UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PLANEJAMENTO BASEADO EM BUSCA E APRENDIZAGEM

RAULCÉZAR MAXIMIANO FIGUEIRA ALVES

Uberlândia - Minas Gerais
2014

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RAULCÉZAR MAXIMIANO FIGUEIRA ALVES

PLANEJAMENTO BASEADO EM BUSCA E APRENDIZAGEM

Dissertação de Mestrado apresentada à Faculdade de Ciência da Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Inteligência Artificial.

Orientador:
Prof. Dr. Carlos Roberto Lopes

Uberlândia, Minas Gerais
2014

DADOS DE CATALOGAÇÃO

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Ciência da Computação a aceitação da dissertação intitulada "**Planejamento Baseado em Busca e Aprendizagem**" por **Raulcézar Maximiano Figueira Alves** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 10 de Fevereiro de 2014

Orientador: _____

Prof. Dr. Carlos Roberto Lopes
Universidade Federal de Uberlândia

Banca Examinadora:

_____

Prof. Dr. José Jean-Paul Z. de S. Tavares
Universidade Federal de Uberlândia

_____

Profª. Drª. Leliane Nunes de Barros
Universidade de São Paulo

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Fevereiro de 2014

Autor:	**Raulcézar Maximiano Figueira Alves**
Título:	**Planejamento Baseado em Busca e Aprendizagem**
Faculdade:	**Faculdade de Ciência da Computação**
Grau:	**Mestrado**

_____

Autor

Dedicatória

Aos meus pais Vagnete e Zoraene e meus irmãos Jozaene e Victor.
A minha namorada Cristiane.

Agradecimentos

Agradeço...

A Deus, por minha vida.

Aos meus pais Vagnete Fagundes Ferreira e Zoraene Carmen Figueira Fagundes pela dedicação, confiança, apoio, carinho e amor incondicional em todos os momentos.

Aos meus irmãos Jozaene Maximiano Figueira Alves Faria e Victor Figueira Fagundes pela serenidade, parceria, carinho e amor.

A minha namorada Cristiane Soares Campos por ter me apoiado durante este projeto. Você me deu muita força, alegria e amor.

A todos da minha família que sempre me apoiaram e acreditaram em mim.

Aos meus amigos do Programa de Pós-Graduação em Ciência da Computação da UFU por terem feito dessa caminhada algo mais descontraído e prazeroso.

Aos meus amigos do Centro de Tecnologia da Informação da UFU pelo incentivo e amizade ao longo desta etapa da minha vida.

A professora Márcia Aparecida Fernandes pelo apoio, orientação e carinho desde a minha graduação.

Principalmente ao professor Carlos Roberto Lopes pelo profissionalismo, apoio, paciência, amizade e sobretudo por me guiar durante minhas buscas.

"Um homem pinta com seu cérebro e não com suas mãos."
(Michelangelo)

Resumo

Durante décadas, vários sistemas de planejamento vêm sendo propostos a fim de encontrar soluções para os mais diversos problemas do dia-a-dia. Porém, apenas recentemente, sistemas de planejamento começaram a obter resultados satisfatórios através de métodos de busca para construir seus planos. Como exemplo de planejador de sucesso, pode-se citar o sistema *Fast Forward* que combina a execução de dois métodos de busca, o *Enforced Hill Climbing* e o *Best First Search*. Técnicas introduzidas pelo *Fast Forward* tem sido também aplicadas em muitos outros planejadores mais recentes.

Apesar da estratégia do *Fast Forward* apresentar um desempenho superior quando comparado a outros métodos, ela também mostra algumas fragilidades. Para alguns problemas de planejamento, o *Enforced Hill Climbing* não consegue escapar de máximos/mínimos locais, o que o leva a ficar preso em "becos sem saída", fazendo com que ele falhe. Caso isso ocorra, uma busca completa é realizada pelo *Best First Search*, que pode falhar por falta de espaço de memória. Note que esses problemas podem afetar qualquer algoritmo de busca.

Este trabalho descreve o desenvolvimento de um planejador, chamado de *SLPlan*, baseado em versões otimizadas dos algoritmos *Enforced Hill Climbing* e *Learning Real-Time A** que tentam reduzir o impacto dos problemas descritos acima para construir planos de forma mais eficiente.

Além disso, a aplicação de técnicas de planejamento na solução de problemas relevantes é tema de especial interesse pela comunidade de planejamento. A geração de processos de negócio (*workflow*) é uma das áreas que podem se beneficiar do uso de planejamento. Nesta dissertação especifica-se também como o planejador desenvolvido foi aplicado na geração automática de modelos de processos de negócio.

Palavras chave: sistemas de planejamento, fast forward, enforced hill climbing, best first search, learning real-time a*.

Abstract

For decades, several planning systems have been proposed in order to find solutions to many of the different problems encountered in everyday life. However, only in recent years, have planning systems started to obtain satisfactory results through the use of search methods for building plans. As a successful planner, *Fast Forward* combines the execution of two search methods, *Enforced Hill Climbing* and *Best First Search*. Techniques introduced by *Fast Forward* have also been applied in many other recent planners.

Although *Fast Forward* strategy presents an enhanced performance when compared to alternative methods, it shows some weaknesses. For some planning problems *Enforced Hill Climbing* cannot escape from local maxima/minima, which means it might get stuck at dead ends, thus leading to a failure. In the event of such occurrences, a complete search is performed by *Best First Search*, which may fail due to lack of memory space. Note that these troubles could affect any search algorithm.

This paper describes the development of a planner, called *SLPlan*, based on optmized versions of the algorithms *Enforced Hill Climbing* and *Learning Real-Time A** that aim to reduce the impact of the troubles cited before in order to build plans more efficiently.

Besides, the application of planning techniques in solving relevant problems is of particular concern for the planning community. The generation of business processes (*workflow*) is one of the areas that can benefit from the use of planning. This dissertation also specifies how *SLPlan* was applied to the automatic generation of business processes models.

Keywords: planning systems, fast forward, enforced hill climbing, best first search, learning real-time a*.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

A*	Algoritmo para busca de caminhos ótimos em grafos
ADL	*Action Description Language*
AIPS	*Artificial Intelligence Planning Systems*
BFS	*Best First Search*
BPMN	*Business Process Modeling Notation*
EHC	*Enforced Hill Climbing*
FF	*Fast Forward*
HC	*Hill Climbing*
IA	Inteligência Artificial
ICAPS	*International Conference on Automated Planning Scheduling*
IPC	*International Planning Competition*
LRTA*	*Learning Real-Time A**
PDDL	*Planning Domain Definition Language*
STRIPS	*STanford Research Institute Problem Solver*
WfMC	*Workflow Management Coalition*
XML	*eXtensible Markup Language*
XPDL	XML *Process Definition Language*

Capítulo 1

Introdução

Planejamento é um campo da Inteligência Artificial que tem como objetivo controlar sistemas complexos de forma autônoma. Esta abordagem se tornou muito usada para resolver problemas práticos como os encontrados nas áreas de: logística, exploração espacial, jogos, projetos, robótica, etc. Sistemas de planejamento, ou simplesmente planejadores, visam gerar uma sequência de ações, denominada plano, necessária para alcançar a meta, solução do problema, a partir de uma configuração inicial do mesmo.

Os planejadores atuais usam técnicas de busca para selecionar ações para construir seus planos. Os grandes desafios desses planejadores envolvem o melhoramento de algoritmos de busca e a construção de heurísticas de planejamento usadas para guiar tais algoritmos. Um planejador de referência que faz uso dessa abordagem é o *Fast Forward* (FF) [Hoffmann e Nebel 2001]. Ele é composto por dois algoritmos de busca, o *Enforced Hill Climbing* (EHC) [Hoffmann e Nebel 2001] e o *Best First Search* (BFS) [Coppin 2004]. Esses algoritmos são guiados por uma heurística derivada de grafos de planos relaxados, extraídos a partir dos problemas a serem resolvidos.

O FF obteve o primeiro lugar na segunda competição internacional de planejamento. Vários planejadores de sucesso surgiram a partir das ideias propostas pelo FF, como: Metric-FF [Hoffmann 2002], Conformant-FF [Hoffmann e Brafman 2006], Contingent-FF [Albore et al. 2009], Probabilistic-FF [Domshlak e Hoffmann 2007], MACRO-FF [Botea et al. 2005], MARVIN [Coles e Smith 2007], POND [Department 2006], JavaFF [Coles et al. 2008], FD [Helmert 2006] e LAMA [Richter e Westphal 2010].

Após executar experimentos utilizando o método proposto pelo FF, em alguns problemas usados nas competições internacionais de planejamento, foi possível identificar algumas situações que podem prejudicar a execução de algoritmos de buscas e, consequentemente, degradar a performance de sistemas de planejamento. Para alguns problemas o EHC não consegue escapar de máximos/mínimos locais, ficando preso em "becos sem saída", o que faz com que ele venha a falhar. Quando isso ocorre, uma busca completa é realizada pelo BFS que pode vir a falhar por falta de espaço de memória.

Os problemas identificados podem ocorrer em qualquer método de busca. Diante

disso, foi desenvolvido o planejador *Searching and Learning - Planner* (SLPlan), descrito neste trabalho, que utiliza dois métodos de busca baseados nos algoritmos *Enforced Hill Climbing* (EHC) e *Learning Real-Time A** (LRTA*) [Korf 1990]. Esses métodos sofreram adaptações a fim de mitigar os impactos causados pelos problemas identificados durante os experimentos. Desse modo, um dos objetivos deste trabalho é analisar a eficiência do planejador desenvolvido, para que ele possa ser utilizado na resolução de problemas de planejamento independente do domínio.

Como resultado desta análise, foi possível constatar uma redução significativa no tempo médio e na taxa de falhas por falta de espaço em memória, durante a geração dos planos, utilizando os métodos desenvolvidos neste trabalho.

Atualmente, outro problema comum de extrema relevância, que vem sendo abordado nas organizações é a modelagem de processos de negócio (*workflow*), estudado no Gerenciamento de Processos de Negócio (do inglês *Business Process Management* (BPM)) [Weske 2007]. Processos de *workflow* podem ser facilmente relacionados a problemas de planejamento, onde se tem inicialmente um conjunto de tarefas/ações que devem ser colocadas em uma sequência lógica (*workflow/plano*) a fim de satisfazer um dado objetivo [Casati et al. 1996]. Geralmente, processos de negócio são modelados manualmente com ajuda de um editor, porém esta abordagem demanda tempo e é passível de erros. Então, levando em consideração a parte prática do uso de planejamento, outro objetivo deste trabalho foi aplicar o planejador SLPlan na geração automática de *workflows*.

1.1 Contribuição

As principais contribuições deste trabalho são:

- implementação do planejador SLPlan baseado nos algoritmos EHC e LRTA*;
- utilização do planejador SLPlan como motor de geração automática de *workflow*.

1.2 Organização da Dissertação

O Capítulo 2 apresenta conceitos básicos de busca e planejamento em Inteligência Artificial, juntamente com os principais trabalhos relacionados ao contexto da dissertação.

O Capítulo 3 propõe um novo planejador, descrevendo sua implementação, além dos algoritmos usados por ele.

Na primeira parte do Capítulo 4 é descrito o estudo comparativo realizado entre os algoritmos desenvolvidos e versões semelhantes a eles. Já na segunda parte, foi feita uma comparação entre o planejador desenvolvido e o planejador FF.

O Capítulo 5 mostra a aplicação do novo planejador em um sistema de *workflow*.

Por fim, o Capítulo 6 conclui o trabalho trazendo perspectivas de trabalhos futuros.

Capítulo 2

Fundamentos Teóricos e Trabalhos Correlatos

Neste capítulo são discutidos os fundamentos teóricos e trabalhos correlatos, objetivando propiciar uma melhor compreensão do trabalho desenvolvido. Como o trabalho consistiu na construção de um sistema de planejamento composto por algoritmos de busca, este capítulo foi estruturado da seguinte forma: na Seção 2.1 são descritos conceitos e alguns algoritmos de busca relacionados ao tema; na Seção 2.2 são apresentados conceitos e alguns sistemas de planejamento, bem como seus avanços obtidos ao longo da história.

2.1 Busca

Uma busca consiste em determinar uma sequência de passos que leva de um estado inicial até um estado meta, cuja complexidade está relacionada ao tamanho do espaço de busca. Algoritmos de busca são fundamentais para a solução de muitos problemas de Inteligência Artificial (IA).

A solução de um problema de busca é composta por uma sequência de estados, sendo que cada estado representa parte da solução. Estes estados são divididos em: estado inicial, estados intermediários e estado meta. Para o conjunto de estados dado como solução de um problema, a partir do estado inicial, os estados intermediários são percorridos até alcançar o estado meta. Entre cada estado pode-se associar um custo/distância. Normalmente, a melhor solução consiste no somatório máximo ou mínimo deste parâmetro.

Um algoritmo de busca gera múltiplas escolhas de caminho, dentre as quais ele opta por uma delas e deixa as outras para serem analisadas posteriormente, onde a decisão de qual será escolhida primeiro define a estratégia de busca.

Existem alguns critérios para se avaliar uma estratégia de busca, como:

- Completude: se existe uma solução a estratégia será capaz de encontrá-la?
- Qualidade da solução: a solução encontrada é a solução ótima?

- Complexidade de tempo: quanto tempo ela demora para obter a solução?
- Complexidade de espaço: de quanta memória ela necessita para realizar a busca?

O processo de busca pode ser comparado ao processo de se construir uma árvore de busca cuja raiz é o estado inicial, os estados folhas correspondem a estados que não possuem sucessores devido ao fato de não terem sido expandidos ainda ou porque foram expandidos e geraram um conjunto vazio. E a cada passo, o algoritmo busca um estado folha para expandir, até que a meta seja encontrada [Russell e Norvig 2009].

2.1.1 Busca Cega

A busca cega, também conhecida como busca não-informada, não possui nenhuma informação adicional sobre o problema que ajude a guiar a busca. Portanto, tudo que se pode fazer é expandir um estado, ou seja, gerar estados sucessores a partir de um outro estado qualquer, e distinguir um estado meta de um não-meta. Os algoritmos mais conhecidos que fazem uso dessa estratégia são:

- *Busca em largura*: é uma estratégia simples onde o estado inicial é expandido primeiro, e depois seus sucessores, os sucessores de seus sucessores, e assim por diante, até que se encontre a meta. Os estados expandidos são armazenados em um fila *FIFO*, onde ao serem gerados, são colocados no fim da fila, e o primeiro (cabeça) é escolhido para ser expandido [Russell e Norvig 2009].
  - Completude: é completa se o estado meta estiver em uma profundidade finita d.
  - Qualidade da solução: é ótima se os custos das transições entre os estados são todos iguais.
  - Complexidade de tempo: considerando que a cada nível da árvore de busca, um estado gere b sucessores (fator de ramificação), tem-se uma sequência de custos nos níveis de: $b + b^2 + b^3 + ... + b^d = O(b^d)$, onde d é o nível onde se encontra a meta e o algoritmo para. Isso se deve ao fato de que um estado é avaliado se é meta ou não após sua geração, ou seja, a cada sucessor gerado durante a expansão é feito um teste de meta sobre ele, com isso a busca sempre é interrompida no nível em que a meta é encontrada.
  - Complexidade de espaço: da mesma forma que a complexidade de tempo, também se gasta $O(b^d)$ para se armazenar os estados gerados durante a busca.
- *Busca em profundidade*: os estados expandidos nessa estratégia são armazenados em uma fila *LIFO*, onde ao serem gerados são colocados na cabeça da fila, e o estado a ser expandido é retirado também da cabeça da fila. As propriedades da busca em profundidade dependem da estratégia usada [Russell e Norvig 2009].

  - Completude: a versão da busca em profundidade em grafos é completa, sendo o espaço de estados finito, já que ela evita estados repetidos e caminhos redundantes, pois eventualmente todos os estados podem ser expandidos. Mas essa busca em árvore não é completa, devido a possível ocorrência de *ciclos*, que até podem ser evitados com uso de memória para o algoritmo, mas que não impede o aparecimento de caminhos redundantes.
  - Qualidade da solução: mesmo que a busca seja completa, ela pode ser obrigada a descer por ramos mais profundos à esquerda, antes de encontrar a meta em um ramo mais a direita, dessa forma ambas as versões não podem ser ótimas.
  - Complexidade de tempo: considerando a situação acima, a busca terá no máximo $O(b^m)$ de tempo, sendo que m é a profundidade máxima da árvore de busca.
  - Complexidade de espaço: após avaliar todos os estados de um ramo a esquerda e não encontrar a solução, o algoritmo pode ir para um ramo da direita e retirar os estados do outro ramo da memória, já que eles não serão mais necessários para a busca. Logo a complexidade de tempo é $O(b \times m)$.

- *Busca em profundidade limitada*: a falha da busca em profundidade em espaço de estados infinito, pode ser contornada fixando um limite l máximo de profundidade. Mas existe um inconveniente que é quando a meta se encontra em níveis inferiores ao limite estipulado, ou seja, $d > l$. A busca em profundidade convencional pode ser vista como uma particularidade da busca em profundidade limitada onde $l = \infty$ [Russell e Norvig 2009].

  - Completude: devido ao fato da meta poder estar em níveis inferiores ao limite l, a busca não é considerada completa.
  - Qualidade da solução: por ela não ser completa, consequentemente ela não pode ser ótima também.
  - Complexidade de tempo: como l funciona como profundidade máxima para a busca, o tempo é de $O(b^l)$.
  - Complexidade de espaço: pelo mesmo motivo que a complexidade de tempo, a complexidade de espaço é no máximo $O(b \times l)$, considerando as podas feitas em ramos mais a esquerda.

- *Busca em profundidade iterativa*: faz um incremento gradual no limite da profundidade da busca até que a solução seja encontrada, que ocorre quando o limite chega em d, ou seja, no nível onde se encontra a meta. Ela combina os benefícios das buscas em profundidade e largura [Russell e Norvig 2009].

  - Completude: é completa assim como a busca em largura, desde que, o fator de ramificação b seja finito.

  - Qualidade da solução: é ótima se os custos são não decrescentes.
  - Complexidade de tempo: consome $O(b^d)$ de tempo como a busca em largura.
  - Complexidade de espaço: consome $O(b \times d)$ como a busca em profundidade convencional, sendo que d neste caso é sempre a profundidade máxima da busca.

- *Busca bidirecional*: são feitas duas buscas simultâneas, uma progressiva partindo do estado inicial e outra regressiva partindo da meta, e se espera que as duas se encontrem no "meio". A motivação para isso é que, $b^{d \div 2}$ de cada busca é melhor que b^d de uma busca completa de um lado ao outro. Ela é implementada substituindo o teste de meta por um teste que verifica quando as fronteiras das duas buscas se interceptam. A dificuldade de se usar essa busca é que, dependendo do problema, a meta nem sempre é bem conhecida e clara, e talvez as transições para estados predecessores na busca regressiva podem ser difíceis de serem determinadas [Russell e Norvig 2009].

  - Completude: é completa se o fator de ramificação é finito e ambas as direções usam busca em largura.
  - Qualidade da solução: é ótima se os custos são iguais e ambas as direções usam busca em largura.
  - Complexidade de tempo: consome $O(b^{d \div 2})$ de tempo com as buscas em largura.
  - Complexidade de espaço: assim como na complexidade de tempo, consome-se $O(b^{d \div 2})$ para as buscas em largura.

- *Busca de custo uniforme*: expande o estado com menor custo até o momento, isso é feito armazenando os estados em uma *fila de prioridade* ordenada pelo custo acumulado $g(n)$ para se chegar em cada estado. Um dos primeiros algoritmos a usar essa estratégia foi o algoritmo de Dijkstra [Dijkstra 1959] [Russell e Norvig 2009].

  - Completude: é completa se o fator de ramificação é finito e os custos $\geq \ell$ (sendo ℓ positivo).
  - Qualidade da solução: é ótima, pois diferentemente da busca em largura, a busca de custo uniforme faz obrigatoriamente o teste de meta ao selecionar um estado para ser expandido, e não durante sua criação. Isso impede que o algoritmo escolha soluções sub-ótimas.
  - Complexidade de tempo: se todos os custos forem iguais, pode-se ter uma complexidade igual a da busca em largura, mas pelo fato do teste de meta ser feito ao selecionar um estado para expansão, pode ocorrer a criação de um nível inferior ao da meta, logo $O(b^{d+1})$. Mas no pior caso, se exploraria mais

caminhos do que o necessário, pois a busca é guiada pelos custos dos estados, e não está diretamente ligada a profundidade onde a meta se encontra. Sendo C^* o custo para se chegar a solução ótima, e cada transição para se chegar até a meta tem um custo de ℓ, tem-se um complexidade de tempo igual a $O(b^{1+\lfloor C^* \div \ell \rfloor})$.

– Complexidade de espaço: assim como na complexidade de tempo, consome-se $O(b^{1+\lfloor C^* \div \ell \rfloor})$ de espaço.

2.1.2 Busca Heurística

A busca heurística, ou busca informada, utiliza um conhecimento específico do problema na escolha do próximo estado a ser expandido. Esse conhecimento pode ser representado por uma função heurística que estima o custo de se chegar ao estado meta a partir de um estado qualquer. Essa estimativa pode ser obtida por meio de uma versão relaxada do problema, ou seja, uma solução que contém menos restrições do que uma solução ideal. Uma heurística $h(s)$ serve para estimar a distância de um estado s até a meta, onde algoritmos de busca priorizam por expandir estados que tem menor valor de h, ou seja, estados que acredita-se estarem mais próximos à meta para que a busca seja executada o mais rápido possível.

Figura 2.1: Heurística como estimativa da distância até a meta

Considerando esta abordagem, existem dois tipos de algoritmos: *off-line* e *on-line* [Furcy 2004]. Nos algoritmos *off-line*, a fase deliberativa antecede totalmente a fase de execução, com isso, eles dispõem de todo tempo e espaço necessários para encontrar a solução, e só depois partir para a execução efetiva. Em contrapartida, os algoritmos *on-line*, também denominados de algoritmos de busca em tempo real, alternam as fases de deliberação e execução durante o processo de busca. Esses, possuem restrições de tempo e informação, ou seja, não se tem toda a informação sobre o espaço de busca, e após um certo tempo é necessário que ele retorne uma solução. De uma forma geral, algoritmos *off-line* tem um tempo de resposta maior que os *on-line*. Na sequência desta seção, serão listados alguns algoritmos que fazem uso de heurística para guiar seus processos de busca.

Hill Climbing (HC) é um algoritmo de melhoramento iterativo que se inicia com uma solução arbitrária do problema [Russell e Norvig 2009]. A partir desse estado, ele seleciona o primeiro estado sucessor cujo valor heurístico é melhor que o anterior, repetindo

esse processo até que nenhuma melhoria seja encontrada. O HC frequentemente produz melhores resultados do que outros algoritmos quando se tem um tempo restrito para se executar a busca, como no caso de sistemas de tempo real. Este método é bom para encontrar ótimos locais, mas não se tem garantias de se encontrar a melhor solução dentre todas as possíveis soluções para o problema. Este algoritmo não é completo, pois ele para quando nenhum sucessor do estado corrente tem valor heurístico melhor do que o seu, ou seja, ao entrar em um máximo/mínimo local.

Durante a fase de expansão do HC padrão, o primeiro sucessor, cujo valor heurístico é melhor do que o do estado corrente, é escolhido para continuar a busca. Ao invés disso, em uma adaptação do HC conhecido como *Steepest Ascent Hill Climbing* [Coppin 2004], todos os sucessores são avaliados, e só depois o melhor deles é escolhido. Este método funciona como uma busca em profundidade ordenada pela heurística. Mesmo com essa modificação, o algoritmo ainda pode entrar em um máximo/mínimo local onde a solução desejada pode não ser encontrada, sendo assim não completo.

Uma forma de resolver o problema de máximos/mínimos locais é a sucessiva exploração do espaço de estados, assim como é feito no *Random-Restart Hill Climbing*. Ele executa uma série de buscas com o HC, onde os estados iniciais são gerados aleatoriamente quando a busca entra em máximos/mínimos locais.

Já o *Enforced Hill Climbing* (EHC), é um método guiado por heurística que combina os algoritmos HC com busca em largura a fim escapar de máximos/mínimos locais. Ele foi introduzido pelo planejador Fast Forward (FF) [Hoffmann e Nebel 2001] como parte de sua estratégia de busca, e é um algoritmo muito usado nos planejadores atuais por ser extremamente ágil e tentar resolver problemas o mais rápido possível. Apesar de conseguir escapar de máximos/mínimos locais com frequência, ele ainda pode não encontrar a solução dependendo do caminho que a busca prosseguiu, ficando preso em um "beco sem saída" fazendo com que ele falhe. Devido ao fato dele fazer parte do planejador desenvolvido neste trabalho, porém com algumas otimizações, o EHC é descrito de forma mais detalhada a seguir, juntamente com seu pseudo-código (Algoritmo 1).

O algoritmo usa duas listas: *aberta* e $fechada$. Todos os estados que serão avaliados são armazenados na lista *aberta*. Os estados já avaliados são mantidos na lista $fechada$.

O EHC inicia instanciando o $s_{corrente}$ (estado corrente) como sendo o estado inicial (linha 1), e verifica se $s_{corrente}$ é a meta (linhas 2-4). A variável $melhorValor$ sempre guarda o melhor valor heurístico encontrado durante a busca, e é inicializada com o valor do estado corrente $h(s_{corrente})$ ainda no início do algoritmo (linha 5). Depois, o estado $s_{corrente}$ é inserido em ambas as listas, *aberta* (linha 6) e $fechada$ (linha 7).

O laço principal (linhas 8-27) varre todos os estados na lista *aberta*. Se esta lista se torna vazia, significa que a busca entrou em um "beco sem saída", ou seja, o algoritmo não foi capaz de inserir mais nenhum estado na lista pelo fato de que todos os estados que ele podia encontrar já foram visitados.

Algoritmo 1 EHC($s_{inicial}, s_{meta}$)

```
 1: s_corrente ← s_inicial
 2: se s_corrente = s_meta então
 3:     retorna sucesso
 4: fim se
 5: melhorValor ← h(s_corrente)
 6: fechada.insere(s_corrente)
 7: aberta.insere(s_corrente)
 8: enquanto aberta ≠ vazia faça
 9:     s_corrente ← aberta.remove()
10:     enquanto s' ∈ suc(s_corrente) faça
11:         se s' ∉ fechada então
12:             fechada.insere(s')
13:             se s' = s_meta então
14:                 retorna sucesso
15:             senão
16:                 se h(s') < melhorValor então
17:                     melhorValor ← h(s')
18:                     aberta ← {}
19:                     aberta.insere(s')
20:                     para laço, interrompe a expansão
21:                 senão
22:                     aberta.insere(s')
23:                 fim se
24:             fim se
25:         fim se
26:     fim enquanto
27: fim enquanto
28: retorna falha
```

Internamente é feita a expansão do estado corrente $s_{corrente}$ (linhas 10-26), que é obtido removendo o primeiro estado da lista *aberta* (linha 9). Durante essa expansão são considerados apenas estados que não foram visitados (linha 11). Então, para cada sucessor s' obtido, ele é inserido na lista $fechada$ (linha 12), e um teste é executado para verificar se ele não é a meta (linhas 13-15).

Após isso, ainda dentro da expansão, é verificado se o valor heurístico do sucessor é menor do que o melhor valor heurístico encontrado até o momento ($melhorValor$) (linha 16). Se for menor, $melhorValor$ será atualizado com esse novo valor (linha 17), a lista *aberta* é esvaziada, e somente o sucessor selecionado será mantido para que ele possa ser escolhido como estado corrente na próxima iteração (linhas 18-19). Então a expansão para (linha 20), economizando tempo, sendo que os próximos sucessores teriam valores heurísticos similares aos encontrados até o momento.

Porém, se o valor heurístico de s' não é menor que $melhorValor$, s' será mantido na lista $fechada$ (linha 22). Então, quando a busca entrar em um máximo/mínimo local, onde os valores heurísticos dos estados gerados param de decrescer, o algoritmo executa uma busca em largura a partir dos estados da lista $fechada$. Desta forma, ele continua realizando a busca até que ele encontre a meta ou um estado que tenha valor heurístico menor que $melhorValor$, o que faz com que ele escape do máximo/mínimo local e volte a sua execução normal.

Modificações vêm sendo feitas para melhorar a eficiência do EHC. O Guided Enforced Hill Climbing (GEHC) [Akramifar e Ghassem-Sani 2010] foi proposto com o objetivo de evitar "becos sem saída" durante a busca. Para isso, ele usa uma função que ordena os

estados a serem expandidos. Ele é mais rápido e examina menos estados que o EHC, apesar de que para alguns problemas a qualidade de suas soluções é um pouco inferior.

Uma generalização do EHC, conhecida como *Stochastic Enforced Hill Climbing* (SEH) [Wu et al. 2011], é um novo método de busca estocástica. Ao invés de buscar por uma sequência de passos determinísticos para encontrar um estado melhor, este algoritmo constrói uma busca em largura local sobre um *Processo de Decisão Markoviano* em torno do estado corrente, e espera-se sair desse processo com um estado de melhor valor.

O *Best First Search* (BFS) é um algoritmo de busca completo, *off-line* e similar ao *Steepest Ascent Hill Climbing* [Coppin 2004]. Porém, ao invés de escolher o melhor sucessor gerado durante a expansão, ele escolhe o melhor estado dentre todos os estados encontrados durante a busca, mas que ainda não foram explorados. Em termos práticos, isso significa que o BFS sempre segue o melhor caminho disponível na árvore de busca atual, ao invés da abordagem de busca em profundidade empregada pelos outros algoritmos baseados no HC. A implementação do BFS é idêntica a busca de custo uniforme, exceto pelo uso da função $h(n)$ (função heurística) ao invés de $g(n)$ (custo acumulado) para ordenar a fila de prioridade. Esse algoritmo não é completo, mas utilizando uma estrutura para armazenar os estados já visitados, geralmente denominada *lista fechada*, ou simplesmente *fechada*, ele consegue evitar *loops*, sendo assim completo. Assim como o EHC, o BFS é bastante usado para compor estratégias de busca nos planejadores da atualidade.

Talvez um dos algoritmos de busca mais conhecidos da IA seja o $A*$. Ele também é um algoritmo *off-line*, mas que combina busca de custo uniforme com busca heurística utilizando a seguinte função de avaliação para determinar quão bom é um estado: $f(n) = g(n) + h(n)$, sendo que $g(n)$ é o custo gasto até se chegar ao estado n e $h(n)$ é o custo estimado para se atingir o estado meta a partir de n. Este algoritmo foi descrito primeiramente por [Hart et al. 1968] sendo chamado de algoritmo A, mas quando usado com uma heurística apropriada (admissível, ou seja, não superestima o valor da solução real) atinge a solução ótima, e passa a se chamar A*. Apesar dos benefícios deste algoritmo, ele é impraticável em larga escala porque mantêm em memória todos os caminhos possíveis durante o processo de busca.

O algoritmo Learning Real Time A* (LRTA*) [Korf 1990] pode ser visto como uma versão *on-line* do A* que, durante a busca pela solução é atualizado o valor de heurística do estado corrente, fazendo com que ele "aprenda" se um dado caminho é bom ou não, e a medida em que ele é executado adquire-se mais informação sobre o ambiente, possibilitando a escolha de caminhos melhores. Como este algoritmo serviu de base para a criação de um novo método de busca que também faz parte da estratégia de busca implementada no planejador desenvolvido neste trabalho, será discutido um pouco mais sobre seu funcionamento (Algoritmo 2) e listado outros algoritmos adaptados a partir dele.

O LRTA* começa localizado no estado inicial s_{inicio}. No passo seguinte ele se desloca

Algoritmo 2 LRTA*(s_{inicio}, s_{meta})

```
1:  s_corrente ← s_inicio
2:  laço
3:      s' ← arg min_{s''∈succ(s_corrente)}(c(s_corrente, s'') + h(s''))
4:      se s_corrente = s_meta então
5:          h(s_corrente) ← h_0(s_corrente)
6:      senão
7:          h(s_corrente) ← max(h(s_corrente), min_{s''∈succ(s_corrente)}(c(s_corrente, s'') + h(s'')))
8:      fim se
9:      se s_corrente = s_meta então
10:         retornar sucesso
11:     fim se
12:     s_corrente ← s'
13: fim laço
```

para o estado sucessor mais promissor de acordo com sua função de avaliação. A busca termina quando se encontra o estado meta s_{meta}, sendo que a cada iteração, o algoritmo atualiza a estimativa da distância até a meta, melhorando a informação sobre o espaço de estados para auxiliar a busca. Sua execução pode ser descrita da seguinte forma. Um valor h é associado a cada estado. O valor $h_0(s)$ corresponde ao valor inicial de h para o estado s. Primeiro o estado inicial s_{inicio} é definido como sendo o estado corrente $s_{corrente}$. O LRTA* seleciona o estado para o qual deve se mover. Isso é feito avaliando o somatório mínimo $c(s_{corrente}, s'') + h(s'')$ dos estados adjacentes ao estado corrente. Se ocorrer a igualdade desta soma entre alguns estados, é escolhido um estado através de um sorteio (linha 3). A seguir, o valor de heurística do estado corrente $h(s_{corrente})$ é atualizado. A atualização é realizada com a melhor avaliação obtida (linhas 4-8). Posteriormente, o movimento é realizado para o estado selecionado (linha 12), e o procedimento realiza uma nova iteração. Ao final, quando a meta é encontrada, o LRTA* termina sua execução com sucesso. Para se chegar à solução ótima é necessário executar o algoritmo repetidas vezes, até que as heurísticas dos estados parem de sofrer atualizações.

Uma das primeiras vezes onde se tentou resolver problemas de planejamento através de algoritmos de busca heurística foi com o *Action Selection for Planning* (ASP). Esse método usa uma variação do LRTA* proposto por Korf, chamada de B-LRTA*, e uma heurística feita especificamente para problemas de planejamento. Ele intercala busca e execução, mas não chega a criar um plano, ou seja, não guarda as ações que ele executou para chegar até a solução, o que é necessário em um sistema de planejamento. O ASP [Bonet et al. 1997] é um mecanismo de seleção de ações que decide a cada passo qual ação deve ser executada. O B-LRTA* seleciona bons movimentos mais rápido, sem requerer várias repetições do algoritmo, onde basicamente ele simula n movimentos do LRTA*, e repete essa simulação m vezes, e somente então ele executa o movimento ou ação.

