# EXPLORAÇÃO DOS PARADIGMAS BIDIRECIONAL E PARALELO EM ALGORITMOS DE BUSCA HEURÍSTICA

LUIS HENRIQUE OLIVEIRA RIOS

# EXPLORAÇÃO DOS PARADIGMAS BIDIRECIONAL E PARALELO EM ALGORITMOS DE BUSCA HEURÍSTICA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: LUIZ CHAIMOWICZ

Belo Horizonte

Abril de 2012

© 2012, Luis Henrique Oliveira Rios.
Todos os direitos reservados.

Rios, Luis Henrique Oliveira
R586e Exploração dos paradigmas bidirecional e paralelo em algoritmos de busca heurística / Luis Henrique Oliveira Rios. — Belo Horizonte, 2012
xxiv, 115 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Luiz Chaimowicz

1. Computação - Teses. 2. Inteligência artifical - Teses. I. Orientador. II. Título.

CDU 519.6*82(043)

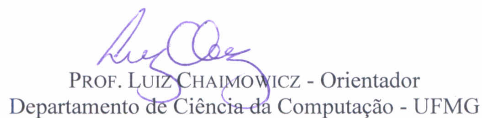
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# FOLHA DE APROVAÇÃO

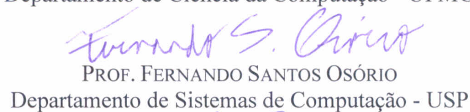
Exploração dos paradigmas bidirecional e paralelo em algoritmos de busca heurística

**LUIS HENRIQUE OLIVEIRA RIOS**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

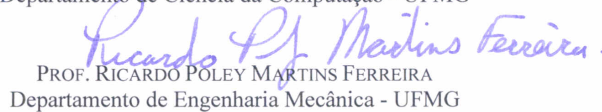


PROF. LUIZ CHAIMOWICZ - Orientador
Departamento de Ciência da Computação - UFMG



PROF. FERNANDO SANTOS OSÓRIO
Departamento de Sistemas de Computação - USP

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG



PROF. RICARDO POLEY MARTINS FERREIRA
Departamento de Engenharia Mecânica - UFMG

Belo Horizonte, 26 de abril de 2012.

# Resumo


O A* é um importante algoritmo de busca heurística em Inteligência Artificial. A heurística proporciona uma diminuição significativa no esforço computacional da busca. Entretanto, em muitos contextos isso não é suficiente. Com o intuito de lidar melhor com essa questão, várias extensões do algoritmo A* tem sido propostas.

O objetivo central deste trabalho é investigar formas de melhorar o desempenho do A* através de abordagens bidirecionais e paralelas para propor novos algoritmos. Suas contribuições, portanto, são uma forma de organizar os principais algoritmos de busca baseados no A* que foram propostos na literatura e dois novos algoritmos de busca heurística bidirecional paralela chamados PNBA* e BPBNF.

A classificação das extensões do A* exposta neste trabalho é uma forma de organizar os principais algoritmos de busca baseados no A* presentes na literatura. Ela é estruturada em seis classes (bidirecional, incremental, *memory-concerned*, paralela, *anytime* e tempo-real) não excludentes entre si.

O PNBA* é uma implementação paralela do NBA* (algoritmo de busca heurística bidirecional) para ambientes computacionais de memória compartilhada. Seus dois processos de busca são executados em paralelo. Em todos os domínios empregados nos experimentos, o PNBA* foi mais rápido do que o A* e o NBA*.

O BPBNF generaliza a idéia do algoritmo PNBA* para mais de dois processadores e também reduz o tempo de execução do PBNF (algoritmo no qual ele se baseia). A comparação empírica dos desempenhos evidenciou uma clara superioridade do BPBNF em relação ao A*. Se comparado ao PBNF, em dois dos três domínios empregados também foi possível notar a sua superioridade.

Portanto, este trabalho mostra ser viável e factível a combinação dos paradigmas bidirecional e paralelo para redução do tempo de execução do algoritmo de busca heurística A*, mantendo a admissibilidade.



**Palavras-chave:** busca heurística bidirecional paralela, classificação de algoritmos de busca heurística, A*, PNBA*, BPBNF.

# Abstract

A* is a very important heuristic search algorithm in Artificial Intelligence. The use of a heuristic provides a significant reduction in the computational efforts of the search algorithm. However, in many contexts this is not sufficient. In order to better deal with this issue, several extensions of the A* algorithm have been proposed.

The goal of this dissertation is to investigate ways of improving the performance of A* through bidirectional and parallel approaches to propose new algorithms. Therefore, the contributions are: a way of organizing the main search algorithms based on A* that have been proposed in the literature and two new parallel bidirectional heuristic search algorithms called PNBA* and BPBNF.

We discuss and organize the main extensions of A* in six different classes: bidirectional, incremental, memory-concerned, parallel, anytime and real-time. These classes are not mutually exclusive and represent the main objectives and characteristics of the majority of A* extensions found in the literature.

The PNBA* is a parallel implementation of NBA* (a bidirectional heuristic search algorithm) for computational environments with shared memory. Its two search processes are executed in parallel. We show in our experiments that PNBA* is faster than A* and NBA* in three different application domains.

The BPBNF algorithm generalizes the idea of PNBA* for more than two processors and also reduces the execution time of PBNF (an algorithm in which it is based on). Our experiments have showed a clear superiority of BPBNF relative to A*. When compared to PBNF in two of the three tested domains, it was also possible to note BPBNF supremacy.

Therefore, this dissertation shows the viability and the feasibility of combining the bidirectional and parallel paradigms in order to reduce the run time of A* while keeping its admissibility.

**Keywords:** bidirectional parallel heuristic search, heuristic search algorithms classification, A*, PNBA*, BPBNF.

# Lista de Figuras

2.1 Representação gráfica de $f(x)$, $g(x)$ e $h(x)$. . . . . . . . . . . . . . . . . . 8
2.2 Heurística consistente interpretada como uma desigualdade triangular. . 8
2.3 Espaço de estados visitado na busca heurística bidirecional e unidirecional. 12
2.4 *Duplicate detection scope* e *interference scope*. . . . . . . . . . . . . . . . . . . . 32
2.5 As seis classes que compõe a classificação dos algoritmos baseados no A*. 43
2.6 Exemplos de extensões do A* e as classes as quais pertencem. . . . . . . 44

3.1 *Grids* com arestas de custo uniforme e não uniforme e seus respectivos grafos. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
3.2 As configurações inicial e final do *15-puzzle*. . . . . . . . . . . . . . . . . . . 55
3.3 Exemplo de *grid* gerado automaticamente para os experimentos. . . . . . 58
3.4 Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme. . . . . . . . . . . . . . . . . . . . . . . . . 62
3.5 Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme. . . . . . . . . . . . . . . . . . . . . 64
3.6 Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme. . . . . . . . . . . . . . . . . . . . . . . . 66
3.7 Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme. . . . . . . . . . . . . . . . . . 67
3.8 Tempo total absoluto e relativo no domínio do *15-puzzle*. . . . . . . . . . 68
3.9 Número de expansões absoluto e relativo no domínio do *15-puzzle*. . . . 70

4.1 Tempo total relativo para os algoritmos PBNF e BPBNF com duas, três e quatro *threads* nos três domínios avaliados. . . . . . . . . . . . . . . . . . 80
4.2 Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme. . . . . . . . . . . . . . . . . . . . . . . . . 83
4.3 Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme. . . . . . . . . . . . . . . . . . . . . 85

4.4 Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme. . . . . . . . . . . . . . . . . . . . . . . . . . . 87
4.5 Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme. . . . . . . . . . . . . . . . . . . . . 88
4.6 Tempo total absoluto e relativo no domínio do *15-puzzle*. . . . . . . . . . 89
4.7 Número de expansões absoluto e relativo no domínio do *15-puzzle*. . . . 91
4.8 Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme. . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
4.9 Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme. . . . . . . . . . . . . . . . . . . . . . 96
4.10 Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme. . . . . . . . . . . . . . . . . . . . . . . . . . . 98
4.11 Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme. . . . . . . . . . . . . . . . . . . . . 99
4.12 Tempo total absoluto e relativo no domínio do *15-puzzle*. . . . . . . . . . 101
4.13 Número de expansões absoluto e relativo no domínio do *15-puzzle*. . . . 103

# Lista de Tabelas

3.1 Tempo absoluto para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme. . . . . . . . . . . . . . . . . . . . . . . . . 63
3.2 Número de expansões e tempo total relativo para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme. . . . . . . . 65
3.3 Tempo absoluto para o domínio do *15-puzzle*. . . . . . . . . . . . . . . . . . 69
3.4 Número de expansões e tempo total relativo para o domínio do *15-puzzle*. 71

4.1 Tempo absoluto para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme. . . . . . . . . . . . . . . . . . . . . . . . . 84
4.2 Número de expansões e tempo total relativo para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme. . . . . . . . 86
4.3 Tempo absoluto para o domínio do *15-puzzle*. . . . . . . . . . . . . . . . . . 90
4.4 Número de expansões e tempo total relativo para o domínio do *15-puzzle*. 92
4.5 Tempo absoluto para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme. . . . . . . . . . . . . . . . . . . . . . . . . 97
4.6 Número de expansões e tempo total relativo para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme. . . . . . . . 100
4.7 Tempo absoluto para o domínio do *15-puzzle*. . . . . . . . . . . . . . . . . . 102
4.8 Número de expansões e tempo total relativo para o domínio do *15-puzzle*. 104

# Lista de Algoritmos

2.1 O algoritmo A*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2 O algoritmo NBA*. . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.3 O algoritmo IDA*. . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.4 O algoritmo AWA*. . . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.1 O algoritmo PNBA*. . . . . . . . . . . . . . . . . . . . . . . . . . 51

# Lista de Acrônimos

| *Acrônimo* | *Significado* |
|---|---|
| A* | A-estrela |
| AA* | *Adaptive A** |
| ARA* | *Anytime Repairing A** |
| ANA* | *Anytime Nonparametric A** |
| AWA* | *Anytime Weighted A** |
| BDD | *Binary Decision Diagram* |
| BDD-IDA* | *Binary Decision Diagram Iterative-Deepening A** |
| BHFFA | *Bidirectional Heuristic Front-to-Front Algorithm* |
| BPBNF | *Bidirectional Parallel Best-NBlock-First* |
| D* | *Dynamic A** |
| DDD | *Delayed Duplicate Detection* |
| DEC-POMDP | *Decentralized partially-observable Markov decision problem* |
| DNA | *Deoxyribonucleic Acid* (Ácido desoxirribonucleico) |
| FSA* | *Fringe-Saving A** |
| GAA* | *Generalized Adaptive A** |
| GPS | *Global Positioning System* (Sistema de Posicionamento Global) |
| HDA* | *Hash Distributed A** |
| iA* | *incremental A** |
| IDA* | *Iterative-Deepening A** |
| KBFS | *K-Best-First Search* |
| HPA* | *Hierarchical Path-Finding A** |
| LRTA* | *Learning Real-Time A** |
| MAA* | *Multi-agent A** |
| MT-D* Lite | *Moving Target D* Lite* |
| NBA* | *New Bidirectional A** |
| NPC | *Non Player Character* (Personagem Não Jogável) |

(A listagem continua na página seguinte)

| *Acrônimo* | *Significado* |
|---|---|
| PA* | *Parallel A** |
| PEA* | *Partial Expansion A** |
| PRA* | *Parallel Retracting A** |
| PBA* | *Parallel Bidirectional A** |
| PBA*S | *Parallel Bidirectional A* Search* |
| PBNF | *Parallel Best-NBlock-First* |
| PNBA* | *Parallel New Bidirectional A** |
| RBFS | *Recursive Best-First Search* |
| RNA | *Ribonucleic Acid* (Ácido Ribonucleico) |
| RTA* | *Real-Time A** |
| RTS | *Real Time Strategy* (Estratégia em Tempo Real) |
| SMA* | *Simple Memory-Bounded A** |
| SMAG* | *Simple Memory-Bounded A* Graph* |
| TBA* | *Time-Bounded A** |

