significa que todos satisfazem as condições de extração da solução, e a extração é então chamada. Na chamada, g recebe as pré-condições de todos as ações contidas no nível i do plano que precisam ser satisfeitas na regressão, que será feita na próxima chamada de extração. $\prod$ receberá o resultado da extração, e se o valor recebido for *failure*, o algoritmo aborta e não haverá solução para o problema. Caso contrário, concatena $\prod$ com as ações executadas de π_i. Caso g não esteja vazio, seleciona-se

qualquer um dos subobjetivos p em g. Para cada ação a pertencente ao nível A_i, se p pertence a efeitos positivos de a, e para qualquer ação b contida em π_i, se a dupla (a, b) não for *mutex* entre si, ou seja, não estiverem em μA_i, adiciona-se a em *resolvers*. Se *resolvers* estiver vazio, quer dizer que não existe uma solução, visto que nele é incluído todas as ações possíveis para executar no dado momento. Caso contrário, o algoritmo escolhe de modo não-determinístico uma ação a para regredir e tentar achar uma solução, eliminando algum efeito de a caso esteja presente nos subobjetivos g, e adicionando a ação escolhida em π_i. Caso exista um plano para a solução, ao longo da execução o algoritmo vai eliminando de g os subobjetivos que satisfaçam para extração de plano no GraphPlan. Isso se repete ação por ação até encontrar uma solução ou falhar.

No exemplo do Jantar Surpresa ocorre a tentativa de extração da solução, visto que todos os subobjetivos estão no nível *2* e não há *mutex* entre eles (Figura 5.4). Os subobjetivos *jantar* e *presente* são gerados a partir das ações *cozinhar* e *embrulhar*, respectivamente. Já ¬*lixo* é gerado a partir de duas ações: *carregarLixo* e *reciclarLixo*. A partir dessas informações, sabe-se que o GraphPlan precisa considerar dois conjuntos de ações no nível *1*:

- A_0 = {*carregarLixo*, *cozinhar*, *embrulhar*}; e
- A_1 = {*reciclarLixo*, *cozinhar*, *embrulhar*}.

Como o subobjetivo ¬*lixo* pode ser gerado por duas ações diferentes - e as ações são sempre executadas paralelamente - é necessário avaliar dois conjuntos diferentes de ações, um com *carregarLixo* e outro com *reciclarLixo*. Analisando os dois conjuntos, percebe-se que há *mutex* em ambos, tornando-os inconsistentes: *carregarLixo* é *mutex* com *cozinhar* e *reciclarLixo* é *mutex* com *embrulhar*, ambos devido a *Interferência*.

Portanto, a extração da solução falha. Isso significa que, até o dado momento, não existe nenhuma possibilidade de solução para o problema proposto, levando o GraphPlan novamente à expansão do grafo.

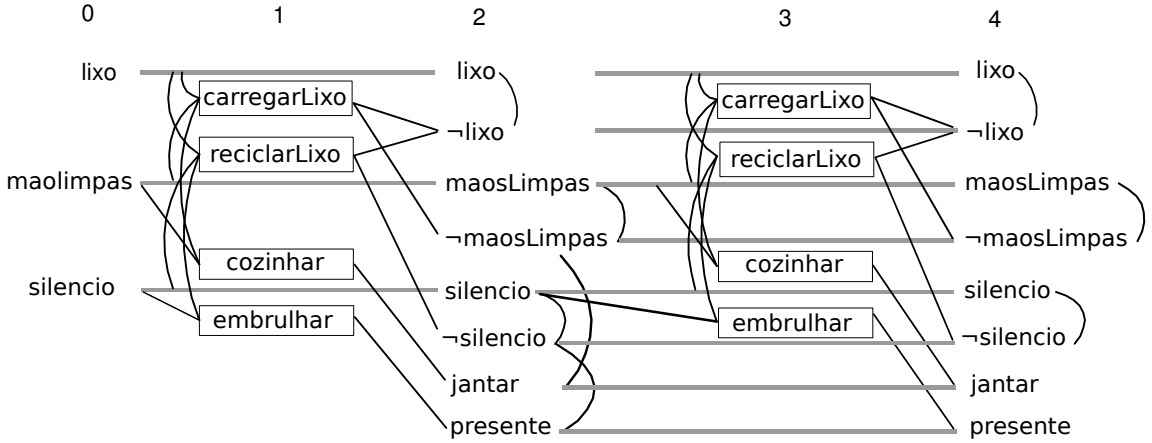

Figura 5.7: Grafo expandido para o nível *4*.

No nível *0* haviam apenas *3* proposições (estado inicial), e a partir dessas 3 proposições foram geradas outras novas proposições através das ações no nível *1*, inclusive a conservação dessas 3 proposições (a conservação das proposições ocorre em todos os níveis para todas as proposições, pois a opção de não alterar a proposição é uma alternativa que pode ser usada). O nível *4* é igual ao nível *2,* pois todas as proposições são as mesmas. Porém, no nível *4* há 5 proposições que foram conservadas e que não existiam no nível *0*. Isso é muito importante, pois agora, cada

proposição subobjetivo possui uma gama maior de possibilidades de escolhas para considerar durante a extração da solução. Analisando-se os subobjetivos presentes no nível *4*:

- ¬*lixo* é suportado por *carregarLixo*, *reciclarLixo* e uma ação de conservação;
- *jantar* é suportado por *cozinhar* e uma ação de conservação; e
- *presente* é suportado por *embrulhar* e uma ação conservadora.

Com isso, as relações de *mutex* foram reduzidas, mesmo não existindo novas proposições no nível *4*. Agora, é possível tentar extrair novamente uma solução para achar o plano.

Para começar a extração, seleciona-se no nível anterior (nível *3*) um conjunto de ações e/ou conservações não *mutex* que satisfaçam as proposições subobjetivos no nível atual (nível *4*):

- *carregarLixo*: resulta em ¬*lixo*;
- *jantar* é suportada pela conservação desta. Neste caso não precisa ser necessariamente uma ação (pode ser a manutenção da mesma); e
- *embrulhar*: resulta em *presente*.

Nenhuma destas escolhas são mutuamente exclusivas (*mutex*). Então, as escolhas utilizada no nível *3* (o conjunto de ações escolhido) é consistente. O cuidado deve ser com relação às pré-condições das ações escolhidas, caso existam. Apenas *carregarLixo* não possui pré-condição, logo não há problema nenhum no fato dela ser executada no plano atual que está sendo gerado. A ação *embrulhar* requer *silencio*, e *jantar* requer que exista sua conservação, ambos no nível *2*. Já que ambos estão contidos no nível *2*, o algoritmo executa mais um passo de extração.

A extração agora parte para o nível *1*. As últimas proposições encontradas do atual plano foram *jantar* e *silencio*, que podem ser suportadas por *cozinhar* e conservação, respectivamente. Estas duas ações não são *mutex*, então as escolhas para o nível *1* são consistentes. As pré-condições dessas duas ações resultam, no nível *0*, nas proposições *maosLimpas* e *silencio*. Essas proposições estão contidas no estado inicial, ou seja, o caminho até então é consistente. Logo, um plano para o problema foi encontrado.

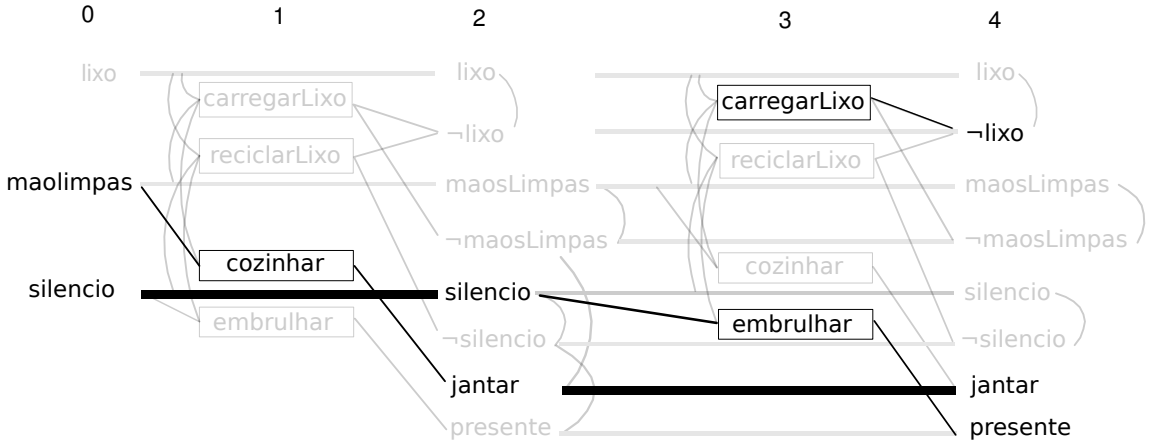

Figura 5.8: Um dos possíveis planos para o problema do Jantar Surpresa.

A seguir, descrevemos todas as etapas do algoritmo seguindo a formalização de Ghallab [10]. Todos os pseudocódigos do GraphPlan utilizados neste trabalho foram retirados de seu livro.

Algoritmo 3 Algoritmo do GraphPlan [10].

Graphplan(A, s_0, g)

begin

i←0, $\bigtriangledown$←∅, P_0←s_0
G←$\langle P_0 \rangle$
until [g $\subseteq$ P_i and $g^2 \cap \mu P_i = \emptyset$] or Fixedpoint(G) **do:**
    i←i+1
    G←**Expand**(G)
if g $\nsubseteq$ P_i or $g^2 \cap \mu P_i \neq \emptyset$ **then return** (failure)
$\prod$=**Extract**(G,g,i)
if Fixedpoint(G) **then** $\eta \leftarrow |\nabla(k)|$
else $\eta \leftarrow 0$
while $\prod \leftarrow$failure **do:**
    i←i+1
    G←**Expand**(G)
    $\prod$←**Extract**(G,g,i)
    if $\prod$ = failure and Fixedpoint(G) **then**
        if $\eta = |\nabla(k)|$ **then return** (failure)
        $\eta \leftarrow |\nabla(k)|$
return ($\prod$)

end

No pseudocódigo do Ghallab et al [10], um grafo G é uma tupla de níveis de ações, proposições e *mutexes* (relações de exclusão mútua entre ações e entre proposições) G = (P_0, A_1, μA_1, P_1, μP_1, ... ,A_i, μA_i, P_i, μP_i) tal que:

- P_i representa um nível de proposição;
- A_i representa um nível de ações (o pseudocódigo de Ghallab considera A_i e P_{i+1} como um único nível, ou seja, as ações e seus efeitos resultantes);
- A_i representa um nível de *mutexes* encontrados em A_i (ver Seção 5.1.2); e
- μP_i representa um nível de *mutexes* encontrados em P_i (ver Seção 5.1.3).