O LRTA*-k é um algoritmo baseado no LRTA* onde a única diferença se encontra na estratégia de atualização das heurísticas. O LRTA* atualiza a heurística de um único estado por iteração. Já o LRTA*-k [Hernández e Meseguer 2005] atualiza as heurísticas de até k estados, não necessariamente distintos, por iteração seguindo uma estratégia de propagação limitada. Em geral, o LRTA* pode ser visto como um caso particular do

LRTA*-k onde $k = 1$.

Atualizar mais de um estado por vez, como o LRTA*-k faz, pode resultar em um aprendizado mais eficiente. Seguindo essa mesma ideia, o *Prioritized*-LRTA* (P-LRTA*) [Rayner et al. 2007] não atualiza apenas a heurística do estado corrente, mas também as heurísticas de n estados armazenados em uma fila. No início, o algoritmo atualiza o estado corrente considerando apenas os vizinhos imediatos. Depois, cada vizinho do estado corrente é candidato a entrar em uma fila de atualização, considerando a diferença dada pela atualização da heurística do estado corrente. Uma vez que a atualização do estado corrente é feita, uma série de atualizações priorizadas se inicia, onde no máximo n estados são tomados do topo da fila e atualizados.

Já o *Local Search Space*-LRTA* (LSS-LRTA*) é uma versão do LRTA* que usa o A* para determinar um espaço de busca local e aprender mais rápido [Koenig e Sun 2009]. Ele usa o A* para realizar uma busca em direção à meta a partir do estado corrente enquanto um *lookahead* > 0 de estados tiverem sido expandidos, ou a meta esteja pronta pra ser expandida. Os estados expandidos pelo A* formam o espaço de busca local. Então, o algoritmo *Dijkstra* altera os valores heurísticos dos estados no espaço de busca local. Para finalizar, ele faz um movimento para o caminho encontrado pelo A* até que ele chegue ao fim deste caminho (saindo do espaço de busca local), ou até que os custos dos caminhos comecem a crescer.

Algoritmos de busca em tempo real, como LRTA*, empregam técnicas de aprendizagem que frequentemente guiam o processo para soluções pobres, que revisitam os mesmos estados repetidas vezes, especialmente nas primeiras iterações. O D-LRTA* [Bulitko et al. 2008] tenta resolver este problema selecionando automaticamente submetas. Isso é feito por uma fase de pré-processamento onde ele analisa um espaço de busca, e computa uma base de dados de submetas que serão usadas em tempo de execução pelo LRTA*. Porém, assim como na versão original do LRTA*, a heurística ainda é calculada considerando a meta global e não submetas.

Outra variação do LRTA* que também usa uma base de dados de submetas é o KNN-LRTA* [Bulitko e Björnsson 2009]. Mas ele é mais simples e mais eficiente em relação ao consumo de memória que o D-LRTA*, que demanda muita memória e possui um módulo de pré-processamento muito complexo. O KNN-LRTA* é baseado no conceito de "aprendizado preguiçoso" dos k vizinhos mais próximos (do inglês, K Nearest Neighbors-KNN). Uma vez que a base de dados de submetas é construída, o algoritmo busca na base casos similares ao problema que ele tem em mãos para resolver, e faz uso disso para sugerir submetas para o LRTA*.

A principal característica de algoritmos do tipo LRTA* é o aprendizado por meio de atualização das heurísticas dos estados. O *F-cost Learning Real-Time A** (F-LRTA*) [Sturtevant e Bulitko 2011] realiza o aprendizado para ambos os custos *g(n)* e *h(n)*. Como o algoritmo sabe de onde a busca vem (*g(n)*), ele é capaz de evitar erros que ele cometeu

anteriormente realizando podas em porções irrelevantes do espaço de estados. Além disso, aprender uma melhor estimativa de onde a busca está indo ($h(n)$), permite que o algoritmo siga uma estratégia mais agressiva em direção a meta.

2.2 Planejamento

Diariamente nos deparamos com problemas, para os quais devemos pensar em uma sequência lógica de ações para chegarmos a uma solução. A área de Planejamento se preocupa com exatamente isso. Planejamento é um processo deliberativo onde escolhemos, dentre todas as opções, as ações que desejamos executar, e as organizamos em uma estrutura chamada *plano* para que possamos ir de um estado inicial a um estado meta. O planejamento de ações é um aspecto fundamental do comportamento inteligente que vem sendo estudado desde a década de 70 [Newell e Simon 1963], e sua automatização é assunto que motiva diversas pesquisas em IA.

Atualmente, o planejamento automatizado tem-se tornado muito útil para resolução de problemas práticos em diversos domínios como: projeto, fabricação, logística, jogos, exploração espacial, entre outros. Formalmente, o objetivo do planejamento em IA é obter uma sequência de transformações para mover o sistema de um estado inicial até um estado meta, dando-se uma descrição das possíveis transformações.

O planejamento automatizado é realizado por sistemas computacionais chamados de sistemas de planejamento, ou simplesmente, planejadores. Um planejador, assim com um algoritmo de busca, deve receber um estado inicial e um estado meta como entrada. Além disso, ele deve receber um conjunto de ações que podem ser executadas no problema.

No planejamento, um estado do mundo pode ser representado por um conjunto de proposições verdadeiras $\mathcal{P}$. Já uma ação α pode ser representada por uma tupla da forma $(pre(\alpha), add(\alpha), del(\alpha))$, onde $pre(\alpha)$ é um conjunto de proposições que devem ser verdadeiras no estado em que a ação α é executada (pré-condições da ação); $add(\alpha)$ denota um conjunto de proposições que passarão a ser verdadeiras no estado após a execução da ação α (efeitos positivos da ação); e $del(\alpha)$ é o conjunto de proposições que passaram a ser falsas após a execução da ação α (efeitos negativos da ação).

Dado um estado $\mathcal{P}$ e uma ação α, dizemos que α é aplicável em $\mathcal{P}$ se e somente se $pre(\alpha) \subseteq \mathcal{P}$. Então, caso a ação seja aplicável, um novo estado resultante será criado com as proposições de $\mathcal{P} - del(\alpha) + add(\alpha)$, denotado por $res(\mathcal{P},\alpha)$.

Um problema de planejamento é definido por uma tupla da forma $(\mathcal{A}, I, \mathcal{G})$, onde $\mathcal{A}$ é o conjunto das possíveis ações que podem ser executadas no domínio do problema, I representa o estado inicial, e $\mathcal{G}$ representa a meta a ser alcançada.

Seja β um problema de planejamento do tipo $(\mathcal{A}, I, \mathcal{G})$, e γ uma sequência de ações $= (\alpha_1,\alpha_2,...,\alpha_n)$, dizemos que γ é uma solução (um *plano* solução) para o problema de planejamento β se contém apenas ações do domínio do problema, e é executável, ou seja,

$(\forall\, i / 1 < i \leq n, pre(\alpha_i) \subseteq I \wedge res(\alpha_i - 1, s), s \in G)$.

Como exemplo, na Figura 2.2 é apresentado uma instância de um problema de planejamento chamado "Mundo dos Blocos". O problema consiste em mover os blocos usando as ações *Mover* e *MoverParaMesa*. O objetivo é alterar o estado inicial do mundo para que seja obtido o estado meta. Os estados são representados por uma conjunção de predicados de primeira ordem, onde *Sobre(b,x)* diz que um bloco b está sobre x, sendo x um bloco ou a própria mesa, e *Livre(b)* diz que não há nada sobre o bloco b.

Os estados inicial e meta são representados graficamente pela Figura 2.2. Suas descrições na forma lógica são dadas abaixo:

- **Estado Inicial:** *Sobre(A,mesa)* $\wedge$ *Sobre(B,mesa)* $\wedge$ *Sobre(C,mesa)* $\wedge$ *Bloco(A)* $\wedge$ *Bloco(B)* $\wedge$ *Bloco(C)* $\wedge$ *Livre(A)* $\wedge$ *Livre(B)* $\wedge$ *Livre(C)* $\wedge$ *Livre(mesa)*.
- **Estado Meta:** *Sobre(A,B)* $\wedge$ *Sobre(B,C)* $\wedge$ *Sobre(C,mesa)*.

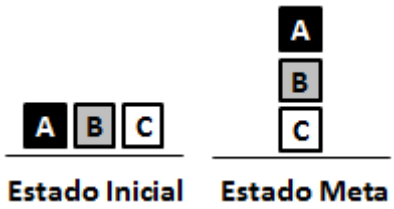

Figura 2.2: Exemplo: instância do problema "Mundo dos Blocos"

Como já foi dito, as ações são operadores que mudam um dado estado, e são compostas por pré-condição, que define quando a ação pode ser aplicada a um estado, e efeito, que adiciona e/ou remove características de um estado, gerando assim um novo estado. A ação *Mover(b,x,y)* move um bloco b que está sobre x para y, se ambos b e y estão livres. Um bloco livre significa que não existe outro bloco sobre ele. Depois que o movimento é feito, x fica livre mas y não. Quando $x = mesa$ a ação *Mover(b,x,y)* tem o efeito *Livre(mesa)*, mas a mesa não ficará livre (sem blocos). Entretanto, como citado em [Russell e Norvig 2009], consideramos que *Livre(mesa)* é verdadeira pois supomos que *Livre(mesa)* significa que existe espaço livre para colocar blocos sobre a mesa. A ação *MoverParaMesa(b,x)* move um bloco b que está sobre x para a *mesa*. Essas ações são descritas abaixo:

- **Mover(b,x,y):**
  - **pré-condição:** *Sobre(b,x)* $\wedge$ *Livre(b)* $\wedge$ *Livre(y)* $\wedge$ *Bloco(b)* $\wedge$ *Bloco(y)* $\wedge$ $(b \neq x) \wedge (b \neq y) \wedge (x \neq y)$
  - **efeito:** *Sobre(b,y)* $\wedge$ *Livre(x)* $\wedge\neg$ *Sobre(b,x)* $\wedge\neg$ *Livre(y)*
- **MoverParaMesa(b,x):**
  - **pré-condição:** *Sobre(b,x)* $\wedge$ *Livre(b)* $\wedge$ *Bloco(b)* $\wedge$ *Bloco(x)* $\wedge$ $(b \neq x)$

– **efeito:** $Sobre(b,mesa) \wedge Livre(x) \wedge\neg\ Sobre(b,x)$

Para evitar algumas redundâncias durante a geração do plano, algumas pré-condições são adicionadas as ações $(b \neq x,\ b \neq y,\ x \neq y,\ Bloco(b),\ Bloco(x),\ Bloco(y))$. As pré-condições $Bloco(b)$, $Bloco(x)$ e $Bloco(y)$, são responsáveis por exigir que cada elemento dado como entrada seja um bloco. Já as pré-condições $b \neq x$, $b \neq y$ e $x \neq y$, determinam que os elementos sejam diferentes. Com estas estratégias evitamos que sejam consideradas algumas ações redundantes. Por exemplo, a ação $MoverParaMesa(A,mesa)$ é redundante pois não altera o estado do mundo.

Assim como em uma busca, os estados podem ser expandidos gerando estados sucessores. A expansão de um estado é feita aplicando todas as ações do problema cujas pré-condições são satisfeitas pelas proposições do estado. A Figura 2.3 mostra passo-a-passo a geração de um novo estado ao aplicarmos a ação $Move(A,mesa,B)$ sobre estado inicial.

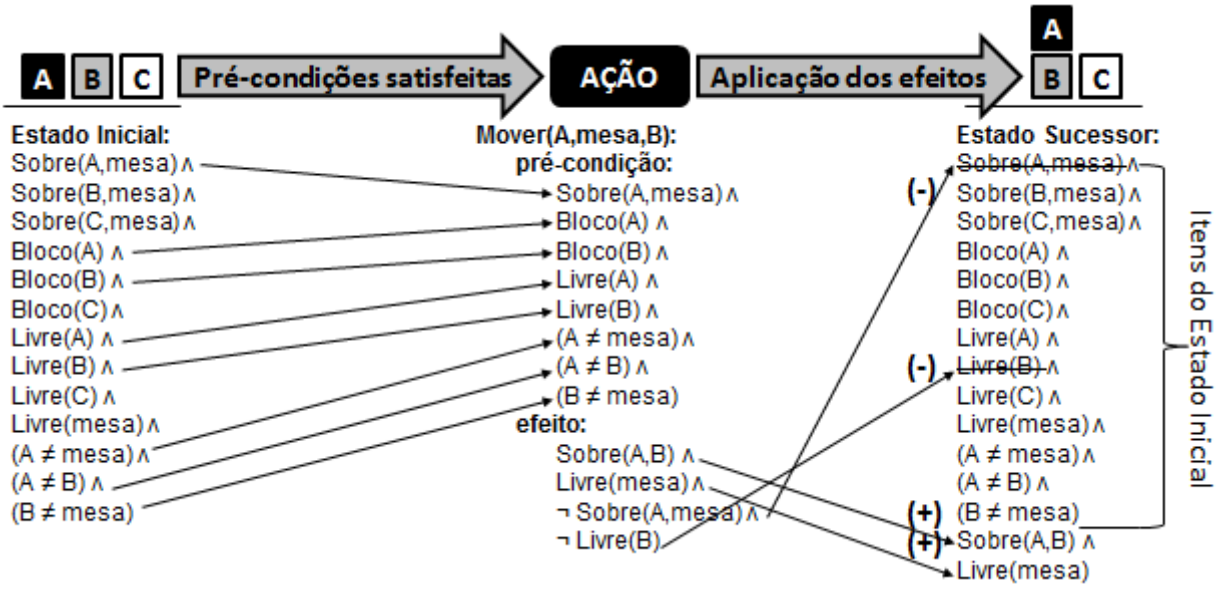

Figura 2.3: Exemplo: aplicando uma ação sobre o estado inicial

Para encontrar a solução de um problema, um planejador pode realizar sucessivas expansões a partir do estado inicial até encontrar o estado meta, guardando as ações usadas no caminho, construindo assim o *plano* (Figura 2.4).

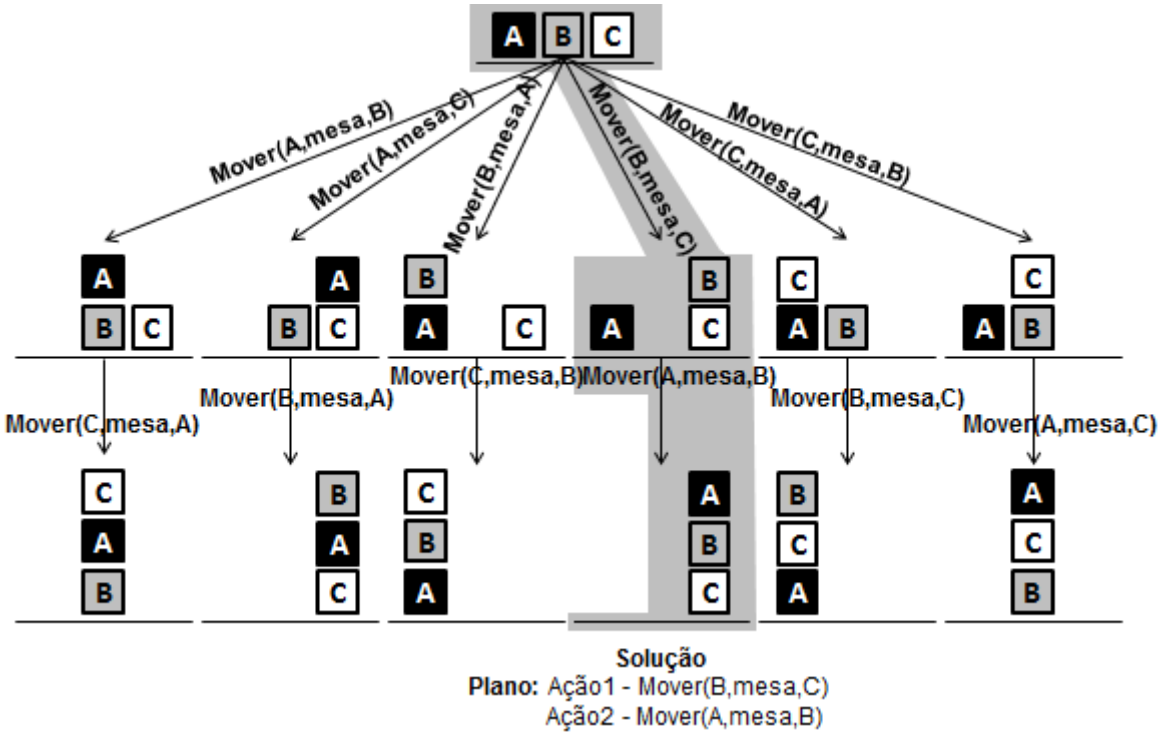

Figura 2.4: Exemplo: sucessivas expansões até encontrar a meta

2.2.1 Planejadores

Nessa seção é apresentado um breve histórico sobre sistemas de planejamento. Ela se inicia por listar alguns planejadores antigos e suas principais técnicas para resolver problemas de planejamento. Depois, são descritos alguns planejadores de sucesso que vêm utilizando diferentes métodos para construir seus planos.

Uma formulação clássica de problemas de planejamento define três entradas para o planejador, ou sistema de planejamento: a descrição do estado inicial em alguma linguagem formal, a descrição do estado meta dentro desta mesma linguagem e a descrição das possíveis ações que podem ser usadas no problema, também respeitando as regras da linguagem usada. A saída do planejador é uma sequência de ações chamado de plano, que ao ser executada em um certo domínio nos levará do estado inicial ao estado meta.

Por décadas, vários sistemas de planejamento foram propostos a fim de encontrar soluções para diferentes tipos de problema. Os primeiros esforços no desenvolvimento de sistemas de planejamento tinham o objetivo de realizar provas de teoremas usando *cálculo de situação* [Green 1969] ou *cálculo de eventos* [Kowalski e Sergot 1986]. Mas, no início da década de 70, um grande progresso foi feito na construção de planejadores computacionais para resolver problemas práticos usando representação de estados do mundo.

No entanto, somente ao final da década de 90 do século passado, os planejadores começaram a obter resultados satisfatórios com tempo computacional menor do que os planejadores desenvolvidos até então. Esses novos planejadores, denominados planejado-

res de busca heurística, iniciaram uma nova era nas pesquisas da área de planejamento.

Planejadores Clássicos

O estudo sobre planejadores originou-se a partir de trabalhos científicos que tentavam criar, principalmente com o uso de lógica de primeira ordem, solucionadores gerais de problemas nos mais diferentes domínios. Mas, nessa época, os planejadores eram tidos como métodos de busca aplicados à provadores de teoremas e, somente na década de 70, as pesquisas passaram a se concentrar no desenvolvimento de sistemas computacionais focados em descrição de estado do mundo.

Planejamento possui um conceito muito simples, onde um planejador começa em um estado inicial e tem uma meta particular que ele deve alcançar. Para chegar a meta, o planejador desenvolve um plano que servirá para uma execução futura [Coppin 2004].

O planejador *STandford Research Institute Problem Solver* (STRIPS) [Fikes e Nilsson 1971] foi o precursor dos atuais sistemas de planejamento, que introduziu as seguintes noções: estado do ambiente como sendo um conjunto de predicados, ações como sendo transformadores do ambiente, a ideia de decomposição da meta, e a busca baseada em *espaço de estados*, que tem como característica um processo de busca que percorre um grafo onde os nós são estados e as transições são feitas por meio das ações.

Durante uma busca, a cada passo, é necessário escolher um estado para que ele gere novos estados sucessores, dentre os quais um deles será escolhido novamente para continuar a busca. Este processo de gerar novos estados a partir de um estado atual é chamado de expansão, que no caso de planejamento é feito da seguinte forma: a partir de um estado composto por um conjunto de predicados é necessário escolher, dentre todas as ações possíveis, aquelas cujas pré-condições são satisfeitas pelos predicados do estado. A partir dessas ações selecionadas, elas são aplicadas sobre o estado, onde seus efeitos podem adicionar e/ou remover predicados, criando assim novos estados.

Apesar do STRIPS ter sido superado por inúmeros métodos de planejamento mais sofisticados, a linguagem que ele propôs para representar problemas de planejamento ainda é extremamente utilizada, mesmo com o desenvolvimento de novas linguagens de planejamento mais poderosas como *Action Description Language* (ADL) [Pednault 1989] e *Planning Domain Definition Language* (PDDL) [Mcdermott et al. 1998].

O planejador STRIPS possui algumas fraquezas, e talvez a mais relevante delas seja o fato dele tratar submetas como sendo independentes, o que causa um problema conhecido como *Anomalia de Sussman*. Como resultado, não há como garantir que o planejador possa alcançar a solução. A fim de resolver este problema, o planejador *Nets of Action Hierarchies* (NOAH) [Sacerdoti 1975] introduziu a ideia de planejamento hierárquico que usa uma rede abstrata de operadores, e o conceito de não-linearidade.

Outra tentativa de resolver o problema enfrentado pelo STRIPS foi a introdução da *teoria causal* pelo *System for Interactive Planning and Execution* (SIPE) [Wilkins 1988],

um planejador do tipo STRIPS. No SIPE ações possuem gatilhos, pré-condições, condições, e efeitos. Informalmente, quando um gatilho se torna "verdadeiro", o SIPE irá checar em sequência se as pré-condições são "verdadeiras" para a situação anterior, e se as condições são "verdadeiras" para a nova situação. Se todas essas condições são "verdadeiras", o planejador então irá deduzir os efeitos.

Planejadores com Busca Heurística

Mesmo com tantos avanços, os sistemas de planejamento citados na Seção 2.2.1 não obtiveram resultados satisfatórios ao tentar resolver problemas considerados simples, em um tempo computacional razoável. Mas graças às inúmeras e promissoras pesquisas relacionadas a algoritmos de busca, principalmente aqueles guiados por funções heurísticas, pesquisadores começaram a tentar resolver problemas de planejamento adaptando algoritmos e heurísticas dentro de planejadores a fim de melhorar a eficiência desses sistemas.

Heurística é um procedimento para resolver problemas através de um enfoque intuitivo, em geral racional, no qual a estrutura do problema passa a ser interpretada e explorada inteligentemente para obter uma solução razoável. Na IA, heurísticas são critérios, métodos, ou princípios para decidir entre vários cursos de ação alternativos, aquele que parece ser mais efetivo para atingir uma meta.

Heurísticas admissíveis (que não superestimam o valor real da solução) podem ser obtidas a partir de soluções ótimas de versões relaxadas do problema, ou seja, versões com menos restrições que o problema original. Dois exemplos clássicos de funções que podem ser usadas para calcular heurísticas são: distância Euclidiana e distância de Manhattan, que tem como objetivo estimar quantos passos serão gastos para chegar ao estado meta a partir do estado corrente. Mas encontrar uma boa função de heurística para problemas de planejamento não é tão trivial. Para ilustrar este cenário serão apresentados alguns planejadores que introduziram novas funções heurísticas para planejamento.

O GRAPHPLAN é um sistema muito simples, em linhas gerais, ele gera um grafo relaxado formado por ações e fatos (predicados) a partir do estado inicial, ignorando as listas de itens que devem ser removidos dos estados ao se aplicar uma ação. O sistema GRAPHPLAN [Blum e Furst 1995] determina as relações entre os diversos predicados e ações do grafo, relação essas ligadas a exclusão mútua, chamadas de *mutex*. Após o sistema resolver essas inconsistências, ele consegue encontrar o plano que representa a solução para o problema. O GRAPHPLAN não fazia uso de heurística, mas a partir dele foi possível abstrair ideias para a construção de funções heurísticas usadas nos planejadores mais recentes.

O planejador *Heuristic Search Planner* (HSP) surgiu em 1998, utilizando uma função heurística proposta por [Bonnet e Geffner 1998] que guiava o procedimento de busca Hill Climbing (HC). A função de heurística do HSP gera uma estimativa do número de passos necessários para se chegar até a meta, a partir do estado corrente. A razão dela gerar

apenas uma estimativa é que ela considera as conjunções da meta como independentes, com isso, para se calcular o valor da heurística para um estado basta somar a quantidade de ações necessárias para se chegar em cada uma das partes da meta. Além disso, a lista de itens que serão deletados do estado é ignorada, fazendo com que só se adicione itens ao estado, aumentando a chance dele chegar mais rápido aos predicados da meta e finalizar o cálculo da heurística.

O *Fast Forward* (FF) é um planejador que se baseou nas técnicas apresentadas pelo HSP e GRAPHPLAN. Sua heurística é calculada a partir de grafos de planos relaxados gerados pelo GRAPHPLAN, ignorando a lista de itens a serem deletados como é feito no HPS e no próprio GRAPHPLAN. Porém, ele não trata as inconsistências geradas pelos *mutex*, portanto esse cálculo serve apenas como estimativa para guiar o processo de busca até a solução.

Assim, como o HSP, o planejador FF [Hoffmann e Nebel 2001] faz uma busca para frente no espaço de estados usando a função heurística descrita acima, que estima a distância até a meta ignorando a lista de itens a serem deletados nos estados, que é uma forma de se relaxar o modelo para se chegar a solução com um menor custo. Porém a heurística do FF não assume os fatos da meta como sendo independentes, como o HSP, pois é gerada a partir do grafo de plano relaxado tendo um valor mais preciso. Outra diferença foi a introdução de uma nova estratégia de busca para tentar escapar de máximos locais, que combina o algoritmo *Hill Climbing* (HC) com busca sistemática, chamada de *Enforced Hill Climbing* (EHC).

Algoritmos de melhoramento iterativo, como o HC, são passíveis ao problema de ficarem presos em máximos/mínimos locais. O planejador HSP, que usa sua heurística para guiar a busca com o HC, tenta resolver esse problema com uma escolha aleatória do melhor sucessor ao ficar preso nesses máximos/mínimos locais. Já o EHC utilizado pelo FF, ao ficar preso em máximos/mínimos locais, utiliza uma *busca em largura* , que busca dentre todos os vizinhos um que forneça um valor de heurística melhor do que o estado corrente. Caso não se encontre, aprofunda-se mais um nível (vizinho dos vizinhos), e assim por diante, até se encontrar um estado melhor que o corrente. Então, o caminho do estado corrente até este novo estado encontrado é adicionado ao plano, e a busca volta ao normal, a partir deste novo estado, interrompendo a *busca em largura* pois já se saiu do máximo/mínimo local no qual se estava preso. Com isso, o EHC não se perde em máximos/mínimos locais ou em platôs, mas não é garantido que ao entrar em um deles ele não fique preso em "becos sem saída", e fique impossibilitado de encontrar a solução. Logo, nesse caso a execução do EHC é interrompida.

O FF usa também o conceito de "ações úteis", uma melhoria proposta pelo planejador que filtra ações mais promissoras a serem utilizadas e, em conjunto com o EHC, acelera o processo de busca. Mas caso o EHC venha a falhar, o planejador começa uma nova busca do zero, utilizando o algoritmo completo Best First Search (BFS) guiado pela mesma

heurística do EHC.

A execução do FF é feita da seguinte forma:

- através do EHC, a cada passo:
  - coletar dados que ajudam a estimar a distância da solução (heurística);
  - realizar a busca local em direção ao "melhor estado" (heurística mais promissora);
  - seguir para este estado, esquecendo o passado;
- caso o FF se perca em caminhos que não chegam à solução, ele deve parar a execução e tentar resolver o problema novamente por meio de uma busca completa, neste caso com BFS.

O FF teve desempenho superior em todos os domínios de problemas da "2ª Competição Internacional de Planejamento" (do inglês, *International Planning Competition* (IPC)) que ele participou, e que depois disso serviu como base para o desenvolvimento de outros planejadores como: Metric-FF, Conformant-FF, Contingent-FF, Probabilistic-FF, MACRO-FF, MARVIN, POND e recentemente JavaFF e Ffha. Embora ele tenha sido o melhor sistema de planejamento nessa competição, ele não conseguiu repetir o feito, apesar de ter ficado entre os melhores na próxima competição. Isso aconteceu porque ele não foi estendido para manipular problemas com restrições temporais na época, com isso sistemas de planejamento que podiam planejar com restrições temporais solucionaram mais problemas durante a competição, conseguindo assim os melhores desempenhos.

Na mesma linha do FF, outros planejadores recentes combinam várias técnicas de busca. Por exemplo, o *Fast Downward* (FD) [Helmert 2006] computa os planos a partir de busca heurística no espaço de estados que podem ser visitados a partir do estado inicial. Apesar disso, o FD usa uma avaliação heurística diferente, chamada *causal graph heuristic* (heurística baseada em grafos causais), que não ignora a lista de exclusão como o HSP e FF. O FD também usa otimizações nos algoritmos de busca dentro da sua estratégia, como descrito a seguir:

- *Greedy* BFS: uma versão modificada de [Russell e Norvig 2009] BFS, com uma técnica chamada *deferred heuristic evaluation* (avaliação de heurística deferida) para mitigar a influência negativa de ramificações amplas. Também trata com *preferred operators* (operadores preferenciais), introduzidos pelo FD e que são similares as "ações úteis" do FF.
- *Multi-heuristic* BFS: uma variação do *Greedy* BFS que avalia o espaço de busca usando múltiplas heurísticas para estimação, mantendo *listas abertas* separadas para cada heurística, e também fornecendo suporte aos *preferred operators*.

- *Focused Iterative-Broadening Search*: um novo algoritmo de busca simples que não usa heurística, porém reduz o espaço vasto de possibilidades de busca focando num conjunto de operadores limitados derivados do *causal graph* (grafo causal).

O planejador LAMA [Richter e Westphal 2010] segue a mesma ideia dos outros sistemas citados acima usando busca heurística. Ele obteve a melhor performance entre todos os planejadores na trilha "satisficing" da IPC em 2008. O LAMA usa uma variação da heurística do FF e a heurística derivada de *landmarks*, onde fórmulas proposicionais devem ser verdadeiras em todas as soluções de uma tarefa de planejamento. Dois algoritmos de busca heurística são implementados no LAMA: o *Greedy* BFS, com a tarefa de encontrar a solução o mais rápido possível, e o *Weighted* A*, que permite um balanceamento entre a rapidez e a qualidade de se encontrar uma solução. Ambos os algoritmos são variações de métodos usuais, usando *listas abertas* e *listas fechadas*.

As melhorias realizadas em sistemas de planejamento não são apenas avanços em funções heurísticas, mas também na criação e modificação de algoritmos de busca.

Em [Akramifar e Ghassem-Sani 2010], uma nova modificação do EHC, chamada *Guided Enforced Hill Climbing* (GEHC) foi proposta para evitar caminhos sem saída durante a busca, adaptando uma função de ordenação para expandir seus sucessores. Ele é mais rápido e examina menos estados que o EHC porém, em alguns domínios, a qualidade é um pouco inferior.

Uma abordagem de planejamento de ações baseadas no *Simulated Annealing* (SA) é descrita em [Rames Basilio Junior e Lopes 2012], que exibe resultados significantes comparados com algoritmos baseados em EHC. Uma das dificuldades é encontrar os parâmetros iniciais usados pelo SA.

[Xie et al. 2012] apresenta um método que usa *Greedy* BFS direcionado e uma combinação de *random walks* e avaliação direta de estados, que faz um equilíbrio entre exploração de estados não visitados e exploração de estados já avaliados. Essa técnica foi implementada no planejador Arvand, que competiu na IPC em 2011, o que melhorou a cobertura e a qualidade do planejador em relação a sua versão anterior. O algoritmo executa melhor que outros planejadores, especialmente em domínios onde muitos caminhos são gerados entre o estado inicial e a meta. Apesar disso, ele não possui boa performance em problemas que requerem busca exaustiva em grandes regiões do espaço de busca.

Uma variação do algoritmo BFS é introduzida por [Stern et al. 2011], chamada *Potential Search* (PTS), que é designada para resolver problemas de busca com corte de custos. Ele ordena os estados na *lista aberta* de acordo com seu potencial, que é avaliado pela relação entre uma heurística fornecida e o custo da solução ótima. Esse algoritmo exibiu bons resultados em relação a outros com propostas semelhantes, como o *Weighted* A*. Porém, as vezes a solução ideal pode não ser encontrada, pois um estado com potencial um pouco mais alto, contudo mais distante da meta, seria selecionado contra um estado mais próximo da meta com potencial pior.

Na maioria dos trabalhos apresentados, mesmo com algumas modificações, os algoritmos não são capazes de encontrar a solução para todos os problemas usando somente uma técnica de busca. Em geral, é necessário usar uma combinação de técnicas de busca. Quando a técnica principal falha em encontrar a solução, o planejador usa uma busca completa para alcançar a meta e garantir a completude. Como resultado, muito tempo e esforço podem ser desperdiçados durante a primeira fase desses planejadores.

2.2.2 Heurística em Planejamento

Em planejamento, uma heurística nos dá uma estimativa de quantas ações o plano de um estado qualquer ainda necessita para alcançar o estado meta. Neste caso, o caminho até a solução é composto por estados cujos valores heurísticos vão decrescendo a partir do estado inicial em direção à meta. Já estados fora desse caminho tendem a ter valores heurísticos piores (mais altos).