# Lista de Conceitos e Definições

- **Algoritmo de busca admissível**: assegura o cômputo da solução ótima (caso haja uma solução);
- **Algoritmo de busca completo**: sempre encontra uma solução (caso algum caminho entre $s$ e $t$ exista);
- ***Duplicate detection scope* de** $a$: o conjunto de *nblocks* sucessores de $a$ no grafo abstrato;
- **Grafo inverso**: versão de um grafo onde cada aresta original $(x, y)$ é substituída por uma aresta $(y, x)$ com o mesmo peso;
- **Heurística admissível**: heurística que nunca superestima o custo real, ou seja, $h(x) \leq h^*(x)$ para todos nós $x$;
- **Heurística consistente / monotônica**: garante que $h(x) \leq d^*(x, y) + h(y)$ para todos nós $x$ e $y$ (ou $h(x) \leq d(x, y) + h(y)$ para qualquer aresta $(x, y)$);
- ***Interference scope* de** $a$: o conjunto de *nblocks* cujos *duplicate detection scopes* interferem com o *duplicate detection scope* do *nblock* $a$;
- ***Nblock***: subregião do espaço de estados. Definido através da função de mapeamento, é corresponde a um nó abstrato;
- **Processo de busca**: uma busca em execução. Também pode ser entendido como uma "instância" de uma busca;

# Lista de Notações e Símbolos

| *Símbolo* | *Significado* |
|---|---|
| | **Comum a busca unidirecional, bidirecional e paralela** |
| $s$ | *start* (nó inicial do caminho que será calculado no grafo a ser explorado) |
| $t$ | *goal* (nó final do caminho que será calculado no grafo a ser explorado) |
| $(x, y)$ | aresta do grafo que conecta o nó $x$ ao nó $y$ |
| *closed* / $\mathcal{M}$ | estruturas de dados para controlar os nós já expandidos[1]. Aquela armazena os nós já expandidos, e essa armazena os nós ainda não expandidos |
| | **Busca unidirecional** |
| *open* | estrutura de dados que armazena os nós candidatos a expansão |
| $d(x, y)$ | peso da aresta do grafo que conecta o nó $x$ ao nó $y$ |
| $d^*(x, y)$ | custo do menor caminho de $x$ até $y$ (nesse caso, não necessariamente dois nós vizinhos) |
| $f(x)$ | estimativa de $f^*(x)$. É igual a $g(x) + h(x)$ |
| $g(x)$ | estimativa de $g^*(x)$ |
| $h(x)$ | estimativa de $h^*(x)$ |
| $f^*(x)$ | custo do menor caminho de $s$ até $t$ que passa por $x$. É igual a $g^*(x) + h^*(x)$ |
| $g^*(x)$ | $d^*(s, x)$ |
| $h^*(x)$ | $d^*(x, t)$ |
| | **Busca bidirecional** |

(A listagem continua na página seguinte)

[1] O objetivo das duas estruturas de dados é exatamente o mesmo: evitar a repetição da expansão de nós. No entanto, para preservar a notação original de alguns algoritmos de busca heurística bidirecional discutidos neste trabalho, emprega-se as duas formas.

| *Símbolo* | *Significado* |
|---|---|
| $open_p$ | estrutura de dados que armazena os nós candidatos a expansão do processo de busca $p$ |
| $\mathcal{L}$ | custo da melhor solução encontrada até o momento |
| $p$ | identificador do processo de busca. Os valores possíveis são 1 e 2 |
| $s_p$ | nó inicial para o processo de busca $p$ do caminho que será calculado no grafo a ser explorado. Os valores para $s_1$ e $s_2$ são $s$ e $t$ respectivamente |
| $t_p$ | nó final para o processo de busca $p$ do caminho que será calculado no grafo a ser explorado. Os valores para $t_1$ e $t_2$ são $t$ e $s$ respectivamente |
| $d_p(x, y)$ | peso da aresta do grafo explorado pelo processo de busca $p$ que conecta o nó $x$ ao nó $y$ |
| $d_p^*(x, y)$ | custo do menor caminho de $x$ até $y$ (nesse caso, não necessariamente dois nós vizinhos) no grafo explorado pelo processo de busca $p$ |
| $f_p(x)$ | estimativa de $f_p^*(x)$. É igual a $g_p(x) + h_p(x)$ |
| $g_p(x)$ | estimativa de $g_p^*(x)$ |
| $h_p(x)$ | estimativa de $h_p^*(x)$ |
| $f_p^*(x)$ | custo do menor caminho de $s_p$ até $t_p$ que passa por $x$. É igual a $g_p^*(x) + h_p^*(x)$ |
| $g_p^*(x)$ | $d_p^*(s_p, x)$ |
| $h_p^*(x)$ | $d_p^*(x, t_p)$ |
| $F_p^*$ | $\min\{f_p(x) \mid x$ é um candidato a expansão no processo de busca $p\}$ |
| $F_p$ | limite inferior para $F_p^*$ |
| | **Busca paralela** |
| $\sigma(a)$ | número de *nblocks* sucessores de $a$ que estão sendo utilizados por algum processador (de modo direto ou indireto) |
| | **Busca bidirecional paralela** |
| $\sigma_p(a)$ | mesmo que $\sigma(a)$ mas restrito ao contexto do processo de busca $p$ |

# Sumário

**Resumo** vii

**Abstract** ix

**Lista de Figuras** xi

**Lista de Tabelas** xiii

**Lista de Algoritmos** xv

**Lista de Acrônimos** xvii

**Lista de Conceitos e Definições** xix

**Lista de Notações e Símbolos** xxi

**1 Introdução** 1

1.1 Formulação e resolução de problemas de busca . . . . . . . . . . . . . 2

1.2 Objetivo e contribuições . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Organização deste documento . . . . . . . . . . . . . . . . . . . . . . 4

**2 Referencial teórico** 5

2.1 Aplicações da busca heurística . . . . . . . . . . . . . . . . . . . . . . 6

2.2 A* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.3 Classificação dos algoritmos baseados no A* . . . . . . . . . . . . . . 11

2.3.1 Classe bidirecional . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.3.2 Classe incremental . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.3.3 Classe *memory-concerned* . . . . . . . . . . . . . . . . . . . . . 23

2.3.4 Classe paralela . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

2.3.5 Classe *anytime* . . . . . . . . . . . . . . . . . . . . . . . . . . . 33

2.3.6 Classe tempo-real . . . . . . . . . . . . . . . . . . . . . . . . . . 38
2.3.7 Considerações finais . . . . . . . . . . . . . . . . . . . . . . . . . 41
2.4 Busca heurística bidirecional paralela . . . . . . . . . . . . . . . . . . . . 45

**3 PNBA* 49**
3.1 Descrição do algoritmo . . . . . . . . . . . . . . . . . . . . . . . . . . 49
3.2 Avaliação empírica . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
3.2.1 *Pathfinding* em *grids* . . . . . . . . . . . . . . . . . . . . . . . . 53
3.2.2 *n-puzzle* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
3.2.3 Metodologia . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
3.2.4 Resultados e discussão . . . . . . . . . . . . . . . . . . . . . . . . 59

**4 BPBNF 73**
4.1 Descrição do algoritmo . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.2 Avaliação empírica . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.2.1 Metodologia . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.2.2 Resultados e discussão . . . . . . . . . . . . . . . . . . . . . . . . 78

**5 Conclusão 105**
5.1 Trabalhos futuros . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107

**Referências Bibliográficas 109**

# Capítulo 1

# Introdução

A busca em um espaço de estados é um paradigma genérico para resolução de problemas amplamente utilizado em Ciência da Computação. É, freqüentemente, um dos componentes fundamentais de agentes inteligentes de resolução de problemas (do inglês, *problem solving agents*) empregados em diversos contextos. A busca em um espaço de estados, portanto, é um dos tópicos abordados pela área de Inteligência Artificial. Nesse cenário, é comum dividi-la em duas partes básicas: busca com adversários e busca com um único agente.

A busca com um único agente aborda problemas onde apenas um agente atua no ambiente. Nesses casos, assume-se um ambiente estático (não se modifica enquanto o planejamento é realizado), discreto, completamente observável (ou seja, o agente dispõe de acesso a uma descrição completa e fidedigna do ambiente) e determinístico (o próximo estado do ambiente depende apenas do estado atual e da ação realizada). Neste trabalho, as discussões dar-se-ão em torno dos algoritmos de busca com um único agente.

Com o intuito de focar a busca e evitar a exploração de estados desnecessários, alguns algoritmos empregam uma heurística: uma informação específica do domínio que os permitem estimar custos. O A* é um dos algoritmos de busca heurística mais conhecidos em Inteligência Artificial. Infelizmente, a diminuição significativa no esforço computacional da busca proporcionada pela adoção da heurística nem sempre é suficiente. No pior caso, o número de estados visitados pelo algoritmo A* varia exponencialmente em função do comprimento da solução ótima [Pearl, 1984]. Como conseqüência, tanto a complexidade de tempo quanto a de espaço são exponenciais.

A utilização da busca heurística para um único agente é possível em diversas áreas da Ciência da Computação. Há trabalhos descrevendo essas aplicações nas

áreas de Banco de Dados, Jogos Digitais, Robótica, Pesquisa Operacional e Bioinformática. Na maior parcela desses casos, porém, a elevada demanda computacional oferece dificuldades e/ou limita a sua adoção nesses contextos.

## 1.1 Formulação e resolução de problemas de busca

Diversas aplicações possuem um conjunto de características comuns que possibilita a sua representação como um problema de busca. Essas características são [Nilsson, 1998; Russell & Norvig, 2003]: um estado inicial, um estado final, um conjunto de operadores que modificam (transformam) os estados (também chamado de função sucessora) e uma função de custo opcional que provê o custo de produzir um estado a partir de outro (de acordo com a regra de transformação aplicada). A aplicação da função sucessora a partir do estado inicial define o espaço de estados [Korf, 1996]. Seu tamanho (número de estados) é drasticamente afetado pelas características do problema e pode até ser infinito.

A execução da busca consiste em sistematicamente aplicar os operadores até encontrar a meta. Ou seja, a intenção é transformar o estado inicial no estado final aplicando as regras disponíveis. Normalmente, o interesse está nos passos utilizados para encontrar o estado final (ou seja, na seqüência de operadores aplicados) que constitui um planejamento em certos contextos. Porém, a importância também pode ser uma prova de que tal transformação é impossível, ou seja, que não há solução. A função de custo proporciona uma métrica para a solução. Portanto, é possível impor restrições e avaliar sua qualidade. Por exemplo, o objetivo pode ser encontrar a seqüência de passos com o menor custo.

A estratégia utilizada para explorar o espaço de estados - a forma como os estados são tratados - levam a diferentes algoritmos. É possível diferenciá-los pela utilização do conhecimento específico do domínio. Isso permite a sua classificação em duas categorias básicas. Aqueles que não aplicam esse tipo de informação são conhecidos como estratégias de busca sem informação, busca cega ou de força bruta. Exemplos de algoritmos que pertencem a essa categoria são: busca em largura, busca em profundidade, busca uniforme (também conhecido como algoritmo de Dijkstra) e busca bidirecional. O segundo tipo de estratégia de busca é conhecida como busca com informação ou busca heurística. Como foi dito, esse tipo de algoritmo assume a presença de um conhecimento específico do problema. Essa informação extra permite que os algoritmos encontrem a solução mais eficientemente (ou seja, permite focar a busca e evitar a exploração de estados desnecessários).

O A* [Hart et al., 1968] é um dos algoritmos de busca heurística mais estudados em Inteligência Artificial. Ele assume a existência de uma função de estimativa que expressa a heurística. Dado um estado, essa função é capaz de gerar uma estimativa do menor custo para atingir o estado objetivo a partir desse. Se essa função possui certas propriedades (detalhadas na seção 2.2), o algoritmo A* é admissível - garantidamente encontra a solução ótima caso uma exista.

## 1.2 Objetivo e contribuições

A comunidade científica, com o intuito de encontrar formas de lidar melhor com a complexidade computacional do algoritmo A*, tem despendido um grande esforço. Conseqüentemente, várias extensões do algoritmo A* têm sido propostas. No contexto deste trabalho, destacam-se duas idéias: a paralelização e o uso do paradigma bidirecional.