O algoritmo começa inicializando o primeiro nível ($i = 0$) e o primeiro nível de proposições P_0 como o estado inicial S_0. A tabela de memoização $\bigtriangledown$ (ver Seção 6.1) é inicializada juntamente e armazenará tuplas de erros (será usada como apoio caso em algum ponto da execução já tenha passado pelo mesmo erro). G armazenará P_0 e representará o grafo montado até certo instante de execução. O grafo é expandido até o momento em que os subobjetivos g estejam contidos no último nível (P_i), onde todos os pares possíveis desses subobjetivos não devem aparecer em μP_i (pois significaria que existe um ou mais *mutexes*) ou se o grafo já está estabilizado e é representado pelo ponto fixo, também chamado de *Fixedpoint* (momento a partir do qual os níveis do grafo são sempre iguais). Se as iterações de expansão terminarem e pelo menos uma das condições

anteriores não for satisfeita, o algoritmo termina, pois não há mais possibilidade de existir uma solução, uma vez que nem as proposições possíveis, nem os *mutexes* irão se alterar em níveis subsequentes. Sendo possível uma tentativa de extração, o método da extração é chamado com os valores do grafo G, subobjetivos g e do nível corrente i, e o algoritmo armazena o resultado em $\prod$. Se o grafo já estiver estabilizado, então o algoritmo armazena em η os valores já contidos no índice k da tabela de memoização (caso contrário η será vazio). Se o resultado em $\prod$ não for *failure*, retorna $\prod$, pois um plano foi encontrado. Enquanto $\prod$ for *failure*, o algoritmo cria um nível novo, expande o grafo e tenta extração novamente armazenando o resultado em $\prod$. Se $\prod$ for *failure* ou ponto fixo, o algoritmo realiza duas verificações: se parte do plano encontrado até então já estiver em um índice k da tabela de memoização, o algoritmo retorna *failure,* pois este erro já foi analisado anteriormente; caso contrário, armazena uma tupla contendo dados deste erro no índice k da tabela de memoização. O algoritmo segue iterando neste laço até encontrar a solução.

6 Otimizações

Muitas formas de otimizações foram estudadas para o GraphPlan e continuam sendo até hoje. Embora ele seja um algoritmo de planejamento melhor do que vários outros da mesma geração, ainda assim, pode ser otimizado com técnicas e análises feitas ao longo dos últimos anos. A proposta deste trabalho é estudá-las e tentar aplicar o maior número de otimizações possíveis, e que ao mesmo tempo surtam efeitos significativos. Os benefícios destas otimizações dependem do problema de planejamento a ser resolvido. Algumas otimizações citadas na proposta deste trabalho não serão exploradas (*Action Schemata, Type Analysis & Simplification*, *Solution Extraction as Constraint Satisfaction* e *Generic Types*), devido às mesmas possuírem um nível de complexidade cuja implementação não seria viável dentro do cronograma do trabalho de conclusão. Nas próximas seções serão descritas as otimizações selecionadas para realização deste trabalho.

6.1 Memoização (*Memoization*)

A memoização consiste em analisar um conjunto de subobjetivos em um determinado nível k do grafo e verificar se os mesmos são insolúveis. Se forem, os mesmos são armazenados em uma tabela de memoização. Desta forma, antes de executar a extração da solução, essa tabela é consultada com o intuito de verificar se os subobjetivos do nível atual já foram identificados como insolúveis. A consulta a essa tabela evita uma busca exaustiva a cada execução da extração da solução. Esta técnica baseia-se em Programação Dinâmica, que é uma técnica computacional de otimização da resolução de algoritmos recursivos, onde cálculos repetidos são armazenados na memória do computador, e utilizados ao invés de se refazer computações com parâmetros que já foram calculados anteriormente.

6.2 Armazenamento de Mutexes e Invariantes Deduzidas Automaticamente (*Utilizing Automatically Inferred Invariants in Graph Construction and Search*)

A ideia desta otimização [8] é aprimorar os testes de exclusão mútua durante a construção do grafo, e encontrar relações além das definidas no algoritmo original do GraphPlan. As relações de exclusão mútua que normalmente não são identificadas pelo algoritmo original são relações do tipo *persistent* e *non-persistent mutex*. *Persistent mutex* corresponde a propriedades lógicas invariantes de um determinado domínio, ou seja, uma propriedade que não se altera ao aplicar-se um conjunto de ações (transformações). Essa relação inclui um caso especial, chamado de *permanent mutex*, no qual indica conflitos entre pré-condições e efeitos das ações. *Non-persistent mutex* correspondem a relações de exclusão mútua que podem mudar de estado conforme o

grafo se expande. O objetivo é evitar que estes *mutexes* sejam recalculados desnecessariamente a cada expansão do grafo. Para certos domínios o algoritmo original não consegue extrair estas relações devido a estas estarem "escondidas" durante a construção do grafo. Para identificar (pré-compilar) essas propriedades, pode-se utilizar o sistema TIM [6], que deduz automaticamente uma coleção de invariantes a partir de descrições (propriedades) do domínio. A dedução dessas invariantes pode ser utilizada em um pré-processamento gerando uma coleção de relações de exclusão mútua. Esta coleção pode ser armazenada em uma matriz, a partir de um algoritmo que baseia-se nas invariantes geradas pelo sistema TIM. Esse algoritmo compara duas proposições a fim de determinar se são *mutex*, e então a matriz é indexada pelos nomes das proposições que estão sendo comparadas. Essa matriz pode ser consultada de forma muito eficiente, e com isso, o GraphPlan pode determinar essas relações durante o processo de construção de grafo. A identificação dessas relações garante que não haverá a necessidade de nova consulta a essa matriz com relação a um determinado par de proposições ou ações, uma vez que foram identificados como uma relação de exclusão mútua.

Análises de domínios conhecidos (*Gripper*, *Logistics*, e etc.) revelaram que o GraphPlan infere os dois tipos de *mutexes* (*persistent* e *non-persistent mutex*) a partir de cada uma das quatro formas de inferência de invariantes utilizadas pelo sistema TIM. A partir destas análises, concluiu-se que existe uma pequena economia na construção de grafo fazendo uso do pré-processamento dos *mutexes*.

6.3 Hipótese do Mundo Fechado (*Closed World Assumption*)

A Hipótese do Mundo Fechado [18] consiste em considerar que, se um fato não foi declarado, ele deve ser falso. Esse conceito possui dois tipos de implementação: *simples* e *relaxada* (*lazily*). Na implementação simples define-se que qualquer proposição não explicitamente conhecida como verdadeira no estado inicial pode ser presumida como falsa. Uma possível implementação para esta abordagem no GraphPlan seria "fechar" o nível *0* do grafo a fim de adicionar a negação de todas as proposições possíveis (todas que não foram declaradas como verdadeiras). Porém, como pode haver um número muito grande de proposições possíveis, deve-se restringir essa abordagem somente para:

- Proposições descritas nas pré-condições de alguma ação; ou
- Proposições que fazem parte do objetivo.

Inicialmente, a abordagem *relaxada* adiciona somente proposições conhecidas como verdadeiras para o estado inicial (nível *0*). Ao expandir o grafo para um nível de ação *i*, realiza-se os seguintes procedimentos: suponha-se que a ação **A** possui a proposição $\neg$**P** como pré-condição; se a proposição $\neg$**P** está no nível *i-1* do grafo, simplesmente faz a ligação. No entanto, se $\neg$**P** não está do nível *i-1* do grafo, deve-se verificar se a sua negação (ou seja, **P**) está no nível *0*. Se **P** não está do nível *0*, então $\neg$**P** deve ser adicionado no nível *0*, e a partir disto, deve-se efetuar manutenções de *mutexes* e ações referente a $\neg$**P** até o nível atual do grafo.

A implementação *relaxada* é considerada uma solução melhor, pois pode reduzir o tamanho do grafo e consequentemente o custo de expansão, além de não ser necessário mudanças para a extração da solução.

6.4 Otimizações não Implementadas

Durante o desenvolvimento deste trabalho, algumas das otimizações estudadas no TC1 se mostraram mais complexas do que inicialmente estimado. Desta forma, foi necessário descartar algumas das otimizações estudadas no TC2, uma vez que seu esforço de desenvolvimento requer mais tempo do que o disponível.

6.4.1 Foco na Regressão (*Regression Focussing*)

De acordo com Weld [18], Kambhampati mostra que é possível através da expansão regressiva do Grafo de Planejamento acelerar o processamento do GraphPlan sem incompletude. Kambhampati propôs uma modificação na expansão do grafo: primeiro expande-se o Grafo de Planejamento regressivamente a partir dos subobjetivos, com um nível de ação e de proposição. Após isso, expande-se o grafo progressivamente a partir da intersecção do estado inicial com o grupo de proposições obtido na regressão, juntamente com as relações de *mutex* e as ações ground[1] que foram adicionadas durante a regressão, para então executar a extração da solução, se necessário. Se a extração falhar na busca por um plano, então o método de Kambhampati executa mais uma vez a expansão no grafo para gerar mais um par de níveis. Uma nova intersecção (maior que a anterior) é gerada com o estado inicial, e continua a expansão progressiva. Embora essa implementação reconstrói o grafo a partir do zero a cada etapa, o Grafo de Planejamento resultante é bem menor segundo Weld, a ponto de superar o GraphPlan sem esta otimização na maioria dos problemas testados.

6.4.2 Expansão Estática do Grafo (*In-Place Graph Expansion*)

Essa otimização visa evitar trabalho duplicado durante a regressão, focando na exploração de alguns aspectos referentes às repetições quando o grafo estabiliza (momento a partir do qual os níveis do grafo são sempre iguais).

- Proposições são monotonicamente[2] crescentes se: a proposição p estiver presente no nível i e em todos os níveis de proposição subsequentes.
- Ações são monotonicamente crescentes se: a ação a estiver presente no nível i e em todos os níveis de ação subsequentes.
- *Mutexes* são monotonicamente decrescentes se: o *mutex* m entre as ações a e b estiverem presente no nível i, então m estará presente em todos os níveis anteriores em que a e b aparecem.
- Subobjetivos inconsistentes são monotonicamente decrescentes se: os subobjetivos q, r e s são inalcançáveis no nível i, então serão inalcançáveis em todos os níveis anteriores de proposições.

A partir do momento em que os níveis estabilizam-se, pode-se usar um grafo bipartido com nodos de ação e proposição. Arestas de proposições para ações representam pré-condições e arestas de ações para proposições representam efeitos. Proposição, ação, *mutex* e subobjetivos inconsistentes são todos anotados com um valor numérico. Para nodos de proposição e ação, esse

[1] *Ground actions:* são instâncias de operadores PDDL/STRIPS onde todas as variáveis estão ligadas com objetos do domínio. Em termos gerais, a diferença é a mesma que entre Classes (Operadores/Ações não-ground) e Objetos (Ações ground).

[2] *Monotonicidade*: significa que algo sempre muda da mesma forma. Por exemplo, em uma função monotonicamente crescente: $\forall_{x,y}\ (y > x) \rightarrow (f(y) > f(x))$.

valor numérico representa o primeiro nível no grafo em que o nodo de proposição ou ação aparece. Para nodos *mutex* ou subobjetivo inconsistente, esse valor representa o último nível em que estas relações se mantêm. Ao adicionar esse conjunto de anotações, pode-se intercalar a expansão para a frente e para trás. Utilizando-se desta ideia, o custo de tempo da fase de expansão do grafo pode diminuir, mas a contabilidade desses valores pode ser bastante complicada.

6.4.3 Exploração de Simetria (*Detection and Exploitation of Simmetry*)

Fox e Long definem uma relação de simetria entre objetos [7] quando um ou mais objetos possuem funções idênticas, ou seja, não são distinguíveis de forma útil ao planejador. Por exemplo, supõe-se uma caixa com inúmeros parafusos idênticos um ao outro, que é usada na construção de alguma coisa. A maioria dos planejadores, mesmo os mais modernos, calcularia todas as possibilidades de parafusos para serem usados em um dado momento, ou pior ainda, calcularia soluções alternativas de uso dos parafusos. Como todos são idênticos e estão à disposição para uso, não faz muito sentido calcular várias permutações possíveis de parafusos, já que não mudaria em nada o plano para o problema, visto que todas as permutações são equivalentes. Ou seja, essa otimização busca evitar decisões repetidas ou desnecessárias pelo planejador. Antes de analisar essa otimização mais profundamente, é necessário uma prévia análise em alguns conceitos importantes que a exploração de simetria exige, como predicados estáticos [3] e inferência de tipos.