As funções heurísticas de planejamento desenvolvidas até o momento podem ser agrupadas em quatro famílias distintas:

- *Abstractions*: reduz a escala ou o tamanho do problema, e a solução desse problema reduzido serve como estimativa para o problema original. Alguns exemplos dessa heurística são: *Pattern Database* (PDB) [Edelkamp 2001] usada no planejador FD, e *Merge-and-Shrink* (M&S) [Helmert et al. 2007] usada em um planejador também chamado de M&S (construído sobre a arquitetura do FD).
- *Landmarks*: define fatos que deverão ser verdadeiros ou, até mesmo, estáticos em algum momento do planejamento das ações. Então esses fatos são colocados em uma lista chamada "landmarks", e todos os outros fatos considerados dinâmicos irão compor uma lista chamada "to-do", ao longo do planejamento, cujo tamanho representa a heurística. Um exemplo dessa abordagem é a heurística $h^{\mathtt{LM}}$ [Richter et al. 2008] usada no planejador LAMA.
- *Critical Paths*: procura a solução de maior custo dentro de um conjunto de subproblemas de tamanhos fixos. Esse tamanho é definido por um parâmetro m, sendo a heurística mais conhecida dessa família a h^m, usada por exemplo no planejador HSP-R [Haslum e Geffner 2000]. Usualmente h^1 significa que $m = 1$ e tem-se apenas um caminho crítico resultante que servirá como heurística, já h^2 diz que deve-se considerar a soma dos custos de dois caminhos críticos distintos que juntos podem levar a solução, e isto será usado como estimativa para o problema. Lembrando que se o m for suficientemente grande acaba-se chegando a solução do problema, pois avalia-se todas as possibilidades de combinação de caminhos até a meta, o que eleva o custo da busca. Mas se m for muito pequeno e o caminho crítico resultante não for muito expressivo, acabe-se deixando a heurística pouco admissível. Exemplos: h^1, h^2, h^3, etc.

- *Ignoring Deletes*: todos os fatos que foram verdadeiros em algum momento no planejamento continuarão sendo verdadeiros até o fim, o que aumenta a possibilidade de satisfazer a meta mais rápido, diminuindo o tempo do cálculo da heurística. Alguns exemplos são: h_{add} e h_{max} utilizadas no planejador HSP [Bonnet e Geffner 1998]. Esta técnica foi uma das primeiras a ser usada, e tem-se obtido bons resultados com ela ao longo da história. Na sequência, é abordado de forma mais explícita uma dessas heurísticas, a chamada h_{FF}, usada em vários planejadores como FF, FD, LAMA, etc; e que também foi utilizada no planejador descrito neste trabalho.

Considerando os grupos de heurísticas citados, podemos concluir que uma forma muito comum de se criar funções heurísticas em IA é simplesmente relaxar o problema, ou seja, imaginar o problema com menos restrições para encontrar uma solução relaxada que servirá como estimativa. Sendo um *plano* (sequência de ações) a solução para um problema de planejamento, uma proposta de heurística para planejamento seria desenvolver um modo de gerar planos relaxados, que na verdade seriam uma aproximação dos planos reais. Uma técnica bastante usada é a de ignorar a lista de itens a serem deletados de um estado ao se aplicar uma ação (*Ignoring Deletes*), o que pode gerar fatos contraditórios como: um bloco A estar sobre os blocos B e C ao mesmo tempo (considerando o exemplo do "Mundo dos Blocos"); mas isso não é algo que compromete a eficiência da busca pois servirá apenas como estimativa durante a resolução do problema.

A heurística h_{FF} introduzida pelo planejador FF, é baseado neste conceito. Ela usa o sistema GRAPHPLAN para gerar um grafo de planos relaxados (do inglês, *Relaxed Planning Graph* (RPG)) que ignora a lista de itens a serem deletados, e posteriormente esse RPG é usado para extrair a heurística.

O RPG é feito alternando *camadas de fatos* e *camadas de ações*. Camadas de fatos são conjuntos de predicados gerados a partir dos fatos do estado inicial e da aplicação de ações que estão em uma camada de ações, onde os predicados que deveriam ser removidos são ignorados. Já as camadas de ações são conjuntos de ações cujas pré-condições foram satisfeitas pelos fatos de uma camada de fatos.

Considerando que essas camadas se alternam, uma camada de fatos $F(n)$ é usada para determinar quais ações podem aparecer na camada de ação $A(n+1)$, ou seja, a camada de ação $A(n+1)$ é composta por ações cujas pré-condições são satisfeitas pelos fatos contidos na camada de fatos $F(n)$.

A camada de fatos $F(n+1)$ é igual a camada $F(n)$, somada com os fatos gerados pela lista de itens a serem adicionados de acordo com os efeitos das ações da camada de ações $A(n+1)$. Sendo assim, camadas de fatos ficam cada vez maiores ao passo que mais ações se tornam aplicáveis, considerando o grande número de fatos.

A Figura 2.5 ilustra um exemplo de um problema de planejamento de logística. Nele, caminhões trafegam entre depósitos para transportar pacotes. O estado inicial mostra o

caminhão estacionado no depósito A, e o pacote no depósito B. O objetivo é colocar o pacote no depósito A, onde as possíveis ações são:

- $mover(c, d_1, d_2)$: mover o caminhão c que está no depósito d_1 para o depósito d_2.
- $carregar(p, c, d)$: carregar o caminhão c com o pacote p que estão no depósito d.
- $descarregar(p, c, d)$: descarregar o pacote p no depósito d que está no caminhão c.

Os fatos que podem ser encontrados nesse problema são:

- $Em(c, d)$: caminhão c está no depósito d.
- $Em(p, d)$: pacote p está no depósito d.
- $No(p, c)$: pacote p está dentro do caminhão c.

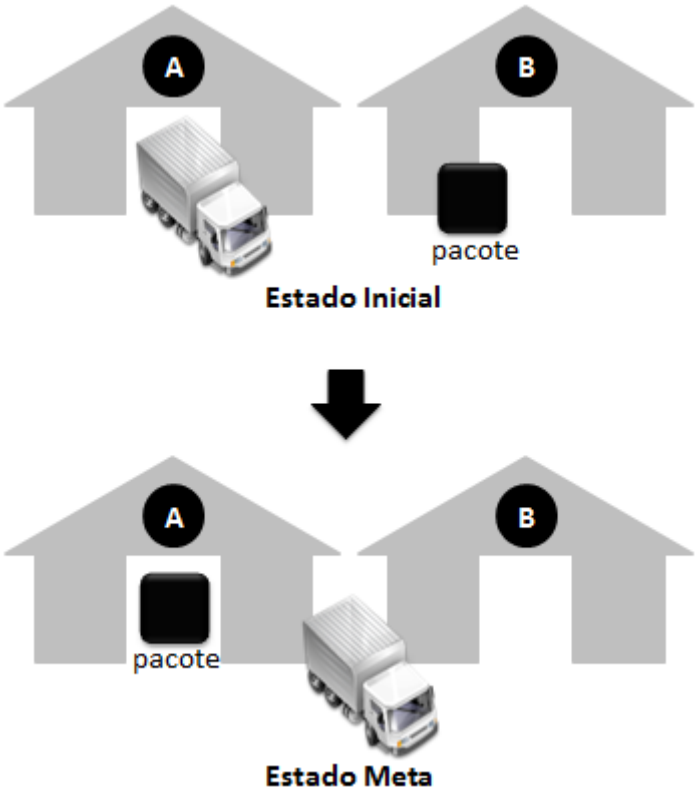

Figura 2.5: Exemplo: problema de logística

A Figura 2.6 mostra o cálculo do RPG considerando o estado inicial $Em(caminhao, A) \wedge Em(pacote, B)$ e a meta $Em(pacote, A)$ alternando camadas de fatos e de ações. A camada de fatos $F(0)$ contém os fatos do estado inicial, a partir dela a camada de ação $A(1)$ é gerada considerando todas as ações satisfeitas pelos fatos de $F(0)$. Em seguida, uma nova camada de fatos $F(1)$ é criada com os fatos já existentes na camada $F(0)$ e com os fatos adicionados pelas ações da camada $A(1)$, desconsiderando os fatos que deveriam ser

deletados. Neste exemplo, esse processo se repetiu até se construir a camada $F(3)$, cujos fatos contemplam todos os fatos contidos no estado meta.

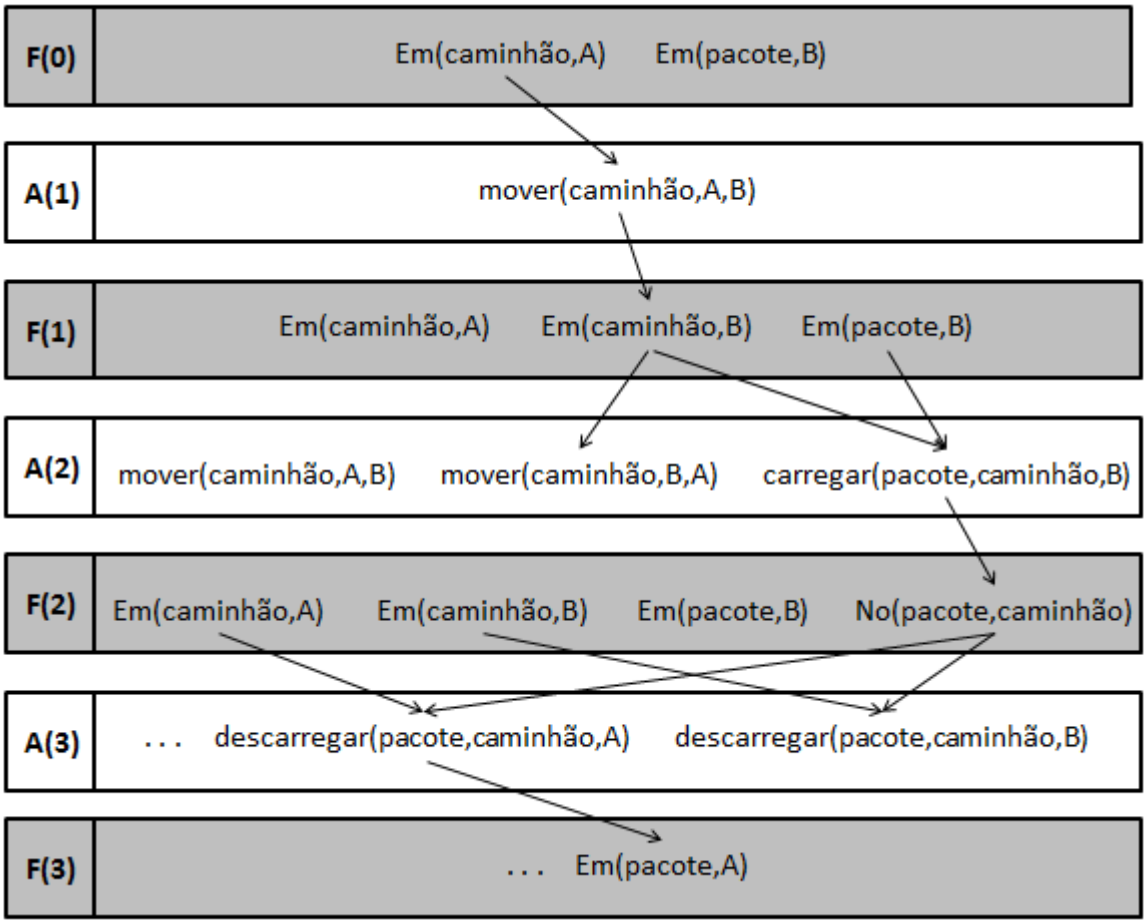

Figura 2.6: Exemplo: geração do RPG

Na verdade, os critérios de parada para a criação do RPG são:

- Gerar uma camada de fatos que possua todos os predicados descritos na meta.
- Gerar uma camada de fatos idêntica a anterior pois, deste ponto em diante, o conjunto de fatos que se tem irão satisfazer sempre as mesmas ações, que gerarão sempre os mesmos fatos, que já estão disponíveis na camada atual.

Portanto, no RPG encontram-se os fatos e suas ligações com as ações. Então, para calcular a heurística h_{FF} a partir do RPG, é necessário realizar uma busca regressiva no grafo relaxado da seguinte forma:

- Inicia-se pela ultima camada de fatos $F(n)$, que contém os fatos requeridos pela meta. Esses fatos que compõem a meta são agrupados em um subconjunto, chamado $G(n)$, dentro da camada de fatos $F(n)$.
- Para cada fato de $G(n)$:

  - Se ele está na camada de fatos anterior, ou seja, em $F(n-1)$, adicione um subconjunto $G(n-1)$ com esse fato dentro de $F(n-1)$. Esse sub-conjunto $G(n-1)$ representa fatos necessários para se chegar a meta e que estão na camada de fatos $F(n-1)$.
  - Caso contrário, escolha uma ação de $A(n)$ e adicione suas pré-condições em $G(n-1)$.

- Parar quando chegar no conjunto $G(0)$, ou seja, ao subconjunto de fatos que levam a meta a partir da camada $F(0)$.

A Figura 2.7 mostra o RPG do problema de logística após ter passado pelo processo de criação dos subconjuntos $G(i)$. Em preto temos os fatos que compõem os subconjuntos $G(i)$ nas camadas de fatos $F(i)$, e em cinza as ações que são satisfeitas por esses fatos e que representam ações que se acredita serem necessárias para se chegar a meta.

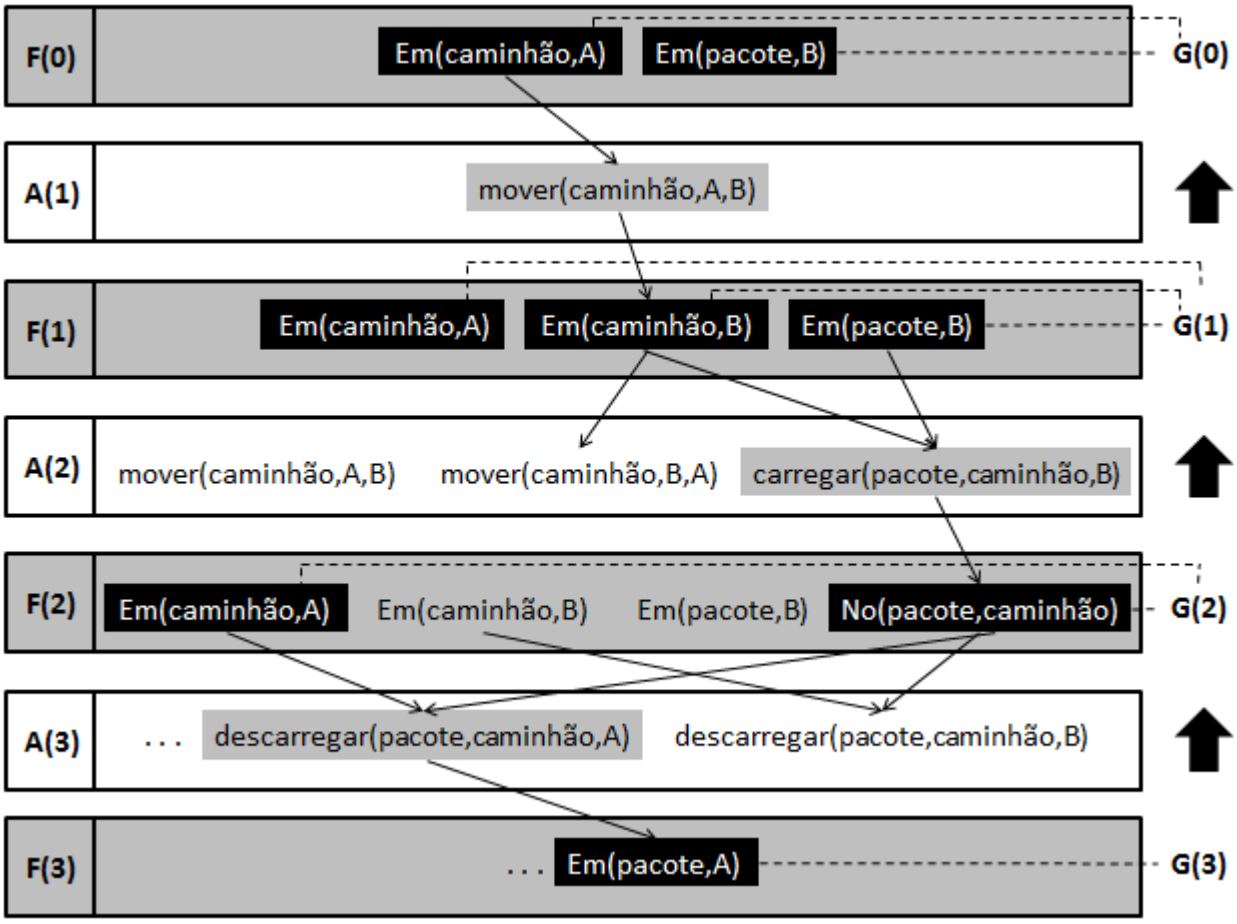

Figura 2.7: Exemplo: regressão no RPG

Sempre temos o caminho de volta no grafo, devido às ligações entre fatos e ações feitas durante a construção progressiva do RPG. A quantidade de ações selecionadas (em cinza) é o resultado da função heurística h_{FF}, ou seja, o numero de ações filtradas no RPG

representa a estimativa de passos necessárias para se chegar a meta, a partir do estado inicial, como é ilustrado na Figura 2.8. O filtro "helpfull actions", usado no algoritmo EHC do planejador FF, usa essas ações por considera-las mais promissoras para a busca, mas não é recomendado que se use esse filtro em algoritmos "completos" pois eles podem perder essa característica.

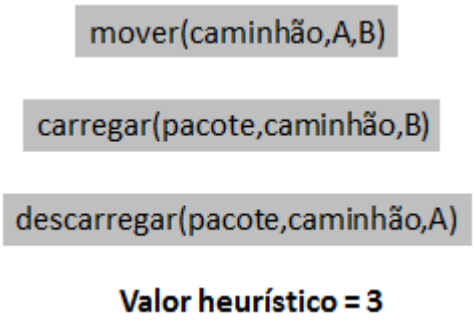

Figura 2.8: Exemplo: valor heurístico extraído do RPG

Observa-se que uma ação necessária para a solução real do problema não foi selecionada, que é $mover(caminhao, B, A)$ após $carregar(pacote, caminhao, B)$. Ela se perdeu, pois os fatos necessários para se chegar a meta já estavam contemplados no RPG, sem que houvesse necessidade de se executar essa ação. Por isso, consideramos esta técnica como uma estimativa, a qual nem sempre condiz com a solução real, mas que é de grande importância para guiar algoritmos de busca e acelerar sua convergência.

Porém, essa heurística não é admissível pois um mesmo fato pode ter sido gerado por diferentes ações durante o RPG, nesse caso a ação que está mais próxima da camada inicial é priorizada, o que pode superestimar a solução real. Portanto algoritmos, como o A*, podem perder a característica de serem ótimos (encontrar a solução ótima). Gerar planos relaxados ótimos, cujos tamanhos representam heurísticas admissíveis como a h_+, são $NP - Hard$.

O cálculo de heurística é geralmente muito caro. Estima-se que o planejador FF gasta cerca de 80% de seu tempo calculando heurística durante a busca. Com isso, uma heurística deve ser bem informativa e podar o espaço de busca o suficiente para compensar seu cálculo. Por exemplo, até 2008 haviam casos em que para encontrar a solução ótima para um problema de planejamento, um planejador que fazia uso de uma busca ótima poderia ter sua performance semelhante a uma busca cega (por exemplo, busca em largura), pelo fato do cálculo da heurística admissível ter um alto custo computacional.

Mas, mesmo considerando esses fatores, planejadores baseados em busca heurística são os que mais obtêm sucesso nos quesitos velocidade e qualidade da solução. Um reflexo disso, são os resultados das competições de planejamento IPCs, onde os campeões são planejadores guiados por heurística.

2.2.3 Linguagens de Definição de Domínios e Problemas de Planejamento

Para que um planejador possa gerar um plano que satisfaça um dado problema, o mesmo deve receber como entrada um estado inicial, uma meta e um conjunto de ações que possam ser aplicadas ao problema. Esses itens de entrada geralmente são estruturados dentro de arquivos, respeitando uma linguagem formal. Portanto, existe uma primeira fase no planejamento que antecede a busca pela solução, que é justamente a fase de interpretação desses arquivos que contêm as entradas do problema.

Além do desenvolvimento do planejador *STandford Research Institute Problem Solver* (STRIPS) em 1970, Nils Nilsson e Richard Fikes introduziram a primeira linguagem formal para descrever domínios e problemas de planejamento. Essa linguagem, que também recebe o nome de STRIPS [Fikes e Nilsson 1971], representa um estado como um conjunto de átomos, e as ações como operadores as quais possuem pré-condições que definem a aplicabilidade das ações e efeitos, que adicionam ou eliminam átomos de um estado para gerar estados sucessores.

Porém, para alguns problemas, era necessária uma descrição mais expressiva incluindo efeitos condicionais e quantificadores lógicos. Foi então, em 1989, que Pednault propôs uma linguagem melhorada para descrever ações chamada *Action Description Language* (ADL) [Pednault 1989]. Na ADL, as ações eram representadas por esquemas, ou seja, operadores cujos objetos são identificados por variáveis. A Tabela 2.1 mostra as principais diferenças entre as linguagens STRIPS e ADL.

Tabela 2.1: Diferenças entre STRIPS e ADL

STRIPS	**ADL**
Somente literais positivos nos estados. Exemplo: $Pobre \wedge Desconhecido$	Literais positivos e negativos nos estados. Exemplo: $\neg Rico \wedge \neg Famoso$
Domínio Fechado: todos os literais não mencionados são considerados falsos.	Domínio Aberto: todos os literais não mencionados são considerados desconhecidos.
Somente literais aterrados no estado meta. Exemplo: $Rico \wedge Famoso$	Variáveis quantificadas no estado meta. Exemplo: $\exists x\ sobre(A, x) \wedge sobre(x, B)$
Efeitos também são conjunções.	Permite efeitos condicionais: um efeito só é válido se sua pré-condição for satisfeita.
Não suporta operação de igualdade.	Aceita igualdade de predicados. Exemplo: $x = y$
Não suporta tipagem.	Variáveis podem possuir tipos. Exemplo: $b : Bloco$

Na tentativa de padronizar e propor ainda mais robustez às linguagens de planejamento McDermott, em 1998, desenvolveu uma linguagem chamada *Planning Domain Definition Language* (PDDL) [Mcdermott et al. 1998]. Ela foi usada pela primeira vez para apresentar os problemas da *International Planning Competition* (IPC), em 1998, e vem evoluindo a cada competição. Essas competições serão descritas com mais detalhes a seguir, na próxima seção.

"A adoção de um formalismo comum para descrever domínios de planejamento promove uma maior reutilização das pesquisas e permite a comparação

> mais direta de sistemas e abordagens de planejamento, e portanto, suporta um progresso mais rápido no campo. Um formalismo comum é um compromisso entre poder expressivo (em que o desenvolvimento é fortemente impulsionado por aplicações potenciais) e o progresso da pesquisa básica (que incentiva o desenvolvimento de fundações bem compreendidas). O papel de um formalismo comum como meio de comunicação exige que ele possua semântica bem clara." [Fox e Long 2002]

A linguagem PDDL foi projetada para ser uma especificação neutra dos problemas de planejamento, ou seja, não ser favorável a nenhum sistema de planejamento. Problemas e domínios descritos através de um formalismo comum, podem ser submetidos a qualquer planejador que entenda a linguagem de descrição. A PDDL foi inspirada por outras notações e formalismo, além da ADL e do STRIPS, como: SIPE-2 [Wilkins 1989], Prodigy 4.0 [Carbonell et al. 1991], UCPOP [Penberthy e Weld 1992], UMCP [Erol et al. 1994] e Unpop [Nguyen e Kambhampati 2000]. Apesar de ser uma nova linguagem, problemas do tipo STRIPS e ADL podem ser descritos através do formalismo PDDL.

Em PDDL, uma tarefa de planejamento é composta por:

- Objetivos: elementos do ambiente que nos interessam.
- Predicados: propriedades dos objetos em que estamos interessados, podendo ser verdadeiras ou falsas.
- Estado: uma descrição do ambiente em determinado instante dada em termos de conjuntos de predicados considerados verdadeiros.
- Estado inicial: estado de inicio do ambiente que se deseja planejar.
- Especificação da meta: predicados que devem ser verdadeiros após a execução do plano.
- Ações: operadores que mudam os estados do ambiente.

Essas tarefas são separadas em dois arquivos:

- Arquivo de domínio: descreve predicados e ações.
- Arquivo de problema: descreve objetos, estado inicial e especificação da meta do problema. Podemos ter diversos arquivos de problemas para um mesmo domínio, alterando as descrições dos objetos, do estado inicial e da meta; o que pode aumentar ou diminuir a complexidade de um problema. Com isso, para um mesmo domínio, podemos encontrar problemas fáceis e difíceis de se resolver.

Um arquivo de domínio tem a seguinte estrutura:

```
(define (domain <nome do domínio>)
```

```
        <código PDDL para predicados>
        <código PDDL para a 1ª ação>
        ...
        <código PDDL para a última ação>

)
```

Sendo <nome do domínio> um texto que identifica o domínio de planejamento. Já a estrutura de um arquivo de problema pode ser descrita como:

```
(define (problem <nome do problema>)

        (:domain <nome do domínio>)
        <código PDDL para objetos>
        <código PDDL para o estado inicial>
        <código PDDL para a especificação da meta>

)
```

Sendo <nome do problema> um texto que identifica um problema de planejamento, e <nome do domínio> nome do arquivo de domínio ao qual o problema se refere.

Desde a primeira competição de planejamento, inúmeras versões da PDDL foram propostas. A versão PDDL-1.2 [Mcdermott et al. 1998] foi a linguagem oficial da 1ª e 2ª IPC em 1998 e 2000, respectivamente, onde se teve a separação de domínios e problemas de planejamento como descrito anteriormente.

Na 3ª IPC, a versão usada foi a PDDL-2.1 [Fox e Long 2003] que introduziu fluentes numéricos que são modelos de recursos não-binários como, por exemplo, nível de combustível, tempo, energia, distância, peso, etc; métricas de planos para permitir a avaliação quantitativa dos planos; e ações contínuas onde pode-se ter variáveis, cumprimentos não-discretos, condições e efeitos. A PDDL2.1 permitiu a representação e solução de muitos outros problemas do mundo real que a versão original ainda não conseguia representar.

A PDDL-2.2 [Edelkamp e Hoffmann 2003] foi usada na 4ª IPC na categoria "determinística". Ela propôs predicados derivados para modelar dependências entre fatos, e literais iniciais temporizados para modelar eventos que podem ocorrer em um determinado momento, independentemente da execução do plano.

A 4ª IPC trouxe a PDDL-3.0 [Gerevini e Long 2005] como linguagem oficial para a categoria "determinística". Ela introduziu o conceito de restrições de trajetória de estados, onde essas restrições podem ser leves ou pesadas. Quando pesadas, as trajetórias dos estados devem ser satisfeitas durante a execução de um plano para que ele seja considerado uma solução mas, caso as restrições sejam leves, não é necessário que as trajetórias realizadas pelos estados as satisfaçam, contudo podem ser consideradas nas métricas de planos.

A versão mais recente dessa linguagem é a PDDL-3.1 que foi usada nas duas últimas competições, 6ª e 7ª IPC em 2008 e 2011, respectivamente. Ela propôs a ideia de objetos fluentes, onde intervalos não são descritos apenas com números, mas também com tipos de objetos do problema.

2.2.4 Competição em Planejamento

Grandes descobertas na área de Escalonamento e Planejamento são abordadas nas conferências bianuais internacionais, *International Conference on Artificial Intelligence Planning and Scheduling Systems* (AIPS). Muitos subcampos da IA tem usado competições para aquecer, direcionar e avaliar o andamento das pesquisas. Nessas competições, os pesquisadores executam seus métodos sobre um *benchmark* comum de problemas, para que ao final seus desempenhos sejam analisados de acordo com os critérios da competição.

Em 1998, durante a 4ª conferência AIPS, foi realizada a primeira competição de planejadores automáticos, chamada de *International Planning Competition* (IPC). Essas competições têm como objetivos principais: compor um conjunto de dados empíricos referente às comparações dos resultados obtidos por cada planejador, propor novos problemas (alguns considerados como problemas práticos do mundo real) que testam os limites das técnicas utilizadas até o momento, levantar novas propostas para essa linha de pesquisa como novas categorias de competição, desenvolver cada vez mais o formalismo de representação de problemas e domínios para termos sistemas de planejamento mais poderosos. Apesar de ter um caráter competitivo, o evento se preocupa em tornar disponível para a comunidade a maior quantidade de dados possíveis sobre os métodos e resultados da competição.

O organizador da IPC-1 [Mcdermott 2000] foi Drew McDermott, um dos criadores da linguagem PDDL. Para o evento foram criadas duas trilhas de competição, uma para problemas do tipo STRIPS e outra para problemas do tipo ADL, nenhuma das quais capazes de manipular problemas com números. Porém, todos os problemas deveriam estar estruturados no formato PDDL, padrão adotado na competição. Com isso, era pré-requisito que todos os planejadores tivessem a capacidade de interpretar arquivos de domínios e problemas PDDL para poderem competir.

Os participantes da trilha STRIPS, que conseguiram realizar seus ajustes para a competição, foram IPP (Jana Koehler), BLACKBOX (Henry Kautz e Bart Selman), HSP (Hector Geffner e Blai Bonet) e STAN (Derek Long e Maria Fox). Todos os planejadores foram escritos nas linguagens de programação *C/C++*, exceto SGP que foi desenvolvido em *Lisp*. Além disso, todos eles foram baseados no sistema GRAPHPLAN, exceto o planejador HSP que usou uma abordagem baseado em heurística. Na trilha STRIPS, o HSP foi quem resolveu o maior número de problemas e encontrou a menor solução com mais frequência, o BLACKBOX teve o menor tempo médio nos problemas que resolveu, e o IPP obteve o menor tempo de execução em um número maior de problemas.

A segunda competição realizada no ano 2000 por Fahiem Bacchus, a IPC-2 [Bacchus 2001], recebeu quinze participantes. Ela foi ampliada para receber sistemas de planejamento completamente automatizados e sistemas que necessitavam de configurações manuais para resolver problemas de determinados domínios. As trilhas de competição STRIPS

e ADL foram mantidas, sem nenhuma mudança no formalismo PDLL. A Tabela 2.2 mostra os competidores da IPC-2. Nessa competição, os planejadores de destaque foram FF e TAL PLANNER, sendo FF completamente autônomo e TAL PLANNER com uso de informações manipuladas para certos domínios.

Tabela 2.2: Competidores IPC-2

Planejador	**Equipe**
FF	Joerg Hoffmann
GRT	Ioannis Refanidis, Ioanis Vlahavas e Dimitris Vraskas
System R	Fangzhen Lin
MIPS	Stefan Edelkamp e Malte Helmert
IPP	Jana Koehler, Joerg Hoffmann e Michael Brener
HSP2	Hector Geffner e Blai Bonet
PropPlan	Michael Fourman
STAN	Derek Long e Maria Fox
BLACKBOX	Henry Kautz, Bart Selman e Yi-Cheng Huang
ALTALT	Biplav Srivastava, Terry Zimmerman, BinhMinh Do, XuanLong Nguyen, Zaiqing Nie, Ullas Nambiar e Romeo Sanchez
TOKEN PLAN	Yannick Meiller e Patrick Fabiani
BDDPLAN	Hans-Peter Stoerr
SHOP	Dana Nau, Hector Munoz-Avila, Yue (Jason) Cao e Ammon Lotem
PBR	Jose Luis Ambite, Graig Knoblock e Steve Minton
TAL PLANNER	Jonas Kvamstrom, Patrick Doherty e Patrik Haslum

Em 2002 ocorreu a IPC-3 [Long e Fox 2003] organizada por Derek Long e Maria Fox. Quatorze planejadores participaram da competição. O principal objetivo da competição era avançar as pesquisas em planejamento temporal e métrico. Mas, para isso, a linguagem PDDL teve que ser estendida, gerando a versão PDDL-2.1 [Fox e Long 2003]. Os critérios utilizados para avaliar os planejadores foram quantidade de problemas tentados, a taxa de plano de sucesso nestes problemas e a qualidade da solução gerada, além da velocidade dos planejadores.

A população de sistemas de planejamento mudou durante as três competições. Alguns planejadores competiram em mais de uma edição do evento, mas nenhum compareceu às três competições. Em parte, isso é reflexo da velocidade do desenvolvimento de sistemas de planejamento, revelando o avanço na área de 1998 a 2002. Mas isso também evidencia o crescente interesse nas competições, o que tem encorajado a criação de mais sistemas para participação no evento. A Tabela 2.3 mostra o resultado da competição, em ordem alfabética, dos competidores, juntamente com os critérios analisados. Apesar de ter sido considerado um dos melhores planejadores da IPC-2, o planejador FF não teve um dos melhores desempenhos nesta competição, pois ainda não estava preparado para resolver alguns problemas relacionados a planejamento temporal e métrico. Mas ainda assim, ele resolveu mais 70 problemas dos 76 que poderiam ser resolvidos através de configurações manuais. Já a baixa performance do planejador IxTex, é caracterizada pelo fato de que ele não conseguiu aceitar alguns domínios e problemas descritos em PDDL.

Houve uma ampliação na IPC-4 realizada em 2004 pelos organizadores Stefan Edelkamp e Joerg Hoffmann. O evento foi dividido em duas vertentes: a competição clássica, que já vinha ocorrendo desde 1998 e, pela primeira vez, uma competição de planejadores probabilísticos. Ainda dentro da competição clássica, separou-se planejadores ótimos

Tabela 2.3: Resultados IPC-3

Planejador	#problemas resolvidos	#problemas tentados	taxa de sucesso
FF	237 (+70)	284 (+76)	83% (85%)
IxTet	9*	*	*
LPG	372	428	87%
MIPS	331	508	65%
SHOP2	899	904	99%
Sapa	80	122	66%
SemSyn	11	144	8%
Simplanner	91	122	75%
Stella	50	102	49%
TALPlanner	610	610	100%
TLPlan	894	894	100%
TP4	26	204	13%
TPSYS	14	120	12%
VHPOP	122	224	54%

(aqueles que garantem a qualidade da solução encontrada) dos sub-ótimos, o que se mostrou muito adequado devido ao fato de que planejadores ótimos demandam mais tempo de execução para encontrar soluções melhores. Para os planejadores sub-ótimos, os resultados foram divididos nas trilhas "proposicional" e "temporal/métrica". Devido a essa vasta diversidade de categorias, foi necessário o desenvolvimento de domínios e problemas mais expressivos [J. Hoffmann 2004] e de melhorias no formalismo, gerando a PDDL-2.2 [Edelkamp e Hoffmann 2003]. A Tabela 2.4 traz os vencedores da vertente clássica do evento, dividida em planejadores ótimos e sub-ótimos, juntamente com suas trilhas.