A adequação do algoritmo A* para execução em ambientes paralelos está ganhando notoriedade com a popularização das máquinas *multicore* e dos *clusters* de computadores. O desafio desses algoritmos é reduzir a contenção (para permitir os maiores períodos de computação possível sem sincronização), mantendo a admissibilidade do algoritmo. O emprego do paradigma bidirecional no A* é uma alternativa viável porque reduz o esforço necessário à computação e oferece algumas oportunidades de paralelização.

As causas da redução do esforço computacional em cada uma dessas estratégias são distintas. É natural, pois, que surja o seguinte questionamento: elas podem ser combinadas para reduzir ainda mais o tempo de execução do A* sem violar a admissibilidade? O presente trabalho responde afirmativamente essa questão ao abordar implementações bidirecionais paralelas do algoritmo A* para ambientes computacionais de memória compartilhada. O objetivo central foi investigar formas de melhorar o desempenho do A* através de abordagens bidirecionais e paralelas para propor novos algoritmos.

Este trabalho destaca-se pelas seguintes contribuições:

- **Uma classificação dos algoritmos inspirados no A***: propõe uma forma de organizar os principais algoritmos de busca baseados no A* que foram propostos na literatura. Chamados de extensões do A* no contexto deste trabalho, eles são classificados de acordo com suas características centrais e com os objetivos motivadores de suas criações;

- ***Parallel New Bidirectional A**** **(PNBA*)**: é um algoritmo de busca heurística bidirecional paralela introduzido nesta dissertação de mestrado. É fruto da paralelização de um algoritmo de busca heurística bidirecional. Se comparado a versão original e ao A*, reduz significativamente o tempo de execução sem comprometer a admissibilidade e a simplicidade da implementação;

- ***Bidirectional Parallel Best-NBlock-First*** **(BPBNF)**: é também um algoritmo de busca heurística proposto neste trabalho. Aplica as idéias da busca bidirecional a um algoritmo de busca heurística paralelo já existente. Assim, permite a utilização de mais de dois processadores e, conseqüentemente, diminui ainda mais o tempo de execução na maioria dos domínios avaliados.

## 1.3 Organização deste documento

No início deste documento encontram-se várias listagens. Duas são especialmente úteis para o leitor: a lista de conceitos e definições e a lista de notações e símbolos. A primeira trata de alguns conceitos no contexto de algoritmos de busca heurística indispensáveis para compreensão deste texto. A outra resume as principais notações empregadas ao longo deste trabalho.

O restante deste documento está estruturado da seguinte forma: o capítulo seguinte trata dos trabalhos relacionados, cobrindo os conceitos utilizados ao longo do texto. Primeiramente, o algoritmo A* é explicado. Uma classificação dos algoritmos de busca heurística nele inspirados é proposta em seguida. Ao final, alguns trabalhos com algoritmos de busca heurística bidirecional paralela são expostos. Nos capítulos 3 e 4 são descritos, respectivamente, os algoritmos PNBA* e BPBNF. Os experimentos realizados com esses (e outros) algoritmos e os resultados obtidos são também discutidos. Por fim, o capítulo 5 apresenta as conclusões e possibilidades de trabalhos futuros.

# Capítulo 2

# Referencial teórico

Neste capítulo, apresenta-se os principais trabalhos relacionados. Primeiramente, a seção 2.1 exemplifica a aplicação da busca heurística com um único agente em vários contextos. Explica-se, em seguida, o algoritmo A* revisando suas propriedades chaves. A seção 2.3 introduz uma classificação dos mais relevantes algoritmos de busca baseados no A* e discute exemplos de algoritmos pertencentes a cada uma das classes propostas. Por fim, estratégias de busca heurística bidirecionais e paralelas são descritas.

No restante deste trabalho, será adotado um conceito matemático formal na interpretação do espaço de estados. É útil concebê-lo como um grafo, onde os estados são nós e uma aresta $(x, y)$ conectando os estados $x$ e $y$ indica que a aplicação de um determinado operador em $x$ gerará $y$ ($y$ é um sucessor de $x$). As arestas podem ser rotuladas com o custo/peso associado à operação. O peso de uma aresta será denotado por $d(x, y)$. O custo do menor caminho de $x$ até $y$ (nesse caso, não necessariamente dois nós vizinhos) será representado por $d^*(x, y)$. Dessa forma, encontrar a solução ótima (aquela com o menor custo) é equivalente a computar o menor caminho entre o nó inicial e o nó final - a partir de agora, denotados por $s$ e $t$ respectivamente.

Na literatura da área de Inteligência Artificial, e também neste documento, classifica-se um algoritmo de busca de acordo com as garantias oferecidas por ele na resposta calculada. Se um algoritmo sempre encontra uma solução (quando algum caminho entre $s$ e $t$ existe), é dito ser *completo*. Por outro lado, se o algoritmo assegura o cômputo da solução ótima (caso haja uma solução), ele é classificado como *admissível*.

## 2.1 Aplicações da busca heurística

A utilização da busca heurística com um único agente ocorre em diversas áreas da Ciência da Computação. Uma delas é na área de Bancos de Dados. A busca em um espaço de estados com um único agente foi aplicada ao problema de otimização de consultas em sistemas de bancos de dados distribuídos [Yoo & Lafortune, 1989]. A justificativa para realização dessa otimização é a redução da quantidade de informação que precisa ser transmitida para computar o resultado de uma consulta. A estratégia empregada reduz o número de relacionamentos referenciados fazendo um processamento prévio. Uma das questões centrais dessa estratégia é a seqüência em que as operações parciais dessa fase serão aplicadas. A ordem da utilização dos operadores no estágio de pré-processamento afeta significativamente o número de relacionamentos considerados durante o cálculo final. Determinar o arranjo ótimo é um problema $\mathcal{NP}$-Difícil. Através da técnica de busca, porém, foi possível calcular a seqüência ótima com um esforço computacional aceitável para a aplicação.

A programação de personagens não jogáveis (NPCs) em Jogos Digitais é também um exemplo do uso desse paradigma. No contexto em questão, os NPCs devem apresentar comportamentos sofisticados e inteligentes. A busca em um espaço de estados é uma das alternativas adotadas para implementá-los. É comum a discussão de algoritmos de busca, principalmente do A*, em livros voltados para o desenvolvimento desse tipo de software como Bourg & Seemann [2004] e Millington [2006]. As discussões focam predominantemente no emprego dessa técnica para concepção dos movimentos de um NPC por um cenário com obstáculos, problema presente em vários estilos de jogos. No entanto, as possibilidades não restringem-se a esse aspecto. Rios & Chaimowicz [2009] empregaram um algoritmo de busca para projetar ferrovias em um jogo de construção e gerência de rotas de transporte. Já Orkin [2003] aborda a elaboração de planos de atuação compostos por ações atômicas definidas previamente. Um algoritmo de busca é responsável por encontrar uma seqüência de ações capaz de satisfazer os objetivos do NPC naquele momento.

Robôs móveis autônomos também utilizam algoritmos de busca heurística no planejamento de caminhos. Tipicamente, possuem uma representação do ambiente (fornecida ou criada através de seus sensores) utilizada para localizarem-se e, então, planejarem suas trajetórias de acordo com os objetivos designados a eles. Uma das formas de computar a trajetória com auxílio desse mapa é através de uma busca [Choset et al., 2005]. Em particular, Likhachev et al. [2005] empregaram essa metodologia para navegação de robôs móveis autônomos em ambientes externos. Entretanto, as aplicações desse paradigma não estão restritas aos robôs móveis.

Ainda no contexto da Robótica, é possível aplicar a busca em um espaço de estados para computar a seqüência de movimentos de um manipulador [Hourtash & Tarokh, 2001; Berenson et al., 2009].

A Pesquisa Operacional emprega a busca como uma técnica genérica para resolução de problemas. Normalmente nesse contexto, a modelagem de um problema através de uma busca em um espaço de estados é conhecida como formulação através do problema do caminho mais curto [Hillier et al., 2004]. Um exemplo é o cálculo da menor rota em mapas rodoviários reais de larga escala [Klunder & Post, 2006]. A importância dessa aplicação relaciona-se, principalmente, com a popularização dos sistemas para navegação de veículos através do Sistema de Posicionamento Global (GPS) e de serviços *on-line* que oferecem gratuitamente acesso a mapas.

Outro exemplo da utilização dessa técnica pode ser encontrado na área da Bioinformática, mais especificamente na Biologia Molecular. Nesse contexto, o alinhamento de múltiplas seqüências de ácido desoxirribonucleico (DNA), ácido ribonucleico (RNA) ou proteínas possui uma grande importância. Ele é fundamental para realização de várias análises e comparações. Seu objetivo é a extração de um padrão comum. Uma das possíveis soluções desse problema envolve formulá-lo como uma busca em um espaço de estados [Yoshizumi et al., 2000; Hansen & Zhou, 2007]. Apesar da dificuldade computacional do problema ($\mathcal{NP}$-Difícil [Wang & Jiang, 1994]), essa abordagem apresenta resultados satisfatórios.

## 2.2 A*

O A* (pronuncia-se *A-estrela*) [Hart et al., 1968] é um dos algoritmo de busca heurística mais conhecidos em Inteligência Artificial e provavelmente um dos mais estudados. É um algoritmo de busca heurística do tipo *best-first*, ou seja, a cada iteração, ele expande o melhor nó disponível usando a métrica denotada por $f(x)$, onde $x$ é um nó do grafo. No caso do A*, a métrica inclui o custo real de chegar até aquele nó e o custo estimado de chegar à meta partindo dele.

Como mencionado, algoritmos de busca com informação empregam uma heurística que provê um conhecimento extra do problema ao algoritmo. Para um nó $x$, $h(x)$ expressa a heurística, que é uma estimativa de $h^*(x) = d^*(x,t)$. Na maioria dos casos, $h^*(x)$ não está disponível *a priori*. O custo do menor caminho encontrado até o momento, $g(x)$, é uma estimativa de $g^*(x) = d^*(s,x)$. Não é necessariamente igual a $g^*(x)$ porque pode haver mais de um caminho de $s$ até $x$ e o algoritmo ainda não considerou todos eles até aquele instante da computação. O custo do menor

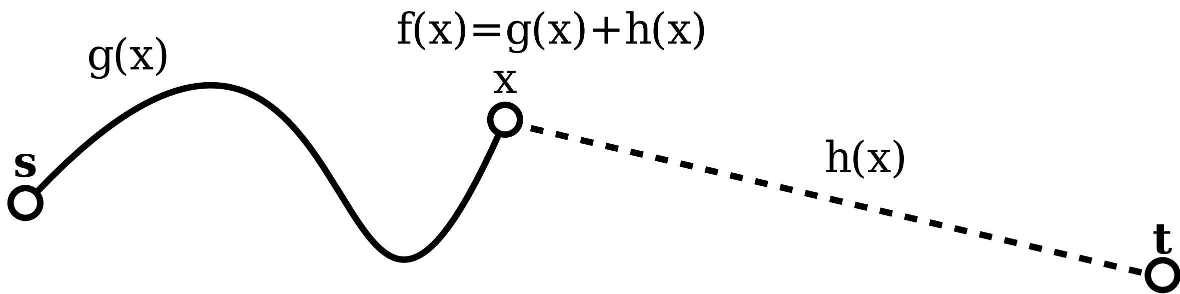



Figura 2.1: Representação gráfica dos conceitos de $f(x)$, $g(x)$, $h(x)$ e da relação entre esses atributos de um nó $x$ do grafo.

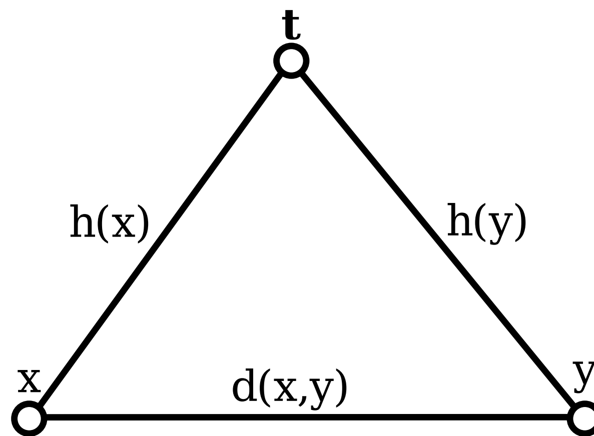



Figura 2.2: Heurística consistente interpretada como uma desigualdade triangular.

caminho que conecta $s$ a $t$ e passa por $x$ é representado por $f^*(x) = g^*(x) + h^*(x)$. O atributo $f(x) = g(x) + h(x)$ é uma estimativa desse custo e é utilizada pelo algoritmo A* na decisão de qual será o próximo nó a ser expandido. A figura 2.1 reproduz graficamente a idéia que sustenta a definição de alguns desses atributos. Emprega-se os termos *f-value*, *g-value* e *h-value* para denotar esses atributos.