O processo de verificação de simetrias é uma análise estática que independe da arquitetura da implementação que explorará as simetrias encontradas, e ocorre em duas fases. Na primeira, grupos de objetos simétricos são automaticamente extraídos da descrição do problema, mais precisamente do estado inicial e do objetivo (esse processo funciona com uma entrada no formato STRIPS. Portanto, neste trabalho seria necessário adaptá-lo ao PDDL). Como mencionado anteriormente, objetos simétricos são definidos como indistinguíveis um ao outro com relação aos seus estados iniciais e finais. Por exemplo, no domínio do *Gripper*[4], bola_1 e bola_2 podem ser considerados simétricos entre si se:

- No estado inicial, bola_1 e bola_2 estão na sala_a; e
- No objetivo, bola_1 e bola_2 estão na sala_b.

Alguns cuidados devem ser tomados com as especificações de estado inicial e objetivo. Informações redundantes, imprecisas ou insuficientes podem causar a perda de simetrias. Por exemplo, se no estado inicial ambas as pinças do robô estão livres, e no objetivo nada está especificado com relação às pinças, isto faz com que as pinças sejam consideradas objetos simétricos. Porém, se apenas uma das pinças é mencionada no estado inicial, então ambas as pinças terão propriedades diferentes que as tornarão assimétricas entre elas. Aparentemente, todas essas características tornam a análise de simetria extremamente frágil. Mas é justamente essa alta sensibilidade que a torna muito poderosa, visto que qualquer divergência nos estados dos objetos é interpretada como uma evidência de que estes objetos não devem ser tratados como totalmente indistinguíveis. A ideia é começar identificando pares de objetos simétricos e começar a formar os grupos simétricos, ou seja, a base destes. Após isso, esses grupos são estendidos adicionando-se outros objetos do mesmo tipo inferido. A informação sobre a tipagem de objetos é fundamental, pois objetos de diferentes tipos não podem ser simétricos, e muitas operações de comparação podem ser evitadas. Isso permite reduzir drasticamente a busca que é realizada para formar os grupos de simetria.

[3] *Predicados Estáticos*: tipo de predicado cujo valor verdade nunca se altera durante a execução de um plano.

[4] *Gripper problem*: problema onde um robô necessita transportar bolas de uma sala a outra, utilizando tanto o braço esquerdo quanto o direito.

Finalmente, na segunda fase da verificação de simetrias, ocorre a identificação e o agrupamento de ações simétricas. Ações são consideradas simétricas quando seu(s) parâmetro(s) são utilizados da mesma coleção de grupos simétricos. Sumarizando, o objetivo de analisar simetrias em um problema é reduzir o número de escolhas de ações que são procuradas.

7 Projeto de Implementação

Durante o semestre avaliamos algumas implementações do GraphPlan. Abaixo seguem as implementações avaliadas.

7.1 Implementações do GraphPlan

- **Emplan** - `http://emplan.sourceforge.net`: Emplan é uma implementação do GraphPlan que incorpora diversas variações do mesmo. O projeto possui a implementação em duas linguagens: Java (JavaGP) e C++.
- **JPlan** - `http://jplan.sourceforge.net`: JPlan é uma implementação em Java do GraphPlan, um planejador independente de domínio. Como é um projeto de código aberto, permite que desenvolvedores e pesquisadores realizem experimentos com planejamento em seus respectivos projetos.
- **PDDL4J** - `http://sourceforge.net/projects/pddl4j`: PDDL4J é uma biblioteca *open source* criada com o objetivo de facilitar implementações de planejadores em Java baseados no PDDL. A biblioteca contém um *parser* da última versão do PDDL 3.0 e uma implementação do GraphPlan.

Avaliando as implementações selecionadas (Emplan, JPlan e PDDL4J), decidimos utilizar o Emplan. O projeto Emplan foi o escolhido devido às seguintes razões: possui ótima documentação, foi projetado pensando em futuras modificações, possui uma versão em Java (linguagem na qual o grupo possui maior conhecimento em relação às outras), e um dos desenvolvedores do projeto estar disponível às possíveis dúvidas referentes à implementação. A implementação em Java do Emplan chama-se JavaGP. O PDDL4J será utilizado juntamente com o JavaGP, pois o Emplan utiliza como linguagem de planejamento o STRIPS. Na próxima seção descrevemos a arquitetura do JavaGP.

7.2 Arquitetura do JavaGP - Emplan

Avaliamos a arquitetura do JavaGP e a mesma se encaixa perfeitamente no objetivo deste trabalho, que é selecionar uma implementação já existente do GraphPlan que permita aplicar um conjunto de otimizações. O JavaGP possui uma documentação completa e foi projetado utilizando padrões de projeto (*Design Patterns*) [9] na implementação de diversos componentes de sua arquitetura. Esta característica do JavaGP resulta em um alto nível de expansibilidade (reutilização), o que permite modificá-lo sem muitas complicações. O JavaGP também utiliza as

bibliotecas do Jason [1] [2] para o modelo de objetos correspondente a representação de Lógica de Primeira Ordem (predicados, termos, variáveis e etc) e operações como comparação e unificação. A seguir serão descritos os principais componentes da arquitetura do JavaGP.

7.2.1 O Grafo de Planejamento

O Grafo de Planejamento é estruturado em níveis conforme a ideia padrão do GraphPlan, onde esses níveis são de proposições (PropositionLevel) e ações (ActionLevel). Na arquitetura do JavaGP, um nível (GraphLevel) é utilizado como objeto base (classe "pai") para proposições e ações.

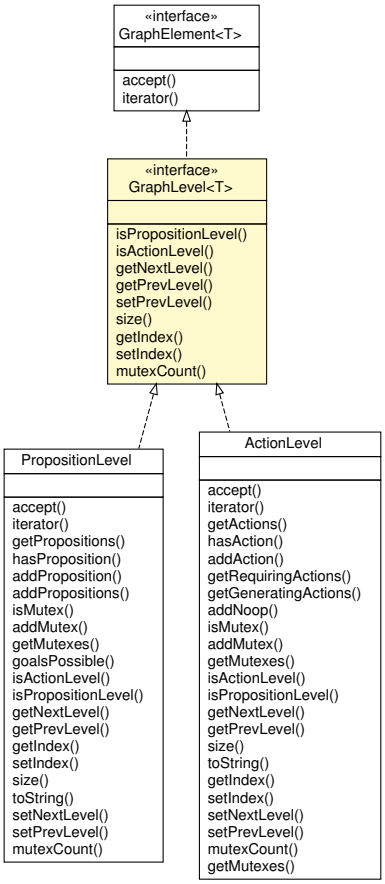

Figura 7.1: UML do nível (GraphLevel) do JavaGP.

Para criar o Grafo de Planejamento, o JavaGP utiliza uma implementação *Factory Pattern* [9] que permite criar diversos objetos relacionados a partir de uma única interface e sem que a classe concreta seja especificada. Utilizando esse padrão de projeto, utilizou-se *Factory* para ações, proposições e termos.

[1] `http://jason.sf.net`

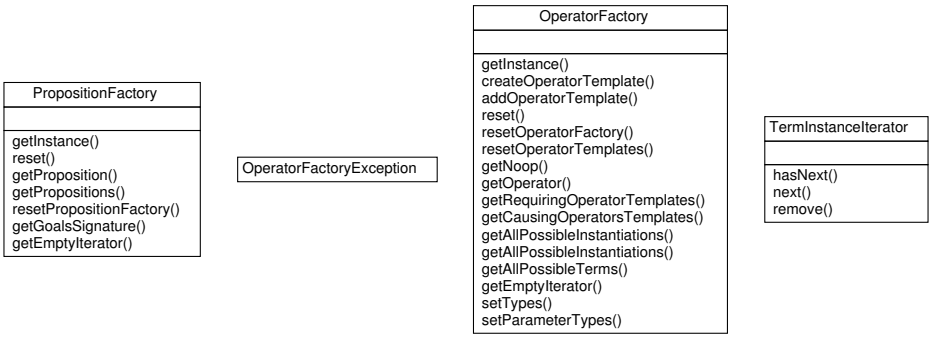

Figura 7.2: Descrição em UML das classes implementadas em *Factory* utilizadas para criar o grafo.

7.2.2 O Algoritmo do GraphPlan

Várias funções componentes do algoritmo do GraphPlan foram implementadas utilizando o padrão *Strategy/Visitor Pattern* [9]. Esse padrão de projeto tem como objetivo representar uma operação a ser realizada sobre elementos da estrutura de um objeto, o que permite criar uma nova operação sem que se mude a classe dos elementos sobre as quais ela opera. É uma maneira de separar um algoritmo da estrutura de um objeto. As Figuras 7.3 e 7.4 mostram em diagramas UML uma ideia da arquitetura do algoritmo.

Pode-se observar (Figura 7.3), que o objeto *PlanningGraph* implementa o Grafo de Planejamento (Figura 7.1) e contém o algoritmo de expansão do grafo, o método *expandGraph*. Como parte do *Visitor Pattern*, a classe *PlanningGraph* implementa o método *accept* da interface. Este método recebe por parâmetro um objeto *visitor* do tipo *GraphElementVisitor* (Figura 7.4), e esse *visitor* chama o método *visitElement* passando como parâmetro a própria instância da classe em que este método está presente (no caso, uma instância de *PlanningGraph*).

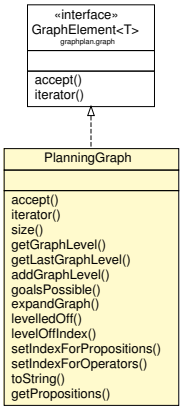

Figura 7.3: UML do Grafo de Planejamento do JavaGP.

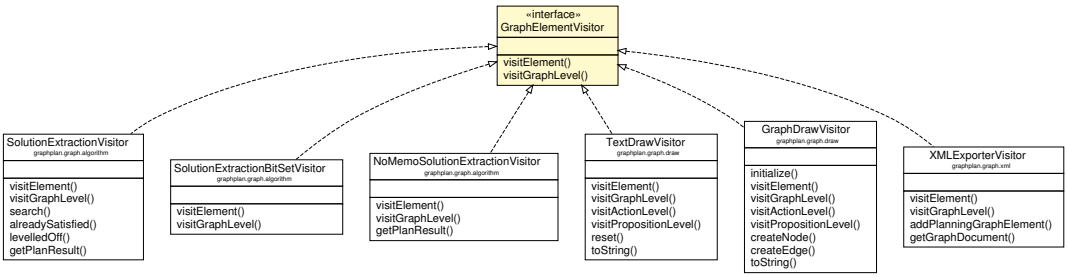

Figura 7.4: UML da extração da solução do JavaGP.

Na Figura 7.4, pode-se observar as classes que fazem parte da extração da solução, ou seja, implementações do *GraphElementVisitor*. As classes *SolutionExtractionBitSetVisitor* (implementação incompleta da extração da solução utilizando mapa de bits) e *NoMemoSolutionExtractionVisitor* (implementação alternativa também utilizando *Visitor Pattern*) não foram utilizadas neste trabalho. As classes *TextDrawVisitor* e *GraphDrawVisitor* permitem representar o Grafo de Planejamento em formato gráfico, e a classe *XMLExporterVisitor* permite salvar um Grafo de Planejamento para continuar o processamento posteriormente a partir do momento em que a execução foi pausada, sendo uma boa alternativa para problemas muito grandes.

8 Integração com PDDL4J - Implementação de um Adapter com PDDL4J

Para a leitura de problemas e domínios em PDDL foi utilizada uma biblioteca chamada PDDL4J. Com o uso dessa biblioteca, não foi necessário a implementação de um *parser* para a leitura de arquivos em PDDL, mas sim o desenvolvimento de um *Adapter*[1] [9] a partir das estruturas de dados providas pelo PDDL4J. A utilização do PDDL4J permite a tipagem nos parâmetros de proposições e ações, e a negação de pré-condições no JavaGP.