Tabela 2.4: Vencedores IPC-4

Planejadores sub-ótimos (trilha: proposicional)		
Colocação	**Planejador**	**Equipe**
1º	Fast Downward	Malte Helmert e Silvia Richter
2º	YAHSP	Vincent Vidal
2º	SGPlan	Yixin Chen, Chih-Wei Hsu e Benjamin W. Wah
Planejadores sub-ótimos (trilha: temporal/métrica)		
Colocação	**Planejador**	**Equipe**
1º	SGPlan	Yixin Chen, Chih-Wei Hsu e Benjamin W. Wah
2º	LPG-TD	Alfonso Gerevini, Alessandro Saetti, Ivan Serina e Paolo Tonelli
Planejadores ótimos		
Colocação	**Planejador**	**Equipe**
1º	SATPLAN'04	Henry Kautz, David Roznyai, Farhad Teydaye-Saheli, Shane Neth e Michael Lindmark
2º	CPT	Vincent Vidal e Hector Geffner

Assim, como a quarta competição, a IPC-5 (2006) e sua organização foi dividida em duas partes: a trilha determinística (antes chamada de planejamento clássico) e a não-determinística (conhecida como planejamento probabilístico). A parte determinística do evento possou a considerar o tempo de CPU e também dar ênfase à qualidade do plano, sendo ela definida nos problemas como uma métrica.

Alguns desses domínios foram inspirados em aplicações do mundo real, outros tinham como objetivo explorar a aplicabilidade e eficiência de planejamento automatizado para problemas que vinham sendo investigados em outros campos da ciência da computação, enquanto domínios de competições mais antigas foram usados como referência para medir o avanço dos planejadores daquela edição [Gerevini et al. 2009]. Os vencedores na catego-

ria "planejadores ótimos" foram MAXPLAN (Zhao Xing, Yixin Chen e Weixiong Zhang) e SATPLAN (Joerg Hoffmann, Henry Kautz, Shane Neph e Bart Selman). Para a categoria "planejadores sub-ótimos", o vencedor foi SGPLAN (Benjamin W. Wah, Chih-Wei Hsu, Yixin Chen e Ruoyun Huang).

Os organizadores da IPC-6, Malte Helmert, Minh Do e Ioannis Refanidis, definiram para a edição de 2008 uma nova categoria chamada de "planejamento com aprendizado", onde os planejadores exploram os domínios a partir de um conhecimento extraído durante um período de treinamento *off-line*, além das categorias determinística e não-determinística. Na principal categoria do evento (determinística, na parte de planejadores não-ótimos) o vencedor foi o planejador LAMA (Richter, Westphal, Helmert e Roger). Além disso, foi consolidado a última versão da linguagem PDDL (PDDL-3.1), que foi usada nessa edição e na edição posterior.

A última competição, organizada por Ángel García Olaya, Sergio Jimenez Celorrio e Carlos Linares López, ocorreu em 2011 (IPC-7) e foi bastante parecida com a de 2008, tanto na parte das categorias e versão PDDL, quanto nos resultados obtidos. Mais uma vez, o planejador que venceu a principal trilha da competição foi o LAMA com sua versão 2011 [Amanda et al. 2012].

A Figura 2.9 ilustra a história da *International Planning Competition* desde a sua criação, com todas as categorias/trilhas, linguagens de planejamento, organizadores e informações sobre cada edição realizada até o momento.

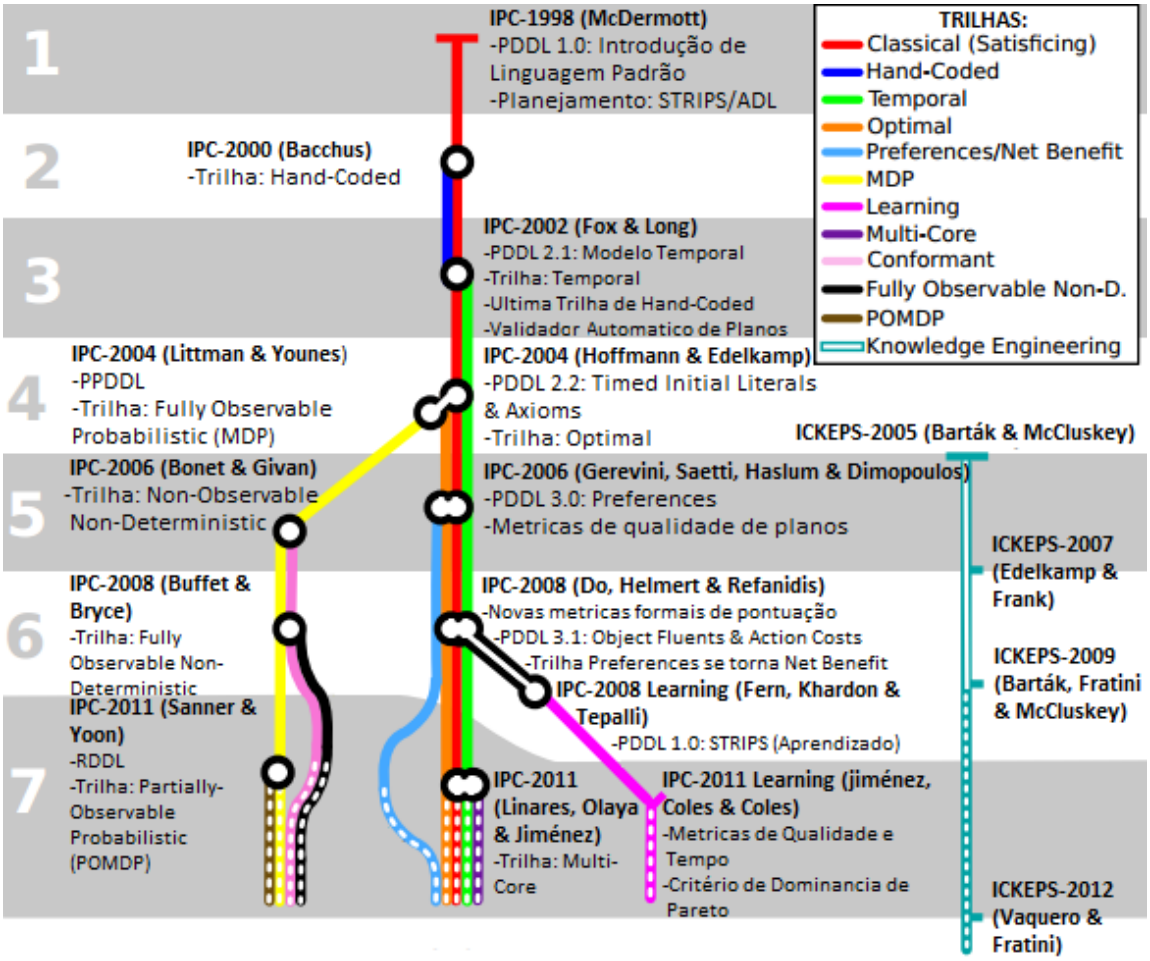

Figura 2.9: História da International Planning Competition

É interessante ressaltar que, todos os planejadores vencedores na história da IPC, sobretudo na principal categoria que é a determinística para planejadores não-ótimos (também conhecida em inglês como "satisficing track"), são sistemas que usam algoritmos de busca heurística para resolver os problemas.

Apesar do planejador construído neste trabalho ser baseado em algoritmos de busca heurística, o objetivo principal ainda não é participar de uma competição como essa. Este planejador desenvolvido, até o momento, apresenta resultados promissores em relação às técnicas de busca utilizadas em planejamento, mas ainda é sensível a problemas não-STRIPS, o que o deixaria em desvantagem em tais competições. Além disso, os planejadores relacionados nas últimas competições aprimoraram métodos que antecedem o processo de busca para melhorar suas performances, o que não foi abordado neste trabalho.

Capítulo 3

Sistema SLPlan

Neste capítulo é descrito o método desenvolvido para resolução de problemas de planejamento. O planejador *Searching and Learning - Planner* (SLPlan) executa uma estratégia de busca composta por dois algoritmos conhecidos, mas que foram otimizados durante a construção do trabalho a fim de melhorar o processo de planejamento.

Na Seção 3.1 são apresentadas situações que podem ocorrer durante a execução de uma busca, e que comumente prejudicam a performance de sistemas de planejamento, compondo assim a motivação para o trabalho. A seguir, na Seção 3.2, é descrita a estratégia de busca utilizada pelo planejador desenvolvido. A partir disso, é mostrado separadamente cada um dos algoritmos que compõe essa estratégia, sendo o algoritmo HB-EHC na Seção 3.2.1, e o algoritmo Adaptive-LRTA* na Seção 3.2.2.

3.1 Motivação para a construção do método

Em planejamento, existem problemas que podem ser considerados simples por terem algumas características, como: o estado meta se encontrar próximo ao estado inicial sendo necessário visitar poucos estados para se chegar a solução, o fator de ramificação ser baixo, o cálculo de heurística ser simples, etc.. Para este tipo de problema é desnecessário o uso de algoritmos robustos, como algoritmos completos, que tendem a ter um tempo de resposta maior por fazer uma busca mais ampla no espaço de estados. Nestes casos, são utilizados algoritmos mais ágeis que trabalham de forma gulosa, podando grande parte dos estados a serem visitados, alcançando a solução rapidamente. Por outro lado, também existem problemas complexos que exigem uma exploração maior no espaço de estados, o que muitas vezes requer uma busca completa.

Mas, a priori, não se sabe se um problema é simples ou não até que se comece de fato a execução de uma busca. Então surge a dúvida: qual algoritmo de busca deve-se usar, sem saber se o problema é fácil ou difícil de se resolver? Por este motivo, muitos planejadores empregam o uso de estratégias de busca, que na verdade são sequências de algoritmos que podem ser executados pelo planejador para tentar encontrar a solução.

A ordem em que esses algoritmos são chamados é geralmente a seguinte: inicialmente tenta-se resolver o problema o mais rápido possível com algoritmos ágeis não-completos, caso eles encontrem a solução, podemos dizer que o problema era relativamente simples, e o planejador irá retornar o plano gerado; mas como eles não são completos, esses algoritmos podem não chegar a solução e ter suas execuções encerradas, o que indica que o problema pode não ser tão fácil de se resolver. Nestes casos, algoritmos completos são acionados para tentar solucionar o problema.

Com o objetivo de observar o comportamento desses planejadores, foram executados experimentos preliminares com domínios e problemas de planejamento da IPC-3, utilizando principalmente a estratégia de busca proposta pelo planejador FF, que é composta pelos algoritmos EHC e BFS. O EHC é uma otimização do algoritmo *Hill Climbing* feita pelos autores do FF, e que se caracteriza pela rapidez com que resolve problemas considerados relativamente simples. Já o BFS é um algoritmo completo usado em inúmeros planejadores, incluindo grandes campeões das IPCs, por exemplo: FF, FD e LAMA. Através desses experimentos foi possível identificar algumas situações que podem prejudicar a execução de algoritmos de busca e degradar a performance de planejadores, como:

- Problemas de planejamento onde muitos estados, ao serem expandidos, geram os mesmos conjuntos de estados sucessores demandando muito espaço de memória.
- Problemas com fator de ramificação muito alto, onde estados geram muitos sucessores, o que também demanda muita memória e aumenta consideravelmente o espaço de busca.
- Estados que geram eles mesmos como sucessores. Portanto, se o planejador não tratar estados repetidos de alguma forma, a busca pode correr o risco de entrar em *loops*.
- Estados cujos cálculos dos valores heurísticos requerem muito tempo, principalmente para problemas maiores e mais complexos. Portanto, caso esse valor heurístico necessite de ser recalculado toda vez que um estado reaparece durante busca, o processo tende a ser muito moroso.
- Situações em que a busca entra em máximos/mínimos locais e fica preso em "becos sem saída", fazendo com que algoritmos de busca local como o EHC falhem. Nestes casos, a busca recomeça do zero, executando o próximo algoritmo descrito na estratégia, desperdiçando todo tempo gasto na fase anterior e perdendo todos os dados já processados até então.
- Dependendo do problema, onde a primeira fase de busca não consegue encontrar a solução e que se faz necessário a execução de algoritmos completos como o BFS, pode-se ter um problema de "estouro de memória", pois algoritmos completos tendem a manter todos os estados explorados em memória.

A estratégia de busca e as otimizações dos algoritmos desenvolvidos neste trabalho, tentam mitigar e até mesmo anular os impactos causados pelas situações descritas acima, a fim de obter uma melhor eficiência na solução de problemas de planejamento.

3.2 Estratégia de Busca

O planejador SLPlan é um sistema de planejamento automático para resolver problemas do tipo STRIPS descritos no formato PDDL. Ele foi construído utilizando a linguagem de programação *Java* baseado no código-fonte do planejador JavaFF, uma implementação Java do planejador FF. Assim como esses planejadores, o SLPlan também utiliza a heurística h_{FF} para guiar os algoritmos de busca descritos em sua estratégia.

Ao submeter um problema para o SLPlan, ele realizará as seguintes ações:

1. Interpretar o arquivo PDDL do domínio e do respectivo problema.
2. Executar a estratégia de busca.
   (a) Iniciar o HB-EHC, salvando todos os estados explorados dentro de um *heap*.
      i. Se encontrar a meta, retornar o plano gerado.
      ii. Caso contrário, interromper sua execução.
   (b) Iniciar o Adaptive-LRTA*, utilizando o mesmo *heap*.
      i. Retornar o plano gerado.

O HB-EHC é uma otimização do algoritmo EHC, que tenta escapar de máximos/mínimos locais com mais frequência e rapidez, evitando que a busca seja interrompida e todo o tempo e processamento realizado sejam perdidos.

Já o algoritmo Adaptive-LRTA* é uma adaptação do LRTA* para que, caso o algoritmo anterior falhe, ele possa continuar a busca aproveitando os estados já explorados, e que foram armazenados em um *heap*, evitando principalmente o recálculo das heurísticas dos estados. Durante a busca, ele continua salvando estados dentro deste mesmo *heap* e fazendo um balanceamento do mesmo para que não falte espaço durante sua execução. Além disso, ele realiza um número limitado de escolhas gulosas para acelerar a convergência, e podas de estados que também ajudará a economizar espaço. Existem outros benefícios relacionados ao uso desse algoritmo, como o "aprendizado" advindo da atualização contínua dos valores heurísticos dos estados durante a busca, para que ele escape de máximos/mínimos locais e "becos sem saída", além de ser um algoritmo completo como o BFS, o que garante a completude do planejador.

Ambos os algoritmos são descritos de forma mais detalhada nas seções seguintes.

3.2.1 HB-EHC

Nesta seção é apresentado o algoritmo *Heap&Backtracking - Enforced Hill Climbing* (HB-EHC), uma extensão do EHC. As otimizações feitas neste algoritmo tem como objetivo escapar de máximos/mínimos locais com mais frequência e rapidez. E se não for possível escapar destes máximos/mínimos locais, o que faz com que a busca entre em "becos sem saída", ele será capaz de realizar retrocessos para evitar que sua execução falhe. Desse modo, é possível resolver mais problemas que sua versão padrão, evitando muitas vezes o uso de uma busca completa.

Descrição

O algoritmo usa dois *heaps* (Figura 3.1) que funcionam como filas de prioridades. A vantagem de se usar estruturas como o *heap*, deve-se ao baixo custo de suas operações. No caso do HB-EHC, eles desempenham os papeis das listas *aberta* e $fechada$ da versão original do EHC. O propósito da lista $fechada$ é armazenar estados já explorados, ao invés disso o HB-EHC usa um *heap máximo* (Q_{max}) organizado em ordem decrescente pelos valores heurísticos dos estados. Sendo assim, estados cujo valor heurístico são altos, consequentemente ruins por estarem longe da meta, tendem a ficar mais próximos da raiz do *heap*.

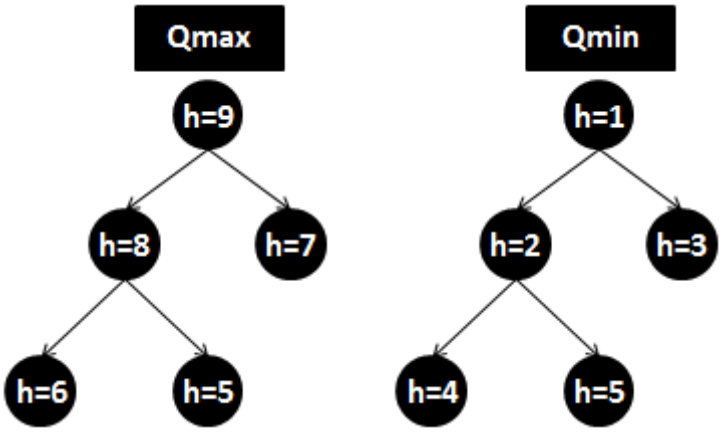

Figura 3.1: HB-EHC: heaps Qmax e Qmin.

Esta estrutura não tem grande impacto para o HB-EHC em si, mas é extremante importante para a estratégia como um todo, pois caso este algoritmo falhe, o próximo algoritmo irá reaproveitar este *heap*. E como será descrito posteriormente, caso a execução do próximo algoritmo comece a ficar sem espaço para armazenar estados, sucessivas remoções serão feitas na raiz deste *heap*, retirando estados ruins para liberar espaço de memória, sendo os custos destas operações de remoção extremamente baixas.

O segundo *heap* corresponde a lista *aberta* que mantém os estados que foram descobertos pela busca, mas que ainda não foram explorados por não terem um valor heurístico

melhor que o do estado corrente. Este *heap* é de uso exclusivo do HB-EHC, e é ordenado de forma inversa em relação ao anterior. Na verdade, este *heap* (Q_{min}) é ordenado de forma crescente pelos valores heurísticos dos estados, ou seja, os melhores estados são mantidos perto da raiz por terem valores heurísticos baixos, o que caracteriza sua grande proximidade ao estado meta. Desta forma, quando a busca entrar em um máximo/mínimo local, ilustrado pela Figura 3.2(a), o algoritmo começará uma *Busca em Largura* dentro de Q_{min}, onde a raiz representa o melhor estado de todos e que será o primeiro a ser explorado. O que não acontece na forma convencional do EHC, que explora os estados na ordem em que eles aparecem na lista *aberta*, onde no pior caso os estados que possuem valores baixos de heurística poderão ser os últimos a serem explorados, fazendo com que ele demore a sair do máximo/mínimo local.

Expandindo o melhor estado do *heap*, existe uma grande chance de se produzir sucessores com melhores valores heurísticos, ou até mesmo o próprio estado meta. Isso faz com que a busca escape de máximos/mínimos locais já nas primeiras tentativas, como podemos ver na Figura 3.2(b), acelerando a convergência.

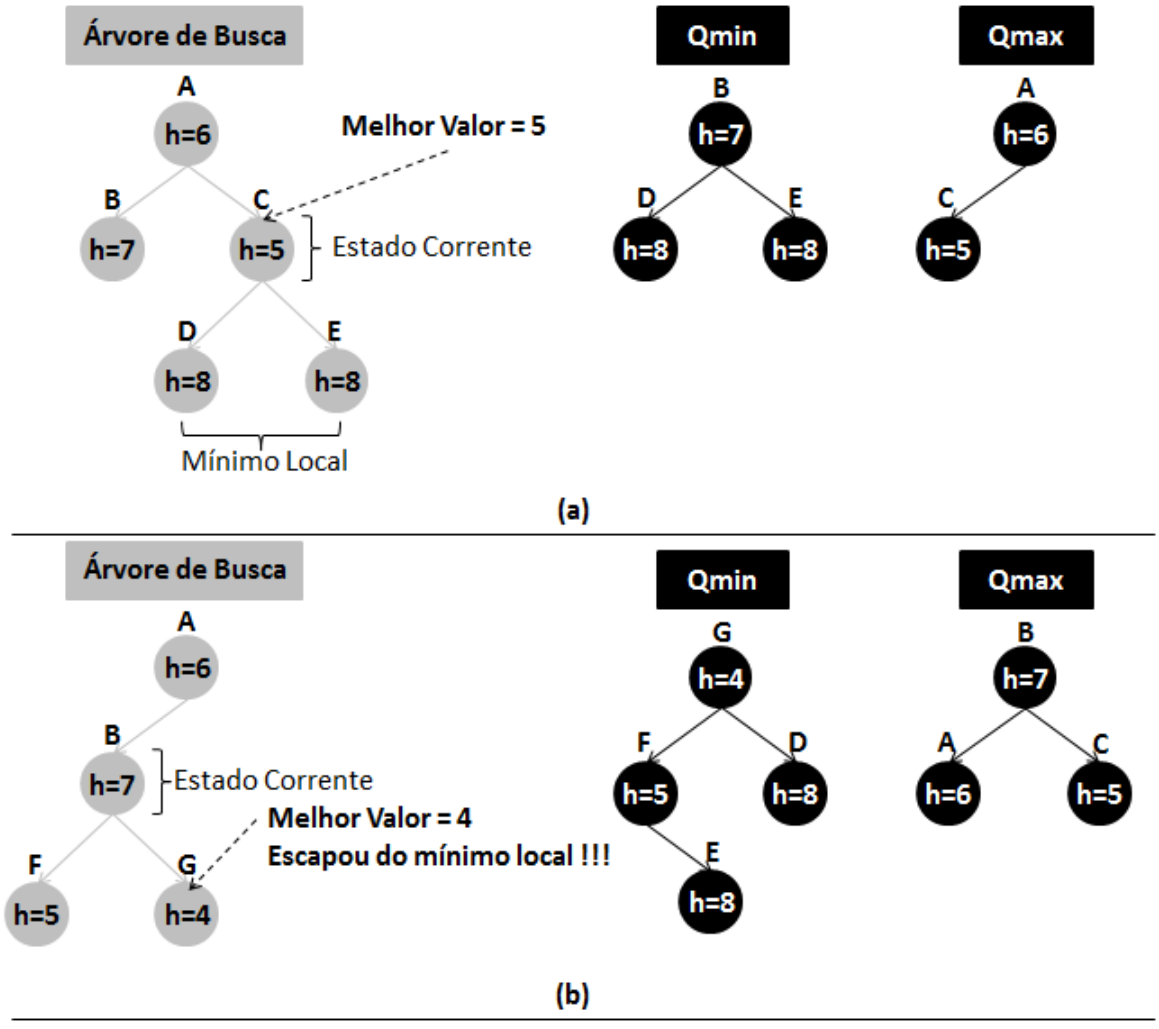

Figura 3.2: HB-EHC: escapando de ótimos locais mais rápido através do *heap* Q_{min}.

Percebeu-se que quando o algoritmo tende a armazenar muitos estados em Q_{min} e não gerar estados sucessores melhores, um algoritmo de busca local se torna um método ineficiente para resolver o problema corrente, e é realmente necessário a execução de uma busca completa. Sendo assim, a fim de não desperdiçar tempo com uma busca que não irá encontrar a solução, é determinado uma capacidade máxima para este *heap*, para que quando ela for excedida o algoritmo pare, antecipando a execução do próximo algoritmo descrito na estratégia (Figura 3.3).

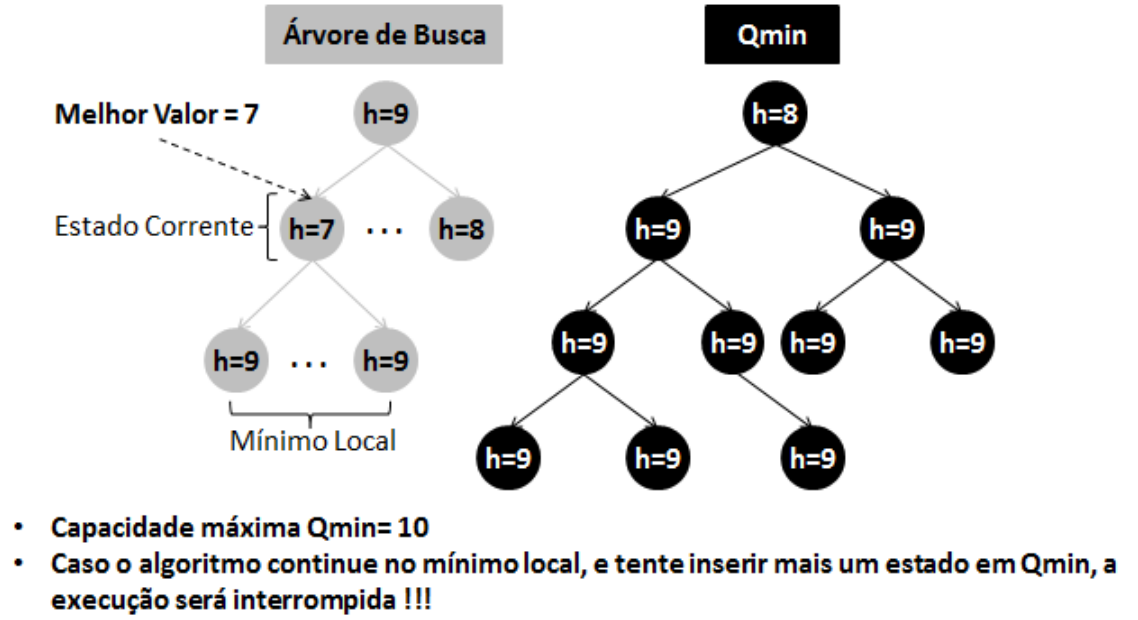

Figura 3.3: HB-EHC: antecipando a busca completa ao exceder a capacidade máxima de Q_{min}.

Mesmo com as otimizações descritas acima, o algoritmo ainda pode ficar preso em "becos sem saída". Isto ocorre quando todos os estados que estavam em Q_{min} foram avaliados e nenhum deles tinha um valor heurístico bom suficiente para escapar do máximo/mínimo local, como é mostrado na Figura 3.4(a). Nesses casos o EHC falharia. Para resolver isso, foi proposto fazer um retrocesso até o estado predecessor do melhor estado que se tinha antes da busca entrar no máximo/mínimo local (Figura 3.4(b)). Após isso, o algoritmo não mais escolherá os mesmos estados que guiaram a busca para o "beco sem saída", porque eles já foram explorados e se encontram no *heap* Q_{max}. Desta forma, outros estados que possivelmente levaram a busca até a solução, serão avaliados.

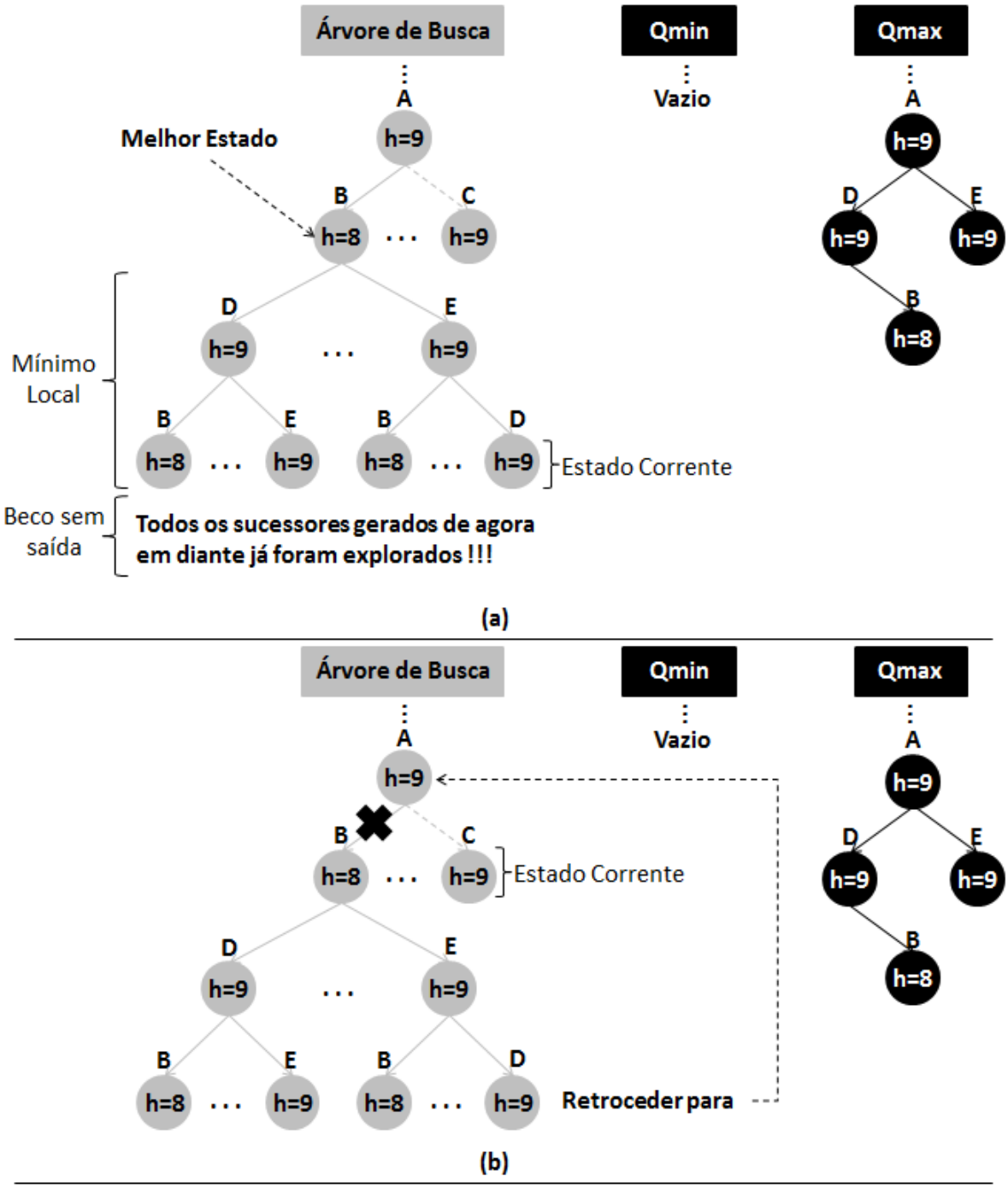

Figura 3.4: HB-EHC: retrocedendo para sair de becos sem saída.

O HC mantém somente o estado corrente em memória, mas o EHC armazena todos os estados visitados na lista $fechada$, assim como o HB-EHC faz com o *heap* Q_{max}. Através desse armazenamento, o algoritmo pode retroceder para qualquer estado já explorado em caso de "becos sem saída". Porém, para que possa haver o retrocesso para o predecessor do melhor estado que se tinha antes do máximo/mínimo local, todo estado sucessor que é gerado durante uma expansão deve manter uma referencia do estado pai que o gerou.

O uso de retrocessos deve ser moderado, caso contrário o algoritmo se comportaria como um algoritmo completo, o que não é uma característica de algoritmos como o EHC.

Além do mais, o uso excessivo de retrocessos pode gerar muitos *overshoots* durante a busca até se encontrar a solução, o que pode levar mais tempo do que a execução de algoritmos completos como por exemplo o BFS. Para evitar este problema, o algoritmo tem um parâmetro para limitar o número de retrocessos que ele pode realizar. Se esse parâmetro é excedido, a busca é interrompida. O HB-EHC também pode parar quando a busca retrocede para o estado inicial, o que caracteriza a ineficiência do método para resolver o dado problema. Em ambos os casos, o próximo algoritmo da estratégia é chamado para resolver o problema, aproveitando os estados já processados.

Algoritmo

Como é descrito no Algoritmo 3, a execução do HB-EHC pode ser descrita da seguinte forma. Ele usa o *heap* Q_{min} para armazenar os estados que ele poderá explorar, e o *heap* Q_{max} para guardar os estados já explorados.

O HB-EHC começa por inicializar os *heaps* (linhas 1-2). A variável $s_{corrente}$ (estado corrente) é atribuída com o valor do estado inicial (linha 3), e um teste é feito para verificar se $s_{corrente}$ já não satisfaz a meta especificada (linhas 4-6). A variável $melhorValor$ que mantém a melhor heurística encontrada durante a busca, é inicializada com o valor heurístico do estado corrente $h(s_{corrente})$ (linha 7). Depois, $s_{corrente}$ é adicionado ao *heap* Q_{max} (linha 8) e também ao *heap* Q_{min} (linha 9).

Em seguida o laço principal (linhas 10-42) varre todos os estados em Q_{min}. Se Q_{min} ficar vazio, significa que a busca está presa em um "beco sem saída", ou seja, o algoritmo não foi capaz de adicionar mais nenhum estado neste *heap* pelo fato de que todos os estados que ele poderia encontrar já foram explorados e adicionados em Q_{max}. Este problema é tratado com o uso de retrocesso, e é descrito posteriormente.

Internamente, existe a expansão do estado corrente (linhas 12-33) que é obtido removendo o primeiro estado de Q_{min} (linha 11). Durante a expansão são considerados somente sucessores que ainda não foram explorados ainda (linha 13). Então, para cada sucessor s', o mesmo é adicionado em Q_{max} (linha 14), $s_{corrente}$ é referenciado como sendo pai de s' (linha 15), e um teste de meta é feito sobre s' (linhas 16-18).

Depois disso, ainda dentro da expansão, é verificado se o valor heurístico de s' é menor que o melhor valor heurístico encontrado até o momento ($melhorValor$) (linha 19). Caso seja verdadeiro, $melhorValor$ será atualizado com este novo valor (linha 20). A variável $melhorEstadoLocal$, que armazena o melhor estado encontrado até o momento, recebe s' (linha 21). O *heap* Q_{min} é limpo, e somente o sucessor selecionado será mantido para que ele possa ser escolhido como estado corrente na próxima iteração (linhas 22-23). Após isso, a expansão é interrompida (linha 24) economizando tempo, sendo que os próximos sucessores possivelmente teriam valores heurísticos similares aos que já foram encontrados.