No contexto do A*, a heurística pode ter duas propriedades importantes. Se $h(x) \leq h^*(x)$ para todos nós $x$ (ou seja, a heurística nunca superestima o custo real) ela é classificada como *admissível*. Uma heurística é considerada *consistente* ou *monotônica* se $h(x) \leq d^*(x,y) + h(y)$ para todos nós $x$ e $y$ (ou $h(x) \leq d(x,y) + h(y)$ para qualquer aresta $(x,y)$). Como sugerido na figura 2.2, essa restrição pode ser interpretada como um tipo de desigualdade triangular: cada lado do triângulo não pode ser maior que a soma dos demais. Toda heurística consistente é também admissível [Russell & Norvig, 2003]. A importância desses conceitos será tratada a seguir.

O A* é completo se as seguintes condições são satisfeitas [Pearl, 1984]: os nós

devem ter um número finito de sucessores e o peso associado as arestas deve ser positivo. É também admissível se uma terceira condição é assegurada: a admissibilidade da heurística. A imposição de uma restrição mais rigorosa (consistência) à heurística permite a utilização de uma versão do A* mais simples e eficiente onde um nó é expandido no máximo uma vez.

O algoritmo 2.1 apresenta uma versão do A* que assume o uso de uma heurística consistente. Duas estruturas de dados são utilizadas durante a busca para gerenciar a expansão dos nós: *open* e *closed*.

A primeira contém todos nós da fronteira da busca (candidatos a expansão). Assim, *open* auxilia o algoritmo durante a seleção do nó mais promissor para expansão utilizando os $f$*-values*, no caso do A*, como seu critério de ordenação. É comumente representada através de um *heap* binário.

A função da estrutura de dados *closed* é armazenar os nós já expandidos. Geralmente implementada com a assistência de uma tabela *hash*, é empregada para evitar a expansão do mesmo vértice várias vezes.

```
1: g(s) ← 0
2: f(s) ← g(s) + h(s)
3: parent(s) ← NULL
4: open ← {s}
5: closed ← ∅
6: while open ≠ ∅ do
7:    x ← argmin{f(n) | n ∈ open}
8:    if x = t then
9:       return solução ótima encontrada
10:   open ← open\{x}
11:   closed ← closed ∪ {x}
12:   for all arestas (x, y) do grafo do
13:      if y ∉ closed then
14:         if y ∉ open then
15:            g(y) ← g(x) + d(x, y)
16:            f(y) ← g(y) + h(y)
17:            parent(y) ← x
18:            open ← open ∪ {y}
19:         else if g(x) + d(x, y) < g(y) then
20:            g(y) ← g(x) + d(x, y)
21:            f(y) ← g(y) + h(y)
22:            parent(y) ← x
23: return não existe solução
```

Algoritmo 2.1: O algoritmo A* para uma heurística consistente.

Antes de começar a execução do laço responsável por expandir os nós, $g(s)$ é

iniciado com zero e parent($s$) com nulo. Quando o A* é utilizado para a solução de problemas onde é relevante conhecer as transformações que levaram à solução; é comum manter um apontador de cada nó para seu pai - o nó expandido que o alcançou com menor custo. Essa é a função do atributo *parent* na versão do algoritmo apresentada. Portanto, nesses casos o A* mantém uma árvore de busca explícita com todos os nós gerados, um subconjunto (que espera-se ser pequeno) do espaço de estados. Após as atualizações dos atributos de $s$, ele é incluído em *open* e inicia-se a expansão dos nós.

O laço principal é executado enquanto houver nós em *open* e o nó $t$ não for removido dela. A cada iteração, o nó com o menor $f$-*value* é removido dessa estrutura de dados para expansão e incluído em *closed*. Seus sucessores são gerados e aqueles que ainda não foram expandidos (não estão em *closed*) são inseridos em *open*. É possível que um ou mais deles já estejam lá. Nesse caso, um caminho alternativo partindo de $s$ foi encontrado. Se o custo do novo caminho é menor, o *g-value* será atualizado.

Existem dois critérios para o término da computação. Se não existem mais nós a serem expandidos (*open* está vazia), não há uma solução. Por outro lado, se $g(t)$ é selecionado para expansão, a solução ótima foi encontrada (se a heurística é pelo menos admissível). Uma maneira de recuperar os nós que compõe o caminho ótimo é iterar na lista encadeada criada através dos apontadores (atributo *parent* de cada nó). A presença do valor nulo durante essa operação representa o início do caminho, ou seja, a descoberta do nó $s$.

Uma das conseqüências da adoção de uma heurística consistente é que um nó $x$ só é expandido se o custo do menor caminho de $s$ até $x$ já foi encontrado [Nilsson, 1998]. Garante-se, assim, a expansão de um nó no máximo uma vez. Outro importante efeito é a monotonicidade não decrescente da seqüência de $f$-*values* dos nós expandidos. Essa característica dos $f$-*values* é desejável para implementação de versões paralelas do algoritmo discutidas adiante.

A principal desvantagem desse algoritmo é que o número de nós expandidos pode variar exponencialmente com o comprimento do caminho ótimo [Pearl, 1984]. Ou seja, tanto a complexidade de tempo quanto a de espaço são exponenciais em função do número de arestas do caminho ótimo. Cabe ressaltar que a base dessa função exponencial está diretamente relacionada com a função sucessora do problema.

## 2.3 Classificação dos algoritmos baseados no A*

Desde a proposição do A*, vários algoritmos de busca semelhantes a ele foram apresentados pela comunidade científica com o intuito de torná-lo mais adequado ao contexto de certas aplicações e/ou de atenuar os problemas causados por sua grande demanda computacional. Uma possível taxonomia desses algoritmos, chamados aqui de extensões do A*, é discutida nesta seção. Ela fundamenta-se nas particularidades e nos objetivos motivadores da criação dos algoritmos. Uma versão preliminar dessa classificação foi apresentada em Rios & Chaimowicz [2010].

Outras organizações também são possíveis. Edelkamp & Schrödl [2012] publicaram um trabalho durante a realização deste trabalho onde separam os algoritmos de busca heurística de duas formas distintas: busca heurística com restrições de tempo e busca heurística com restrições de memória. Cada uma delas possui vários subtipos.

Na classificação a ser apresentada, as extensões do A* são categorizadas através de seis classes: bidirecional, incremental, *memory-concerned*, paralela, *anytime* e tempo-real. É importante ressaltar que elas não são excludentes entre si. Ou seja, um algoritmo pode pertencer a mais de uma delas. Ao longo da discussão, para cada classe, examina-se as suas principais características, as aplicações (nos casos onde o uso é mais restrito) e detalha-se alguns dos seus algoritmos mais representativos.

### 2.3.1 Classe bidirecional

Algoritmos desta classe dividem a exploração do espaço de estados em duas buscas com o objetivo de reduzir a demanda computacional do A*. Uma busca parte de $s$ e a outra de $t$. A motivação principal é redução do número de estados expandidos como ilustrado na figura 2.3. Esse almejado resultado já foi obtido em outros algoritmos (por exemplo, na busca em largura) através da utilização da abordagem bidirecional. Com a diminuição do número de nós expandidos, há, conseqüentemente, uma redução no tempo e no espaço necessários à computação.

Mais formalmente, suponha que $b$ denote o fator de ramificação (*branching factor*) - o número máximo de sucessores de um nó. Seja $d$ o comprimento da solução ótima. A complexidade de tempo e espaço da busca em largura é $O(b^d)$ [Russell & Norvig, 2003]. Quando emprega-se a estratégia bidirecional, tem-se duas árvores de busca cada uma com um número de nós que é $O(b^{\frac{d}{2}})$. A complexidade para ambas grandezas, então, será $O(b^{\frac{d}{2}}) + O(b^{\frac{d}{2}}) = O(b^{\frac{d}{2}})$ que, apesar de ainda exponencial, é muito menor do que $O(b^d)$ [Russell & Norvig, 2003]. O desejo, por conseguinte, é,

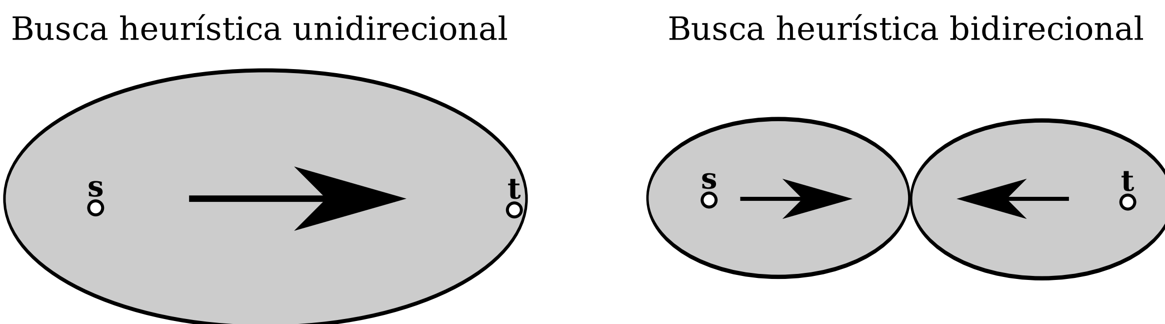



Figura 2.3: Comparação do espaço de estados visitado na busca heurística bidirecional e unidirecional. O espaço de estados explorado é representado pelas áreas das formas geométricas. A figura representa a situação almejada pelos algoritmos desta classe: o espaço de estados visitado por duas buscas é menor do que o espaço de estados visitado por uma única busca.

com o uso dessa abordagem, conseguir também uma redução nos recursos computacionais exigidos pelo A* para realização de buscas.

Nem todo domínio pode ser explorado por algoritmos de busca bidirecional. É preciso que a única meta seja explícita[1] e que seja possível determinar os predecessores dos estados dada uma ação [Russell & Norvig, 2003]. No caso de domínios com ações reversíveis, a última condição é sempre satisfeita. A heurística deve também conseguir estimar o custo do caminho ótimo de um nó qualquer até $s$ e de $t$ até um nó arbitrário.

Com base no trabalho de Kaindl & Kainz [1997], propõe-se a divisão dos algoritmos de busca heurística bidirecional em duas subclasses/subtipos. A primeira realiza as duas buscas simultaneamente[2], enquanto no outro subtipo as buscas são realizadas em seqüência.

Nesta classe, particularmente, será apresentada uma pequena história e evolução dos algoritmos pertencentes ao primeiro subtipo, que agrupa a grande maioria do total. Inicialmente, a comunidade científica deparou-se com grandes obstáculos para conseguir superar o desempenho do A* através da utilização de versões de algoritmos de busca heurística bidirecional. O texto a seguir fundamenta-se na discussão presente no trabalho de Kaindl & Kainz [1997], uma boa referência para abordagens bidirecionais do A* mais antigas.

[1]Se existem várias metas enumeráveis e em uma quantidade finita, ainda sim é praticável a busca bidirecional. Basta criar um nó postiço e fazer com que a expansão dessas metas gere esse nó que passa a ser o único alvo.

[2]O termo "simultaneamente" não deve ser confundido nesse contexto com "paralelamente". Ele denota apenas que as buscas não são seqüenciais, ou seja, há um intercalamento entre suas execuções. Normalmente, para alcançar o efeito desejado, alterna-se entre a execução das buscas.