8.1 Implementação

Ao analisar as estruturas de dados do PDDL4J (*PDDLObject*) após o *parser,* identificou-se uma certa similaridade com as estruturas utilizadas pelo JavaGP (*DomainDescription*), ou seja, objetos de domínio (ações, proposições, tipos, etc) e problema (estado inicial e objetivo) representados de forma congênere. Devido a essa similaridade, a implementação do *Adapter* não interferiu nas estruturas de dados do JavaGP, o qual foi um fator muito importante para escolha do PDDL4J. A única alteração feita na classe *DomainDescription* foi a adição dos tipos e parâmetros de proposições e ações. O *Adapter* desenvolvido chama-se *PDDLPlannerAdapter*, e tem como objetivo realizar a tradução do *PDDLObject* (PDDL4J) para *DomainDescription* (JavaGP) sempre que o JavaGP utilizar a linguagem PDDL.

A Figura 8.1 mostra um diagrama de sequência em UML representando os passos da execução do *parsing* e as mensagens passadas entre os objetos.

[1] *Adapter*: padrão que tem como principal objetivo facilitar a conversão da interface de um classe para outra interface, fazendo com que várias classes possam trabalhar em conjunto independentemente das interfaces originais.

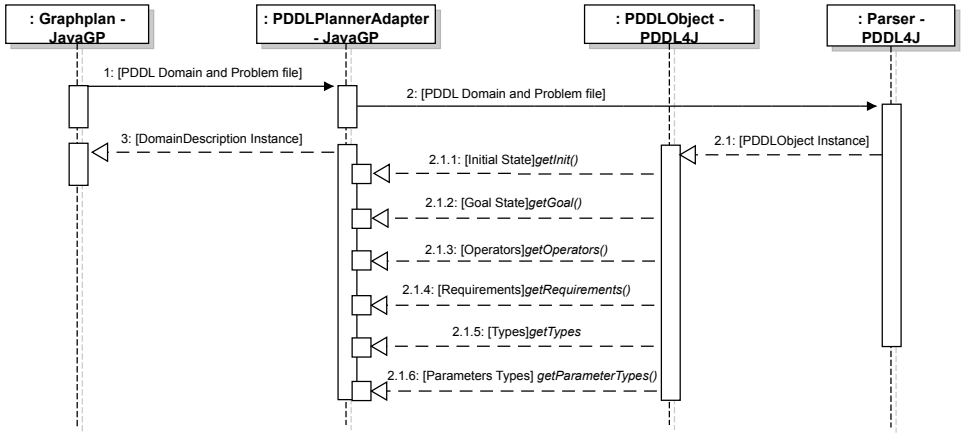

Figura 8.1: Diagrama de sequência em UML do *PDDLPlannerAdapter*.

8.2 Tipos

A utilização do PDDL4J permitiu a tipagem de proposições e ações no JavaGP, o que facilitou a instanciação dos mesmos durante a execução do GraphPlan. Sem a utilização de tipos, o JavaGP testava todas as combinações de parâmetros para instanciar cada ação e proposição, dadas as pré-condições fornecidas. O *Dock Worker Robots*, por exemplo, possui a ação *move*, e seus argumentos são, respectivamente: um robô, uma localização inicial e uma localização final.

- move(ROBOT, FROM, TO);

Sem a tipagem, os exemplos a seguir de ações desnecessárias seriam instanciadas e acrescentadas no conjunto de ações possíveis:

- move(ROBOT, TO, FROM);
- move(TO, FROM, ROBOT);
- move(FROM, ROBOT, TO).

8.3 Negação de Pré-Condições

O PDDL4J permitiu a utilização de pré-condições negativas na especificação de ações do domínio, o que não era possível usando STRIPS. Com esta característica foi possível a implementação da otimização Hipótese do Mundo Fechado (*Closed World Assumption*), a qual será descrita com mais detalhes no próximo capítulo.

9 Correções do JavaGP

Antes de iniciar o desenvolvimento das otimizações, identificou-se que o JavaGP necessitava de algumas correções. Algumas dessas correções já haviam sido identificadas e documentadas no *SourceForge* por usuários do JavaGP.

9.1 Mutexes

Alguns problemas de consistência foram encontrados na instanciação de *mutexes* (Algoritmos 4 e 5). Um deles refere-se ao *Suporte Inconsistente*, onde o método, no último retorno, retornava *false* quando na verdade deveria retornar *true*. Se as proposições sendo analisadas forem uma a negação da outra, então existe um *mutex* entre elas e o método retorna *true* (pois encontrou um *mutex*). Senão, o algoritmo analisa todas as maneiras de atingir as duas proposições, verificando par a par essas maneiras: se em algum momento for encontrado alguma maneira de antigir as proposições, ou seja, se as duas ações consideradas não forem *mutex* entre elas, então essas duas proposições não são *mutex*. O método vai retornar false somente se for encontrado uma maneira de atingir essas duas proposições, pois conforme visto anteriormente, *Suporte Inconsistente* existe somente se não há nenhum jeito de atingir as duas proposições. Se nenhum jeito for encontrado, então existe *mutex* e o último retorno é *true*, e não *falso* como no algoritmo original.

Algoritmo 4 Algoritmo de validação do *mutex Suporte Inconsistente* - Parte 1

```
InconsistentSupport(proposition1, proposition2, previousLevel)
begin
    if (proposition1 == !proposition2 ) then
        return TRUE
    foreach operator1 in previousLevel.getGeneratingActions(proposition1) do:
        foreach operator2 in previousLevel.getGeneratingActions(proposition2) do:
            if ( !previousLevel.isMutex(operator1, operator2) ) then
                return FALSE
    /*O retorno aqui deve ser TRUE e não FALSE */
    return TRUE
end
```

O outro problema encontrado foi em um trecho de código que realiza a verificaçao dos *mutexes*

de *Suporte Inconsistente* e *Interferência*. A comparação de efeitos ocorria com os efeitos da própria ação sendo analisada, o que ocasionava em alguns *mutexes* instanciados incorretamente.

Algoritmo 5 Algoritmo de validação do *mutex Suporte Inconsistente* - Parte 2

```
begin
    ...
    foreach proposition in operator1.getEffects() do:
        foreach precondition in operator2.getPreconds() do:
            if ( proposition.isMutex(precondition) ) then
                return TRUE
        /*Neste laço, os efeitos eram iterados a partir do operator1 ao invés de
            operator2 */
        foreach effect in operator2.getEffects() do:
            if ( proposition.isMutex(effect) ) then
                return TRUE
    ...
end
```

9.2 Seleção de Ações

Na realização de testes no JavaGP, notou-se uma certa demora na extração da solução. Ao notar esse fato, decidiu-se realizar uma investigação do porquê a demora na extração da solução, e identificou-se que essa demora ocorria na seleção de ações a partir de um nível de ações e um conjunto de subobjetivos.

Para realizar essa seleção de ações, o JavaGP utilizava um iterador (*ActionSetIterator*) que gera um conjunto de ações satisfazendo um conjunto de subobjetivos em um determinado nível de ações. Embora eficaz, esse iterador não era eficiente, pois não gerava um conjunto mínimo de ações a partir de um nível de ações e um conjunto de subobjetivos. Além disso, uma quantidade muito grande de conjuntos nulos eram gerados desnecessariamente devido à implementação da geração de conjuntos. Identificada essa ineficiência, decidiu-se implementar a seleção de ações conforme descrição do livro de Ghallab, o algoritmo chamado *GP-Search*.

Após a aplicação do *GP-Search*, o JavaGP mostrou-se mais eficiente, obtendo um ganho de performance considerável. No Capítulo 11 (*Resultados - Avaliações de Desempenho*) poderá ser avaliada com mais clareza essa correção.

10 Otimizações Implementadas

10.1 Heurísticas

Uma seleção heurística de nodos é uma forma de classificar um conjunto de nodos de acordo com suas relativas vantagens, e são usadas para resolução de escolhas não-determinísticas [10]. Na maior parte dos casos, a heurística de seleção de nodos é suscetível a erros no que diz respeito a um nodo recomendado pela heurística não ser garantidamente sempre a melhor escolha: este nodo pode não guiar à melhor solução, ou até mesmo a uma solução. Por isso, as heurísticas precisam ser o mais informativas possíveis, para que escolhas incorretas sejam, em grande parte, descartadas (quanto mais escolhas incorretas descartadas, mais informativa é a heurística).

O Grafo de Planejamento gerado pelo GraphPlan é análogo a uma heurística, mas as heurísticas também podem ser usadas dentro do GraphPlan para melhorá-lo. Isto é feito no processo de extração da solução [10] tornando-o priorizado por heurísticas.

Para fazer uso de heurística, existe um princípio chamado de *relaxamento*, onde algumas restrições do problema original são eliminadas com o intuito de deixar o problema mais fácil de ser resolvido (embora ajude, o GraphPlan não muda de complexidade com esta técnica). Esta estratégia de modelagem é uma aproximação do problema original, onde é mais fácil de ser resolvido e ainda por cima fornece informações relevantes sobre o problema original. Todos os conceitos de heurísticas citados até então estão inclusos no *relaxamento* do GraphPlan.

10.1.1 Ordenação de Ações

10.1.1.1 Ordenação de Ações *Resolvedoras*

A ordenação de *resolvedores* é uma heurística [10] que sempre seleciona preferencialmente a ação *resolvedora* que aparece mais tarde no Grafo de Planejamento. Uma ação *resolvedora* é uma ação que satisfaz uma proposição na parte da extração da solução, ou seja, uma ação que possui a proposição satisfeita em um de seus efeitos. Para a heurística foi utilizada uma estrutura de dados para armazenar as ações, ordenando-as pelo seus índices que correspondem ao nível em que elas aparecem pela primeira vez. Se uma ação **A** aparece no grafo em um nível *i*, então não faz sentido incluir **A** no conjunto possível de escolhas (na extração) nos níveis anteriores a *i*. Esta ordenação de ações não permite que isso aconteça. A implementação consistiu em adicionar um atributo na classe *OperatorImpl* para representação do índice, e uma estrutura de dados como atributo para armazenar os operadores na classe *PlanningGraph*.

10.1.1.2 Ordenação de Ações *No Operation First*

Antes de realizar qualquer refatoração no JavaGP, um dos problemas que este apresentava era a seleção de ações na extração da solução. Em muitos casos, esta seleção optava por uma ação

que produziria a proposição em si, quando na verdade o correto seria optar por um *No operation*. Isso acarretava na utilização desnecessária de ações no plano final de um problema.

Segundo experimentos de Kambhampati [11], a utilização desta heurística em alguns domínios pode piorar drasticamente o desempenho do GraphPlan, pois dependendo do domínio o número de *Noops* gerados pode ser muito grande. Mesmo após essa constatação, essa heurística estará disponível no JavaGP para ser utilizada quando desejado.

10.1.2 Ordenação de Proposições

10.1.2.1 Ordenação de Proposições Subobjetivos

A ordenação de proposições a partir de subobjetivos é uma heurística [10] que sempre seleciona primeiro a proposição objetivo que aparece mais cedo no Grafo de Planejamento, porque pode ser fácil de encontrar uma solução para esta proposição no momento da extração da solução.

10.1.2.2 Ordenação de Proposições que leva ao Menor Número de *Resolvedores*

A ordenação pelo número de *resolvedores* é uma heurística [10] que consiste em selecionar primeiro a proposição p que leva ao menor número de *resolvedores*, isto é, a proposição p alcançada pelo menor número de ações. Se p é alcançado por apenas uma ação, então será melhor processado quanto mais cedo possível e não será necessário que o GraphPlan faça *backtracking*. A partir do momento em que o algoritmo de GraphPlan retrocede, ele analisa o plano gerado até então em busca de uma solução para o objetivo proposto começando o retrocesso do ponto de parada da expansão, e revisando todos os níveis gerados até então.