Mas se o valor heurístico de s' não é menor que $melhorValor$, ele será armazenado em Q_{min} (linha 26). Então, quando a busca entrar em um máximo/mínimo local, onde os

valores heurísticos dos estados gerados param de decrescer, o algoritmo usa uma *Busca em Largura* com os estados de Q_{min}. Desta forma, ele continua a busca até que ele encontre a meta ou um estado que tenha um valor heurístico melhor que $melhorValor$. Considerando que Q_{min} é ordenado de forma crescente pelos valores heurísticos, estados perto da raiz são os mais promissores e serão explorados primeiro. Desse modo, é possível escapar mais rápido de máximos/mínimos locais.

Ter muitos estados no *heap* Q_{min} significa que a busca está presa há muito tempo em um máximo/mínimo local. Então, provavelmente, este não é o melhor método para resolver o dado problema. Por esta razão, se a capacidade deste *heap* exceder o parâmetro c que determina sua capacidade máxima, o algoritmo interrompe sua execução (linhas 27-29).

No caso da busca entrar em um máximo/mínimo local e acabar em um "beco sem saída", o algoritmo irá retroceder até o predecessor do melhor estado que se tinha antes de entrar neste máximo/mínimo local (linha 39). Mas o uso excessivo de retrocesso pode afetar a performance da busca. Desta forma, o número de retrocessos é limitado pelo parâmetro r (linha 34). Além disso, se a busca retroceder até o estado inicial ($s_{inicial}$), a execução será interrompida (linhas 35-36). Baseado em experimentos, foi observado que os melhores resultados foram obtidos quando a capacidade máxima de Q_{min}, representada pelo parâmetro c, era igual a 30, e o limite de retrocessos r era igual a 15.

Propriedades

Assim como o EHC, o algoritmo HB-EHC consiste basicamente em um laço repetitivo que a cada iteração seleciona, dentre os sucessores do estado corrente, aquele estado que tem o melhor valor heurístico, até que este estado seja a própria meta. Quando o algoritmo não consegue fazer escolhas gulosas durante cada expansão, ele passa a realizar *Buscas em Largura* até conseguir voltar a realizar escolhas gulosas. Considerando isto, é descrito abaixo as propriedades relativas à completude, complexidade de tempo e complexidade de espaço do algoritmo.

- Completude: o objetivo deste algoritmo não é ser completo, mas sim tentar resolver o problema o mais rápido possível sem que haja necessidade de uma busca completa, sendo assim sua completude não é algo desejado. Para tanto, existe um outro algoritmo na sequência da estratégia de busca que garante a completude do planejador. Conforme foi descrito na Seção 3.2.1, existem três situações onde o algoritmo pode falhar, sendo elas: exceder a capacidade máxima do *heap* Q_{min}, retroceder ao estado inicial, e exceder o limite máximo de retrocessos definido pelo parâmetro r.
- Complexidade de Tempo:

  Teorema 3.1. *Seja d o número de iterações gastas para encontrar a solução, b o fator de ramificação durante as expansões, e n o número de estados armazenados*

Algoritmo 3 HB-EHC($s_{inicial}, s_{meta}, c, r$)

```
1:  Q_max ← {}
2:  Q_min ← {}
3:  s_corrente ← s_inicial
4:  se s_corrente = s_meta então
5:      retornar sucesso
6:  fim se
7:  melhorValor ← h(s_corrente)
8:  Q_max.adiciona(s_corrente)
9:  Q_min.adiciona(s_corrente)
10: enquanto Q_min ≠ vazio faça
11:     s_corrente ← Q_min.remove()
12:     enquanto s' ∈ sucessores(s_corrente) faça
13:         se s' ∉ Q_max então
14:             Q_max.adiciona(s')
15:             s'.pai ← s_corrente
16:             se s' = s_meta então
17:                 retornar sucesso
18:             senão
19:                 se h(s') < melhorValor então
20:                     melhorValor ← h(s')
21:                     melhorEstadoLocal ← s'
22:                     Q_min ← {}
23:                     Q_min.adiciona(s')
24:                     interromper o laço e parar a expansão
25:                 senão
26:                     Q_min.adiciona(s')
27:                     se Q_min.tamanho > c então
28:                         retornar falha
29:                     fim se
30:                 fim se
31:             fim se
32:         fim se
33:     fim enquanto
34:     se (Q_min = vazio) ∧ (r > 0) então
35:         se melhorEstadoLocal = s_inicial então
36:             retornar falha
37:         senão
38:             r ← r − 1
39:             Q_min.adicina(melhorEstadoLocal.pai)
40:         fim se
41:     fim se
42: fim enquanto
43: retornar falha
```

em um heap, então a complexidade de tempo do algoritmo HB-EHC é de ordem $O(d \times b^d \times log(n))$.

Prova. Suponha que em cada uma das d iterações, o algoritmo no pior dos casos entrasse em máximos/mínimos locais, onde é necessário realizar uma *Busca em Largura* no *heap* Q_{min}. Sabendo que o tempo gasto por uma *Busca em Largura* é de $O(b^d)$ (Seção 2.1.1), tem-se um custo até o momento de $O(d \times b^d)$. Além disso, as operações unitárias de maior custo dentro das expansões são as de manipulação dos *heaps* que tem custo de $O(log(n))$, onde n representa a quantidade de estados dentro de um *heap*. Portanto a complexidade de tempo do algoritmo é de $O(d \times b^d \times log(n))$.

- Complexidade de Espaço:

Teorema 3.2. *Seja d o número de iterações gastas para encontrar a solução, b o fator de ramificação durante as expansões, então a complexidade de espaço do algoritmo HB-EHC é de ordem* $O(d \times b^d)$.

Prova. Exitem apenas duas estruturas capazes de armazenar dados durante a busca, que são os *heaps* Q_{min} e Q_{max}. O *heap* Q_{min} tem sua capacidade limitada pelo parâmetro c do algoritmo, que é uma constante. Já o *heap* Q_{max} armazena todos os estados explorados durante a busca. Suponha então que, em cada uma das d iterações o algoritmo no pior dos casos necessitasse de realizar uma *Busca em Largura*, e no fim acabar tendo armazenado todos os estados em Q_{max}. Sabendo que o espaço necessário para uma *Busca em Largura* é de $O(b^d)$ (Seção 2.1.1), tem-se um custo total de espaço para o algoritmo de $O(d \times b^d)$.

3.2.2 Adaptive-LRTA*

Nesta seção é apresentado o algoritmo Adaptive-Learning Real Time A* (Adaptive-LRTA*), uma extensão do LRTA*. As otimizações feitas neste algoritmo tem como objetivo fazer uma busca completa mais eficiente no espaço de estados para encontrar a solução. O Adaptive-LRTA* só será executado caso o primeiro algoritmo descrito na estratégia de busca tenha falhado, no entanto ele irá aproveitar os estados já processados anteriormente.

Como ele é responsável por manter a completude do planejador, sua busca deve obrigatoriamente encontrar a solução; sendo assim, é necessário realizar um balanceamento dos estados em memória caso não haja mais espaço suficiente para armazenamento durante a busca. Para acelerar sua convergência, escolhas gulosas são feitas durante algumas iterações. Por fim, uma de suas principais vantagens é o uso de "aprendizagem" para melhor avaliar estados que já apareceram na busca, escapando automaticamente de máximos/mínimos locais e "becos sem saída".

Descrição

O algoritmo usa um *heap máximo* para armazenar os estados gerados durante a busca. Este *heap* é o mesmo usado pelo HB-EHC, o Q_{max}, utilizado para armazenar os estados já explorados e que são ordenados de forma decrescente, ou seja, estados considerados ruins por terem valores heurísticos maiores ficam próximos a raiz. Desta forma, caso o algoritmo venha a precisar de espaço para continuar a busca, esses estados que se encontram mais próximos da raiz serão removidos, pois as chances de se usar esses estados novamente são remotas.

Manter os estados em algum tipo de estrutura previne o recálculo de suas heurísticas cada vez que eles aparecerem na busca, como pode ser visto na Figura 3.5. Nos casos em que a meta está muito afastada do estado em questão, este cálculo é muito lento. Além do mais, armazenar estados garante que quando a heurística de um estado for atualizada através do aprendizado do algoritmo, ela não será reinicializada quando o mesmo reaparecer na busca.

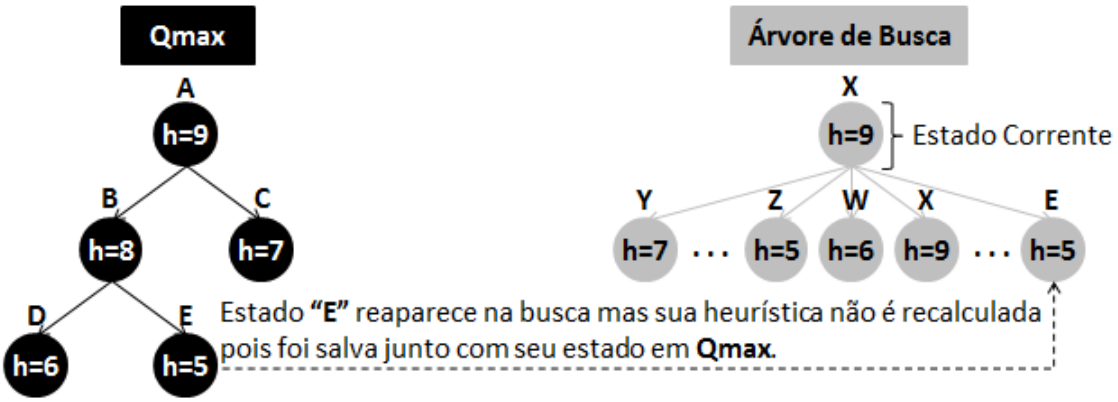

Figura 3.5: Adaptive-LRTA*: armazenamento de estados e seus valores heurísticos.

O Adaptive-LRTA* usa a heurística de planejamento h_{FF}, bem como vários planejadores de sucesso. Considerando essa heurística como uma boa função para estimar o número de ações necessárias para alcançar a meta, acredita-se que estados com um alto valor heurístico não serão promissores para a busca. Baseado neste aspecto, e pelo fato de que existem problemas de planejamento que geram muitos estados sucessores com o mesmo valor heurístico durante a expansão do estado corrente, podar sucessores ruins é uma opção eficiente para economizar espaço. Por esta razão, apenas alguns dos sucessores que estão empatados com o menor valor heurístico durante uma expansão serão salvos no *heap*, enquanto os outros serão podados como pode-se ver na Figura 3.6. Baseado em experimentos, foi observado que a taxa de poda de 70% proporcionou os melhores resultados.

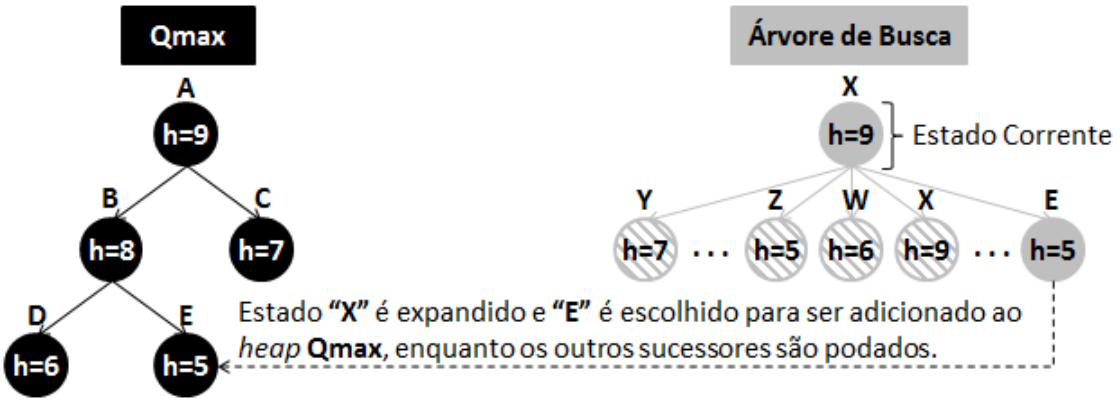

Figura 3.6: Adaptive-LRTA*: poda de sucessores durante a expansão.

O processo de aprendizagem é uma das grandes vantagens do algoritmo, que é próprio de algoritmos do tipo LRTA*. Isto é feito atualizando o valor heurístico do estado corrente, usando o resultado obtido da função de avaliação do melhor sucessor, que permite uma avaliação mais precisa deste estado caso ele reapareça na busca. Então, a heurística de um estado pode diminuir se a busca está convergindo para a meta passando por ele, ou aumentar caso contrário. Neste caso, estados que tinham sido podados em um primeiro

momento terão a oportunidade de ser adicionados ao *heap* se os valores heurísticos dos vizinhos que foram selecionados anteriormente aumentar, como é mostrado na Figura 3.7.

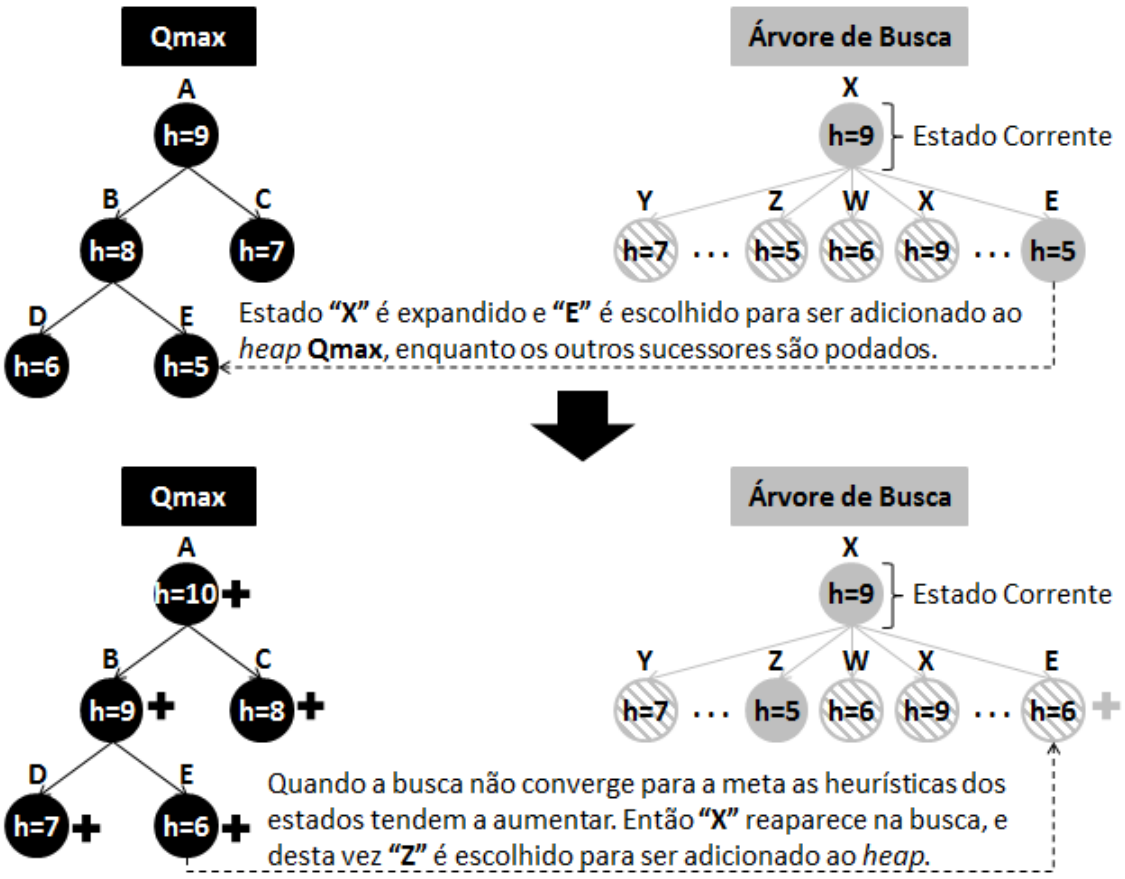

Figura 3.7: Adaptive-LRTA*: processo de aprendizagem.

O aprendizado também possibilita que o algoritmo faça retrocessos escapando facilmente de máximos/mínimos locais e até de "becos sem saída". Para permitir isso, cada estado deve guardar uma referência de seu estado pai. Então, após a expansão de um estado, não só seus sucessores são avaliados, mas também o seu estado pai. Portanto, se o estado corrente gerou sucessores piores que o seu antecessor, o algoritmo pode optar por retroceder para este estado antecessor e expandi-lo novamente, onde outro caminho será selecionado senão aquele que gerou o máximo/mínimo local onde se teve um acréscimo na heurística (Figura 3.8). Sem retrocesso, a execução poderia falhar e todo o processamento seria descartado.

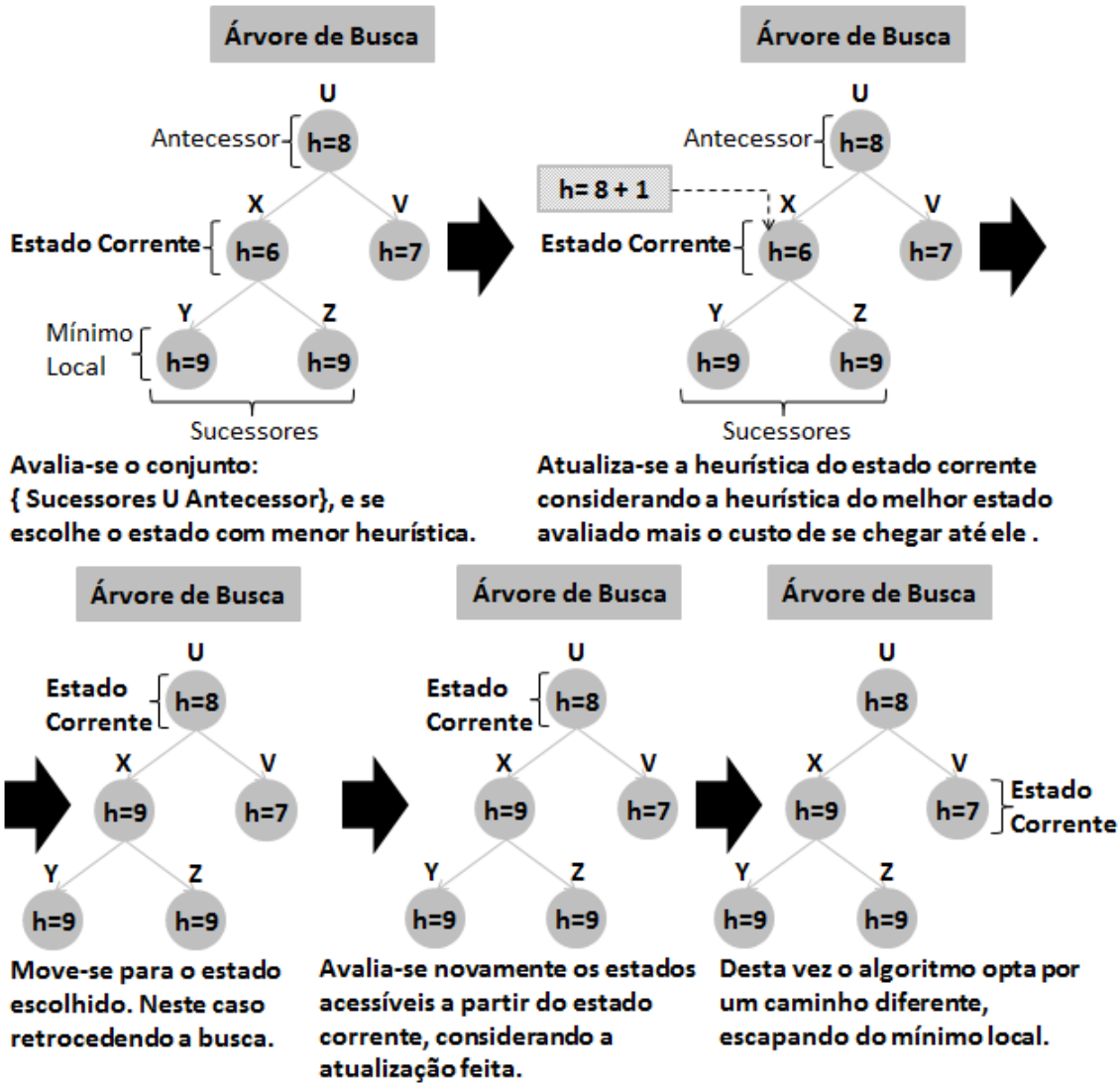

Figura 3.8: Adaptive-LRTA*: escapando de máximos/mínimos locais.

Caso um estado seja gerado diversas vezes durante a busca, apenas uma instância dele permanecerá no *heap* Q_{max}, o que também economiza espaço. Além disso, quando um estado gerar ele mesmo como sucessor, este sucessor será podado para evitar *loops*.

Um parâmetro importante do algoritmo é o que representa a *capacidade máxima do heap*, que determina quantos estados poderão ser armazenados em Q_{max}. Se a busca exceder este parâmetro, a raiz do *heap*, que sempre contém o pior estado dentre todos os já explorados, será removida repetidas vezes até que a capacidade seja restabelecida. Esta situação é ilustrada na Figura 3.9.

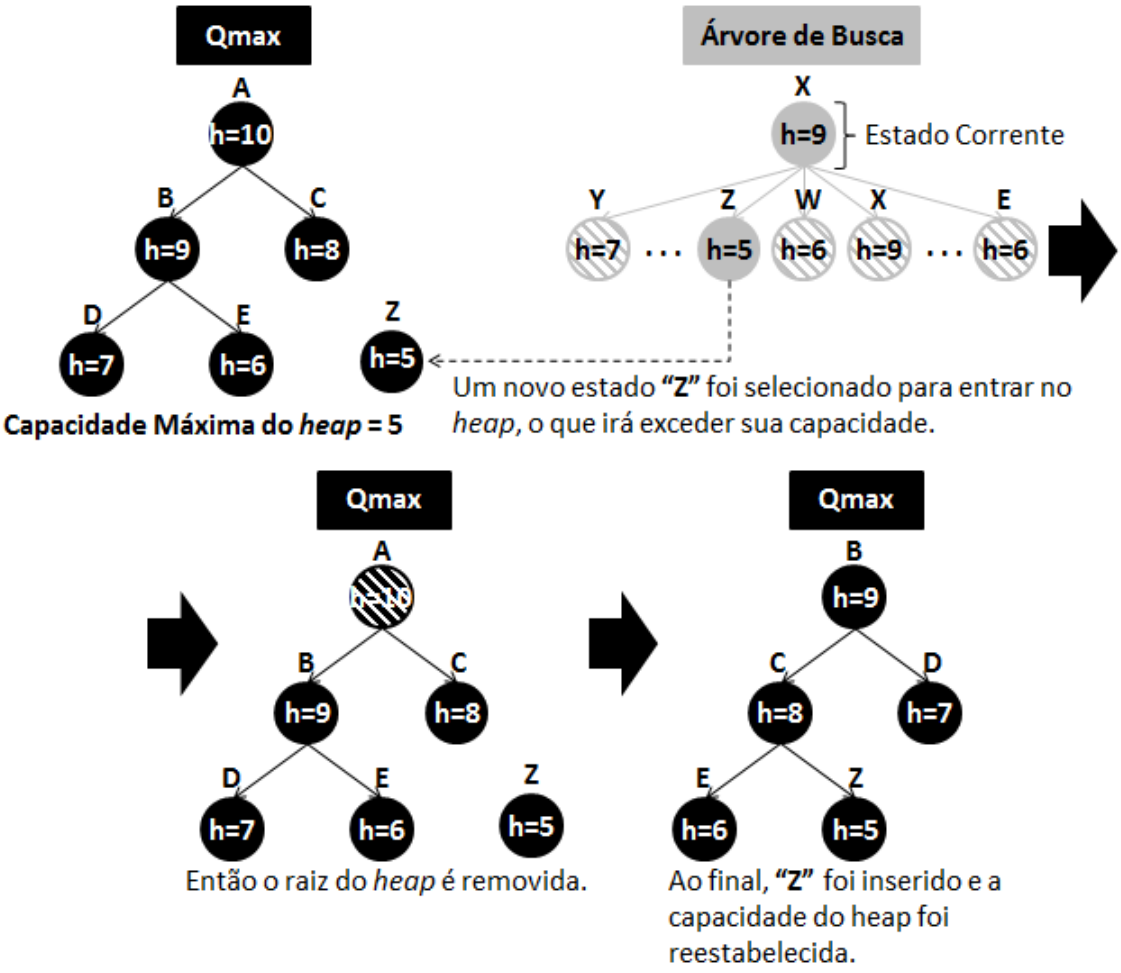

Figura 3.9: Adaptive-LRTA*: balanceamento do *heap* Q_{max}.

Como foi dito, a heurística h_{FF} estima o número de passos entre a meta e um dado estado. Considerando os n passos entre o estado inicial e a meta (dado pelo valor heurístico do estado inicial), o algoritmo tenta realizar n iterações gulosas para tentar chegar a solução o mais rápido possível, o que é muito eficaz especialmente para problemas não tão complexos onde este valor representa o número exato de iterações para se alcançar a meta, como é mostrado na Figura 3.10.

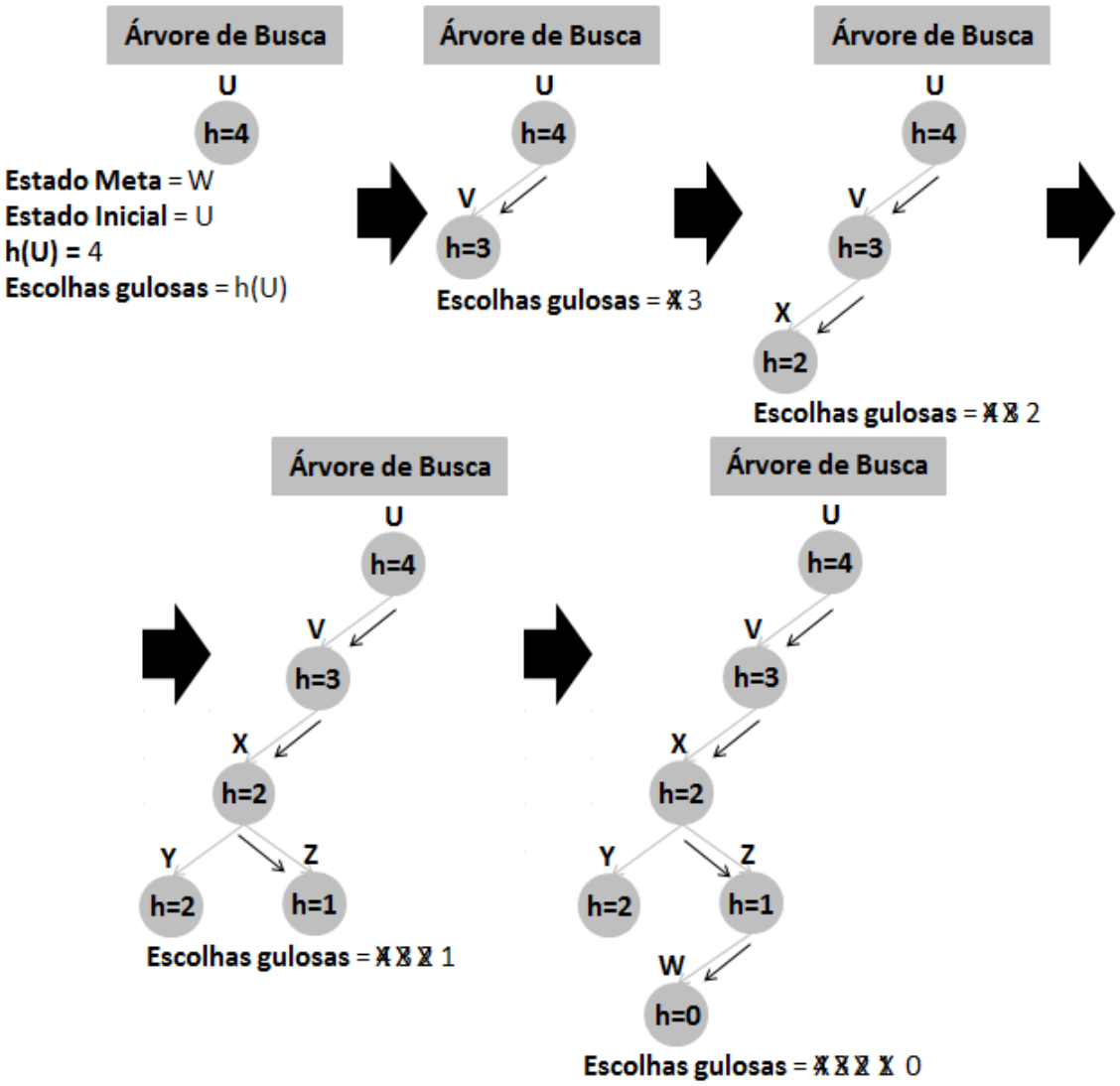

Figura 3.10: Adaptive-LRTA*: encontrando a solução a partir de iterações gulosas.

Mesmo se o Adaptive-LRTA* não encontrar a solução durante essas iterações gulosas, a busca terá encontrado um estado mais próximo da meta muito rápido, acelerando a convergência (Figura 3.11). Caso as escolhas gulosas guiem a busca para um máximo/mínimo local, o algoritmo conta com características como "aprendizado" e "retrocesso" para escapar deles.

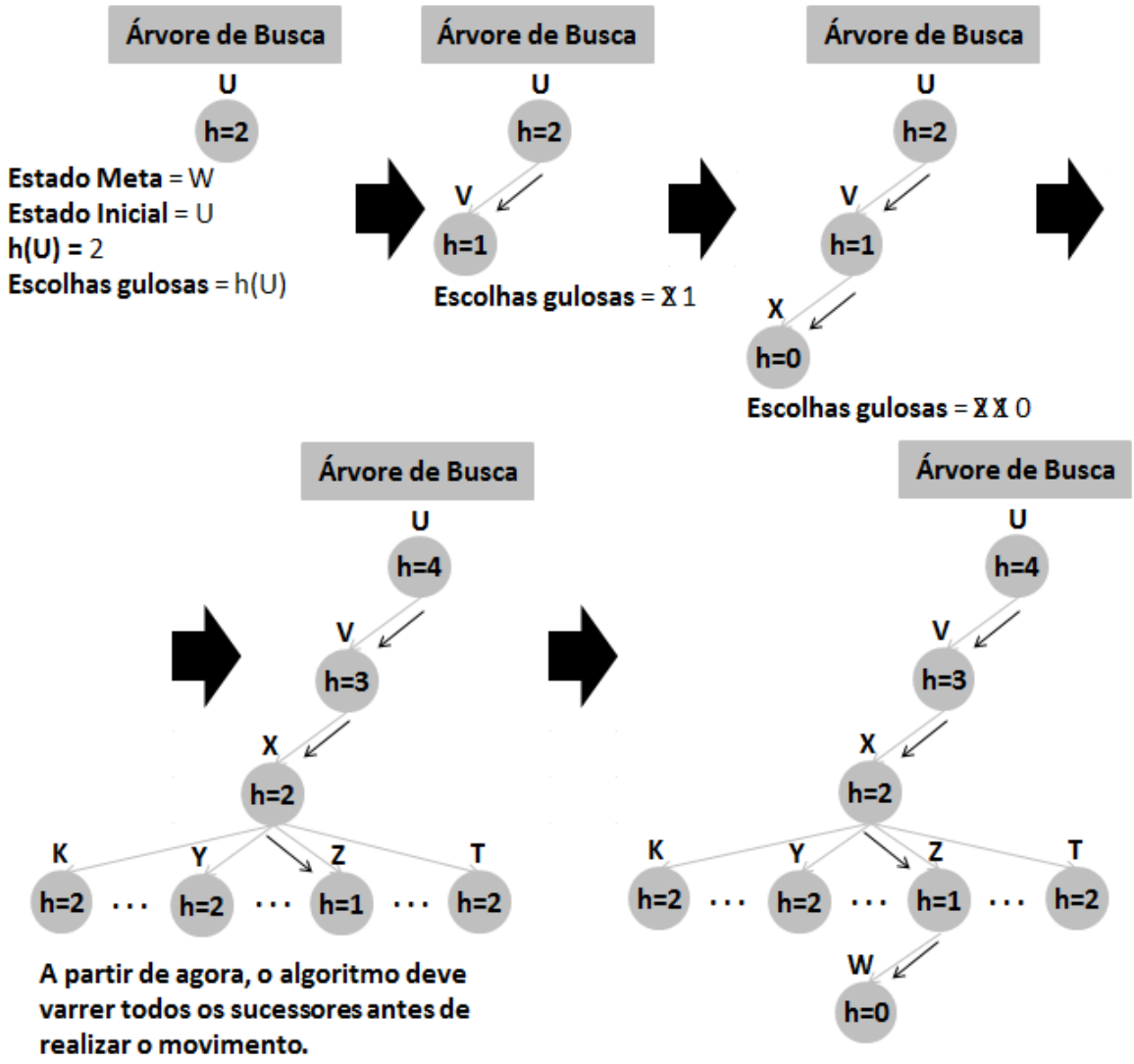

Figura 3.11: Adaptive-LRTA*: acelerando a convergência com iterações gulosas.

O LRTA* deve ser executado repetidas vezes até que os valores heurísticos dos estados parem de sofrer atualizações para encontrar a solução ótima. Mas o principal objetivo de um planejador é encontrar a solução o mais rápido possível. Por esta razão, será realizada apenas uma execução do algoritmo Adaptive-LRTA*. A característica de aprendizagem continua muito útil, mesmo com apenas uma rodada do algoritmo, pois é muito comum um estado reaparecer várias vezes durante a busca, o que faz com que sua avaliação fique cada vez mais precisa.

Algoritmo

O Algoritmo 4 mostra o pseudo-código do Adaptive-LRTA*. Ele começa por inicializar o *heap* Q_{max} (linha 1). A variável $s_{corrente}$ (estado corrente) é inicializada com o valor do estado inicial (linha 2) e adicionada ao *heap* (linha 3). O número de escolhas gulosas que o algoritmo pode fazer é mantido na variável n, que começa com o valor de $h(s_{corrente})$ (heurística do estado corrente/inicial) (linha 4). Depois, o laço principal se inicia (linhas

5-42). Primeiramente, o balanceamento do *heap* Q_{max} é feito caso necessário (linhas 6-8). Ele consiste em remover a raiz de Q_{max} repetidamente até que sua capacidade fique menor do que o parâmetro c (capacidade máxima do *heap*). Isso é necessário para disponibilizar memória suficiente durante a busca. Logo após, um teste é feito para verificar se $s_{corrente}$ satisfaz a meta especificada. Quando a meta é alcançada, o algoritmo para sua execução e retorna sucesso (linhas 9-11).