Os primeiros trabalhos a tratar a busca heurística bidirecional não apresentaram bons resultados (ganhos no desempenho quando comparados às respectivas versões unidirecionais). De modo geral, a computação por eles efetuada pode ser divida em duas fases/etapas. O princípio da primeira coincide com o início da computação. Ela termina quando as duas fronteiras de busca se encontram, ou seja, no momento em que a primeira solução é obtida. Com o encerramento dessa etapa, começa a segunda fase. O seu término coincide com o fim da execução do algoritmo.

Uma espécie de consenso, fundamentado por esses resultados ruins, foi estabelecido: as fronteiras das buscas passavam uma pela outra sem se encontrarem próximo ao meio. Essa hipótese, ilustrada através da metáfora dos mísseis [Pohl, 1969], seria mais tarde rebatida por Kaindl & Kainz [1997]. Na realidade, a maior parte do esforço é realizado após o término da primeira fase cujo encerramento é marcado pelo encontro das duas fronteiras de busca. Ou seja, uma parcela predominante do tempo é empregado para garantir a otimalidade da solução na segunda fase do processamento.

Essa crença motivou a criação dos algoritmos *front-to-front*, onde cada processo de busca "mira", através da heurística, na fronteira do outro processo de busca ao invés de almejar sua raiz. Ou seja, a idéia opunha-se as abordagens anteriores denominadas *front-to-end*. O *Bidirectional Heuristic Front-to-Front Algorithm* (BHFFA) [de Champeaux & Sint, 1977], para atingir esse comportamento, calcula a heurística de um nó considerando informações de todos os nós da fronteira do processo de busca oposta. A equação leva em conta o custo estimado até um nó daquela fronteira mais o custo já calculado desse nó até a raiz da árvore de busca oposta. Politowski & Pohl [1984] apresentaram uma técnica onde o processo de busca de tempos em tempos muda o seu "alvo" com base no nó mais promissor da fronteira do processo de busca oposto. Essas estratégias foram de fato capazes de reduzir o número de expansões com relação aos algoritmos *front-to-end*, pois a heurística que utilizam é mais exata (já que o "alvo" está mais próximo). Entretanto, resultavam em algoritmos computacionalmente caros (porque o custo médio da expansão de um nó aumenta muito) ou em algoritmos que não garantem a qualidade da solução [Kaindl & Kainz, 1997].

Ikeda et al. [1994] propuseram uma versão bidirecional do A* (*front-to-end*) utilizando uma nova abordagem que "traduz" o A* no algoritmo de Dijkstra [Dijkstra, 1959]. A versão bidirecional de Dijkstra apresenta uma redução expressiva no esforço computacional (causada pela diminuição do número de expansões necessárias) quando comparada a sua versão unidirecional sendo, portanto, a motivação dessa tentativa. Klunder & Post [2006] realizaram uma minuciosa comparação em-

pírica no domínio de mapas rodoviários de larga escala e incluíram esse algoritmo em seu estudo. Os resultados mostram claramente uma redução no tempo de execução com relação ao A* unidirecional.

Mais recentemente, esses resultados inspiraram a elaboração do algoritmo *New Bidirectional A** (NBA*) [Pijls & Post, 2010, 2009]. Ele é a base para os dois algoritmos de busca heurística bidirecional paralela propostas neste trabalho e será descrito na subseção 2.3.1.1.

Os algoritmos do segundo subtipo de busca heurística bidirecional também foram apresentados com o intuito de superar as dificuldades encontradas nos primeiros algoritmos do subtipo exposto anteriormente. A diferença principal está na forma como os dois processos de busca são conduzidas. Ao invés de intercalá-los, sua execução ocorre seqüencialmente. Dillenburg & Nelson [1994] propõe uma forma genérica, denominada *Perimeter Search*, fundamentada nessas idéias para tornar bidirecional os algoritmos de busca. Em particular, quando aplicada ao A*, dá origem ao PS* que será agora detalhado.

Primeiramente, uma busca em largura com profundidade limitada é conduzida a partir de $t$ para gerar o perímetro. A segunda busca (que parte de $s$ e é realizada pelo algoritmo A*) computa a heurística com relação a um nó $r$ da fronteira do perímetro e adiciona a ela o custo (obtida através da primeira busca) do menor caminho de $r$ até $t$. Se $x$ é o nó a ser expandido, a heurística é computada utilizando o $r$ que minimiza a soma do custo estimado do caminho ótimo de $x$ até $r$ e do custo do menor caminho de $r$ até $t$. O PS* obteve um bom desempenho com relação ao A* no domínio avaliado.

Kaindl & Kainz [1997] introduzem algumas formas de aprimorar dinamicamente a heurística para estratégias de busca semelhantes ao *Perimeter Search*. A idéia é melhorar o aproveitamento da informação adquirida a partir da primeira busca realizada. Uma delas é o método intitulado *Add*. Suponha que a primeira busca tenha sido realizada (através do A*) partindo de $t$ e que a heurística empregada era consistente. Para cada um dos nós na fronteira de *closed* (conjunto de predecessores imediatos dos nós de *open*) computada pela primeira busca; calcula a diferença do custo do menor caminho de $t$ até o nó e o custo estimado pela heurística (da segunda busca) do menor caminho do nó até $t$. Note que a diferença representa o erro da estimativa provida pela heurística. A menor diferença encontrada é adicionada a heurística de um nó (não contemplado pela primeira busca) para criar uma estimativa, também consistente, mais próxima do valor real. Ela é utilizada na segunda busca. Os resultados dos experimentos realizados em diferentes domínios com esse método e outros também discutidos mostraram uma redução significativa

no tempo de execução e número expansões para alguns dos algoritmos aos quais foram aplicados.

#### 2.3.1.1 NBA*

O *New Bidirectional A** (NBA*) [Pijls & Post, 2010, 2009] é um algoritmo de busca heurística bidirecional baseado no A*. Diferentemente do A*, para encontrar a solução ótima é necessário que a função heurística seja consistente (e não apenas admissível). Essa é também a sua principal diferença para abordagens bidirecionais nas quais baseou-se. Ao contrário desses algoritmos, o NBA* é mais genérico e sua condição para uma segunda fase de processamento reduzida é apenas a adoção de uma heurística consistente.

Como qualquer algoritmo de busca heurística bidirecional do primeiro subtipo apresentado, dois processos de busca são conduzidos simultaneamente. Uma técnica comum para efetuar essa execução simultânea nesse contexto é alternar a execução de cada processo de busca. Essa abordagem foi empregada pelos autores do NBA*. O processo de busca número 1 opera no grafo original, enquanto o processo número 2 lida com o grafo inverso (cada aresta original $(x, y)$ é substituída por uma aresta $(y, x)$ com o mesmo peso). Além disso, no segundo processo de busca, o nó inicial original ($s$) torna-se a meta e a meta original ($t$) passa a ser o nó inicial. Utiliza-se a notação $s_1$, $t_1$, $s_2$ e $t_2$ para o nó inicial e nó final de cada um dos processos de busca.

Antes de iniciar a explicação do algoritmo NBA*, a notação empregada até o momento será estendida. Seja $d_p(x, y)$ e $d_p^*(x, y)$, respectivamente, o mesmo que $d(x, y)$ e $d^*(x, y)$, mas restrito ao processo de busca $p \in \{1, 2\}$. O subscrito será adicionado também a outros elementos resultando em $g_p(x)$, $h_p(x)$ e $f_p(x) = g_p(x) + h_p(x)$, estimativas de $g_p^*(x) = d_p^*(s_p, x)$, $h_p^*(x) = d_p^*(x, t_p)$ e $f_p^*(x) = g_p^*(x) + h_p^*(x)$ respectivamente. Portanto, a semântica é equivalente, mas está restrita ao processo de busca $p$.

Como o A*, o NBA* utiliza uma estrutura de dados para controlar a expansão de nós. Utilizando a notação original do algoritmo, $\mathcal{M}$ contém todos os nós que estão no "meio", *i.e.*, entre as duas fronteiras de busca. Inicialmente, todos os nós estão em $\mathcal{M}$ e têm seus $g_p$*-values* iguais a $\infty$.

Além de $\mathcal{M}$, $\mathcal{L}$ é uma variável compartilhada, lida e escrita pelos dois processos de busca. Ela contém o custo da melhor solução encontrada até então pelo algoritmo e é inicializada com o valor infinito ($\mathcal{L} \leftarrow \infty$). $\mathcal{L}$ é empregada no critério de poda juntamente com $F_p$ (um limite inferior de $F_p^*$, o menor $f_p$*-value* na fronteira

do processo de busca $p$). As variáveis $g_p$ e $F_p$ são escritas somente por um processo, mas lidas pelos dois.

Os nós das fronteiras dos processos de busca são armazenados nas estruturas de dados $open_1$ e $open_2$. Antes que um nó possa ser inserido em alguma dessas estruturas de dados, ele deve se tornar *labeled* ($x$ é um nó *labeled* se $g_1(x)$ ou $g_2(x)$ é finito). Um vértice é adicionado à estrutura de dados que representa a fronteira do processo de busca que o gerou. Por questão de conveniência, para simplificar o código, um nó cujo $g_p(x)$-*value* já finito é reduzido também é incluído na fronteira (mesmo já estando lá). Em ambos os casos, o nó ainda deve estar em $\mathcal{M}$ (ou seja, não ter sido expandido).

O algoritmo 2.2[3] apresenta uma versão do NBA*[4] que assume o uso de uma heurística consistente. A primeira parte corresponde ao código necessário para inicializar as variáveis e estruturas do algoritmo. O outro trecho apresenta as ações de um turno do processo de busca 1. As atividades do turno do processo de busca 2 são semelhantes. Para obtê-las basta substituir adequadamente os subscritos (de 1 para 2 e vice-versa). Durante a execução, alterna-se os turnos correspondentes a cada um dos processo de busca.

A cada turno do processo de busca 1, o nó $x \in open_1$ com o menor $f_1$-*value* finito é selecionado para expansão. Se ainda pertence a $\mathcal{M}$, é removido de $\mathcal{M}$ e podado (não expandido) se $f_1(x) \geq \mathcal{L}$ [5] ou $g_1(x) + F_2 - h_2(x) \geq \mathcal{L}$. Nesse caso, $x$ é classificado como *rejected*. A seqüência de $f_1$-*values* e $f_2$-*values* é monotonicamente não decrescente, pois a heurística é consistente. Portanto, é seguro podar um nó cujo $f_1$-*value* seja maior do que $\mathcal{L}$. O segundo critério para poda é também um limite inferior para o custo do caminho de $s_1$ até $t_1$ que passa $x$. A diferença é o uso de informações do processo de busca 2. O resto da subtração $F_2 - h_2(x)$ é o menor valor possível de $g_2(x)$ ao longo da computação (lembre-se de que $x$ está em $\mathcal{M}$ neste momento). Pijls & Post [2010, 2009] apresentam as provas de que essas

[3] Alguns algoritmos apresentados neste trabalho (o 2.2, por exemplo) realizam comparações do tipo $v = \infty$ e $v \neq \infty$. Ao realizar tais comparações, assume-se que o infinito possui representação própria através de um padrão específico; pois isso é realidade na maioria das linguagens de programação e arquiteturas computacionais. Portanto, é possível determinar se duas variáveis armazenam o padrão correspondente a $\infty$.

[4] O pseudocódigo do NBA* diferencia-se em alguns aspectos da versão apresentada pelos autores, pois é mais voltada para implementação. As principais diferenças são a presença explícita das estruturas de dados que representam as fronteiras de busca e da alternância (também explícita) dos turnos correspondentes a cada um dos processos de busca. Os conjuntos com os nós *rejected* e *stabilized* do algoritmo original não são exibidos por serem desnecessários na prática.

[5] A condição original é $f_1(x) - h_1(t_1) \geq \mathcal{L}$. Entretanto, como na maioria das heurísticas (incluindo aquelas que serão utilizadas nesse trabalho) $h_1(t_1) = h_2(t_2) = 0$, esse termo foi simplificado para $f_1(x) \geq \mathcal{L}$.