10.2 Memoização (*Memoization*)

A implementação da tabela de memoização no JavaGP consiste em uma lista que contém uma tabela de proposições subobjetivos mapeada pelo índice k do grafo, onde esses subobjetivos são insolúveis. No JavaGP, a tabela de memoização é representada pelo objeto *MemoizationTable* (Figura 18).

A tabela de memoização é utilizada na extração da solução em dois momentos:

1. Antes de executar a extração é realizada a seguinte consistência. Dado subobjetivos selecionados em um índice k do grafo, se os mesmos são insolúveis, ou seja, estão na tabela de memoização, a extração da solução não é executada para estes, caso contrário, a extração é executada.

2. Após a seleção de ações a partir de subobjetivos em um índice k do grafo. Se a seleção de ações não retornar nenhum conjunto, os subobjetivos selecionados para o índice k serão armazenados na tabela de memoização, pois são insolúveis.

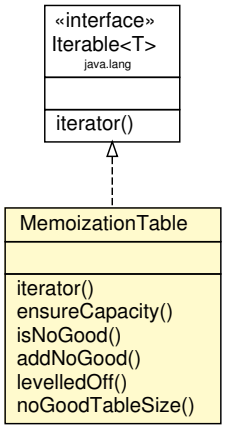

Figura 10.1: Diagrama em UML do *MemoizationTable*.

10.3 Mutexes Estáticos

Os *mutexes* do tipo *Interferência* e *Efeitos Inconsistentes*, uma vez encontrados entre duas ações, nunca sofrerão qualquer alteração, permancendo estaticamente no grafo independentemente dos *mutexes* de proposições antes e depois destas proposições (ver exemplo a seguir). Como visto anteriormente (Seção 5.1.2), esta otimização consiste em pré-compilar estes dois tipos de *mutexes*, ou seja, armazenar regras que serão usadas ao longo da execução do algoritmo como consulta para estes *mutexes*, ao invés de realizar as verificações habituais de *Interferência* e *Efeitos Inconsistentes*. Essas regras são criadas a partir da análise de todas as ações pertencentes ao domínio, par a par, verificando se as condições destas ações sendo analisadas unificam. Unificação é um algoritmo usado em problemas de satisfatibilidade, onde o objetivo é encontrar um conjunto mínimo de substituições para tornar duas fórmulas idênticas. O método de unificação utilizado no JavaGP, é da biblioteca Jason.

Tomando como exemplo os *Efeitos Inconsistentes*, a análise é feita par a par com os efeitos de todas ações. Se uma ação a possui um efeito que não unifica com o efeito de uma ação b, e se a assinatura[1] dos dois efeitos forem a mesma, então existe um *mutex* a ser armazenado entre a e b. Como procura-se por efeitos que sejam negações uns dos outros, tanto nos *Efeitos Inconsistentes* quanto na *Interferência*, a etapa de verificação de unificação funciona como um primeiro filtro, pois uma proposição nunca vai unificar com sua negação, já que a negação faz parte do predicado. Após isso, verifica-se então se a assinatura de ambas as proposições são idênticas. Caso seja, é um *mutex* estático e deve ser armazenado na tabela de *mutexes* estáticos, pois a negação não faz parte da assinatura. Se há argumentos nos predicados, estes são armazenados de acordo com sua indexação, de forma a identificar exatamente com quais argumentos ocorrem este *mutex*. O mesmo processo ocorre para os *mutexes* de *Interferência*.

[1] A assinatura refere-se ao nome do predicado, juntamente com seus argumentos e a ordem com que estão dispostos.

Algoritmo 6 Algoritmo de pré-compilação dos Mutexes Estáticos.

```
preCompileStaticMutexes(operators)

begin

    /*Criar No Operations para as proposições dos Efeitos e Pré-condições*/
    operators.addNoops(getNoops(operators));
    foreach op1 in operators do:
        foreach op2 in operators do:
            if( op1.getSignature() == op2.getSignature() ) then
                continue
            foreach effect1 in op1.getEffects() do:
                /*Efeitos inconsistentes*/
                foreach( proposition1 in op2.getEffects() ) do:
                  if( !effect1.unifies(effect2)
                  and effect1.getSignature() == effect2.getSignature() ) then
                        populateStaticMutexTable( effect1, effect2, op1, op2 )
                /*Interferência*/
                foreach( proposition1 in op2.getPreconds() ) do:
                  if( !effect1.unifies(proposition1)
                  and effect1.getSignature() == proposition1.getSignature() ) then
                        populateStaticMutexTable( effect1, proposition1, op1, op2 )

end
```

Utilizando o domínio do *Gripper* com exemplo, supõe-se as seguintes comparações:

- **Efeitos Inconsistentes**
  - Ações: pick(OBJ, ROOM, GRIPPER) e drop(OBJ, ROOM, GRIPPER).

  Ambas as ações possuem como um de seus efeitos, respectivamente, carry(OBJ, GRIPPER) e ¬carry(OBJ, GRIPPER). As duas proposições não unificam por causa da negação da segunda, e possuem a mesma assinatura. Logo, há um *mutex* estático e é armazenado na tabela, juntamente com os respectivos argumentos, pois estes mesmos efeitos podem não ser *mutex* com outros diferentes argumentos.

- **Interferência**
  - Ações: noop_carry(OBJ,GRIPPER) e drop(OBJ,ROOM,GRIPPER).

  A ação drop possui como um de seus efeitos ¬carry(OBJ,GRIPPER), e noop_carry possui como uma de suas pré-condições carry(OBJ,GRIPPER). Ambas não unificam, pois uma delas possui negação no predicado, mas a assinatura de ambas é igual. Então, realiza-se o mesmo processo de armazenamento citado anteriormente nos *Efeitos Inconsistentes*.

Além da tabela de *mutexes* estáticos, foi adicionada uma outra tabela auxiliar de memoização que armazenará a assinatura das ações com *mutexes* estáticos, pois o custo de acesso a esta tabela é mais barato do que o da tabela de *mutexes* estáticos.

10.4 Hipótese do Mundo Fechado (*Closed World Assumption*)

No conceito da Hipótese do mundo fechado considera-se que, se um fato não foi declarado, ele deve ser presumido como falso. Esse conceito possui dois tipos de implementação: *simples* e *relaxada (Lazily)*.

10.4.1 Simples

A implementação do tipo *simples* foi feita da seguinte forma. Para toda proposição **P** do estado objetivo, se **P** não está no estado inicial, então a negação desta proposição (¬**P**) será adicionada no estado inicial.

Algoritmo 7 Implementação *simples* da Hipótese do Mundo Fechado.

foreach $g \in$ Goals **do**:

if InitialState $\nsubseteq g$ **then**
InitialState $\cup \neg g$

10.4.2 Relaxada *(Lazily)*

Para implementação do tipo *relaxada* foi necessário extender a expansão do Grafo de Planejamento, pois é preciso efetuar a manutenção de *mutexes* e ações referentes às proposições adicionadas no nível *0* até o nível atual do grafo. Com essa necessidade, projetou-se uma nova estrutura para o uso de pré-condições negativas no JavaGP. Como a expansão do grafo possui complexidade polinomial, não sendo portanto um agravante de performance, decidiu-se que ao invés de realizar manutenção dos *mutexes* e níveis do grafo, optou-se simplesmente pela deleção do Grafo de Planejamento. Com isso, ao identificar que a Hipótese do Mundo Fechado pode ser usada, o novo método de expansão cria um novo Grafo de Planejamento.

No JavaGP, o objeto responsável pela manutenção do Grafo de Planejamento é o *PlanningGraph*. Esse objeto contém os níveis do grafo (Ações e Proposições), algoritmo de expansão do Grafo de Planejamento, e efetua a manutenção de *mutexes* a cada nível que o grafo é expandido.

Devido a ótima estrutura do *PlanningGraph*, decidiu-se manter a estrutura atual, sobreescrevendo somente alguns métodos para a realização desta otimização. Na Figura 10.2 segue diagrama em UML referente à estrutura criada para a implementação *relaxada*.

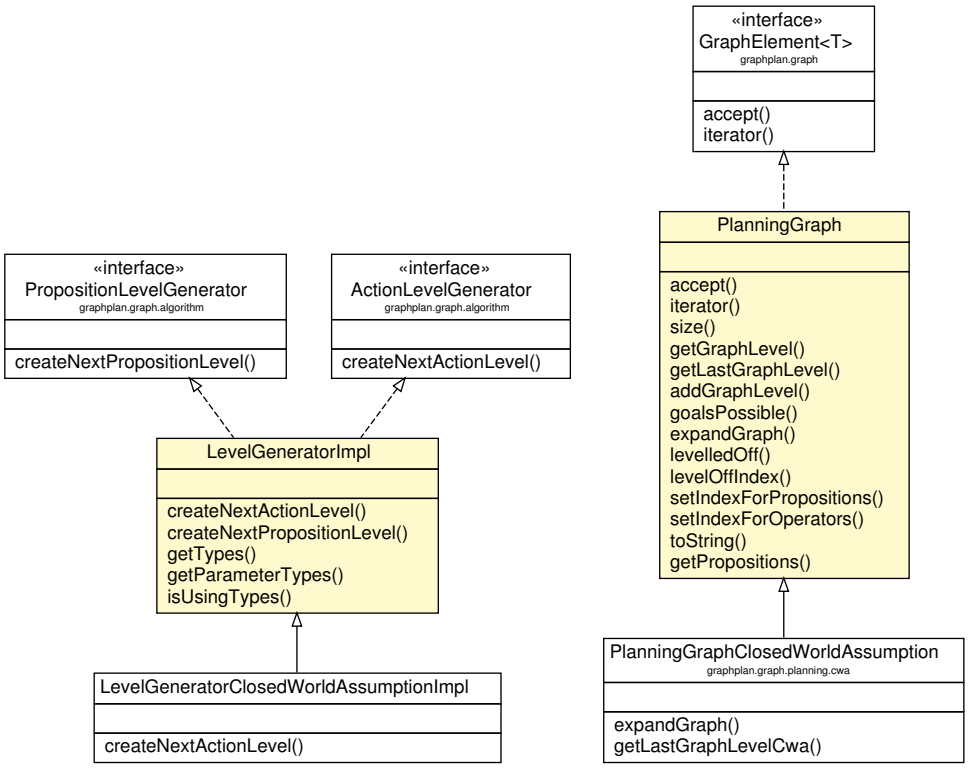

Figura 10.2: UML da estrutura da implementação *relaxada*.

A partir desta estrutura, no objeto *PlanningGraphClosedWorldAssumption,* o método *expandGraph* foi sobreescrito e sofreu algumas alterações para suprir as necessidades da implementação *relaxada* (6.3), ou seja, se uma ação **A** possui a proposição ¬**P** como pré-condição, e está no nível *i-1* do grafo, simplesmente faz a ligação. No entanto, se ¬**P** não está do nível *i-1* do grafo, deve-se verificar se a sua negação (ou seja, **P**) está no nível *0*. Se **P** não está do nível *0*, então ¬**P** deve ser adicionado no nível *0*, e a partir disto, exclue-se o atual Grafo de Planejamento e o reconstrói a partir da adição de ¬**P** no estado inicial.

Nos resultados, será exibido como a Hipótese do Mundo Fechado mostrou-se eficaz na busca de planos no problema *Dock Worker Robots.*

10.5 Inferência de Tipos

Essa otimização foi suprida pela pela tipagem de objetos utilizando o comando "(:requirements :typing)" na definição do domínio em PDDL, o que facilita a identificação dos objetos na instanciação do mesmo.