A variável $melhorValor$ armazena a menor soma $c(s_{corrente}, s) + h(s)$ dentre todos sucessores s avaliados, a partir do estado corrente (incluindo o antecessor do estado corrente), onde $c(s_{corrente}, s)$ é o custo de se alcançar s a partir de $s_{corrente}$ e $h(s)$ representa o custo estimado para se chegar a meta a partir de s. Esses sucessores referenciam $s_{corrente}$ como sendo seu antecessor, o que permite retrocessos. Além disso, caso o estado sucessor gerado seja igual ao estado $s_{corrente}$, o mesmo será podado. Para cada iteração, $melhorValor$ é reinicializado com ∞ (linha 12).

No próximo passo, a fase de expansão se inicia (linhas 13-29). Começa-se por verificar se o algoritmo pode realizar escolhas gulosas. Para permitir isso, o número restante de escolhas gulosas mantido na variável n deve ser maior que zero, e o valor heurístico de s deve ser maior que o de $s_{corrente}$ (linha 14). Se todas as condições forem satisfeitas, $melhorValor$ recebe o resultado da soma $c(s_{corrente}, s) + h(s)$ (linha 15), um conjunto S armazena o sucessor selecionado s (linha 16), o número de escolhas gulosas possíveis é decrementado (linha 17), e a expansão é interrompida (linha 18).

Mas se escolhas gulosas não puderem ser executadas, o algoritmo funciona da forma descrita a seguir. O conjunto S guardará todos os sucessores que possuem valores iguais a $melhorValor$. Então, caso a soma $c(s_{corrente}, s) + h(s)$ de um sucessor seja menor que $melhorValor$ (linha 20), $melhorValor$ recebe $c(s_{corrente}, s) + h(s)$ (linha 21), S é limpo, restando apenas o sucessor s.

Após a expansão, de todos os sucessores remanescentes em S, um deles é escolhido aleatoriamente (linha 30). O valor heurístico do estado corrente é atualizado com $melhorValor$, que representa a melhor avaliação dos sucessores analisados (linha 31). Com este novo valor, o estado deve ser atualizado em Q_{max}. Esta atualização deve aproximar o estado da raiz se sua heurística piorar (linha 32), ou afastá-lo caso contrário. O sucessor selecionado é adicionado ao *heap* caso ele já não esteja lá (linhas 33-35). O mesmo acontece com uma porcentagem dos sucessores que estão em S, e o restante deles é podado para economizar espaço de memória (linhas 36-40). A taxa de poda é dada pelo parâmetro p. Finalmente, o movimento é feito em direção ao sucessor selecionado (linha 41), e o algoritmo parte para a próxima iteração.

Algoritmo 4 Adaptive-LRTA*($s_{inicial}, s_{meta}, p, c$)

```
    Q_max ← {}
 2: s_corrente ← s_inicial
    Q_max.adiciona(s_corrente)
 4: n ← h(s_corrente)
    laço
 6:     enquanto Q_max.tamanho() > c faça
            Q_max.remove()
 8:     fim enquanto
        se s_corrente = s_meta então
10:         retornar sucesso
        fim se
12:     melhorValor ← ∞
        para s ∈ sucessores(s_corrente) faça
14:         se (n > 0) ∧ (h(s) < h(s_corrente)) então
                melhorValor ← c(s_corrente, s) + h(s)
16:             S ← {s}
                n ← n − 1
18:             interromper o laço e parar a expansão
            senão
20:             se c(s_corrente, s) + h(s) < melhorValor então
                    melhorValor ← c(s_corrente, s) + h(s)
22:                 S ← {s}
                senão
24:                 se c(s_corrente, s) + h(s) = melhorValor então
                        S ← S ∪ {s}
26:                 fim se
                fim se
28:         fim se
        fim para
30:     s' ← S{random(S.tamanho())}
        h(s_corrente) ← melhorValor
32:     Q_max.atualizaChave(s_corrente, h(s_corrente))
        se s' ∉ Q_max então
34:         Q_max.adiciona(s')
        fim se
36:     para i ← 0 até (S.tamanho() × p) faça
            se S{i} ∉ Q_max então
38:             Q_max.adiciona(S{i})
            fim se
40:     fim para
        s_corrente ← s'
42: fim laço
```

Propriedades

As propriedades do algoritmo Adaptive-LRTA* são similares as da versão original do LRTA* proposto por Korf. Nas seguintes explicações e teoremas são levantadas as teorias que garantem que a completude do LRTA* ainda é válida para o Adaptive-LRTA*, além das análises de complexidade de tempo e espaço do algoritmo.

- Completude:

  Teorema 3.3. *Em um problema com espaço de estados finito, custos de transições positivas e valores heurísticos finitos, onde a meta é alcançada a partir de qualquer*

estado, o algoritmo Adaptive-LRTA encontra a solução.*

Prova. A completude pode ser provada quase da mesma forma para qualquer algoritmo da família LRTA*. Por contradição, assume-se que o algoritmo Adaptive-LRTA* é incapaz de encontrar a solução caso ela exista. Para que isso aconteça em um problema com espaço de estados finito, deve existir um ciclo tal que o Adaptive-LRTA* fica preso para sempre, onde não há a presença do estado meta.

Considerando que a meta é alcançável a partir de qualquer estado, deve haver pelo menos uma transição que guia a busca para fora do ciclo. Então, tudo que se deve mostrar é como o algoritmo consegue encontrar esta transição para escapar do ciclo. Para todos os estados encontrados na busca, cada um tem um valor heurístico. Para estados já visitados, este valor heurístico é o valor corrente salvo com o estado na memória. Para os não salvos, este valor é dado pela função heurística.

Para cada movimento, o algoritmo faz uma avaliação dos estados acessíveis a partir do estado corrente considerando seus valores heurísticos, e o custo entre cada um deles e o estado corrente. A avaliação mais baixa é usada para atualizar a heurística do estado corrente. Então o movimento é feito para o melhor estado encontrado. Mesmo empregando o conceito de escolhas gulosas no algoritmo, o efeito permanece o mesmo. Agora, considere a situação onde o estado corrente entrou em um ciclo infinito hipotético. Então, a menor avaliação dos estados acessíveis a partir dele é maior que seu próprio valor heurístico, o que aumenta seu valor. Lembrando que nessas condições, escolhas gulosas nunca são usadas.

Desta forma, cada iteração do algoritmo em volta deste ciclo aumenta o valor da menor avaliação. Além disso, os valores de todos os estados do ciclo aumentam sem limites. Em algum momento, o valor heurístico de um estado, que faz parte de um caminho que guia a busca para fora do ciclo, terá valor menor que o resto dos outros estados. Isto faz com que o algoritmo saia do ciclo, o que seria impossível, considerando a hipótese.

A adaptação referente a política de podas de estados não impacta na propriedade de completude, pois ela só é empregada quando não existe mais espaço de memória para salvar todos os estados do espaço de estados finito do problema. Então, se o LRTA* pode resolver o problema sem precisar de espaço extra (mantendo sua completude), o Adaptive-LRTA* também pode resolver o problema sem realizar podas de estados da memória.

Para garantir que o algoritmo seja ótimo, deve-se executar o Adaptive-LRTA* repetidas vezes até que os valores heurísticos dos estados parem de sofrer atualizações, assim como no LRTA*. Mas o principal objetivo deste algoritmo é encontrar a solução o mais rápido possível, desta forma somente uma execução do Adaptive-LRTA* é realizada, desconsiderando o fator ótimo neste caso.

- Complexidade de Tempo:

  Teorema 3.4. *A complexidade de tempo do algoritmo Adaptive-LRTA* por movimento é de ordem $O(|S| \times log(c))$, onde S é o conjunto dos melhores estados acessíveis a partir do estado corrente, e c é a quantidade de estados que podem ser salvos no heap pelo algoritmo.*

  Prova. Como foi descrito no Algoritmo 4, para cada movimento do Adaptive-LRTA* existem três tarefas importantes que afetam a complexidade de tempo do algoritmo. A primeira é o balanceamento da memória (linhas 6-7), a segunda é o processo de expansão (linhas 13-29), e a terceira está relacionada à inserção e atualização de uma taxa de sucessores dentro do *heap* que tem o menor valor heurístico encontrado durante a expansão (linhas 36-40).

  Considere o pior caso que acontece quando:

  - O limite de escolhas gulosas foi excedido.
  - Todos os sucessores tem o mesmo valor heurístico, desta forma todos os sucessores são armazenados em S.
  - O parâmetro p tem valor de 100%, o que significa que todos os estados em S deverão ser inseridos ou atualizados no *heap*.
  - O *heap* já está cheio, ou seja, o parâmetro c que representa a capacidade máxima do *heap* foi excedido.

  Avaliando o processo de expansão, nota-se que todos os sucessores serão armazenados em S, logo, o fator de ramificação nesse caso é igual a $|S|$. Sendo todas as operações dentro da expansão na ordem de *O(1)*, o custo total da expansão é de $|S| \times O(1)$, ou seja, *O(|S|)*.

  Se todos os estados de S devem ser salvos, o número de operações no *heap* é igual a quantidade de estados armazenados em S. Qualquer operação no *heap* neste caso tem custo de $O(log(c))$, o que significa que esta tarefa é de ordem $O(|S| \times log(c))$.

  Imagine que a última iteração tenha inserido $|S|$ estados no *heap* também. Desta forma, no início da iteração corrente, o algoritmo deve realizar o balanceamento do *heap* removendo sua raiz $|S|$ vezes, a fim de liberar espaço suficiente para a busca. Consequentemente, o custo para isto também é de $O(|S| \times log(c))$ pois $|S|$ remoções no *heap* são necessárias, onde o custo de cada uma é *O(log(c))*.

  Analisando todos os custos, concluí-se que a complexidade de tempo é: $O(|S| \times log(c)) + O(|S| \times log(c)) + O(|S|) < O(a \times |S| \times log(c))$, $\forall$ constante $a \geq 3$.

  $\therefore$ Adaptive-LRTA* é de ordem $O(|S| \times log(c))$ por movimento.

- Complexidade de Espaço:

Teorema 3.5. *A complexidade de espaço do algoritmo Adaptive-LRTA* é de ordem* $O(|S| + c)$*, onde* S *é o conjunto dos melhores estados acessíveis a partir do estado corrente, e* c *é a quantidade de estados que podem ser salvos no heap pelo algoritmo.*

Prova. No pior caso, onde o *heap* está cheio e todos os estados de S devem ser salvos, o algoritmo alcança seu limite de espaço. Neste ponto, existem c estados no *heap* e $|S|$ estados no conjunto S para serem transferidos para o *heap*. Por esta razão, a complexidade de espaço do algoritmo Adaptive-LRTA* é de ordem $O(|S| + c)$.

Capítulo 4

Experimentos e Discussões

Este capítulo introduz detalhes relacionados aos testes executados para analisar a performance do planejador desenvolvido, o SLPlan. Como o SLPlan é baseado numa estratégia de busca composta por dois algoritmos, HB-EHC e Adaptive-LRTA*, foram realizados primeiramente experimentos separados para comparar o desempenho de cada um deles com outros algoritmos de mesma característica. Depois, foram executados testes para avaliar a estratégia como um todo, para isso se usou a estratégia proposta pelo planejador FF como comparação.

Para tanto, este capítulo foi estruturado da seguinte forma: na Seção 4.1 é mostrada a metodologia empregada nos testes; a seção 4.2 apresenta os domínios e problemas de planejamento que compõe o *benchmark*; a Seção 4.3 detalha as baterias de testes realizadas para avaliar os algoritmos e o planejador desenvolvido, além de discutir os resultados obtidos.

4.1 Metodologia

Os experimentos foram executados em duas partes. A primeira tinha como objetivo avaliar as melhorias feitas em cada algoritmo da estratégia de busca do planejador SLPlan. A finalidade disto era confirmar o bom desempenho dos algoritmos em comparação com versões semelhantes a eles. No caso do HB-EHC, que é um método não completo porém bastante ágil, utilizou-se os algoritmos *Hill Climbing* (HC) e *Enforced Hill Climbing* (EHC). Já o algoritmo Adaptive-LRTA* foi comparado com a versão original do LRTA* e uma versão melhorada conhecida como LRTA*-K, onde todos são métodos completos e utilizam "aprendizado" durante as buscas para melhorar a convergência para a solução.

A segunda parte dos testes é destinada a avaliar a estratégia de busca, ou seja, analisar a eficiência do planejador como um todo. Para isso, foi utilizado o planejador FF como comparação. Lembrando que o planejador FF tem a seguinte estratégia: executa-se o algoritmo EHC para resolver o problema e caso ele falhe uma busca completa é feita com o algoritmo BFS, onde todos os dados já processados são descartados. Já o SLPlan

executa o HB-EHC armazenando os dados em um *heap* e caso ele falhe o algoritmo Adaptive-LRTA* é acionado, aproveitando os dados armazenados neste *heap*.

Foi montado um *benchmark* com 6 domínios de planejamento contendo mais de 160 problemas. Para a primeira fase de testes, onde se avaliou os algoritmos de forma separada, somente alguns domínios foram utilizados. Já na segunda, ambos os planejadores, SLPlan e FF, foram submetidos a todos os domínios.

Os experimentos foram realizados neste *benchmark* sob as mesmas condições de *hardware* e *software*. As configurações de *hardware* e *software* incluem: Sistema Operacional linux OpenSuse 12.1, processador Intel Centrino (dual core de 1.4GHz cada núcleo), 1.9GiB de memória e Disco Rígido SATA-82801 de 160GB de espaço.

Foram realizadas 50 execuções, de cada método, para cada um dos problemas do *benchmark*. Além disso, foi estipulado durante os testes um tempo máximo de 30 minutos para as execuções, o mesmo tempo utilizado nas competições de planejamento da IPC. Nos resultados, foram considerados apenas problemas onde todos os métodos avaliados conseguiram finalizar suas execuções dentro do tempo máximo estipulado, seja por terem encontrado a solução, por não terem encontrado a solução (no caso de algoritmos não-completos), ou por terem encerrado sua execução por falta de espaço de memória (no caso de algoritmos completos).

4.2 Benchmark

O *benchmark* usado inclui problemas de planejamento da categoria STRIPS das seguintes edições da International Planning Competition: IPC-3, IPC-4 e IPC-5. No total, o *benchmark* é constituído por 162 problemas. A IPC é um mecanismo importante para alavancar pesquisas na área de planejamento. O objetivo geral da IPC é impulsionar o estado da arte das tecnologias de planejamento através da apresentação de novos desafios científicos para a comunidade de pesquisadores a cada edição [Gerevini et al. 2009]. A descrição de cada domínio de planejamento utilizado, bem como o seu número de problemas e a edição da IPC na qual o mesmo foi apresentado, podem ser vistos na Tabela 4.1.

Geralmente, cada domínio tem cerca de 20 instâncias de problemas associados a ele, onde esses problemas são ordenados de forma crescente de acordo com sua complexidade, dada pelo número de constantes e meta estabelecida [Long e Fox 2006]. No *benchmark* proposto, o domínio DEPOTS contêm 22 problemas, onde os problemas 01-06 são pequenos, 07-09 aumenta-se o número de superfícies, 10-12 são inseridas mais localidades, 13-15 tem mais superfícies e localidades, 16-18 são adicionados mais recursos como caminhões e guinchos, e 19-22 aumenta-se a escala do problema.

No domínio DRIVERLOG existem 20 problemas distribuídos da seguinte forma: 01-08 são problemas simples, 09-13 são adicionadas mais localidades e 14-20 são aplicados mais

recursos.

Já o ROVERS é um domínio relativamente simples, exceto pelos últimos problemas, do total de 20. Sendo assim, os problemas 01-18 são fáceis, porém, 19-20 são de difícil resolução.

O domínio ZENOTRAVEL também possui 20 problemas onde 01-07 são problemas simples, 08-13 aumenta-se a complexidade e 14-20 são problemas difíceis.

O PIPESWORLD é um domínio muito complexo e tem 50 problemas, onde até mesmo os 10 primeiros são muito difíceis de se resolver.

Para finalizar, o domínio TPP possui 30 problemas sendo 01-08 fáceis, 09-11 de nível médio e 12-30 são quase impossíveis de se resolver, considerando o seu fator de ramificação.

Tabela 4.1: Domínios de planejamento do *benchmark*

Domínio	**Competição**	**Descrição**	**#Problemas**
DEPOTS	IPC-3	Domínio de gerenciamento de depósitos, onde caminhos transportam caixas que depois deverão ser empilhadas em pálets em seus destinos.	22
DRIVERLOG	IPC-3	Este domínio envolve dirigir caminhões que entregam pacotes em locais. A complexidade disto é que caminhões requerem motoristas que, devem caminhar entre os diversos caminhões do pátio para selecionar aqueles que devem ser dirigidos. Os caminhos para caminhar no pátio e as ruas onde os caminhões transitam, formam diferentes mapas entre os locais.	20
ROVERS	IPC-3	O domínio ROVERS foi construído como uma representação simplificada do problema que confronta as missões "Mars Exploration Rover" da NASA lançada em 2003, a missão "Mars Science Laboratory" realizada em 2009 e outras missões similares. A versão STRIPS do problema envolve o planejamento para vários viajantes (rovers), equipados com diferentes, mas possivelmente sobrepostos, conjuntos de equipamentos para atravessar a superfície do planeta. Os viajantes devem se deslocar entre pontos de marcação, coletando dados e transmitindo-os de volta para a nave. A travessia é complicada pelo fato de que algumas sondas estão restritas a viajar sobre certos tipos de terreno e isso faz com que determinadas rotas sejam intransponíveis para algumas das sondas. A transmissão de dados também é limitada pela visibilidade da nave e dos pontos de marcação.	20
ZENOTRAVEL	IPC-3	Este domínio possui ações de embarque e desembarque de passageiros em aeronaves, que podem por sua vez voar em dois níveis diferentes de velocidades entre as diversas localidades.	20
PIPESWORLD	IPC-4	Domínio de oleoduto que modela o problema do transporte de lotes de produtos petrolíferos em uma rede de dutos, com ou sem restrições de tancagem (espaço em tanques de armazenamento intermediários).	50
TPP	IPC-5	O domínio TPP é inspirado pelo problema "Comprador Viajante" que é uma generalização conhecida do Problema do "Caixeiro Viajante". O problema pode ser definido da seguinte forma: tem-se um conjunto de produtos e para cada produto uma demanda conhecida. Tem-se também um conjunto de mercados que podem fornecer uma quantidade limitada conhecida de cada produto a um preço conhecido. O comprador deve selecionar um subconjunto dos mercados de tal forma que possa comprar os produtos listados nas demandas, e construir um caminho que começa de um dado local (chamado depósito) visitando todos os mercados selecionados, e retorne ao local de origem.	30

4.3 Resultados

Considerando o limite máximo de tempo para tentar resolver os problemas do *benchmark*, foram levados em consideração nos resultados somente os problemas onde os métodos avaliados conseguiram encerrar suas execuções dentro deste tempo. Desta forma, execuções que chegaram a solução, ou foram interrompidas por não chegarem a solução (algoritmos não-completos), ou falharam por falta de espaço de memória (algoritmos completos), sem exceder os 30 minutos foram contabilizadas.

4.3.1 Resultados parciais: HB-EHC e Adaptive-LRTA*

O HB-EHC foi comparado aos algoritmos HC e EHC utilizando os domínios: DEPOTS, DRIVERLOG, ZENOTRAVEL e PIPESWORLD. Este teste tem como objetivo analisar o tempo médio e a taxa de completude dos algoritmos. Estes dados podem ser visualizados na Tabela 4.2.

Tabela 4.2: Taxa de completude e tempo médio de execução: HC, EHC e HB-EHC.

Domínio	Problema	Completude			Tempo		
		HC	EHC	HB-EHC	HC	EHC	HB-EHC
DEPOTS	01	90%	100%	100%	0,18	0,20	0,20
	02	0%	100%	100%	.....	0,50	0,56
	03	0%	14%	14%	.....	4,93	6,58
	07	0%	12%	42%	.....	6,17	5,75
	13	0%	100%	100%	.....	29,63	27,94
	16	0%	100%	100%	.....	40,45	41,07
DRIVERLOG	01	0%	100%	100%	.....	0,07	0,07
	02	0%	0%	0%	.....	.....	.....
	03	0%	100%	100%	.....	0,17	0,18
	04	0%	0%	0%	.....	.....	.....
	05	0%	88%	92%	.....	0,42	0,46
	06	0%	0%	0%	.....	.....	.....
	07	0%	100%	100%	.....	0,40	0,41
	08	0%	0%	0%	.....	.....	.....
	09	0%	0%	0%	.....	.....	.....
	10	0%	100%	100%	.....	0,88	0,89
	11	0%	64%	100%	.....	1,40	1,42
	12	0%	0%	0%	.....	.....	.....
	13	0%	52%	70%	.....	5,95	5,87
	14	0%	0%	10%	.....	.....	22,11
ZENOTRAVEL	01	100%	100%	100%	0,60	0,43	0,43
	02	0%	86%	86%	.....	0,09	0,09
	03	0%	100%	100%	.....	0,17	0,17
	04	0%	68%	98%	.....	0,28	0,28
	05	0%	94%	100%	.....	0,39	0,39
	06	0%	100%	100%	.....	0,52	0,51
	07	0%	82%	100%	.....	0,67	0,55
	08	0%	48%	98%	.....	1,35	1,34
	09	0%	70%	100%	.....	2,17	1,72
	10	0%	92%	100%	.....	3,38	2,25
	11	0%	82%	100%	.....	2,87	2,76
	12	0%	96%	100%	.....	4,43	3,51
	13	0%	94%	100%	.....	6,47	5,58
PIPESWORLD	01	100%	100%	100%	0,56	0,44	0,45
	02	0%	0%	0%	.....	.....	.....
	03	0%	10%	78%	.....	14,46	14,66
	04	0%	0%	14%	.....	.....	20,52
	05	30%	20%	92%	3,87	4,16	13,77
	06	0%	38%	82%	.....	10,77	10,68

No domínio DEPOTS, os 3 algoritmos tiveram resultados semelhantes no primeiro problema, exceto pelo fato de que 10% das execuções do HC não chegaram a solução, enquanto o EHC e o HB-EHC encontraram a solução em todas as execuções. No restante dos problemas, o HC não conseguiu encontrar a solução em nenhuma das execuções, já os algoritmos EHC e HB-EHC mantiveram uma taxa de convergência parecida, com exceção

do problema 07 onde o algoritmo HB-EHC teve 30% a mais de execuções bem sucedidas. A cada problema, o EHC e o HB-EHC se alternavam em relação ao menor tempo médio.

Durante a execução dos problemas do domínio DRIVERLOG, nenhuma execução do algoritmo HC encontrou a solução. Apesar do tempo médio do algoritmo EHC ter sido ligeiramente menor na maioria dos problemas, a taxa de completude do HB-EHC foi maior, o que pode ser constatado nos problemas 05, 11, 13 e 14 onde se teve respectivamente 4%, 36%, 18%, e 10% a mais de execuções de sucesso, sendo que no problema 14 nenhuma das execuções do EHC conseguiu alcançar a meta.

As execuções do HB-EHC no domínio ZENOTRAVEL foram mais rápidas ou no máximo iguais as do algoritmo EHC. No primeiro problema, todas as execuções dos 3 algoritmos obtiveram sucesso, mas o HC não conseguiu chegar a solução em mais nenhum problema deste domínio. Além disso, o HB-EHC obteve 100% de aproveitamento das execuções em 10 dos 13 problemas, enquanto o EHC apenas em 3.

Já no domínio PIPESWORLD, o algoritmo HC voltou a resolver alguns problemas. No problema 01, as execuções de todos os algoritmos chegaram à solução, e com um tempo médio bem semelhante. No segundo problema, nenhum algoritmo obteve sucesso. Já no terceiro, o HC não chegou à solução em nenhuma execução, ao contrário dos algoritmos EHC e HB-EHC que tiveram tempos médios parecidos, sendo que o HB-EHC teve 68% a mais de execuções bem sucedidas. No quarto problema, apenas o algoritmo HB-EHC encontrou a solução, com uma taxa de convergência de 14%. Para o problema 05, todos conseguiram encontrar a solução, sendo que no HB-EHC quase todas as execuções obtiveram sucesso. Para o último problema, a taxa de completude do HB-EHC foi 82% contra 38% do EHC, sendo que o HC mais uma vez não obteve sucesso em nenhuma execução.

Através dos testes com o HB-EHC, foi possível constatar um aumento significativo na taxa de convergência do algoritmo por meio de suas otimizações, forçando ele a ser um pouco mais completo, mas sem comprometer seu tempo de execução. Isso se dá, principalmente, pelo fato de que ele escapa mais rápido e com mais frequência de máximos/mínimos locais. Portanto, ao utilizarmos o HB-EHC como primeiro algoritmo na estratégia de busca do planejador, evitaremos o uso excessivo de algoritmos completos, que são caracterizados por demandarem mais espaço de memória e terem um tempo de resposta maior. A taxa de completude dos algoritmos são ilustradas na Figura 4.1.

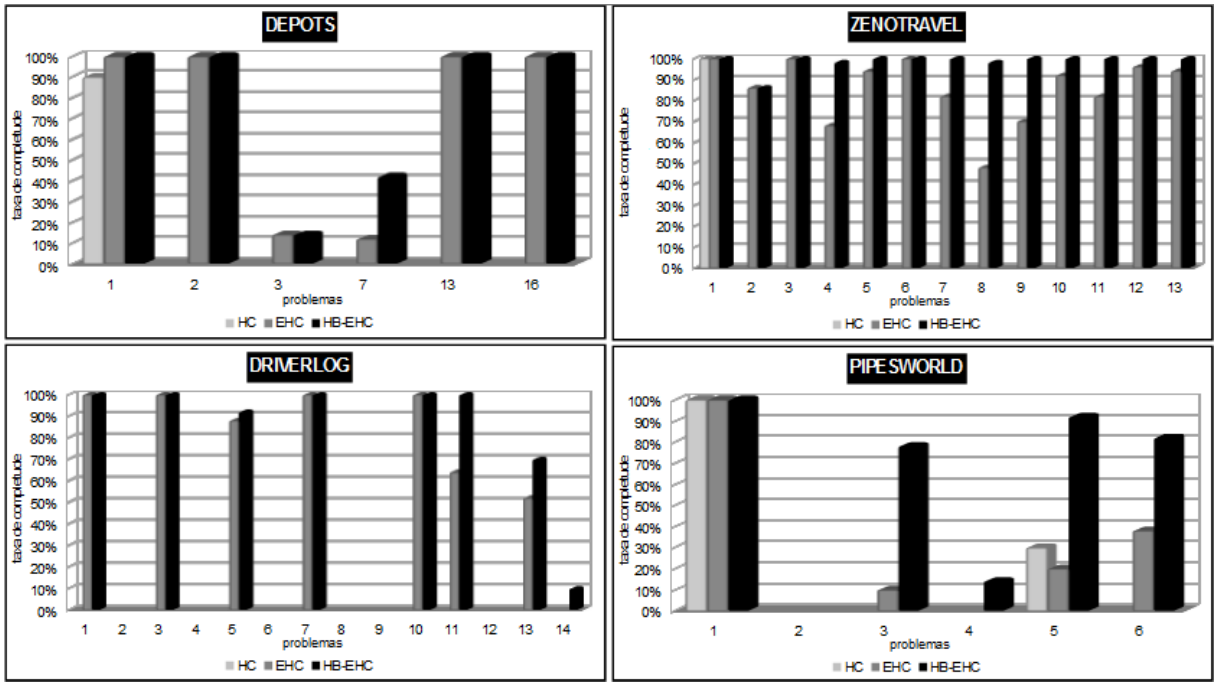

Figura 4.1: Taxa de completude: HC, EHC e HB-EHC.

O algoritmo Adaptive-LRTA* foi comparado a sua versão original, LRTA*, e uma versão otimizada conhecida como LRTA*-K que faz atualizações nas heurísticas de K elementos por iteração, e não só no estado corrente como é feito em outras versões do LRTA*. Nestes testes foram usados os domínios: DEPOTS, DRIVERLOG, PIPESWORLD e TPP. As melhorias feitas no Adaptive-LRTA* tinham como foco acelerar sua convergência e diminuir a taxa de falha do algoritmo por falta de espaço de memória, um problema enfrentado por muitos algoritmos completos. As taxas de falhas por falta de espaço de memória e o tempo médio de execução podem ser vistos na Tabela 4.3.

No domínio DEPOTS o algoritmo Adaptive-LRTA* teve um tempo médio melhor na maioria dos problemas. Ele obteve o pior tempo médio, dentre os 3 algoritmos, somente no problema 08. Isto se deu devido à complexidade do problema. Em algumas execuções neste problema todos os algoritmos obtiveram sucesso, mas em algumas delas a busca foi para caminhos ruins o que fez com que os algoritmos LRTA* e LRTA*-K viessem a falhar por não terem mais espaço livre para armazenar estados, fazendo com que sua execução não chegasse à solução, onde seu tempo não foi contabilizado. Como o Adaptive-LRTA* faz um balanceamento de memória, suas execuções não foram interrompidas por falta de espaço para armazenar estados, fazendo-o encontrar a solução mesmo quando entrava em caminhos ruins durante a busca, o que fez seu tempo médio subir um pouco.

Os tempos médios do Adaptive-LRTA* foram os menores em todos os problemas do domínio DRIVERLOG, exceto no problema 08 onde o LRTA* foi apenas 0,16 segundos mais rápido. Já o LRTA*-K teve novamente problemas por falta de espaço, onde 10% de

Tabela 4.3: Taxa de falha (falta de espaço) e tempo médio de execução: LRTA*, LRTA*-K e Adaptive-LRTA*.

Domínio	Problema	Falha			Tempo		
		LRTA*	LRTA*-K	Adaptive-LRTA*	LRTA*	LRTA*-K	Adaptive-LRTA*
DEPOTS	01	0%	0%	0%	0,50	0,62	0,40
	02	0%	0%	0%	1,27	1,41	0,97
	03	0%	0%	0%	28,94	84,07	22,81
	07	0%	0%	0%	92,94	179,27	102,03
	08	65%	40%	0%	481,54	602,71	705,03
	13	0%	0%	0%	167,66	151,90	95,34
DRIVERLOG	01	0%	0%	0%	0,24	0,38	0,20
	02	0%	0%	0%	0,69	1,07	0,63
	03	0%	0%	0%	0,48	0,65	0,41
	04	0%	0%	0%	0,81	1,04	0,97
	05	0%	0%	0%	1,07	1,32	0,91
	06	0%	0%	0%	0,95	1,59	0,84
	07	0%	0%	0%	1,26	1,35	1,03
	08	0%	0%	0%	2,43	2,93	2,27
	09	0%	0%	0%	4,73	4,78	4,16
	10	0%	0%	0%	2,39	2,65	2,04
	11	0%	0%	0%	57,35	13,22	13,18
	12	0%	0%	0%	201,86	223,18	101,72
	13	0%	0%	0%	40,29	59,52	27,07
	14	0%	10%	0%	118,84	278,25	158,94
PIPESWORLD	01	0%	0%	0%	0,65	0,77	0,51
	02	0%	0%	0%	34,58	38,00	37,67
	03	0%	0%	0%	65,96	54,00	35,90
	04	0%	0%	0%	42,33	43,14	41,82
	05	0%	0%	0%	49,23	51,85	31,44
	06	0%	0%	0%	44,01	56,05	48,16
TPP	01	0%	0%	0%	0,03	0,04	0,04
	02	0%	0%	0%	0,05	0,08	0,05
	03	0%	0%	0%	0,10	0,16	0,07
	04	0%	0%	0%	0,16	0,28	0,10
	05	0%	0%	0%	0,39	0,64	0,28
	06	0%	0%	0%	1,28	1,98	1,16
	07	0%	0%	0%	1,86	3,67	1,53
	08	0%	0%	0%	2,39	4,77	2,04
	09	0%	0%	0%	9,03	16,90	5,62
	10	0%	0%	0%	11,73	21,23	7,35
	11	0%	0%	0%	22,61	42,46	17,89

suas execuções falharam no problema 14.

O algoritmo Adaptive-LRTA* teve o melhor tempo médio em 4 dos 6 problemas contabilizados do domínio PIPESWORLD, considerado o domínio mais difícil do *benchmark*. Como exemplo disso, nenhum dos algoritmos conseguiu finalizar suas execuções para os outros 44 problemas deste domínio em menos de 30 minutos.

Em relação ao domínio TPP, com exceção do primeiro problema onde o LRTA* foi 0,01 segundo mais rápido, o Adaptive-LRTA* mais uma vez obteve o melhor resultado em todos os problemas.

Os resultados obtidos demonstram a eficiência do método Adaptive-LRTA* em relação as outras versões do LRTA*. Graças as escolhas gulosas que o algoritmo faz durante as primeiras iterações, a velocidade de convergência do método aumentou muito como se pode ver na Figura 4.2. Outro ganho que se teve, foi a redução da taxa de falhas por falta de espaço, devido ao balanceamento do *heap* utilizado para armazenar os estados processados. O Adaptive-LRTA* se mostrou um método de busca completo muito eficaz, principalmente para resolver problemas mais complexos, onde se tem um espaço de estados grande. Por esta razão, ele é usado pelo planejador SLPlan caso o algoritmo HB-EHC falhe.