1: {Procedimento para iniciar variáveis do NBA*:}
2: $\mathcal{M} \leftarrow \emptyset$
3: $\mathcal{L} \leftarrow \infty$
4: **for all** nós $x$ do grafo que será explorado **do**
5: $\quad g_1(x) \leftarrow \infty$
6: $\quad g_2(x) \leftarrow \infty$
7: $\quad \mathcal{M} \leftarrow \mathcal{M} \cup \{x\}$
8: **for all** $p \in \{1, 2\}$ **do**
9: $\quad g_p(s_p) \leftarrow 0$ {prepara o nó inicial e a fronteira do processo de busca $p$}
10: $\quad f_p(s_p) \leftarrow g_p(s_p) + h_p(s_p)$
11: $\quad \text{open}_p \leftarrow \{s_p\}$
12: $\quad F_p \leftarrow f_p(s_p)$
13:
14: {Ações de um turno do processo de busca 1 do NBA*:}
15: $x \leftarrow \text{argmin}\{f_1(n) \mid n \in \text{open}_1\}$
16: $\text{open}_1 \leftarrow \text{open}_1 \setminus \{x\}$
17: **if** $x \in \mathcal{M}$ **then**
18: $\quad \mathcal{M} \leftarrow \mathcal{M} \setminus \{x\}$
19: $\quad$ **if** $\left(f_1(x) < \mathcal{L}\right) \wedge \left(g_1(x) + F_2 - h_2(x) < \mathcal{L}\right)$ **then**
20: $\quad\quad$ **for all** arestas $(x, y)$ do grafo explorado pelo processo de busca 1 **do**
21: $\quad\quad\quad$ **if** $\left(y \in \mathcal{M}\right) \wedge \left(g_1(y) > g_1(x) + d_1(x, y)\right)$ **then**
22: $\quad\quad\quad\quad g_1(y) \leftarrow g_1(x) + d_1(x, y)$
23: $\quad\quad\quad\quad f_1(y) \leftarrow g_1(y) + h_1(y)$
24: $\quad\quad\quad\quad \text{open}_1 \leftarrow \text{open}_1 \cup \{y\}$
25: $\quad\quad\quad\quad \text{parent}_1(x) \leftarrow y$
26: $\quad\quad\quad\quad$ **if** $g_1(y) + g_2(y) < \mathcal{L}$ **then**
27: $\quad\quad\quad\quad\quad \mathcal{L} \leftarrow g_1(y) + g_2(y)$
28: $\quad\quad\quad\quad\quad$ meeting_node $\leftarrow y$
29: **if** $\text{open}_1 \neq \emptyset$ **then**
30: $\quad F_1 \leftarrow \min\{f_1(n) \mid n \in \text{open}_1\}$
31: **else**
32: $\quad$ **if** $\mathcal{L} = \infty$ **then**
33: $\quad\quad$ **return** não existe solução
34: $\quad$ **else**
35: $\quad\quad$ **return** solução ótima encontrada

Algoritmo 2.2: O algoritmo NBA* para uma heurística consistente.

condições para podar nós não violam a admissibilidade do algoritmo.

Caso nenhum dos critérios para poda sejam atendidos, todos os sucessores $y$ de $x$ são gerados. Nesse caso, $x$ é classificado como *stabilized* porque $g_1(x)$ não mais será alterado e corresponde a $g_1^*(x)$. Para cada $y$, o algoritmo verifica se um caminho mais curto de $s_1$ até $y$ ou de $s_1$ até $t_1$ (através de $y$) foi encontrado para alterar o conteúdo de $g_1(y)$ ou de $\mathcal{L}$. Assim, para cada nó $y$ gerado é como se $g_1(y)$ e $\mathcal{L}$ fossem atualizados, respectivamente, com os seguintes valores: $\min\{g_1(y), g_1(x) + d_1(x,y)\}$ e $\min\{\mathcal{L}, g_1(y) + g_2(y)\}$.

Ao final do turno, $F_1$ é atualizada com o menor $f_1$*-value* da fronteira. A execução termina quando não há mais candidatos a expansão em um dos lados da busca. Nesse momento, $\mathcal{L}$ armazena o custo da solução ótima ou um valor infinito se não existe solução. A utilização da variável *meeting_node* e dos atributos $parent_1$ e $parent_2$ é uma das formas de recuperar todos os nós que formam o caminho mais curto de $s$ até $t$. O *meeting_node* é o nó componente no caminho de custo $\mathcal{L}$ que possui o $g_1$-*value* e $g_2$-*value* finitos. Ou seja, é um vértice no qual os dois processos de busca se encontraram. Já os atributos $parent_1$ e $parent_2$ possuem função semelhante a propriedade *parent* empregada pelo A*. Através deles, cada um dos processos de busca mantém uma árvore de busca explícita com todos os nós gerados.

### 2.3.2 Classe incremental

Em algumas aplicações da busca heurística em um espaço de estados, é possível que haja alterações nos componentes do problema de busca antes do consumo total do planejamento (supondo não ser instantânea essa última etapa). As modificações refletem-se no espaço de estados e podem invalidar a solução corrente tornando necessário um novo cálculo. A proposta da classe de algoritmos incremental é aproveitar a solução anterior no cálculo atual com o intuito de reduzir o tempo de execução. O objetivo central é, através da utilização de informações das execuções passadas, computar a resposta do problema atual mais rapidamente do que realizar uma busca completa do princípio [Koenig et al., 2004b].

A utilização de algoritmos pertencentes a classe incremental é conveniente em aplicações onde possivelmente será necessário realizar o cálculo da solução de vários problemas de busca semelhantes. Aplicações com essa característica tipicamente possuem ambientes dinâmicos e/ou parcialmente observáveis. Tais propriedades contrapõem a assunção feita por algoritmos de busca clássicos como o A*.

Em ambientes dinâmicos, modificações podem ocorrer enquanto a busca é realizada e/ou durante a aplicação dos resultados do planejamento. Já nos ambientes

parcialmente observáveis, não ocorrem alterações, mas o agente percebe divergências entre o mundo real e o seu modelo enquanto executa a solução devido a incompletude das informações disponíveis. Há, portanto, nesses algoritmos um intercalamento de planejamento e execução em um nível diferente do existente na classe tempo real (seção 2.3.6); pois realiza-se uma busca completa antes de iniciar o uso da solução.

Existem pelo menos três diferentes abordagens para reutilizar a informação [Sun et al., 2009] das buscas anteriores[6]: restaurar o conteúdo da estrutura de dados *open*, atualizar os *h-values* e construir a árvore de busca atual com base na precedente.

No primeiro modo, restaura-se o conteúdo das estruturas de dados *open* e *closed* para que reflitam o estado da busca anterior no instante de tempo onde ela poderia divergir da busca atual. Ou seja, o estado de uma busca realizada pelo A*, representado pelo conteúdo dessas estruturas de dados, é recuperado para permitir o reuso do início da árvore de busca imediatamente anterior. Para ser mais rápido do que uma simples repetição de buscas do A*, esse tipo de algoritmo de busca incremental deve ser capaz de restabelecer o estado da busca de maneira eficiente. Exemplos de algoritmos de busca heurística incremental pertencentes a essa subdivisão são: *incremental A** (iA*) [Sun & Koenig, 2007] e *Fringe-Saving A** (FSA*) [Sun & Koenig, 2007]. Ambos podem lidar somente com o aumento ou decréscimo dos pesos das arestas. Os experimentos descritos por Sun & Koenig [2007] mostraram que o FSA* possui um tempo de execução menor do que o iA*, mais curto do que o correspondente a uma série de buscas do A* e também inferior ao tempo do *Lifelong Planning A** (LPA*) (algoritmo de busca incremental explicado a seguir) - nesse caso para algumas situações específicas.

Uma maneira alternativa de reaproveitar os resultados de buscas passadas é atualizar os *h-values* agregando a eles mais informação. Os algoritmos de busca incremental *Adaptive A** (AA*) [Koenig & Likhachev, 2006a] e *Generalized Adaptive A** (GAA*) [Sun et al., 2008] assim procedem para aproveitar as informações das buscas anteriores na atual. Portanto, antes de iniciar a busca atual, se uma busca já foi realizada e encontrou uma solução, os *h-values* consistentes são alterados de modo a obter uma estimativa também consistente e ainda mais exata. Um dos casos mais simples ocorre quando, entre uma busca e outra, o custo de nenhuma aresta decresce (ou seja, eles apenas mantem-se iguais ou aumentam). Suponha que a busca

[6]Apesar dos autores terem reportado problemas com os experimentos realizados para esse artigo [Koenig, 2011], a classificação de algoritmos incrementais por eles sugerida ainda permanece válida.

anterior encontrou uma solução ótima cujo custo é $d^*(s,t)$. O custo do caminho ótimo de um nó $x$ até a meta $t$ é expresso por $d^*(x,t)$. Pela definição de $g(x)$, tem-se $d^*(s,t) \leq g(x) + d^*(x,t)$ e conseqüentemente $d^*(s,t) - g(x) \leq d^*(x,t)$. Como $d^*(x,t)$ é o custo do caminho ótimo de $x$ até o alvo $t$, a diferença $d^*(s,t) - g(x)$ é uma estimativa admissível desse custo. Pode-se mostrar que a nova heurística obtida é também consistente [Koenig & Likhachev, 2006a], supondo que a heurística empregada inicialmente também era.

O GAA* é uma versão genérica do AA*; pois permite, além da alteração do nó inicial e incremento do custo das arestas, a diminuição dos pesos das arestas e a mudança da meta. Ambos dependem da existência de uma solução e de uma heurística inicial consistente para, então, atualizarem os *h-values*. Cabe ressaltar que é válido utilizar zero como heurística inicial para todos os nós (caso não haja conhecimento prévio do problema suficiente, por exemplo). Os algoritmos, então, a partir da segunda busca, calcularão esses valores com base nas soluções anteriores. As avaliações empíricas realizadas [Sun et al., 2008] mostraram ter o GAA* um tempo de execução menor do que o *D* Lite* (algoritmo de busca incremental detalhado a seguir) e inferior ao tempo correspondente a uma série de buscas do A*. O domínio adotado foi o da busca com alvo móvel (perseguição).

O último subtipo de algoritmo de busca incremental atualiza os *g-values* das buscas anteriores durante a busca corrente corrigindo-os quando necessário. Transformam, portanto, a árvore de busca do problema anterior na árvore de busca do problema atual. Para que possuam um tempo de execução inferior ao correspondente a uma série de buscas do A*, a interseção entre as árvores de busca anterior e atual deve ser grande. Assim, fundamentam-se fortemente na possibilidade de que as mudanças no espaço de estados afetarão somente pequenos ramos da árvore de busca. Se muitas alterações ocorrem próximas à raiz da árvore de busca, aumentam-se as chances do desempenho degradar (como demostrado empiricamente por Koenig et al. [2004b]); pois o esforço computacional evitado com o aproveitamento de parte do processamento anterior não supera o esforço necessário de corrigir e/ou desfazer uma porção da árvore de busca do problema precedente.

Apesar dessas dificuldades, a maioria dos algoritmos de busca heurística incremental pertencem a esta subclasse, principalmente aqueles com maior utilização em aplicações práticas. Alguns exemplos são: *Lifelong Planning* A* (LPA*) [Koenig et al., 2004a], *Dynamic A** (D*) [Stentz, 1994] e *D* Lite* [Koenig & Likhachev, 2002] (abordado detalhadamente na subseção seguinte). Todos eles são capazes de lidar com o incremento e decremento do custo das arestas. Além disso, o D* e o *D* Lite* tratam também alteração do nó inicial. Em razão da relevância que possuem, esses

algoritmos possuem algumas extensões (também contempladas por esse subtipo): *Moving Target D* Lite* (MT-D* Lite) [Sun et al., 2010], *Anytime D** [Likhachev et al., 2005][7], *Field D** [Ferguson & Stentz, 2005] e *Focussed D** [Stentz, 1995].

As principais aplicações dos algoritmos de busca heurística incremental encontram-se na área da Robótica; especialmente dos algoritmos contemplados pelo último subtipo discutido. Nesse contexto é comum a existência de ambientes dinâmicos e informação parcial e/ou imperfeita. Essas características causam alterações nos componentes do problema de busca, gerando uma série de problemas de busca semelhantes como já explicado.

Os algoritmos D* e *D* Lite* foram apresentados precisamente com a intenção de realizar planejamentos de rotas ótimas de forma eficiente em ambientes desconhecidos, parcialmente conhecidos e dinâmicos para navegação de robôs. Grande parte de suas extensões contemplam também problemas nessa área.