Após *parser,* o *PDDLPlannerAdapter* adiciona dois conjuntos no *DomainDescription.*

- *Types*: conjunto de tipos do domínio; e
- *Parameter types*: um conjunto mapeado pela assinatura de uma proposição ou ação, contendo uma lista de parâmetros com os respectivos tipos.

11 Resultados - Avaliações de Desempenho

Para avaliações de performance do JavaGP, realizou-se experimentos com outros domínios de planejamento além do *Dock Worker Robots*. A seguir segue uma breve explicação sobre cada um desses domínios.

- ***Blocks World*** é um domínio de planejamento que consiste de uma superfície plana, ou uma mesa, e um conjunto de blocos. Esses blocos são colocados sobre essa superfície e podem ser empilhados e desempilhados uns sobre os outros. A modelagem em PDDL deste domínio está no Apêndice B deste volume.
- ***Gripper*** é um domínio de planejamento constituído por um robô com duas garras, duas salas e algumas bolas. Ele pode carregar uma bola em cada garra. O objetivo é levar algumas bolas de uma sala para outra, com algumas restrições. A modelagem em PDDL deste domínio está no Apêndice C deste volume.
- ***Hanoi*** é um domínio de planejamento que consiste basicamente em mover todos os círculos inseridos em um dos pilares para um outro pilar. A modelagem em PDDL deste domínio está no Apêndice D deste volume.

Para os nossos experimentos, utilizamos a codificação destes domínios disponível como exemplo do PDDL4J - `http://sourceforge.net/projects/pddl4j` .

11.1 JavaGP (Sem correções e Otimizações)

A fim de avaliar o quanto o JavaGP evoluiu, realizou-se uma avaliação de performance comparando o JavaGP atual com a versão inicial do projeto. Analisando os gráficos, pode-se observar a evolução do JavaGP após as correções e otimizações.

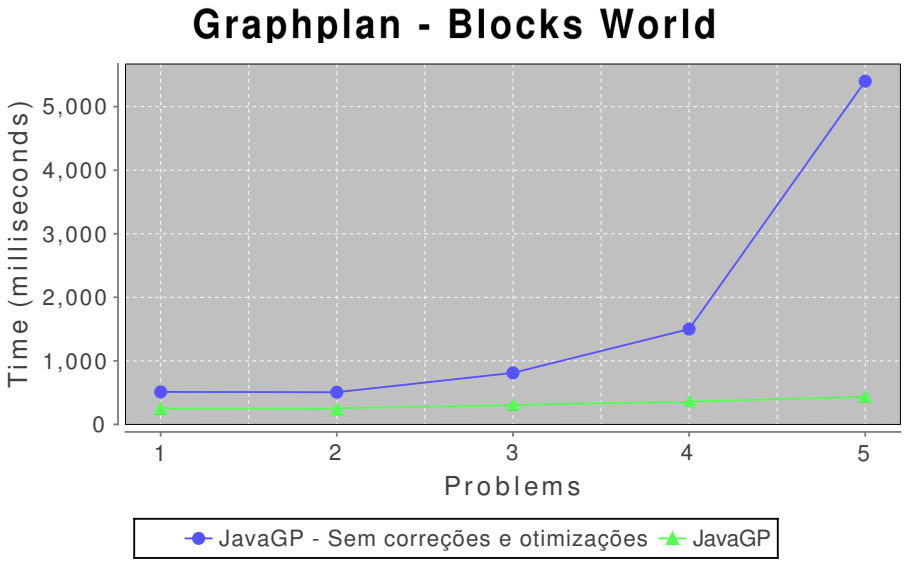

Figura 11.1: JavaGP - *Blocks World*.

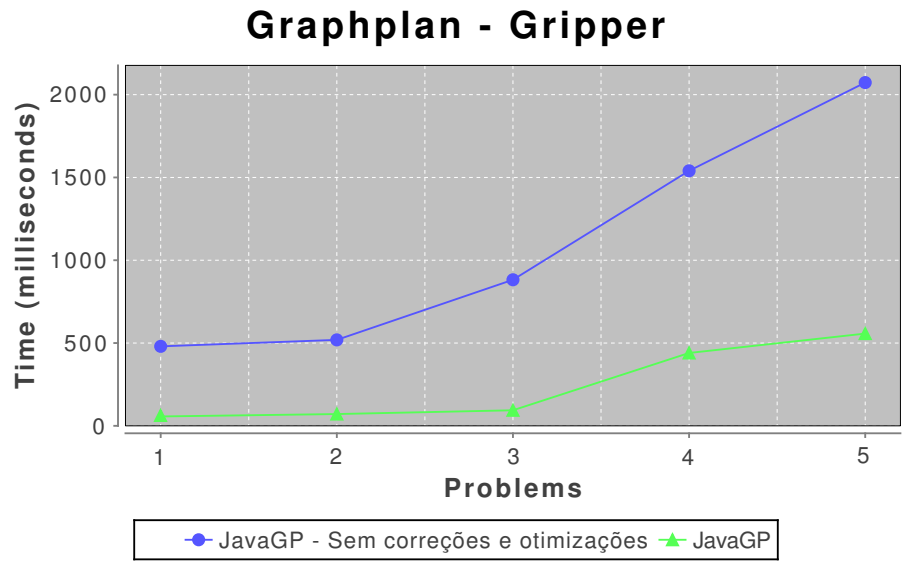

Figura 11.2: JavaGP - *Gripper*.

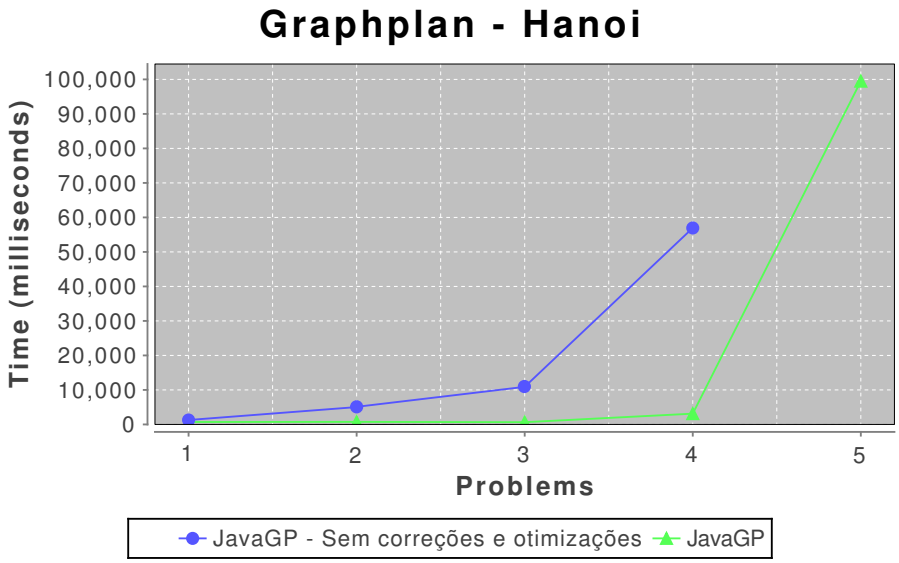

Figura 11.3: JavaGP - *Hanoi*.

Na Figura 11.3, o JavaGP na sua versão original não concluiu a execução para o problema de tamanho 5, pois a execução excedeu o tempo de 5 minutos.

11.2 Heurísticas

Realizando experimentos, constatou-se que, com a combinação de heurísticas pode-se obter um ganho significativo de performance para alguns domínios.

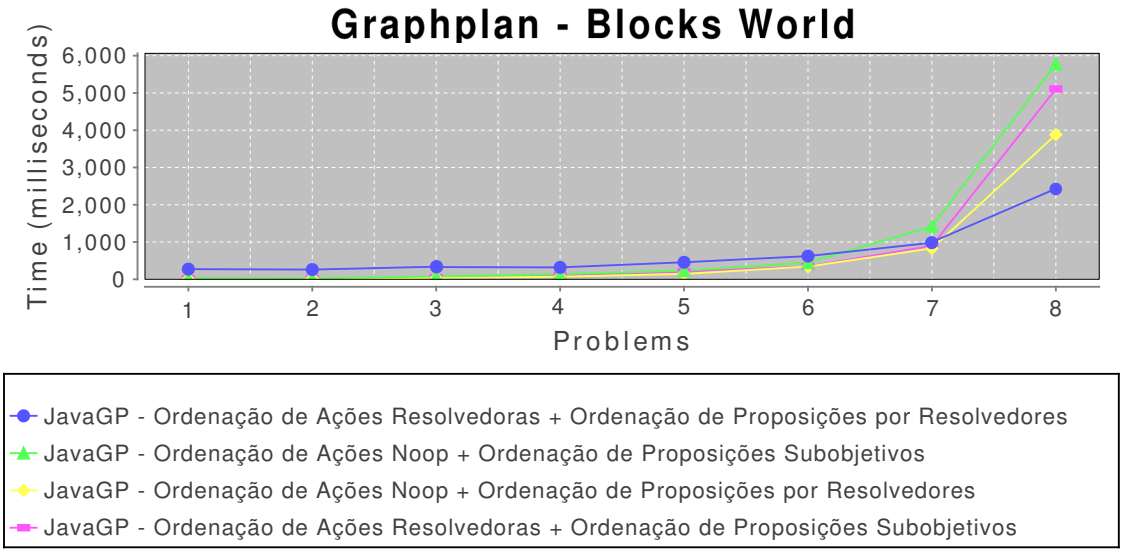

Figura 11.4: JavaGP com Heurísticas - *Blocks World*.

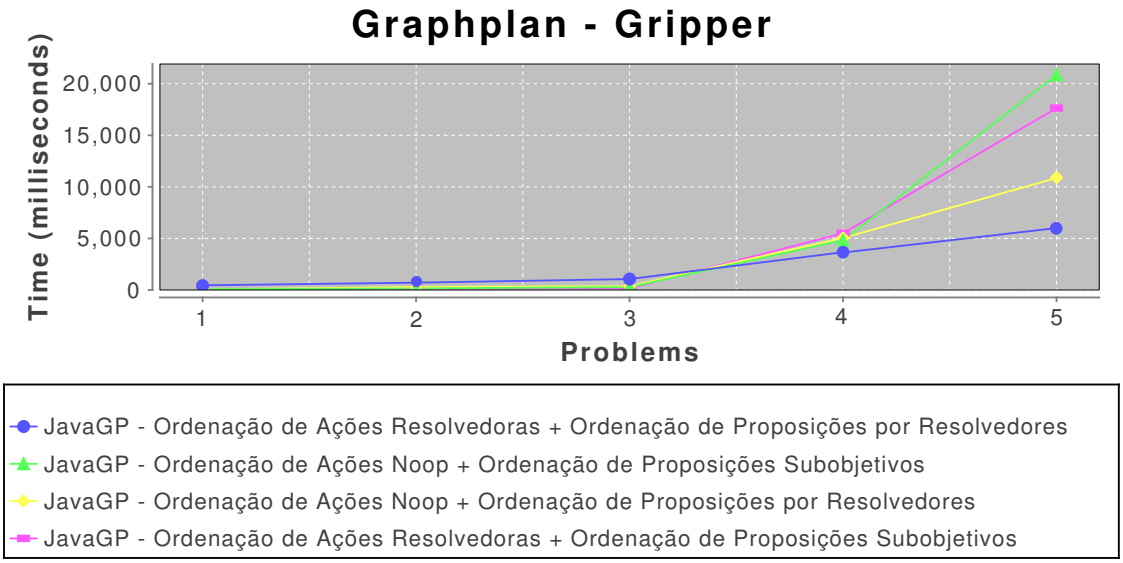

Figura 11.5: JavaGP com Heurísticas - *Gripper*.