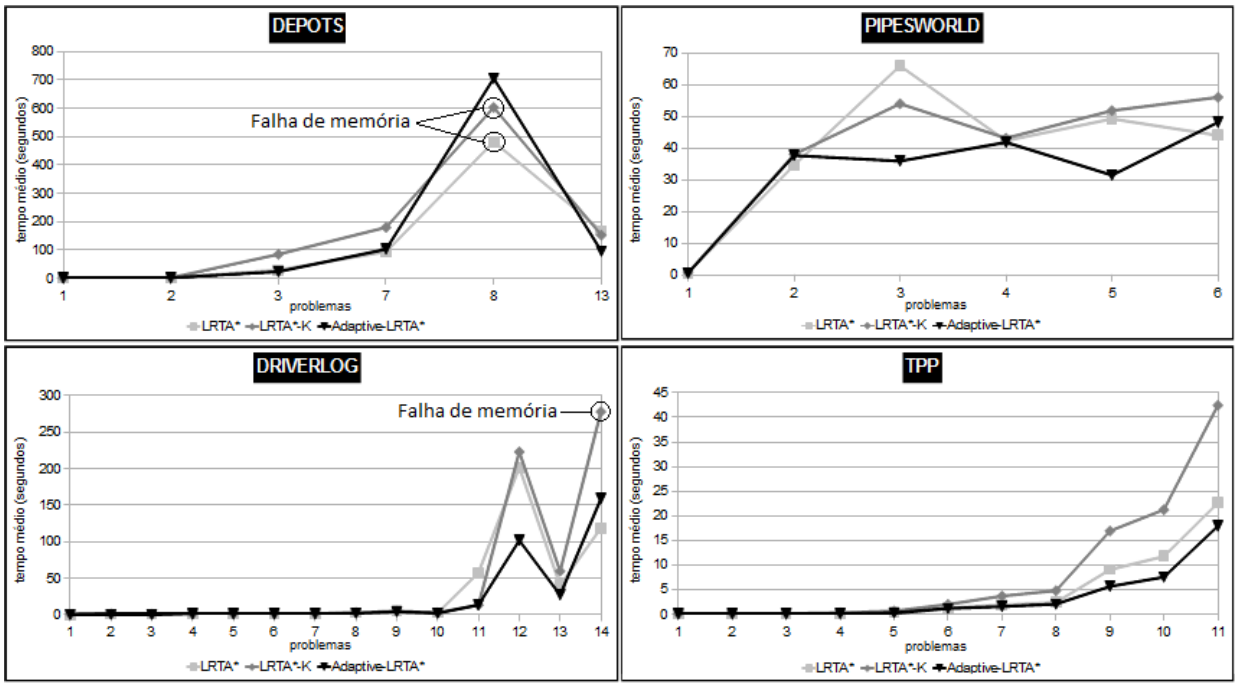

Figura 4.2: Tempo médio: LRTA*,LRTA*-K e Adaptive-LRTA*.

4.3.2 Resultados gerais: Planejador SLPlan

A partir dos resultados obtidos através dos experimentos com HB-EHC e Adaptive-LRTA*, observou-se que uma estratégia de busca composta por esses algoritmos seria um método eficiente para resolução de problemas de planejamento. Para isso, foi criada uma estratégia onde o HB-EHC seria executado primeiro, e caso ele falhasse, o Adaptive-LRTA* seria acionado aproveitando os estados já processados.

A vantagem de se usar um algoritmo não-completo primeiro é que sua execução é rápida, ou seja, para problemas simples ele acha a solução mais depressa que uma busca completa, e caso o problema seja complexo, sua execução não se estenderá por muito tempo, onde o planejador irá requisitar uma busca completa para encontrar a solução. Esta estratégia foi implementada no planejador SLPlan, e é semelhante a do FF.

Como já foi dito, o planejador FF utiliza o EHC como primeiro método, e caso ele falhe, uma nova busca com o BFS começa do zero. Para fins comparativos, foram executados testes em todos os problemas do *benchmark*, utilizando esses dois planejadores. O objetivo era verificar se o planejador SLPlan conseguiria resolver os problemas de forma mais eficiente, reduzindo o tempo de execução e o número de falhas por falta de espaço, em relação ao FF. Tais dados são apresentados na Tabela 4.4.

Nos problemas do domínio DEPOTS, cujas execuções dos planejadores não excederam o tempo estipulado para os testes, o SLPlan teve um tempo médio melhor na maioria dos problemas. Um fato interessante é que o FF teve 8% de execuções falhas no problema

Tabela 4.4: Taxa de falha e tempo médio de execução: FF e SLPlan.

Domínio	Problema	Falha		Tempo	
		FF	SLPlan	FF	SLPlan
DEPOTS	01	0%	0%	0,20	0,20
	02	0%	0%	0,50	0,60
	03	0%	0%	65,36	19,61
	07	8%	0%	202,68	69,16
	08	86%	0%	18,41	733,06
	13	0%	0%	29,63	27,89
	16	0%	0%	40,45	45,73
DRIVERLOG	01	0%	0%	0,07	0,07
	02	0%	0%	0,82	1,02
	03	0%	0%	0,17	0,18
	04	0%	0%	0,95	1,38
	05	0%	0%	0,53	0,56
	06	0%	0%	1,03	2,04
	07	0%	0%	0,40	0,43
	08	0%	0%	2,98	2,11
	09	0%	0%	4,84	4,48
	10	0%	0%	0,88	0,89
	11	0%	0%	5,77	1,43
	12	0%	0%	158,20	92,83
	13	5%	0%	52,33	47,29
	14	16%	0%	204,02	166,99
ROVERS	01	0%	0%	0,11	0,11
	02	0%	0%	0,09	0,09
	03	0%	0%	0,16	0,16
	04	0%	0%	0,11	0,11
	05	0%	0%	0,05	0,05
	06	0%	0%	1,02	1,02
	07	0%	0%	0,43	0,43
	08	0%	0%	0,99	0,98
	09	0%	0%	1,68	1,55
	10	0%	0%	2,09	1,91
	11	0%	0%	2,20	2,14
	12	0%	0%	1,35	1,33
	13	0%	0%	9,00	6,11
	14	0%	0%	2,43	2,42
	15	0%	0%	15,08	6,81
	16	0%	0%	8,92	5,99
	17	0%	0%	25,14	19,79
	18	0%	0%	61,72	41,77
ZENOTRAVEL	01	0%	0%	0,43	0,06
	02	0%	0%	0,10	0,09
	03	0%	0%	0,17	0,18
	04	0%	0%	0,51	0,26
	05	0%	0%	0,43	0,39
	06	0%	0%	0,52	0,51
	07	0%	0%	0,92	0,55
	08	0%	0%	9,86	1,33
	09	0%	0%	7,76	1,66
	10	0%	0%	4,76	2,25
	11	0%	0%	16,09	2,76
	12	0%	0%	7,97	3,51
	13	0%	0%	15,67	5,58
PIPESWORLD	01	0%	0%	0,44	0,35
	02	100%	0%	.....	38,16
	03	0%	0%	136,44	31,83
	04	4%	0%	83,52	90,17
	05	0%	0%	38,35	10,37
	06	4%	0%	49,46	37,52
TPP	01	0%	0%	0,02	0,03
	02	0%	0%	0,04	0,05
	03	0%	0%	0,07	0,07
	04	0%	0%	0,09	0,10
	05	0%	0%	0,21	0,21
	06	0%	0%	0,70	0,74
	07	0%	0%	2,35	1,11
	08	0%	0%	4,50	1,40
	09	0%	0%	28,78	4,90
	10	0%	0%	181,66	11,42
	11	20%	0%	1163,36	24,20

07 e 86% no problema 8, sendo que o SLPlan conseguiu encontrar a solução em todos os problemas.

Para os problemas considerados mais simples no domínio DRIVERLOG, o FF teve uma ligeira vantagem em relação ao tempo médio do SLPlan. Mas para problemas mais complexos, o SLPlan obteve um tempo melhor e sem falhas em suas execuções, diferentemente do FF que teve respectivamente 5% e 16% de falha nos problemas 13 e 14.

O domínio ROVERS possui, na sua grande maioria, problemas relativamente simples. Nele, os primeiros 18 problemas, de um total de 20, puderam ser resolvidos em menos de

30 minutos pelos planejadores. O SLPlan obteve um tempo médio menor ou no máximo igual ao FF.

Para o domínio ZENOTRAVEL, o planejador SLPlan teve o menor tempo médio, exceto no problema 03 onde o FF foi apenas 0,01 segundo mais rápido.

A complexidade do domínio PIPESWORLD fez com que apenas os 6 primeiros problemas pudessem ser resolvidos por ambos os planejadores em menos de 30 minutos. O SLPlan foi mais eficiente na resolução dos problemas, com exceção do 4º. Além disso, houve novamente falhas nas execuções do FF. A pior delas ocorreu no problema 02, onde 100% de suas execuções falharam por falta de espaço de memória.

No domínio TPP, até o problema 07, os tempos de execução foram muito baixos. Mas considerando todos os problemas resolvidos neste domínio, o tempo de execução do SLPlan pareceu ser linear em relação a complexidade dos problemas, já o FF começou a aumentar seu tempo quase que exponencialmente.

Os resultados mostram que o planejador SLPlan conseguiu diminuir o tempo médio de planejamento, em relação ao FF, além de evitar que suas execuções parassem por falta de espaço de memória. Isto se deve pela estratégia de busca montada e pelas melhorias feitas nos algoritmos HB-EHC e Adaptive-LRTA*, que aceleram a convergência para a solução realizando o balanceamento de estados em memória.

Observou-se também que o FF, as vezes, perdia muito tempo tentando resolver um problema mais difícil utilizando o EHC, e só depois de um certo tempo a chamada do BFS ocorria, o que fazia o tempo total de planejamento aumentar. Já o HB-EHC através de seus critérios de parada, como o excesso de estados dentro do *heap* Q_{min} (que caracteriza um longo tempo de espera para sair de máximos/mínimos locais), antecipa a chamada do Adaptive-LRTA* que rapidamente efetuará uma busca completa para o problema.

O tempo médio das execuções pode ser um bom indicador de eficiência para sistemas de planejamento, pois um baixo tempo de resposta, as vezes, é uma das características mais desejadas em um sistema. A Figura 4.3 ilustra o tempo médio de planejamento dos sistemas FF e SLPlan, além de indicar as falhas por falta de memória.

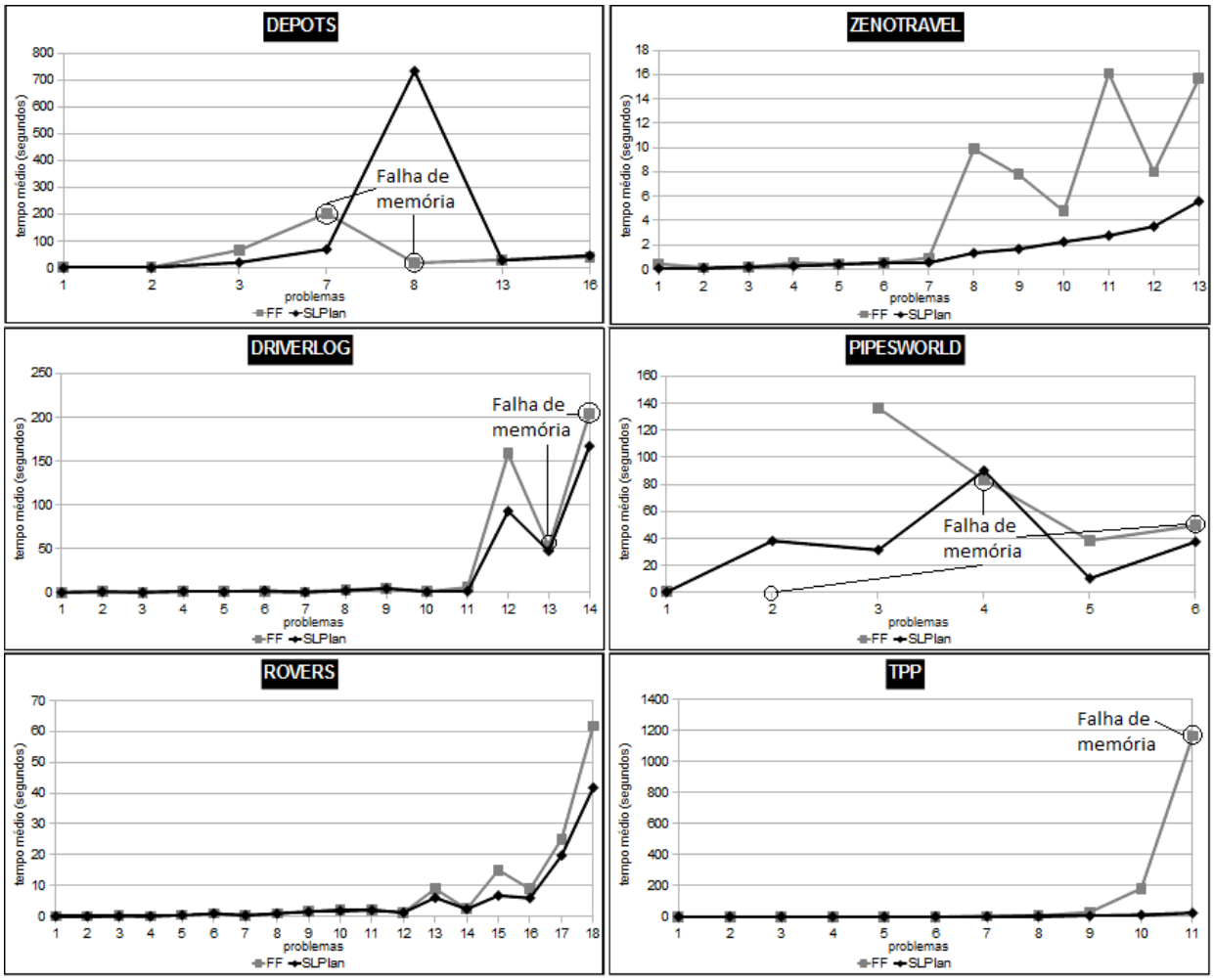

Figura 4.3: Tempo médio: FF e SLPlan.

Capítulo 5

Aplicando planejamento em processos de negócio

Gerenciamento de Processos de Negócio, mais conhecido do inglês *Business Process Management* (BPM), tem recebido recentemente uma atenção considerável pelas áreas de Administração e Ciência da Computação [Weske 2007]. Profissionais da Administração estão interessados em melhorar a satisfação dos clientes, reduzir custos em negociações e estabelecer novos serviços e produtos de baixo custo no mercado. Já as comunidades ligadas à Ciência da Computação se preocupam em desenvolver sistemas mais robustos e escalonáveis para principalmente mapear e modelar processos de negócio.

Uma das possíveis aplicações para sistemas de planejamento, diz respeito a automatização da modelagem de processos de negócio ou, mais conhecido como, *workflow* [van der Aalst e van Hee 2002]. Como processos de negócio, ou *workflow*, envolvem raciocínio com atividades/ações a fim de gerar uma sequência lógica de execução para se alcançar uma determinada meta/objetivo, o uso de planejamento parece adequado. Com isso, o planejador SLPlan foi integrado a um sistema de geração de *workflow* para que ele pudesse construir automaticamente fluxos a partir das possíveis atividades de um processo de negócio.

A Seção 5.1 explica alguns conceitos de BPM e *workflow*. Em seguida, a Seção 5.2 apresenta o sistema de geração de *workflow* Metaplan. Por fim, a Seção 5.3 descreve a integração entre o planejador SLPlan e o sistema Metaplan, além de apresentar experimentos realizados na geração de *workflow* com o uso do planejador desenvolvido neste trabalho.

5.1 Conceitos

Segundo Hammer e Champy, "um processo é um grupo de atividades realizadas numa sequência lógica com o objetivo de produzir um bem ou um serviço que tem valor para

um grupo específico de clientes" [Michael Hammer 1994].

A utilização do Gerenciamento de Processos de Negócio ao longo dos últimos anos vem crescendo de forma bastante significativa, dada a sua utilidade e rapidez com que melhora os processos nas organizações onde já foi implementado. No contexto de BPM, existem várias ferramentas denominadas sistemas de gestão de processos de negócio (sistemas BPM) para auxiliar nas diversas fases do gerenciamento de processos. Os sistemas de BPM buscam oferecer aos usuários maior facilidade e flexibilidade no uso, o que torna a experiência mais agradável, com ferramentas simples e intuitivas. Atualmente, existem ferramentas para monitorar, modelar, simular e executar processos [Van Der Aalst et al. 2003]. Através delas, os gestores podem analisar e alterar processos baseados em dados reais o que auxilia, por exemplo, na remoção de gargalos em certas atividades de um processo.

O BPM pode ser considerado como uma extensão do gerenciamento clássico de *workflow* (*Workflow Management* - WfM). Hoje, o termo *workflow* é mais empregado durante a fase de modelagem de processo, e é usado para referenciar a parte de dependências entre tarefas que devem ser respeitadas durante a execução de um processo de negócio [Reijers 2003]. A diferença entre BPM e *workflow* pode ser descrita da seguinte forma: BPM é uma disciplina de gerenciamento orientada a processo, e não é considerada uma tecnologia; já *workflow* é uma tecnologia de gerenciamento de fluxo encontrada em ferramentas de gerenciamento de processos de negócio [Ko et al. 2009].

Para se realizar a modelagem de processos, pode-se utilizar editores específicos tais como Visio, Bizagi, YAML, *Together Workflow Editor*, etc.. Os editores gráficos geralmente seguem um padrão de símbolos definidos pela Business *Process Model and Notation* (BPMN). Os principais símbolos definidos pela notação BPMN são mostrados na Figura 5.1 e descritos abaixo.

- Processo: define o processo que está sendo modelado. Este símbolo também é conhecido como *Pool* (piscina), pois quando se divide o processo dentre as partes envolvidas nas chamadas *Lanes* (raias), seu desenho se assemelha a uma piscina.
- Envolvidos: define partes distintas envolvidas no processo, onde cada uma delas possui uma região delimitada para agrupar suas atividades.
- Início: marca o início do processo.
- Fim: marca o fim do processo.
- Condição: determina um ponto de decisão no fluxo.
- Tarefa: define uma atividade desempenhada por um dos envolvidos.
- Fluxo: mostra o sentido do fluxo de atividades do processo.

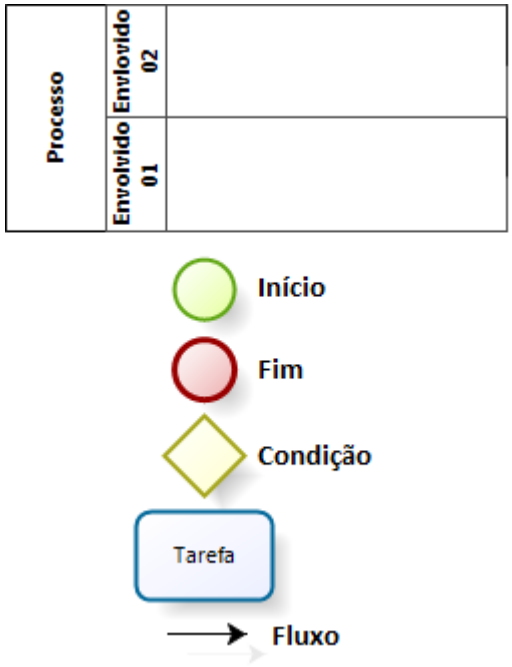

Figura 5.1: Principais símbolos da notação BPMN.

A Figura 5.2 ilustra um exemplo genérico de modelagem de processo.

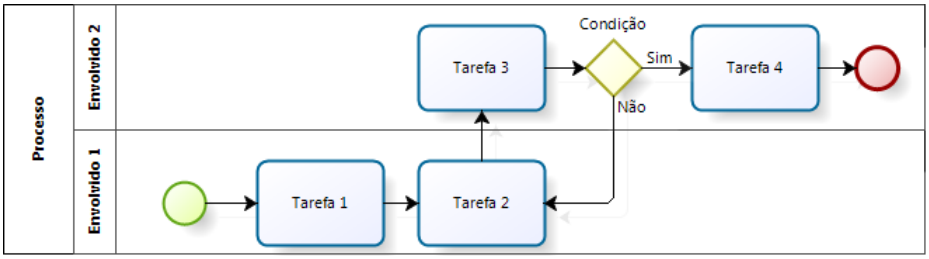

Figura 5.2: Exemplo de modelagem de *workflow*.

Geralmente, existem inúmeras atividades desenvolvidas por diversos setores dentro das organizações, e cabe a pessoa que está modelando o processo definir quais atividades irão o compor e escolher a melhor sequência em que essas atividades deverão ser executadas, respeitando as pré-condições e efeitos de cada uma delas. Apesar da agilidade proporcionada pelas ferramentas de edição de *workflow*, a fase de modelagem pode ser demorada e passível de erros, pois requer o entendimento de todas as atividades do negócio, e se pode não chegar numa boa formatação do processo.

Pelas semelhanças entre planejamento e modelagem de *workflow*, pode-se buscar caminhos para aplicar técnicas desenvolvidas em sistemas de planejamento para gerar fluxos eficientes de atividades, automaticamente, para processos de negócio.

No campo de *workflow*, a meta é a solução de um caso, que pode ser resolvido instanciando-se um processo ou roteiro. O processo deve ser completo, ou seja, deve contemplar todas as situações de exceção previstas, de forma a sempre conduzir ao seu final. No campo de planejamento, a meta é a solução do problema original. A solução

do problema significa satisfazer a meta, que pode ser uma disjunção de submetas, e que é atingida pela execução das atividades do roteiro, em outras palavras, é o plano gerado.

5.2 Metaplan

Alguns sistemas de modelagem de *workflow*, apoiadas em planejamento, para geração automática de processos têm sido propostos [Myers e Berry 1999] [Berry e Drabble 1999] [Shi et al. 2010] [Wainer e de Lima Bezerra 2003] [Aler et al. 2002]. Um sistema de particular interesse é o Metaplan [Melo 2005], resultado de uma pesquisa desenvolvida pela Faculdade de Computação da Universidade Federal de Uberlândia.

O objetivo do Metaplan é integrar ferramentas de modelagem e execução de *workflow* com algoritmos de planejamento. Como plataforma, para a modelagem de processos, foi escolhida a ferramenta Enhydra JaWE, uma ferramenta gráfica de código aberto para modelagem de processos de *workflow*. Ela segue as especificações da Workflow Management Coalition (WfMC) e suporta a linguagem XPDL como formato nativo [Melo 2005].

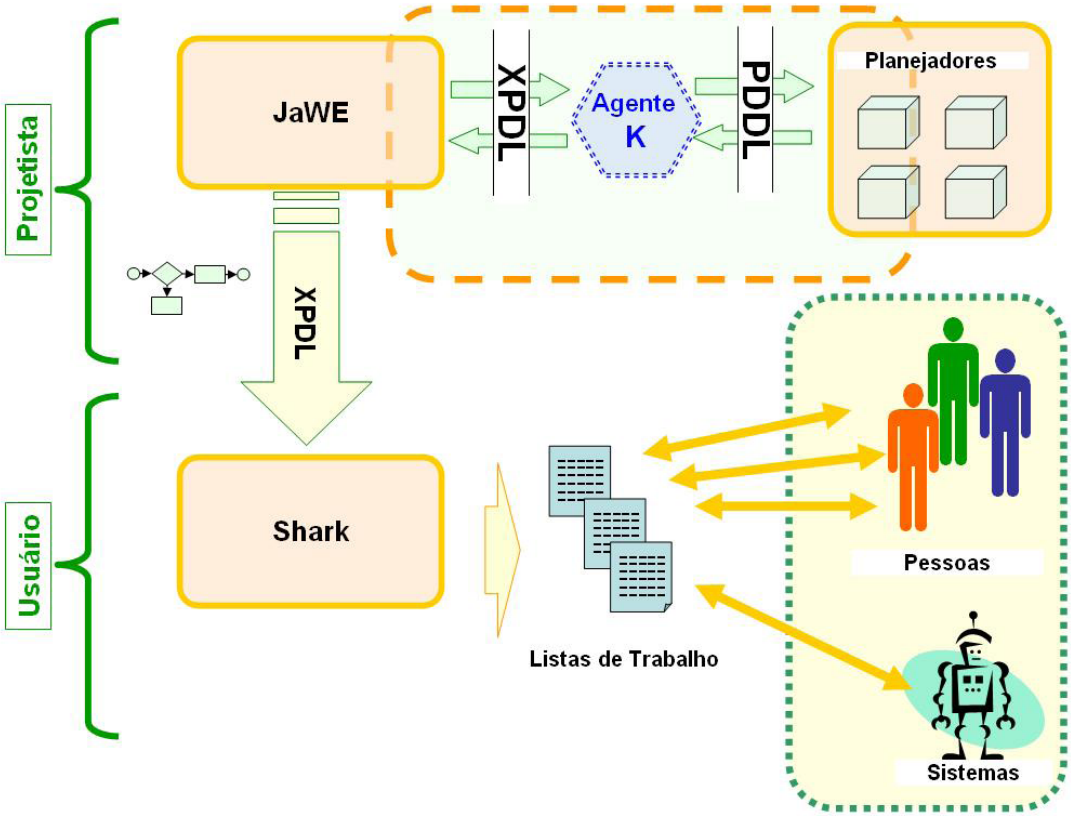

Figura 5.3: Arquitetura do Sistema Metaplan.

O Metaplan proposto nesta abordagem, recebe como entrada um conjunto de dados no formato da linguagem XPDL que descreve, entre outras informações, as atividades do processo. Depois, manipula esta informação, de forma a gerar uma equivalência em

PDDL. Logo após, chama um planejador para a produção de um plano consistente, que leve do estado inicial à meta. A meta, neste caso, é a conclusão do processo de *workflow*.

A seguir descrevem-se as principais funcionalidades da ferramenta Metaplan [Melo 2005]:

- Modelagem de Processos (JaWE): A ferramenta JaWE é modificada para interagir com o agente de planejamento K, que interpreta a definição de processos dada na linguagem XPDL e retorna o processo estruturado na mesma linguagem.
- Máquina de Workflow (*Shark*): A ferramenta *Shark* é utilizada para executar os processos de *workflow* gerados na ferramenta JaWE. A máquina de *workflow* interpreta e instancia os processos, gerando as listas de trabalho correspondentes para cada responsável, em cada passo do processo. Inclui também os subsistemas de Administração e Monitoramento dos processos. Através desta ferramenta, também é possível atribuir agentes autômatos para execução de determinadas atividades. A maior parte destas funcionalidades já é nativa da ferramenta, a qual atende completamente as especificações do WfMC.
- Agente Planejador (Agente K): O agente de planejamento tem a função de converter as especificações da linguagem de modelagem de processos para a linguagem de definição de domínios para planejamento (PDDL), submeter uma requisição de geração de plano para a ferramenta de planejamento e, por fim, converter o resultado (plano gerado) novamente no formato XPDL, devolvendo o controle para a ferramenta de modelagem.

Dessa forma, definidos os requerimentos do *workflow*, em termos de atividades e objetivos a serem alcançados, pode-se automaticamente gerar o processo de *workflow* [Melo 2005].

5.3 Integração entre SLPlan e Metaplan

A integração entre o Metaplan e sistemas de planejamento capazes de interpretar arquivos PDDL dá-se através do Agente *K*, como é ilustrado na Figura 5.3. Sendo assim, o Metaplan é totalmente desacoplado de um planejador específico. Para enfatizar o uso de planejamento na resolução de problemas práticos do cotidiano, integramos o planejador SLPlan ao sistema Metaplan com o intuito de gerar a modelagem de processos de negócio automaticamente. O Algoritmo 5 detalha a integração entre estes sistemas.

Algoritmo 5 Metaplan(*Processo*)

1: $atividades \leftarrow decompoeAtividades(Processo)$
2: $atividades_{pddl} \leftarrow geraPddl(atividades)$
3: $plano \leftarrow SLPlan(atividades_{pddl})$
4: $workflow \leftarrow extraiModelo(plano)$

No planejamento clássico, as ações em um plano conectam-se entre si de forma sequencial, bem como as atividades em um processo de *workflow*. Porém, no *workflow* podem ocorrer tomadas de decisão ou paralelismo.

Quando as atividades envolvem ramificações, elas podem ser de dois tipos: separação ou junção (*Split*/*Join*). As ramificações de separação estão na saída da atividade corrente e podem direcionar para uma execução paralela ou condicional (*AND*/*OR*). Na execução paralela, duas ou mais atividades podem ser habilitadas ao mesmo tempo pelo motor de *workflow*. A saída da execução condicional obriga a escolha de um único caminho; já a sua entrada, se for do tipo *OR* necessita apenas ser atingida por um dos ramos, e se for do tipo *AND* precisa ser atingida por todos os ramos para que a atividade seja habilitada.

A função *decompoeAtividades* (linha 1) é responsável por converter as atividades do processo em ações. Já a função *geraPddl* (linha 2), gera um domínio e um problema de planejamento baseado nas atividades do processo. A integração ocorre na linha 3, onde o planejador SLPlan é invocado sobre o problema gerado. Por fim, na linha 4, o processo é modelado a partir do plano criado pelo SLPlan.

Para testar a efetividade da modelagem automática de processos de negócio, utilizando o planejador desenvolvido neste trabalho, foram criadas duas instâncias do Metaplan. Na primeira, o Agente *K* se liga ao planejador FF (Metaplan-FF). Já na segunda, o Agente *K* se liga ao planejador SLPlan (Metaplan-SLPlan).

Os sistemas Metaplan-FF e Metaplan-SLPlan foram aplicados a problemas de *workflow*, gerados aleatoriamente, com quantidade de atividades de 48 a 3968, variando de 80 em 80. Portanto, foram criados 50 processos de *workflow*. O Exemplo 5.1 apresenta um dos processos utilizados nos experimentos. No Exemplo 5.2 é detalhado o arquivo PDDL de domínio do problema correspondente a esse processo. E por fim, no Exemplo 5.3 é mostrado o plano resultante gerado pelo planejador, e que servirá para construir a modelagem do processo.