Esse é o caso do algoritmo Field D*, fortemente baseado no algoritmo *D* Lite*, que tem sido empregado para o planejamento de rotas de sistemas robóticos reais em ambientes discretos (*grids*). Técnicas tradicionais podem produzir rotas não naturais e demandar deslocamentos desnecessários, pois restringem os movimentos dos robôs a um pequeno conjunto de direções (por exemplo; $0, \frac{\pi}{4}, \frac{\pi}{2}, \ldots, \frac{7\pi}{4}$). O algoritmo em questão pode gerar rotas com traçados suaves em *grids*. Isso é possível porque o Field D* computa rotas que podem entrar e sair de células em posições arbitrárias. Basicamente, ele altera a forma como os nós são extraídos do *grid* e adota uma interpolação para aprimorar a estimativa do custo. Os resultados são rotas mais naturais em ambientes discretos.

O algoritmo *Anytime D** também foi utilizado com sucesso em aplicações robóticas reais. Os robôs avaliados operavam em ambientes dinâmicos ao ar livre onde considerar a velocidade é importante para gerar rotas de traçado suave. A conseqüência natural do aumento das restrições são espaços de busca maiores, nesse caso com quatro dimensões. Como o cômputo da solução ótima pode ser inviável devido ao esforço necessário, o algoritmo é capaz de gerar soluções rapidamente e, então, aprimorá-las o quanto for possível (podendo chegar na melhor resposta) respeitando as restrições de processamento. Quando percebe mudanças no ambiente, reutiliza a informação provida pelas buscas passadas para calcular uma nova rota em pouco tempo. A solução inicial fornecida pelo algoritmo não é ótima, mas seu custo é limitado por um percentual (ajustável) do custo da menor solução.

[7] O algoritmo Anytime D* pertence também à classe *anytime* discutida na seção 2.3.5.

#### 2.3.2.1 *D* Lite*

Como um algoritmo de busca heurística incremental, o *D* Lite* [Koenig & Likhachev, 2002] assume a existência de uma série de problemas de busca semelhantes. Também reutiliza informações de buscas passadas para computar a solução do problema atual de forma mais rápida do que efetuar um novo cálculo completo. Devido a sua importância, especialmente para a Robótica, várias melhorias foram propostas para esse algoritmo, deixando-o mais genérico e eficiente para atender às demandas existentes. No entanto, a discussão apresentada nesta subseção estará restrita a uma versão mais simples do *D* Lite* que assume o uso de uma heurística consistente, utiliza a meta como raiz da árvore de busca[8] e desempata os nós com mesmo $f_2$*-value* em favor dos que têm menor $g_2$*-value*.

O *D* Lite* implementa a mesma estratégia de navegação do *Focussed D** (algoritmo de busca heurística incremental pertencente ao mesmo subtipo), mas de forma diferente. Por ser mais simples e rápido do que o *Focussed D** leva "Lite" no seu nome. Ambos possuem como motivação central a navegação de agentes em ambientes com pelo menos uma das seguintes características: desconhecidos, parcialmente observáveis e/ou dinâmicos.

A elaboração do *D* Lite* teve como base o algoritmo de busca heurística incremental LPA* adaptado ao grafo inverso. Além de manter um $g_2$*-value* para cada nó do grafo, o *D* Lite* emprega também uma segunda estimativa dos $g_2^*$*-values*. O $\text{rhs}_2(y)$, uma segunda estimativa da menor distância de $t$ até $y$, é calculado com base nos $g_2(x)$ - onde $x$ é um predecessor de $y$ no grafo inverso (que é o grafo explorado pelo algoritmo). É, portanto, um *lookahead* para o $g_2$*-value* de uma etapa da execução do algoritmo.

Com base nesses valores, um vértice $x$ é classificado como localmente consistente se $\text{rhs}_2(x) = g_2(x)$. Do contrário, é chamado de localmente inconsistente. Um dos invariantes do algoritmo é a manutenção apenas dos nós localmente inconsistentes na fila de prioridades empregada pelo algoritmo (semelhante à estrutura de dados *open* do A*). A prioridade de um vértice $x$ nessa estrutura de dados é composta de duas parcelas: $\min\{\text{rhs}_2(x), g_2(x)\} + h_2(x)$ e $\min\{\text{rhs}_2(x), g_2(x)\}$. A primeira parcela é similar ao $f_2$*-value*, mas também considera o $\text{rhs}_2$*-value*. A segunda corresponde a menor estimativa de $g_2^*(x)$. Quando há uma comparação dessas cha-

[8] O *D* Lite* expande os nós partindo de $t$ ao invés de $s$. Como o funcionamento é análogo às operações realizadas pelo processo de busca dos algoritmos bidirecionais que exploram o grafo de $t$ em direção a $s$; utiliza-se nessa descrição a notação empregada nos algoritmos bidirecionais. Dessa forma, fica explícito o fato da exploração ser realizada no grafo inverso e evita-se confusão com a notação dos algoritmos onde a busca parte de $s$.

ves para determinar qual nó é mais prioritário, vence aquela cuja primeira parcela é menor. Se houver um empate, aquela com a segunda parcela menor terá prioridade mediante a outra chave.

A versão do *D* Lite* em questão permite alteração da conectividade entre os nós do grafo (ou seja, criação e remoção de arestas), alteração de $s$ e também dos pesos da arestas (tanto para mais quanto para menos). Quando algum desses eventos ocorre, nós podem ficar localmente inconsistentes sendo necessário colocá-los novamente na fila de prioridades do algoritmo. O *D* Lite*, no entanto, não necessariamente altera o $g_2$*-value* de todos nós para que fiquem localmente consistentes. Ao invés disso, utiliza a heurística para focar a busca e atualizar apenas os vértices necessários ao cômputo do menor caminho. Para eliminar algumas inconsistências, pode ser necessário propagar as mudanças para os nós sucessores (diretos e indiretos). Essa é a razão pela qual o tempo de execução é menor quando há poucas modificações próximas à raiz da árvore de busca ($t$).

As condições para interromper o laço do algoritmo responsável por realizar a busca são: a prioridade da meta (no caso $s$, pois a busca é realizada partindo de $t$) é maior do que o nó mais prioritário dentre os localmente inconsistentes e $s$ não está mais localmente inconsistente. Nesse momento, a qualidade da solução não pode mais ser aprimorada. Koenig & Likhachev [2002] apresentam uma discussão completa do *D* Lite* que inclui a prova da admissibilidade desse algoritmo.

### 2.3.3 Classe *memory-concerned*

A proposta dos algoritmos compreendidos por esta classe possuem pelo menos um dos seguintes objetivos: diminuir a quantidade de memória necessária para armazenar as estruturas de dados *open* e *closed* do A*, ser capaz de realizar buscas em ambientes computacionais de memória escassa ou atender a aplicações onde é necessário controlar a quantidade de memória empregada na busca. Conforme mencionado, o número de nós nessas estruturas de dados pode ser exponencial em função do comprimento do caminho ótimo. Portanto, dependendo das características do problema, o A* pode esgotar toda memória disponível antes de encontrar uma solução, tornando sua utilização impraticável. Nessas circunstâncias, como o algoritmo não está preparado para lidar com a ausência de recursos computacionais, o processamento falha e termina sem determinar uma solução.

São três os subtipos básicos de algoritmos na classe *memory-concerned*. Os dois primeiros empregam exclusivamente a memória principal [Zhou & Hansen, 2002a], enquanto o último utiliza também a memória secundária para realizar a computa-

ção. Parte dos algoritmos da classe *memory-concerned*, principalmente aqueles pertencentes aos dois primeiros subtipos mencionados, claramente abordam o compromisso fundamental presente na Ciência da Computação entre tempo e espaço. Sua adoção é conveniente, nesses casos, em aplicações onde pode-se sacrificar o tempo de execução em benefício da redução da quantidade de espaço necessário à computação.

Denominado *memory-efficient*, o primeiro subtipo contempla algoritmos que realizam a busca utilizando uma quantidade de espaço menor do que seria necessário para realizar a mesma tarefa através do A*. Ou seja, a motivação central para a proposição desses algoritmos foi a sua capacidade de realizar buscas com eficiência espacial.

Uma parcela dos algoritmos do primeiro subtipo não mantém as estruturas de dados *open* e *closed*. Como conseqüência, esses algoritmos expandem o mesmo nó várias vezes; mas em contrapartida a demanda por espaço é linear em função do comprimento da solução. Exemplos de algoritmos de busca heurística com essas características são: *Iterative-Deepening A** (IDA*) [Korf, 1985], DFS*[9] [Vempaty et al., 1991], *Binary Decision Diagram Iterative-Deepening A** (BDD-IDA*) [Qian et al., 2005] e *Recursive Best-First Search* (RBFS) [Korf, 1993]. Detalhes do algoritmo IDA* serão apresentados posteriormente na subseção 2.3.3.1.

O restante dos algoritmos de busca heurística do subtipo *memory-efficient* empregam estratégias diversas com o intuito de reduzir a quantidade de memória necessária para realizar a busca. Um deles é o *Partial Expansion A** (PEA*) [Yoshizumi et al., 2000]. O seu objetivo é armazenar somente os nós necessários para determinar a solução ótima. A alteração essencial com relação ao A* é a introdução do conceito de expansão parcial. Ao selecionar um nó da estrutura de dados *open* para expansão, não necessariamente todos seus sucessores serão armazenados. Somente aqueles considerados mais promissores. O algoritmo mantém, para cada nó, um atributo adicional (denotado por $F$) utilizado como prioridade para expansão. O critério para avaliar o quanto um nó é promissor considera uma constante pré-definida e a diferença entre os *F-values* do nó expandido e do seu sucessor.

Inicialmente, o *F-value* de um nó é igual a seu $f$*-value*. Quando ocorre uma expansão parcial, ele passa a conter o menor $f$*-value* dos sucessores não promissores (ou seja, daqueles que não foram armazenados) e é mantido na estrutura de dados *open*. Assim, o algoritmo garante sua admissibilidade, pois posteriormente poderá reconsiderar a expansão dos nós parcialmente expandidos. Além de ser simples se

[9] Segundo os autores, o "acrônimo" foi escolhido para sugerir uma busca em profundidade admissível. Portanto, não há uma correspondência direta com um nome.

comparado aos outros algoritmos da classe *memory-concerned*, é capaz de realizar a computação com uma quantidade de memória significativamente inferior à requerida pelo A*, segundo os experimentos realizados pelos autores.

Já o algoritmo *Frontier-A** [Korf et al., 2005] armazena somente a estrutura de dados *open* (os nós candidatos a expansão) não utilizando recursos, portanto, para manter a estrutura de dados *closed*. Além de evitar a expansão do mesmo nó várias vezes, essa última também é comumente utilizada para recuperar os nós componentes do caminho que representa a solução. O primeiro problema é superado através de uma estratégia de expansão capaz de garantir a não geração de vértices que já foram expandidos (que estariam em *closed*). Isso envolve armazenar para cada nó um vetor de *bits* representando os operadores que poderão ser empregados na sua expansão. O outro obstáculo é superado através de uma técnica de divisão e conquista para recuperar o caminho que representa a solução. Nos experimentos realizados, o *Frontier-A** reduziu significativamente a quantidade de memória necessária ao cômputo das soluções ótimas.

O segundo subtipo de algoritmo de busca heurística da classe *memory-concerned* é chamado de *memory-bounded*. Seus membros preservam as estruturas de dados empregadas pelo A*, mas limitam o seu crescimento. Dessa forma, são capazes de realizar computações com uma quantidade fixa de memória. Esse é o caso dos algoritmos *Simple Memory-Bounded A** (SMA*) [Russell, 1992] e *Simple Memory-Bounded A* Graph* (SMAG*) [Zhou & Hansen, 2002a] que controlam o crescimento das estruturas de dados *open* e *closed*. Quando não há mais espaço disponível, os nós menos promissores são podados para permitir a inserção de novos vértices. Para preservar parte da informação que seria perdida com a poda, a heurística do nó predecessor é atualizada com os dados do nó podado. Nota-se, portanto, que diante da falta de espaço para armazenamento o algoritmo é capaz de prosseguir a computação, diferentemente do A*.