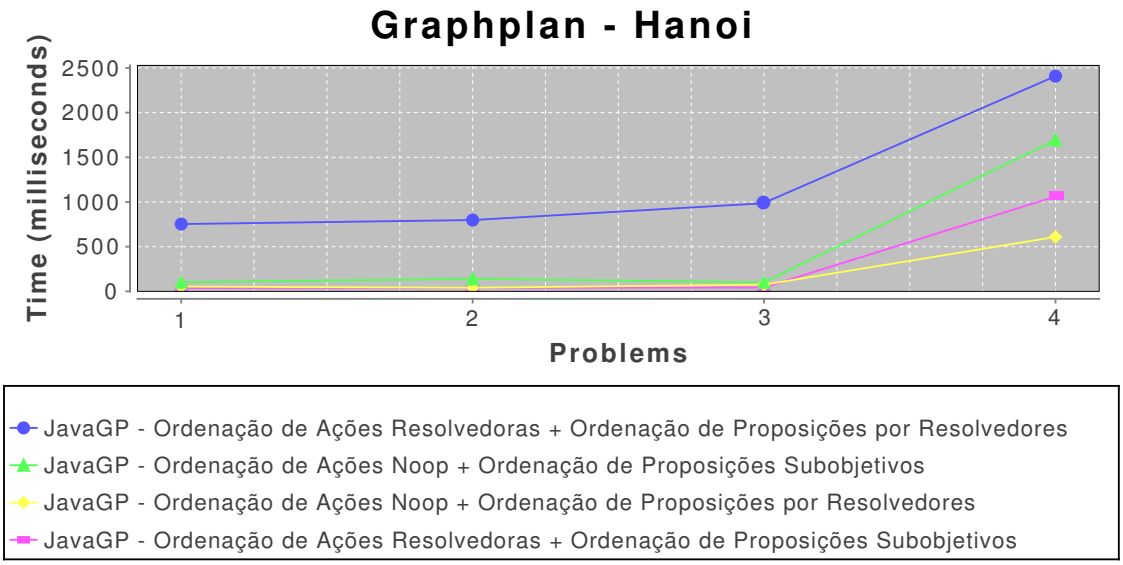

Figura 11.6: JavaGP com Heurísticas - *Hanoi*.

Nas Figuras 11.4, 11.5 e 11.6, é possível observar quais heurísticas combinadas produzem um melhor rendimento do JavaGP.

11.3 Benchmark (Comparações)

Para realização do *benchmark*, utilizou-se de outros planejadores, a fim de obter uma avaliação de performance de como o JavaGP se comporta comparado a outros planejadores. A seguir,

uma breve explicação referente aos planjeadores utilizados.

- **BLACKBOX** é um planejador baseado em técnicas de planejamento em grafo, como o GraphPlan, juntamente com técnicas de planejamento por satisfatibilidade (SatPlan). Mostrou-se um planejador bastante eficiente.
- **JGraphplan** é uma implementação do GraphPlan que acompanha a biblioteca do PDDL4J. O modo como o JGraphplan foi implementado é muito interessante, pois utiliza um mapa de bits na representação do grafo. Essa implementação mostrou-se bastante rápida e confiável, apesar do mesmo não resultar planos corretos para pré-condições negativas. Apesar da biblioteca de *parser* do PDDL4J gerar as pré-condições negativas, realizamos alguns experimentos e analisamos o código, e identificou-se que o algoritmo só trata as pré-condições positivas na expansão do grafo.
- **PDDLGraphplan** é implementação do GraphPlan que se mostrou eficaz mas não eficiente. Para problemas de tamanho médio-grande mostrou-se bastante lenta.
- **PropPlan** é um planejador baseado em busca em espaço de estado utilizando *Ordered Binary Decision Diagrams* [3]. Esse planejador será utilizado exclusivamente para comparações utilizando o domínio *Dock Worker Robots*, devido ao DWR ter pré-condições negativas nas ações e os outros planejadores não tratarem essa característica corretamente.

Alguns dos planejadores citados anteriormente não são desenvolvidos em Java, como o BLACKBOX e PropPlan. O BLACKBOX é desenvolvido na linguagem C. Já o PropPlan é desenvolvido em Standard ML, e é executado em uma interface *WEB*, na qual supõe-se que há intervenção de algum dispositivo computacional para melhorar a performance. Decidiu-se então realizar o *benchmark* com estes planejadores pois não existem planejadores confiáveis em Java em número suficiente para realizar os experimentos.

A partir dos experimentos com heurísticas no JavaGP, serão selecionadas as que obtiveram melhor rendimento conforme o respectivo domínio.

No *benchmark* utilizando o domínio *Blocks World,* o JavaGP selecionou as seguintes heurísticas: **Ordenação de Ações *Resolvedoras*** e **Ordenação de Proposições que leva ao Menor Número de *Resolvedores***. Otimizações selecionadas: **Mutexes Estáticos**, **Memoização** e **Inferência de Tipos**.

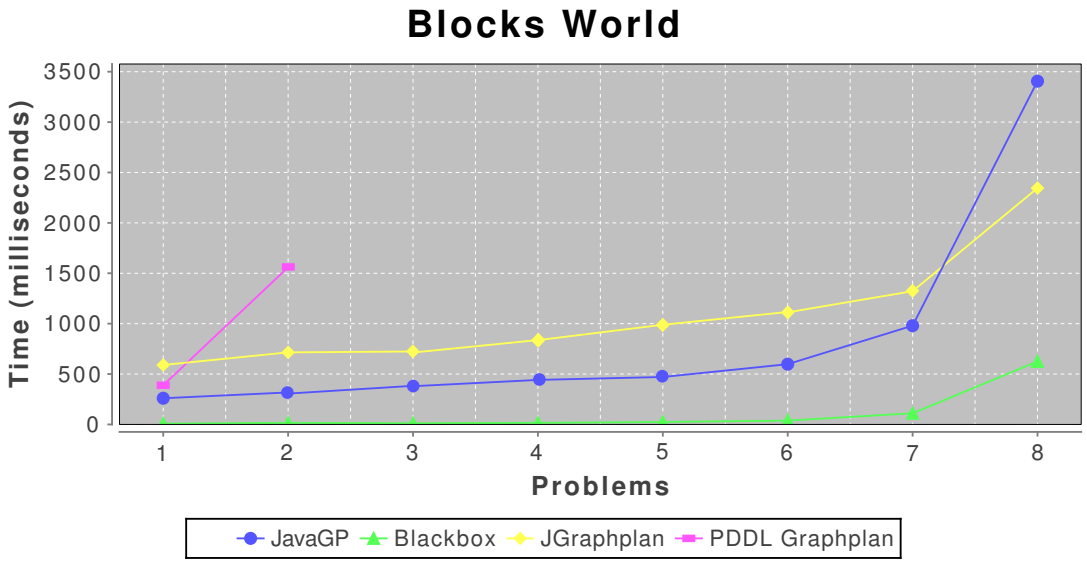

Figura 11.7: Benchmark - *Blocks World*.

No *benchmark* utilizando o domínio *Gripper,* o JavaGP selecionou as seguintes heurísticas: **Ordenação de Ações *Resolvedoras*** e **Ordenação de Proposições que leva ao Menor Número de *Resolvedores***. Otimizações selecionadas: **Mutexes Estáticos**, **Memoização** e **Inferência de Tipos**.

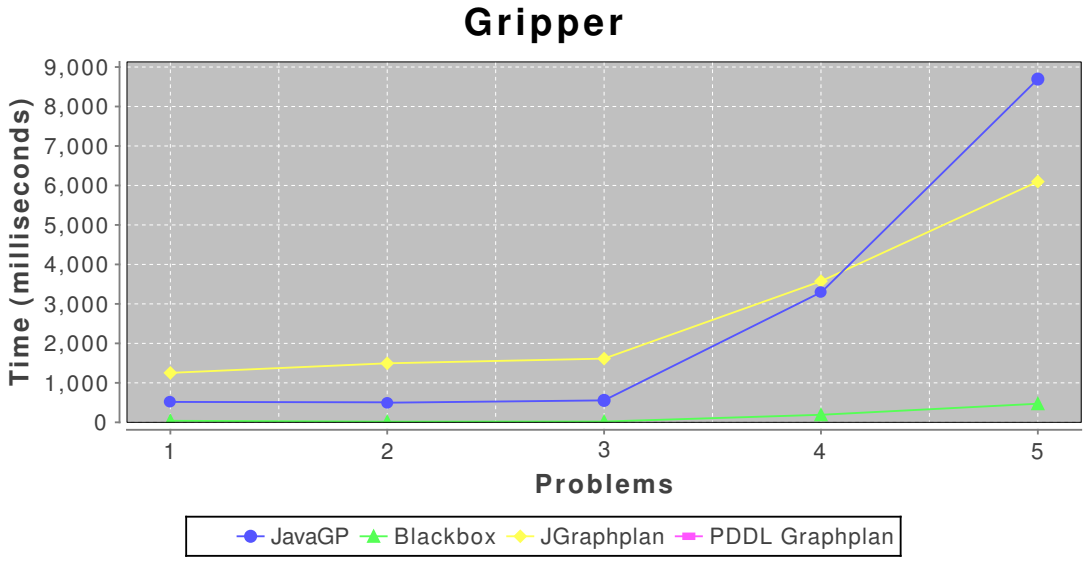

Figura 11.8: Benchmark - *Gripper*.

Porém, no *benchmark* utilizando o domínio *Hanoi,* o JavaGP selecionou as seguintes heurísticas: **Ordenação *No Operation First*** e **Ordenação de Proposições que leva ao Menor Número de *Resolvedores***. Otimizações selecionadas: **Mutexes Estáticos**, **Memoização** e **Inferência de Tipos**.

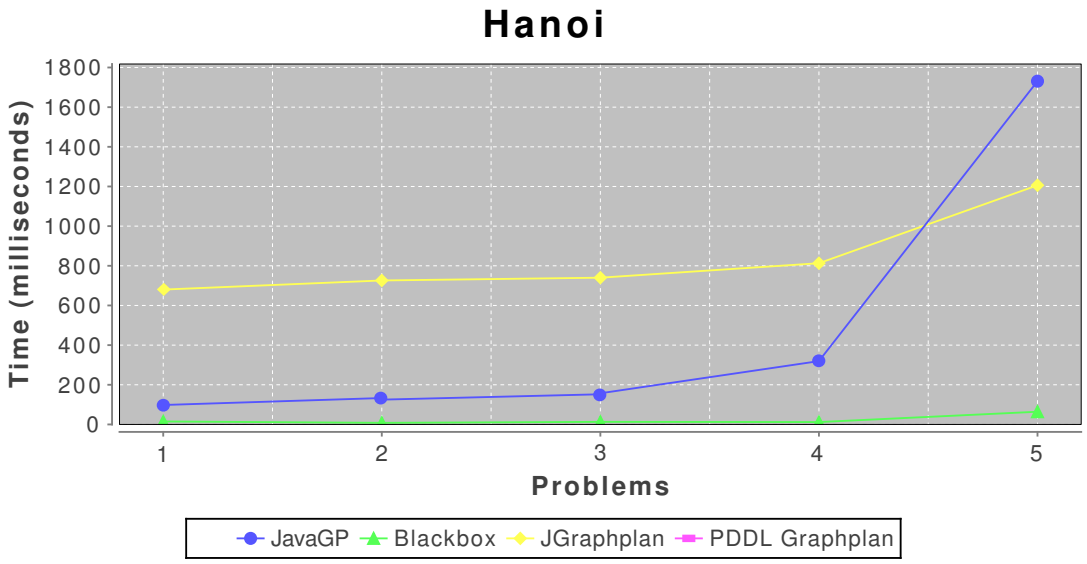

Figura 11.9: Benchmark - *Hanoi*.

Nas Figuras 11.8 e 11.9 é possível notar que o planejador PDDLGraphplan não aparece nas mesmas. Isto ocorreu pelo fato de o planejador não ter concluído a execução para um tempo de até 5 minutos.