Exemplo 5.1. Processo gerado para os testes comparativos entre Metaplan-FF e Metaplan-SLPlan com 48 atividades. A notação utilizada representa a atividade em termos de seu executor, sua pré-condição e seu efeito. Por conveniência, como o executor é o mesmo para todo o processo, ele foi omitido para todas as atividades, exceto a primeira:

```
<PROCESSO nome="processo1">
  <ATIVIDADE nome="A0">
    <EXECUTOR>agente< /EXECUTOR>
    <PRECONDICAO>< /PRECONDICAO>
    <EFEITO>PA0< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="A1">
    <PRECONDICAO>PA0< /PRECONDICAO>
    <EFEITO>PA1< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="B">
    <PRECONDICAO>PA1< /PRECONDICAO>
    <EFEITO>PB1 ou PB2< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K2a">
    <PRECONDICAO>PB1< /PRECONDICAO>
```

```
    <EFEITO>PK2a1 or PK2a2 or PK2a3< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K2a1">
    <PRECONDICAO>PK2a1< /PRECONDICAO>
    <EFEITO>PK21a< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K2a2">
    <PRECONDICAO>PK2a2< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K2a3">
    <PRECONDICAO>PK2a3< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K3a">
    <PRECONDICAO>PK21a< /PRECONDICAO>
    <EFEITO>PK3a1 or PK3a2 or PK3a3< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K3a1">
    <PRECONDICAO>PK3a1< /PRECONDICAO>
    <EFEITO>PK31a< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K3a2">
    <PRECONDICAO>PK3a2< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K3a3">
    <PRECONDICAO>PK3a3< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K4a">
    <PRECONDICAO>PK31a< /PRECONDICAO>
    <EFEITO>PK4a1 or PK4a2 or PK4a3< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K4a1">
    <PRECONDICAO>PK4a1< /PRECONDICAO>
    <EFEITO>PK41a< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K4a2">
    <PRECONDICAO>PK4a2< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K4a3">
    <PRECONDICAO>PK4a3< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K5a">
    <PRECONDICAO>PK41a< /PRECONDICAO>
    <EFEITO>PK5a1 or PK5a2 or PK5a3< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K5a1">
    <PRECONDICAO>PK5a1< /PRECONDICAO>
    <EFEITO>PK51a< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K5a2">
    <PRECONDICAO>PK5a2< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K5a3">
```

```
    <PRECONDICAO>PK5a3< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K6a">
    <PRECONDICAO>PK51a< /PRECONDICAO>
    <EFEITO>PK6a1 or PK6a2 or PK6a3< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K6a1">
    <PRECONDICAO>PK6a1< /PRECONDICAO>
    <EFEITO>PK61a< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K6a2">
    <PRECONDICAO>PK6a2< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K6a3">
    <PRECONDICAO>PK6a3< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K7a">
    <PRECONDICAO>PK61a< /PRECONDICAO>
    <EFEITO>PK7a1 or PK7a2 or PK7a3< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K7a1">
    <PRECONDICAO>PK7a1< /PRECONDICAO>
    <EFEITO>PK71a< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K7a2">
    <PRECONDICAO>PK7a2< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K7a3">
    <PRECONDICAO>PK7a3< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K8a">
    <PRECONDICAO>PK71a< /PRECONDICAO>
    <EFEITO>PK8a1 or PK8a2 or PK8a3< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K8a1">
    <PRECONDICAO>PK8a1< /PRECONDICAO>
    <EFEITO>PK81a< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K8a2">
    <PRECONDICAO>PK8a2< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K8a3">
    <PRECONDICAO>PK8a3< /PRECONDICAO>
    <EFEITO>PB2< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K9a">
    <PRECONDICAO>PK81a< /PRECONDICAO>
    <EFEITO>PK9a1 or PK9a2 or PK9a3< /EFEITO>
  < /ATIVIDADE>
  <ATIVIDADE nome="K9a1">
    <PRECONDICAO>PK9a1< /PRECONDICAO>
    <EFEITO>PK91a< /EFEITO>
  < /ATIVIDADE>
```

```
<ATIVIDADE nome="K9a2">
   <PRECONDICAO>PK9a2< /PRECONDICAO>
   <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K9a3">
   <PRECONDICAO>PK9a3< /PRECONDICAO>
   <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K10a">
   <PRECONDICAO>PK91a< /PRECONDICAO>
   <EFEITO>PK10a1 or PK10a2 or PK10a3< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K10a1">
   <PRECONDICAO>PK10a1< /PRECONDICAO>
   <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K10a2">
   <PRECONDICAO>PK10a2< /PRECONDICAO>
   <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="K10a3">
   <PRECONDICAO>PK10a3< /PRECONDICAO>
   <EFEITO>PB2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="D0">
   <PRECONDICAO>PB2< /PRECONDICAO>
   <EFEITO>PD0< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="D1">
   <PRECONDICAO>PD0< /PRECONDICAO>
   <EFEITO>PD1< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="D2">
   <PRECONDICAO>PD1< /PRECONDICAO>
   <EFEITO>PD2< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="E-alfa0" >
   <PRECONDICAO>PD2< /PRECONDICAO>
   <EFEITO>PE-alfa0< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="E-alfa1">
   <PRECONDICAO>PE-alfa0< /PRECONDICAO>
   <EFEITO>PE-alfa1< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="E-beta0">
   <PRECONDICAO>PD2< /PRECONDICAO>
   <EFEITO>PE-beta0< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="E-beta1">
   <PRECONDICAO>PE-beta0< /PRECONDICAO>
   <EFEITO>PE-beta1< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="E-Fim">
   <PRECONDICAO>(PE-alfa1 and PE-beta1)< /PRECONDICAO>
   <EFEITO>PX< /EFEITO>
< /ATIVIDADE>
<ATIVIDADE nome="X">
   <PRECONDICAO>PX< /PRECONDICAO>
   <EFEITO>FIM< /EFEITO>
```

```
  </ATIVIDADE>
</PROCESSO>
```

Exemplo 5.2. Domínio de planejamento em PDDL gerado pelo Metaplan, e submetido aos planejadores, que corresponde ao processo de *workflow* do Exemplo 5.1:

```
(define (domain processo1)
  (:predicates
    (PK31A)
    (PD0)
    (PD1)
    (PD2)
    (PK51A)
    (SPLIT-20)
    (SPLIT-21)
    (SPLIT-22)
    (SPLIT-23)
    (PK41A)
    (SPLIT-24)
    (PE-ALFA1)
    (SPLIT-25)
    (PE-ALFA0)
    (SPLIT-26)
    (FIM)
    (SPLIT-27)
    (SPLIT-28)
    (SPLIT-29)
    (SPLIT-6)
    (SPLIT-5)
    (PK9A2)
    (SPLIT-8)
    (PK9A1)
    (SPLIT-7)
    (SPLIT-9)
    (PK8A1)
    (PK8A3)
    (PK8A2)
    (PB2)
    (PK6A1)
    (SPLIT-12)
    (PK6A2)
    (SPLIT-13)
    (PK6A3)
    (SPLIT-10)
    (SPLIT-11)
    (PK21A)
    (SPLIT-16)
    (PK7A1)
    (SPLIT-17)
    (PK7A2)
    (SPLIT-14)
    (PK7A3)
    (SPLIT-15)
    (PX)
    (PK9A3)
    (SPLIT-1)
    (SPLIT-2)
    (SPLIT-3)
    (SPLIT-18)
    (SPLIT-4)
    (SPLIT-19)
```

```
    (PE-BETA1)
    (PE-BETA0)
    (PB1)
    (PK2A3)
    (PK2A2)
    (PK3A2)
    (PK3A1)
    (PK2A1)
    (PK3A3)
    (PK5A1)
    (PK5A2)
    (PK5A3)
    (PA1)
    (PK4A1)
    (PK4A2)
    (PK4A3)
    (PA0)
    (PK91A)
    (PK81A)
    (PK10A1)
    (PK61A)
    (PK10A2)
    (PK10A3)
    (PK71A)
)
(:action A0
    :precondition()
    :effect (PA0)
)
(:action A1
    :precondition (PA0)
    :effect (PA1)
)
(:action B-DCA1
    :precondition (SPLIT-1)
    :effect (and (PB1) (not (SPLIT-1)) (not (SPLIT-2)))
)
(:action B-DCA2
    :precondition (SPLIT-2)
    :effect (and (PB2) (not (SPLIT-1)) (not (SPLIT-2)))
)
(:action B
    :precondition (PA1)
    :effect (and (SPLIT-1) (SPLIT-2))
)
(:action K2A-DCA1
    :precondition (SPLIT-3)
    :effect (and (PK2A1) (not (SPLIT-3)) (not (SPLIT-4)) (not (SPLIT-5)))
)
(:action K2A-DCA2
    :precondition (SPLIT-4)
    :effect (and (PK2A2) (not (SPLIT-3)) (not (SPLIT-4)) (not (SPLIT-5)))
)
(:action K2A-DCA3
    :precondition (SPLIT-5)
    :effect (and (PK2A3) (not (SPLIT-3)) (not (SPLIT-4)) (not (SPLIT-5)))
)
(:action K2A
    :precondition (PB1)
```

```
    :effect (and (SPLIT-3) (SPLIT-4) (SPLIT-5))
)
(:action K2A1
    :precondition (PK2A1)
    :effect (PK21A)
)
(:action K2A2
    :precondition (PK2A2)
    :effect (PB2)
)
(:action K2A3
    :precondition (PK2A3)
    :effect (PB2)
)
(:action K3A-DCA1
    :precondition (SPLIT-6)
    :effect (and (PK3A1) (not (SPLIT-6)) (not (SPLIT-7)) (not (SPLIT-8)))
)
(:action K3A-DCA2
    :precondition (SPLIT-7)
    :effect (and (PK3A2) (not (SPLIT-6)) (not (SPLIT-7)) (not (SPLIT-8)))
)
(:action K3A-DCA3
    :precondition (SPLIT-8)
    :effect (and (PK3A3) (not (SPLIT-6)) (not (SPLIT-7)) (not (SPLIT-8)))
)
(:action K3A
    :precondition (PK21A)
    :effect (and (SPLIT-6) (SPLIT-7) (SPLIT-8))
)
(:action K3A1
    :precondition (PK3A1)
    :effect (PK31A)
)
(:action K3A2
    :precondition (PK3A2)
    :effect (PB2)
)
(:action K3A3
    :precondition (PK3A3)
    :effect (PB2)
)
(:action K4A-DCA1
    :precondition (SPLIT-9)
    :effect (and (PK4A1) (not (SPLIT-9)) (not (SPLIT-10)) (not (SPLIT-11)))
)
(:action K4A-DCA2
    :precondition (SPLIT-10)
    :effect (and (PK4A2) (not (SPLIT-9)) (not (SPLIT-10)) (not (SPLIT-11)))
)
(:action K4A-DCA3
    :precondition (SPLIT-11)
    :effect (and (PK4A3) (not (SPLIT-9)) (not (SPLIT-10)) (not (SPLIT-11)))
)
(:action K4A
    :precondition (PK31A)
    :effect (and (SPLIT-9) (SPLIT-10) (SPLIT-11))
)
(:action K4A1
```

```
    :precondition (PK4A1)
    :effect (PK41A)
)
(:action K4A2
    :precondition (PK4A2)
    :effect (PB2)
)
(:action K4A3
    :precondition (PK4A3)
    :effect (PB2)
)
(:action K5A-DCA1
    :precondition (SPLIT-12)
    :effect (and (PK5A1) (not (SPLIT-12)) (not (SPLIT-13)) (not (SPLIT-14)))
)
(:action K5A-DCA2
    :precondition (SPLIT-13)
    :effect (and (PK5A2) (not (SPLIT-12)) (not (SPLIT-13)) (not (SPLIT-14)))
)
(:action K5A-DCA3
    :precondition (SPLIT-14)
    :effect (and (PK5A3) (not (SPLIT-12)) (not (SPLIT-13)) (not (SPLIT-14)))
)
(:action K5A
    :precondition (PK41A)
    :effect (and (SPLIT-12) (SPLIT-13) (SPLIT-14))
)
(:action K5A1
    :precondition (PK5A1)
    :effect (PK51A)
)
(:action K5A2
    :precondition (PK5A2)
    :effect (PB2)
)
(:action K5A3
    :precondition (PK5A3)
    :effect (PB2)
)
(:action K6A-DCA1
    :precondition (SPLIT-15)
    :effect (and (PK6A1) (not (SPLIT-15)) (not (SPLIT-16)) (not (SPLIT-17)))
)
(:action K6A-DCA2
    :precondition (SPLIT-16)
    :effect (and (PK6A2) (not (SPLIT-15)) (not (SPLIT-16)) (not (SPLIT-17)))
)
(:action K6A-DCA3
    :precondition (SPLIT-17)
    :effect (and (PK6A3) (not (SPLIT-15)) (not (SPLIT-16)) (not (SPLIT-17)))
)
(:action K6A
    :precondition (PK51A)
    :effect (and (SPLIT-15) (SPLIT-16) (SPLIT-17))
)
(:action K6A1
    :precondition (PK6A1)
    :effect (PK61A)
)
```

```
(:action K6A2
   :precondition (PK6A2)
   :effect (PB2)
)
(:action K6A3
   :precondition (PK6A3)
   :effect (PB2)
)
(:action K7A-DCA1
   :precondition (SPLIT-18)
   :effect (and (PK7A1) (not (SPLIT-18)) (not (SPLIT-19)) (not (SPLIT-20)))
)
(:action K7A-DCA2
   :precondition (SPLIT-19)
   :effect (and (PK7A2) (not (SPLIT-18)) (not (SPLIT-19)) (not (SPLIT-20)))
)
(:action K7A-DCA3
   :precondition (SPLIT-20)
   :effect (and (PK7A3) (not (SPLIT-18)) (not (SPLIT-19)) (not (SPLIT-20)))
)
(:action K7A
   :precondition (PK61A)
   :effect (and (SPLIT-18) (SPLIT-19) (SPLIT-20))
)
(:action K7A1
   :precondition (PK7A1)
   :effect (PK71A)
)
(:action K7A2
   :precondition (PK7A2)
   :effect (PB2)
)
(:action K7A3
   :precondition (PK7A3)
   :effect (PB2)
)
(:action K8A-DCA1
   :precondition (SPLIT-21)
   :effect (and (PK8A1) (not (SPLIT-21)) (not (SPLIT-22)) (not (SPLIT-23)))
)
(:action K8A-DCA2
   :precondition (SPLIT-22)
   :effect (and (PK8A2) (not (SPLIT-21)) (not (SPLIT-22)) (not (SPLIT-23)))
)
(:action K8A-DCA3
   :precondition (SPLIT-23)
   :effect (and (PK8A3) (not (SPLIT-21)) (not (SPLIT-22)) (not (SPLIT-23)))
)
(:action K8A
   :precondition (PK71A)
   :effect (and (SPLIT-21) (SPLIT-22) (SPLIT-23))
)
(:action K8A1
   :precondition (PK8A1)
   :effect (PK81A)
)
(:action K8A2
   :precondition (PK8A2)
   :effect (PB2)
```

```
)
(:action K8A3
   :precondition (PK8A3)
   :effect (PB2)
)
(:action K9A-DCA1
   :precondition (SPLIT-24)
   :effect (and (PK9A1) (not (SPLIT-24)) (not (SPLIT-25)) (not (SPLIT-26)))
)
(:action K9A-DCA2
   :precondition (SPLIT-25)
   :effect (and (PK9A2) (not (SPLIT-24)) (not (SPLIT-25)) (not (SPLIT-26)))
)
(:action K9A-DCA3
   :precondition (SPLIT-26)
   :effect (and (PK9A3) (not (SPLIT-24)) (not (SPLIT-25)) (not (SPLIT-26)))
)
(:action K9A
   :precondition (PK81A)
   :effect (and (SPLIT-24) (SPLIT-25) (SPLIT-26))
)
(:action K9A1
   :precondition (PK9A1)
   :effect (PK91A)
)
(:action K9A2
   :precondition (PK9A2)
   :effect (PB2)
)
(:action K9A3
   :precondition (PK9A3)
   :effect (PB2)
)
(:action K10A-DCA1
   :precondition (SPLIT-27)
   :effect (and (PK10A1) (not (SPLIT-27)) (not (SPLIT-28)) (not (SPLIT-29)))
)
(:action K10A-DCA2
   :precondition (SPLIT-28)
   :effect (and (PK10A2) (not (SPLIT-27)) (not (SPLIT-28)) (not (SPLIT-29)))
)
(:action K10A-DCA3
   :precondition (SPLIT-29)
   :effect (and (PK10A3) (not (SPLIT-27)) (not (SPLIT-28)) (not (SPLIT-29)))
)
(:action K10A
   :precondition (PK91A)
   :effect (and (SPLIT-27) (SPLIT-28) (SPLIT-29))
)
(:action K10A1
   :precondition (PK10A1)
   :effect (PB2)
)
(:action K10A2
   :precondition (PK10A2)
   :effect (PB2)
)
(:action K10A3
   :precondition (PK10A3)
```

```
    :effect (PB2)
  )
  (:action D0
    :precondition (PB2)
    :effect (PD0)
  )
  (:action D1
    :precondition (PD0)
    :effect (PD1)
  )
  (:action D2
    :precondition (PD1)
    :effect (PD2)
  )
  (:action E-ALFA0
    :precondition (PD2)
    :effect (PE-ALFA0)
  )
  (:action E-ALFA1
    :precondition (PE-ALFA0)
    :effect (PE-ALFA1)
  )
  (:action E-BETA0
    :precondition (PD2)
    :effect (PE-BETA0)
  )
  (:action E-BETA1
    :precondition (PE-BETA0)
    :effect (PE-BETA1)
  )
  (:action E-FIM
    :precondition (and (PE-ALFA1)(PE-BETA1))
    :effect (PX)
  )
  (:action X
    :precondition (PX)
    :effect (FIM)
  )
)
```

Exemplo 5.3. O plano gerado pelo sistema SLPLan, a partir do arquivo PDLL, do Exemplo 5.2 é descrito abaixo. Este plano será repassado ao Metaplan para que ele extraia a modelagem do processo em questão.

```
(A1)
(B)
(B-DCA2)
(D0)
(D1)
(D2)
(E-ALFA0)
(E-ALFA1)
(E-BETA0)
(E-BETA1)
(E-FIM)
(X)
```

A Figura 5.4 apresenta um comparativo entre os sistemas. O eixo horizontal representa os processos gerados ilustrando a quantidade de tarefas de cada um. Já o eixo vertical

mostra o tempo médio de geração dos planos. As condições de execução foram as mesmas descritas na Seção 4.1 do Capítulo 4. Podemos ver que o sistema Metaplan-SLPlan obteve resultados melhores que o Metaplan-FF para maioria dos processos, onde o tempo médio de execução, considerando todos os 50 processos, foi de 0,187 segundos para o Metaplan-FF e 0,171 segundos para o Metaplan-SLPlan.

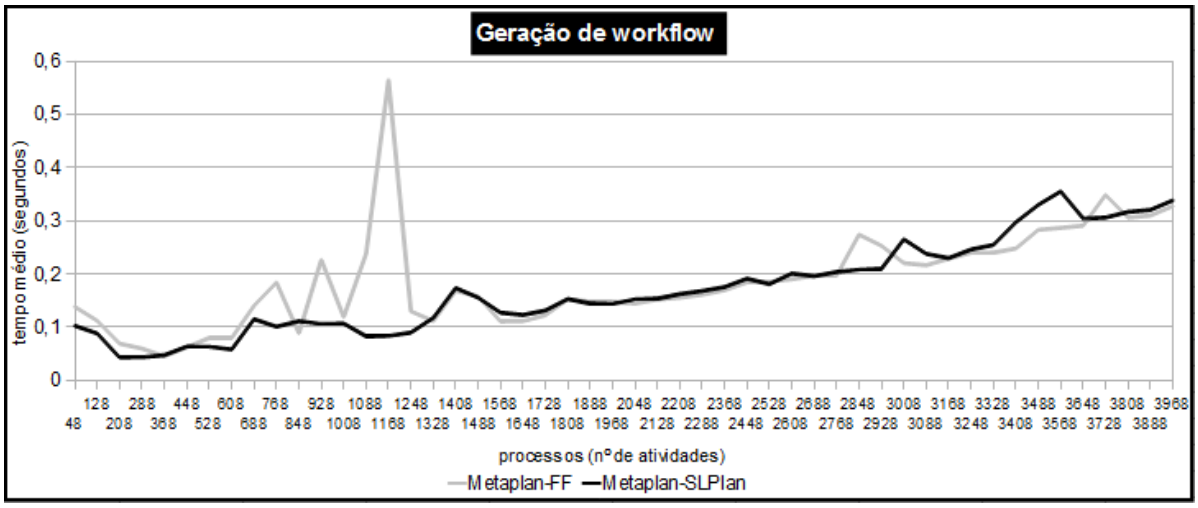

Figura 5.4: Tempo médio: Metaplan-FF e Metaplan-SLPlan.

Capítulo 6

Conclusão e Trabalhos Futuros

Planejamento é um processo deliberativo, onde devemos escolher dentre todas as ações disponíveis, aquelas que possam formar um plano que resolva um dado problema. A automatização de planejamento vem sendo estudada desde a década de 70, e a partir da utilização de técnicas de busca e heurísticas na década de 90, esta área tem sido um dos principais tópicos de pesquisa na Inteligência Artificial.

Nesta dissertação foi apresentado o sistema de planejamento automático SLPlan (Capítulo 3), que faz uso de dois algoritmos de busca. Esses algoritmos, chamados de HB-EHC e Adaptive-LRTA*, são otimizações dos algoritmos EHC e LRTA* que tem como objetivo resolver problemas de planejamento de forma mais eficiente, reduzindo o tempo de execução e a taxa de falhas por falta de espaço de memória no caso de problemas muito grandes.

Para avaliar a viabilidade do planejador, foram feitas duas etapas de testes neste trabalho, descritas na Seção 4.3. A primeira analisava os algoritmos desenvolvidos separadamente, comparando suas performances com outros algoritmos de mesmas características. Sendo assim, o HB-EHC foi comparado com o HC e o EHC, cujo principal objetivo era verificar se sua taxa de completude aumentaria sem prejudicar o tempo de execução, evitando o acionamento de um algoritmo completo. As melhorias obtidas com o HB-EHC foram publicadas em [Alves e Lopes 2013a]. Já o algoritmo Adaptive-LRTA* foi comparado com sua versão original, LRTA*, e uma versão otimizada, chamada LRTA*-K, a fim de demostrar sua capacidade em resolver problemas de forma mais rápida, além de reduzir a taxa de falhas por falta de espaço em memória. Em [Alves e Lopes 2013b] e [Alves et al. 2013], foram publicadas algumas das otimizações que originaram este algoritmo e suas contribuições na área de planejamento.

Na segunda etapa de testes foi avaliado o desempenho geral do planejador SLPlan, bem como sua estratégia de busca, utilizando o planejador FF para fins de comparação. Como resultado, o planejador desenvolvido apresentou resultados promissores, onde se teve um ganho significativo no tempo de planejamento e uma redução da taxa de falhas por falta de espaço devido ao balanceamento de memória feito durante as buscas. Também foi

realizada uma aplicação prática para o planejador SLPlan, onde ele foi usado como motor de geração automática de *workflow* dentro do sistema Metaplan.

Apesar do planejador desenvolvido apresentar uma abordagem promissora, ele ainda leva uma certa desvantagem em relação aos planejadores atuais que participam das competições internacionais de planejamento. Isto se deve pelo fato de que o SLPlan é sensível a problemas não STRIPS, ou seja, ainda é necessário melhorar sua parte de interpretação de arquivos PDDL para problemas que não fazem parte da categoria STRIPS. Com isso, acredita-se que ele seria um método competitivo e que poderia participar futuramente das competições de planejamento sem restrição alguma, pois o trabalho realizado nesta dissertação independe desta nova funcionalidade. Uma outra melhoria que poderia ser colocada em prática, seria o cálculo dinâmico da quantidade de estados que podem ser armazenados dentro do *heap*, aproveitando ao máximo o uso da memória, independente da configuração de *hardware* do computador.

O planejador SLPlan se mostrou bastante eficiente para resolver os problemas do *benchmark* proposto. Mas pesquisas recentes mostram que, os planejadores atuais não conseguem ter bons resultados para todos os problemas de planejamentos criados até então. Com isso, está surgindo uma nova tendência em planejamento que é o "portfólio de planejadores" [Vallati 2012]. Neste caso, o portfólio nada mais é que um aglomerado de planejadores já desenvolvidos, onde um dado problema é submetido ao portfólio, e quando um de seus planejadores encontra a solução, o mesmo retorna o plano gerado. Sendo assim, além de poder ser usado de forma separada, o SLPlan poderá compor um portfólio para que ele possa resolver problemas mais adequados às suas características.

Referências Bibliográficas

[Akramifar e Ghassem-Sani 2010] Akramifar, S. A. e Ghassem-Sani, G. (2010). Fast forward planning by guided enforced hill climbing. *Eng. Appl. Artif. Intell.*, 23(8):1327–1339.

[Albore et al. 2009] Albore, A., Palacios, H., e Geffner, H. (2009). A Translation-Based Approach to Contingent Planning. In Boutilier, C. (editor), *IJCAI*, pp. 1623–1628.

[Aler et al. 2002] Aler, R., Borrajo, D., Camacho, D., e Sierra-Alonso, A. (2002). A knowledge-based approach for business process reengineering, SHAMASH. *Knowl.-Based Syst.*, 15(8):473–483.

[Alves et al. 2013] Alves, R., Lopes, C., e Branquinho, A. (2013). Generating Plans Using LRTA. In *Intelligent Systems (BRACIS), 2013 Brazilian Conference on*, pp. 207–212.

[Alves e Lopes 2013a] Alves, R. M. e Lopes, C. R. (2013a). Automated Planning with Adapted Enforced Hill Climbing. In *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*, pp. 2258–2263.

[Alves e Lopes 2013b] Alves, R. M. F. e Lopes, C. R. (2013b). Solving Planning Problems with LRTA*. In Hammoudi, S., Maciaszek, L. A., Cordeiro, J., e Dietz, J. L. G. (editores), *ICEIS (1)*, pp. 475–481. SciTePress.

[Amanda et al. 2012] Amanda, C., Andrew, C., Olaya, A. G., Jimenez, S., Lopez, C. L., Sanner, S., e Yoon, S. (2012). A Survey of the Seventh International Planning Competition. *AI Magazine*, 33(1):1–8.

[Bacchus 2001] Bacchus, F. (2001). The AIPS '00 Planning Competition. *AI Magazine*, 22(3):47–56.

[Berry e Drabble 1999] Berry, P. M. e Drabble, B. (1999). SWIM: An AI-based System for Workflow Enabled Reactive Control.

[Blum e Furst 1995] Blum, A. L. e Furst, M. L. (1995). Fast Planning Through Planning Graph Analysis. *ARTIFICIAL INTELLIGENCE*, 90(1):1636–1642.

[Bonet et al. 1997] Bonet, B., Loerincs, G., e Geffner, H. (1997). A Robust and Fast Action Selection Mechanism for Planning. In *In Proceedings of AAAI-97*, pp. 714–719. MIT Press.

[Bonnet e Geffner 1998] Bonnet, B. e Geffner, H. (1998). HSP: Heuristic Search Planner. Entry at the AIPS-98 Planning Competition, Pittsburgh.

[Botea et al. 2005] Botea, A., Enzenberger, M., 0003, M. M., e Schaeffer, J. (2005). Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *J. Artif. Intell. Res. (JAIR)*, 24:581–621.

[Bulitko e Björnsson 2009] Bulitko, V. e Björnsson, Y. (2009). kNN LRTA*: Simple Subgoaling for Real-Time Search. In Darken, C. e Youngblood, G. M. (editores), *AIIDE*. The AAAI Press.

[Bulitko et al. 2008] Bulitko, V., Schaeffer, J., Björnsson, Y., Sigmundarson, S., e et al. (2008). Dynamic control in real-time heuristic search.

[Carbonell et al. 1991] Carbonell, J., Etzioni, O., Gil, Y., Joseph, R., Knoblock, C., Minton, S., e Veloso, M. (1991). PRODIGY: an integrated architecture for planning and learning. *SIGART Bull.*, 2(4):51–55.

[Casati et al. 1996] Casati, F., Grefen, P., Pernici, B., Pozzi, G., Sánchez, G., e Pernici, B. (1996). WIDE Workflow model and architecture.

[Coles e Smith 2007] Coles, A. e Smith, A. (2007). Marvin: A Heuristic Search Planner with Online Macro-Action Learning. *J. Artif. Intell. Res. (JAIR)*, 28:119–156.

[Coles et al. 2008] Coles, A. I., Fox, M., Long, D., e Smith, A. J. (2008). Teaching Forward-Chaining Planning with JavaFF. In *Colloquium on AI Education, Twenty-Third AAAI Conference on Artificial Intelligence*.

[Coppin 2004] Coppin, B. (2004). *Artificial Intelligence Illuminated*. Jones and Bartlett Publishers, Inc., USA.

[Department 2006] Department, D. B. (2006). POND: The Partially-Observable and Non-Deterministic Planner.

[Dijkstra 1959] Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271.

[Domshlak e Hoffmann 2007] Domshlak, C. e Hoffmann, J. (2007). Probabilistic Planning via Heuristic Forward Search and Weighted Model Counting. *J. Artif. Intell. Res. (JAIR)*, 30:565–620.

[Edelkamp 2001] Edelkamp, S. (2001). Planning with Pattern Databases. In *PROCEEDINGS OF THE 6TH EUROPEAN CONFERENCE ON PLANNING (ECP-01)*, pp. 13–24.

[Edelkamp e Hoffmann 2003] Edelkamp, S. e Hoffmann, J. (2003). PDDL2.2: The Language for the Classical Part of the 4th International planning Competition. Technical report.

[Erol et al. 1994] Erol, K., Hendler, J., e Nau, D. S. (1994). UMCP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. In *PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE PLANNING SYSTEMS (AIPS 94)*, pp. 249–254.

[Fikes e Nilsson 1971] Fikes, R. e Nilsson, N. J. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artif. Intell.*, 2(3/4):189–208.

[Fox e Long 2002] Fox, M. e Long, D. (2002). PDDL+: Modelling Continuous Time-dependent Effects.

[Fox e Long 2003] Fox, M. e Long, D. (2003). PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:2003.

[Furcy 2004] Furcy, D. A. (2004). *Speeding up the convergence of online heuristic search and scaling up offline heuristic search.* PhD thesis.

[Gerevini e Long 2005] Gerevini, A. e Long, D. (2005). Plan constraints and preferences in PDDL3 - the language of the fifth international planning competition. Technical report.

[Gerevini et al. 2009] Gerevini, A. E., Haslum, P., Long, D., Saetti, A., e Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.*, 173(5-6):619–668.

[Green 1969] Green, C. (1969). Application of theorem proving to problem solving. pp. 219–239. Morgan Kaufmann.

[Hart et al. 1968] Hart, P., Nilsson, N., e Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107.

[Haslum e Geffner 2000] Haslum, P. e Geffner, H. (2000). Admissible Heuristics for Optimal Planning. pp. 140–149. AAAI Press.

[Helmert 2006] Helmert, M. (2006). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246.

[Helmert et al. 2007] Helmert, M., Haslum, P., e Hoffmann, J. (2007). Flexible abstraction heuristics for optimal sequential planning. In *IN ICAPS*, pp. 176–183.

[Hernández e Meseguer 2005] Hernández, C. e Meseguer, P. (2005). LRTA*(k). In Kaelbling, L. P. e Saffiotti, A. (editores), *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pp. 1238–1243. Professional Book Center.

[Hoffmann 2002] Hoffmann, J. (2002). The Metric-FF Planning System: Translating "Ignoring Delete Lists"to Numeric State Variables.

[Hoffmann e Brafman 2006] Hoffmann, J. e Brafman, R. I. (2006). Conformant planning via heuristic forward search: a new approach. *Artif. Intell.*, 170(6):507–541.

[Hoffmann e Nebel 2001] Hoffmann, J. e Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:2001.

[J. Hoffmann 2004] J. Hoffmann, S. Edelkamp, R. E. (2004). Towards realistic benchmarks for planning: the domains used in the classical part of IPC-04 (2004). In *In 4th International Planning Competition, 7-14 ICAPS.*

[Ko et al. 2009] Ko, R. K., Lee, S. S., e Lee, E. W. (2009). Business Process Management (BPM) standards: A survey. *Business Process Management journal*, 15(5).

[Koenig e Sun 2009] Koenig, S. e Sun, X. (2009). Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, 18(3):313–341.

[Korf 1990] Korf, R. E. (1990). Real-time heuristic search. *Artif. Intell.*, 42(2-3):189–211.

[Kowalski e Sergot 1986] Kowalski, R. e Sergot, M. (1986). A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95.

[Long e Fox 2003] Long, D. e Fox, M. (2003). The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59.

[Long e Fox 2006] Long, D. e Fox, M. (2006). The International Planning Competition Series and Empirical Evaluation of AI Planning Systems. In Paquete, L., Chiarandini, M., e Basso, D. (editores), *Proceedings of Workshop on Empirical Methods for the Analysis of Algorithm*.

[Mcdermott 2000] Mcdermott, D. (2000). The 1998 AI Planning Systems Competition. *AI Magazine*, 21:35–55.

[Mcdermott et al. 1998] Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., e Wilkins, D. (1998). PDDL - The Planning Domain Definition Language. Technical Report TR-98-003, Yale Center for Computational Vision and Control,.

[Melo 2005] Melo, J. T. d. (2005). Workflow com técnicas de planejamento apoiado em inteligência artificial. Master's thesis, Universidade Federal de Uberlândia.

[Michael Hammer 1994] Michael Hammer, J. C. (1994). *Reengenharia: revolucionando a empresa em função dos clientes, da concorrência e das grandes mudanças da gerência.*

[Myers e Berry 1999] Myers, K. e Berry, P. (1999). Workflow Management Systems: An AI Perspective. Technical report, AIC, SRI International.

[Newell e Simon 1963] Newell, A. e Simon, H. (1963). GPS, a program that simulates human thought. In Feigenbaum, E. e Feldman, J. (editores), *Computers and Thought*, pp. 279–293. McGraw-Hill, New York.

[Nguyen e Kambhampati 2000] Nguyen, X. e Kambhampati, S. (2000). Extracting Effective and Admissible State Space Heuristics from the Planning Graph. In *In Proc. AAAI-2000*, pp. 798–805. AAAI Press.

[Pednault 1989] Pednault, E. P. D. (1989). ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pp. 324–332, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Penberthy e Weld 1992] Penberthy, J. S. e Weld, D. S. (1992). UCPOP: A Sound, Complete, Partial Order Planner for ADL. pp. 103–114. Morgan Kaufmann.

[Rames Basilio Junior e Lopes 2012] Rames Basilio Junior, R. e Lopes, C. (2012). An approach to action planning based on simulated annealing. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pp. 2085 –2090.

[Rayner et al. 2007] Rayner, D. C., Davison, K., Bulitko, V., Anderson, K., e Lu, J. (2007). Real-time heuristic search with a priority queue. In *In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI*, pp. 2372–2377.

[Reijers 2003] Reijers, H. A. (2003). *Design and Control of Workflow Processes: Business Process Management for the Service Industry*. Número 2617 in Lecture Notes in Computer Science. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Richter et al. 2008] Richter, S., Helmert, M., e Westphal, M. (2008). Landmarks revisited. In *in: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-2008*. AAAI Press.

[Richter e Westphal 2010] Richter, S. e Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks.

[Russell e Norvig 2009] Russell, S. J. e Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition.

[Sacerdoti 1975] Sacerdoti, E. D. (1975). The nonlinear nature of plans. In *Proceedings of the 4th international joint conference on Artificial intelligence - Volume 1*, IJCAI'75, pp. 206–214, Tblisi, USSR, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Shi et al. 2010] Shi, Y., Yang, M., e Sun, R. (2010). Goal-Driven Workflow Generation Based on AI Planning. In Li, D., Liu, Y., e Chen, Y. (editores), *CCTA (3)*, volume 346 de *IFIP Advances in Information and Communication Technology*, pp. 367–374. Springer.

[Stern et al. 2011] Stern, R., Puzis, R., e Felner, A. (2011). Potential Search: A Bounded-Cost Search Algorithm. In *ICAPS*.

[Sturtevant e Bulitko 2011] Sturtevant, N. R. e Bulitko, V. (2011). Learning where you are going and from whence you came: h- and g-cost learning in real-time heuristic search. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence - Volume Volume One*, IJCAI'11, pp. 365–370, Barcelona, Catalonia, Spain. AAAI Press.

[Vallati 2012] Vallati, M. (2012). A Guide to Portfolio-Based Planning. In Sombattheera, C., Loi, N. K., Wankar, R., e Quan, T. T. (editores), *MIWAI*, volume 7694, pp. 57–68. Springer.

[Van Der Aalst et al. 2003] Van Der Aalst, W. M. P., Hofstede, A. H. M. T., e Weske, M. (2003). Business process management: a survey. In *Proceedings of the 2003 international conference on Business process management*, BPM'03, pp. 1–12, Eindhoven, The Netherlands, Berlin, Heidelberg. Springer-Verlag.

[van der Aalst e van Hee 2002] van der Aalst, W. M. P. e van Hee, K. (2002). *Workflow Management: Models, Methods, and Systems*. MIT Press.

[Wainer e de Lima Bezerra 2003] Wainer, J. e de Lima Bezerra, F. (2003). Constraint-Based Flexible Workflows. In *Proceedings of the 9th International Workshop on Groupware: Design, Implementation, and Use (CRIWG 2003)*, volume 2806, pp. 151 – 158. Springer-Verlag, Berlin.

[Weske 2007] Weske, M. (2007). *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Wilkins 1988] Wilkins, D. E. (1988). *Practical planning: extending the classical AI planning paradigm*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Wilkins 1989] Wilkins, D. E. (1989). Can Ai Planners Solve Practical Problems? Technical Report 468, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025. Revised.

[Wu et al. 2011] Wu, J.-H., Kalyanam, R., e Givan, R. (2011). Stochastic Enforced Hill-Climbing. *J. Artif. Intell. Res. (JAIR)*, 42:815–850.

[Xie et al. 2012] Xie, F., Nakhost, H., e Müller, M. (2012). Planning Via Random Walk-Driven Local Search. In *ICAPS*.