O *benchmark* utilizando o domínio *Dock Worker Robots* foi realizando somente entre o JavaGP e o PropPlan. Para esta execução, o JavaGP selecionou as seguintes heurísticas: **Ordenação de Ações *Resolvedoras*** e **Ordenação de Proposições que leva ao Menor Número de *Resolvedores***. Otimizações selecionadas: **Mutexes Estáticos**, **Memoização** e **Inferência de Tipos**. Além destas, utilizou-se da **Hipótese do Mundo Fechado (*Relaxada*)**, pois o domínio *Dock Worker Robots* possui pré-condições negativas em algumas de suas ações.

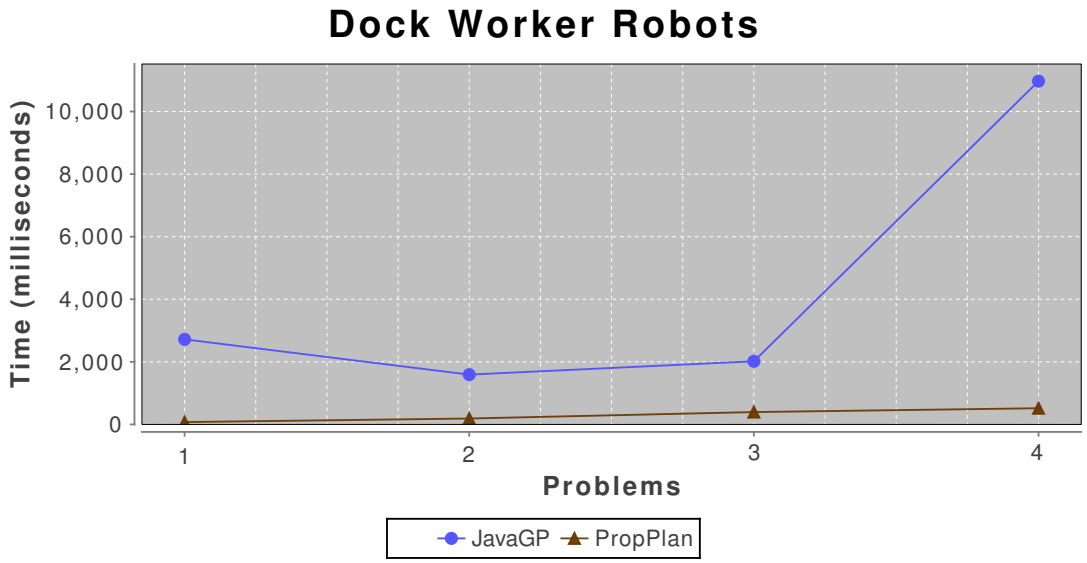

Figura 11.10: Benchmark - *Dock Worker Robots*.

Após comparações, nota-se que o JavaGP tem um desempenho pior que os demais para problemas grandes. Mesmo com esse fato, o JavaGP mostrou-se competitivo com planejadores implementados por profissionais.

12 Conclusões

A experiência de estudar e modificar um código fonte de implementação do GraphPlan foi de grande valor, pois nos proporcionou agregar um bom conhecimento sobre planejamento, *Design Patterns* e boas práticas de algoritmos, além da participação em um projeto *Open Source*. O projeto Emplan possui uma ótima documentação, o que não dificultou a implementação das otimizações e heurísticas no JavaGP. Outro ponto interessante, foi lidar com planejadores desenvolvidos por profissionais da área, comparando-os com o JavaGP, e estudar técnicas de otimizações realizadas por cientistas renomados.

O desenvolvimento de otimizações e correções agregou bastante valor ao projeto Emplan (JavaGP), como as heurísticas e compatibilidade a linguagem PDDL, visto que a versão original possuía alguns erros e interpretava somente a linguagem STRIPS. Com isso, a versão atual do JavaGP pode ser considerada confiável e competitiva com os demais planejadores utilizados neste trabalho, conforme visto nos resultados.

Para trabalhos futuros pretende-se implementar as outras otimizações e algumas refatorações de código. Além disso, alguns experimentos serão realizados, como por exemplo a manutenção do Grafo de Planejamento ao utilizar a Hipótese do Mundo Fechado, ao invés de eliminar por completo o Grafo de Planejamento.

Como o JavaGP faz parte do projeto Emplan, o código fonte e a última versão do planejador estão disponíveis em `http://sourceforge.net/projects/emplan/files/Release_2_Java/`.

A PDDL do Dock Worker Robots

A.1 Domínio

```
(define
       (domain DockWorkRobots)
       (:requirements :strips :typing :negative-preconditions)
       (:types location pile robot crane container)
       (:predicates
              (adjacent ?l1 ?l2 - location)
              (attached ?p - pile ?l- location)
              (belong ?k - crane ?l- location)
              (at ?r- robot ?l- location)
              (occupied ?l- location)
              (loaded ?r- robot ?c - container)
              (unloaded ?r- robot)
              (holding ?k - crane ?c - container)
              (empty ?k - crane)
              (in ?c - container ?p - pile)
              (top ?c - container ?p - pile)
              (on ?k1 - container ?k2 - container))

       (:action move
              :parameters(?r-robot ?from ?to-location)
              :precondition( and(adjacent ?from ?to)
                           (at ?r ?from) (not (occupied ?to)))
              :effect( and(at ?r ?to) (not(occupied :from))
                           (occupied ?to) (not (at ?r ?from)))

       (:action load
              :parameters(?k-crane ?c-container  ?r-robot)
              :vars(?l - location)
              :precondition( and (at ?r ?l) (belong ?k ?l)
                                 (holding ?k ?c) (unloaded ?r))
              :effect( and (loaded ?r ?c) (not (unloaded ?r))
                           (empty ?k) (not (holding ?k ?c))))
```

```
        (:action unload
               :parameters(?k-crane ?c-container  ?r-robot)
               :vars(?l - location)
               :precondition( and (at ?r ?l) (belong ?k ?l)
                                  (loaded ?r ?c) (empty ?k))
               :effect( and (unloaded ?r) (holding ?k ?c)
                            (not (loaded ?r ?c) (not (empty?k))))

        (:action take
               :parameters(?k-crane ?c-container  ?p-pile)
               :vars(?l-location  ?else-container)
               :precondition( and (belong ?k ?l) (attached ?p ?l)
                            (empty ?k) (in ?c ?p)
                            (top ?c ?p) (on ?c ?else))
               :effect( and (holding ?k ?c)
                            (top ?else ?p) (not (in ?c ?p))
                            (not (top ?c ?p)) (not (on ?c ?else))
                            (not (empty ?k))))

        (:action put
               :parameteres(?k-crane  ?c-container  ?p-pile)
               :vars(?else-container  ?l-location)
               :precondition( and (belong ?k ?l) (attached ?p ?l)
                            (holding ?k ?c) (top ?else ?p))
               :effect( and (in ?c ?p) (top ?c ?p) (on ?c ?else)
                      (not (top ?else ?p))
                      (not (holding ?k ?c)) (empty ?k)))
)
```

B PDDL do Blocks World

B.1 Domínio

```
(define (domain blocksworld)
(:requirements :strips :equality)
(:predicates
       (clear ?x)
       (onTable ?x)
       (holding ?x)
       (on ?x ?y)
)

(:action pickup
       :parameters (?ob)
       :precondition (and (clear ?ob) (onTable ?ob))
       :effect (and (holding ?ob) (not (clear ?ob))
              (not (onTable ?ob)))
)
(:action putdown
       :parameters (?ob)
       :precondition (and (holding ?ob))
       :effect (and (clear ?ob) (onTable ?ob)
              (not (holding ?ob)))
)
(:action stack
       :parameters (?ob ?underob)
       :precondition (and (clear ?underob) (holding ?ob))
       :effect (and (clear ?ob) (on ?ob ?underob)
              (not (clear ?underob)) (not (holding ?ob)))
)
(:action unstack
       :parameters (?ob ?underob)
       :precondition (and (on ?ob ?underob) (clear ?ob))
       :effect (and (holding ?ob) (clear ?underob)
              (not (on ?ob ?underob)) (not (clear ?ob)))
))
```

C PDDL do Gripper

C.1 Domínio

```
(define (domain gripper)
(:requirements :strips)
(:predicates
       (room ?r)
       (ball ?b)
       (gripper ?g)
       (atRobby ?r)
       (at ?b ?r)
       (free ?g)
       (carry ?o ?g)
)

(:action move
       :parameters (?from ?to)
       :precondition (and  (room ?from)
              (room ?to) (atRobby ?from))
       :effect (and  (atRobby ?to) (not (atRobby ?from)))
)
(:action pick
       :parameters (?obj ?room ?gripper)
       :precondition (and (ball ?obj)
              (room ?room) (gripper ?gripper)
              (at ?obj ?room) (atRobby ?room) (free ?gripper))
       :effect (and (carry ?obj ?gripper)
                     (not (at ?obj ?room))
                     (not (free ?gripper)))
)
(:action drop
       :parameters (?obj ?room ?gripper)
       :precondition (and (ball ?obj)
              (room ?room) (gripper ?gripper)
              (carry ?obj ?gripper) (atRobby ?room))
       :effect (and (at ?obj ?room) (free ?gripper)
              (not (carry ?obj ?gripper)))
))
```

D PDDL do Hanoi

D.1 Domínio

```
(define (domain hanoi)
(:requirements :strips)
(:predicates
       (clear ?x)
       (on ?x ?y)
       (smaller ?x ?y)
)
(:action move
       :parameters (?disc ?from ?to)
       :precondition (and (smaller ?to ?disc)
              (on ?disc ?from) (clear ?disc) (clear ?to))
       :effect (and (clear ?from) (on ?disc ?to)
              (not (on ?disc ?from)) (not (clear ?to)))
))
```

Referências Bibliográficas

[1] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300, 1997.

[2] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason.* Wiley, 2007.

[3] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[4] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the Classical Part of the 4th International Planning Competition. Technical Report 195, January 2004.

[5] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

[6] Maria Fox and Derek Long. The automatic inference of state invariants in tim. *J. Artif. Intell. Res. (JAIR)*, 9:367–421, 1998.

[7] Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In Thomas Dean, editor, *IJCAI*, pages 956–961. Morgan Kaufmann, 1999.

[8] Maria Fox and Derek Long. Utilizing automatically inferred invariants in graph construction and search. In Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *AIPS*, pages 102–111. AAAI, 2000.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice.* Elsevier, 2004.

[11] Subbarao Kambhampati and Romeo Sanchez Nigenda. Distance-based goal-ordering heuristics for graphplan. In *AIPS*, pages 315–322, 2000.

[12] Craig A. Knoblock, Steven Minton, and Oren Etzioni. Integrating abstraction and explanation-based learning in prodigy. In Thomas L. Dean and Kathleen McKeown, editors, *AAAI*, pages 541–546. AAAI Press / The MIT Press, 1991.

[13] Felipe Rech Meneguzzi. Planejamento proposicional em agentes bdi. Master's thesis, PPGC-C/PUCRS, 2003.

[14] Steven Minton, John Bresina, and Mark Drummond. Total-order and partial-order planning: A comparative analysis. *Journal of Artificial Intelligence Research*, 2:227–262, 1994.

[15] Bernhard Nebel, Charles Rich, and William R. Swartout, editors. *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92). Cambridge, MA, October 25-29, 1992*. Morgan Kaufmann, 1992.

[16] J. Scott Penberthy and Daniel S. Weld. Ucpop: A sound, complete, partial order planner for adl. In Nebel et al. [15], pages 103–114.

[17] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[18] Daniel S. Weld. Recent advances in ai planning. *AI Magazine*, 20(2):93–123, 1999.