O último subtipo de algoritmo da classe *memory-concerned* emprega também a memória secundária para armazenar as estruturas de dados do algoritmo A*. Esse tipo de armazenamento, discos rígidos por exemplo, provêem muito mais espaço a um custo muito reduzido. No entanto, para utilizá-los de forma eficiente os algoritmos devem acessá-los de maneira seqüencial, pois o tempo da operação deslocamento (do inglês *seek*) é muito elevado. O algoritmo *External A** [Edelkamp et al., 2004] emprega uma técnica chamada *Delayed Duplicate Detection* (DDD). De modo simplificado, ela mantém os nós das estruturas de dados *open* e *closed* em um única lista. A cada iteração, o nó com menor $f$*-value* é expandido. Ao final, uma espécie de ordenação externa elimina as repetições. Korf [2004] aplicou o DDD ao *Frontier-A**

para torná-lo apto a empregar também memória secundária na computação.

Relatada por Yoshizumi et al. [2000], uma das aplicações do algoritmo de busca heurística *Partial Expansion A** (PEA*) da classe *memory-concerned* foi na Bioinformática. O alinhamento de múltiplas seqüências para extração de um padrão comum é um dos problemas centrais nessa área, pois os dados produzidos por esse procedimento servem de insumo para várias análises. Como o fator de ramificação é grande (para 7 seqüências é 127 e para 8 seqüências 255), e, conseqüentemente, também é o espaço de estados, a dificuldade computacional desse problema é enorme. Nesse contexto, o PEA* conseguiu realizar o alinhamento de 8 seqüências, diferentemente do A* que excedeu a memória disponível. De acordo com o valor da constante utilizada para definição dos nós promissores, o algoritmo reduziu a necessidade de memória com relação ao A* em 87%, enquanto o número de expansões cresceu apenas 20%.

A Verificação de Modelos (do inglês, *Model Checking*) é adotada para verificar automaticamente propriedades em um modelo construído para expressar características relevantes do sistema que será testado. Qian et al. [2005] propõe o algoritmo BDD-IDA*, uma versão do algoritmo IDA* aperfeiçoado para explorar *binary decision diagrams* (BDDs), e a sua utilização em verificadores de modelos para provar a falsidade de invariantes do sistema. Para esse tipo de tarefa, algoritmos de busca cega tipicamente empregados possuem a desvantagem de processar partes do espaço de estados desnecessárias. Logo, com base nos experimentos realizados, foram reportados aumentos significativos no tempo de execução.

#### 2.3.3.1 IDA*

O *Iterative-Deepening A** (IDA*) [Korf, 1985] é o resultado da combinação da busca em profundidade iterativa (do inglês, *depth-first iterative-deepening*) com o algoritmo de busca heurística A*. A motivação para tal associação é a possibilidade de reduzir a complexidade espacial do A*. O IDA*, como a busca em profundidade iterativa [Russell & Norvig, 2003], possui complexidade de espaço linear em função do comprimento da solução. Mas também, como na busca em profundidade iterativa, alguns vértices, principalmente os mais próximos à raiz, poderão ser expandidos diversas vezes. Assume-se, pois, a opção de sacrificar o tempo de execução em detrimento de reduzir a demanda por memória.

O algoritmo 2.3 apresenta uma versão do IDA* que assume o uso de uma heurística admissível. Nesse caso, o algoritmo garantidamente encontra a solução ótima se uma solução existe. O IDA* realiza uma série independente de buscas em pro-

fundidade limitando o custo máximo de um caminho por um valor (denotado por $f_{\max}$) crescente ao longo das buscas. Como no A*, o custo do caminho de $s$ até $t$ que passa por um determinado nó é a soma das duas parcelas correspondentes ao *g-value* e ao *h-value*.

Inicialmente, o valor de $f_{\max}$ escolhido é o custo estimado de $s$ até $t$, ou seja, $h(s)$. O procedimento recursivo DFS realiza uma busca em profundidade limitada e retorna o custo da menor solução por ela encontrada dentro do limite especificado. Os parâmetros necessários são: um nó, o custo da raiz até o nó informado e o custo máximo de um caminho para aquela busca em profundidade. As linhas 13-15 são responsáveis por determinar o próximo valor de $f_{\max}$ armazenado na variável global next_$f_{\max}$. Ele será o menor *f-value* que excedeu o limite atual. Se a heurística é admissível, a forma como o primeiro e os subseqüentes valores de $f_{\max}$ são selecionados garante o cálculo do custo do menor caminho de $s$ até $t$ quando um valor finito é retornado pela chamada de DFS na linha 6. Korf [1985] apresenta as provas formais.

### 2.3.4 Classe paralela

As dificuldades causadas pela grande demanda de recursos computacionais do A* (tanto de tempo quanto de espaço) podem ser atenuadas através da execução paralela. Os algoritmos desta classe são apropriados para execução paralela, ou seja, exploram as possibilidades de concorrência para resolver problemas de busca com um tempo menor e/ou para conseguir solucionar instâncias maiores.

A paralelização do A* possui alguns desafios. Como já foi descrito, o A* emprega duas estruturas de dados para auxiliá-lo na busca denominadas *open* e *closed*. A primeira concentra todos os nós candidatos a expansão. Por se tratar de um algoritmo do tipo *best-first*, a cada iteração o A* seleciona o nó mais promissor dessa estrutura de dados para expansão. A outra estrutura de dados evita a expansão do mesmo nó várias vezes e é consultada ao longo da expansão de um nó, mais especificamento no momento da geração dos seus sucessores, para evitar repetições. Quando for necessário acessar essas estruturas de dados, um dos desafios é evitar a contenção entre os elementos computacionais, a fim de permitir os maiores períodos de computação possível sem sincronização.

O modelo computacional para o qual os algoritmos são desenvolvidos afeta substancialmente o seu funcionamento. Faz-se necessário, então, a sua separação de acordo com essas diferenças. O critério utilizado para proposição dos dois subtipos (memória compartilhada e memória distribuída) foi a arquitetura da memória do

```
1: {Procedimento IDA*:}
2: next_fmax ← h(s) {Variável global alterada pelo procedimento DFS}
3: loop
4:    fmax ← next_fmax
5:    next_fmax ← ∞
6:    solution_cost ← DFS(s, 0, fmax)
7:    if solution_cost ≠ ∞ then
8:       return solução ótima encontrada
9:    else if next_fmax = ∞ then
10:      return não existe solução
11:
12: {Procedimento (recursivo) DFS(x, c, fmax):}
13: if c + h(x) > fmax then
14:    if next_fmax > c + h(x) then
15:       next_fmax ← c + h(x)
16:    return ∞
17: if x = t then
18:    best_solution_cost ← c
19: else
20:    best_solution_cost ← ∞
21:    for all arestas (x, y) do grafo do
22:       solution_cost ← DFS(y, c + d(x, y), fmax)
23:       if best_solution_cost > solution_cost then
24:          best_solution_cost ← solution_cost
25: return best_solution_cost
```

Algoritmo 2.3: O algoritmo IDA* para heurística admissível.

modelo computacional.

Burns et al. [2009] avaliaram empiricamente o desempenho de algumas abordagens simples para paralelização do A* em arquiteturas de memória compartilhada. Uma delas resultou no algoritmo chamado *Parallel A** (PA*), onde basicamente cada *thread* pode manipular as estruturas de dados *open* e *closed* (pois o acesso exclusivo é garantido através de primitivas de sincronização) e expandir nós em paralelo. Outra abordagem, baseada no algoritmo *K-Best-First Search* (KBFS) [Felner et al., 2003], seleciona um determinado número de nós mais promissores e delega a expansão de cada um a uma *thread* diferente. Na implementação, uma *thread* mestre é responsável por extrair os nós da estrutura de dados *open*, aguardar a expansão e inserir os nós produzidos nas devidas estruturas de dados (porque somente ela pode acessar *open* e *closed*).

Constatou-se nesses casos um aumento no tempo de execução com relação ao

A* no domínio empregado - *pathfinding* em *grids*[10] com conectividade quatro. Uma possível explicação para esse resultado, especialmente da versão paralela do KBFS, é o esforço extra causado pela necessidade de sincronização. Como o tempo de expansão de um vértice é muito pequeno se comparado a outras operações realizadas, o esforço trazido pela paralelização supera os benefícios.

Os resultados encontrados por Vidal et al. [2010] reforçam a hipótese de que abordagens simples para a paralelização do A* não obtém resultados satisfatórios em todos domínios (principalmente naqueles cujo o tempo de expansão de um vértice é muito baixo quando comparado a outras operações efetuadas). Ao contrário do relatado por Burns et al. [2009], nos experimentos por eles realizados houve um significativo corte no tempo de execução. Porém, os próprios autores admitiram ter sido conseqüência direta de uma característica do domínio: o grande esforço que deve ser empreendido na expansão de um nó, especialmente no momento de computar as heurísticas. Logo, o trecho do algoritmo paralelizado responde a uma grande parcela do tempo, e o custo trazido pela paralelização passa a ter uma influência muito menor na soma final.

A ineficácia dessas estratégias sugestiona, pois, a necessidade de abordagens mais sofisticadas para criar algoritmos mais genéricos capazes de reduzir o tempo de execução em diferentes domínios. É o caso do algoritmo *Parallel Best-NBlock-First* (PBNF) [Burns et al., 2009] que emprega uma estratégia mais elaborada para paralelizar o A* em ambientes computacionais de memória compartilhada e atingir bons resultados em domínios diversos. Devido a sua importância neste trabalho (é base para criação do algoritmo apresentado no capítulo 4), ele será discutido em detalhes na subseção 2.3.4.1.

Há também os algoritmos de busca heurística paralela inspirados no A* para ambientes computacionais de memória distribuída. Além de reduzir o tempo de execução, esses algoritmos podem também atenuar os problemas causados pela falta de memória necessária aos cálculos; pois a quantidade de espaço total será a soma da contribuição de cada elemento computacional. Um dos desafios comuns em algoritmos distribuídos é a fase de finalização. Como não há acesso ao estado global da computação, deve haver uma coordenação para realizar o término da computação. No caso do A*, dependendo da forma de operação (por exemplo, se cada elemento da computação é responsável por parte dos nós), deve existir um critério extra já que a primeira solução encontrada não necessariamente é ótima. Nesses casos, é comum a utilização da seguinte condição: nenhum dos elementos compu-

[10] O domínio do *pathfinding* em *grids* é comumente empregado para avaliação experimental de algoritmos de busca heurística e é tratado na subseção 3.2.1.

tacionais deve possuir nós que possam diminuir o custo da solução já calculada.

Um dos algoritmos de busca heurística paralela deste subtipo é o *Parallel Retracting A** (PRA*)[11] [Evett et al., 1995]. Ele mantém as estruturas de dados *open* e *closed* na memória local de cada processador. A cada iteração, o processador expande o nó mais promissor em sua estrutura de dados *open*. Uma função de mapeamento (*hash*) é responsável por determinar a qual processador um nó está associado. A comunicação entre os processadores ocorre através de troca de mensagens. Os experimentos mostraram um significativo *speedup* no domínio do *15-puzzle*[12].

O *Hash Distributed A** (HDA*) [Kishimoto et al., 2009] é também um algoritmo que estende o A* para ambientes computacionais de memória distribuída. Combina duas técnicas: comunicação assíncrona entre processos e uma função *hash* que distribui uniformemente os estados entre os processadores disponíveis - a última é uma influência do algoritmo PRA*. Cada processador mantém suas estruturas de dados *open* e *closed* em sua memória local. A grande inovação do HDA* é o uso de comunicação assíncrona que permite o envio de mensagens sem interromper a expansão de nós. Os processadores dão preferência ao recebimento de mensagens para, então, expandirem o nó mais promissor de *open*. Nos diversos domínios avaliados, a redução de tempo foi significativa se comparada ao A*, até mesmo em ambientes computacionais de memória compartilhada também analisados.

#### 2.3.4.1 PBNF

O *Parallel Best-NBlock-First* (PBNF) [Burns et al., 2009] é um algoritmo de busca heurística paralelo para ambientes computacionais de memória compartilhada. Com o intuito de lidar com as dificuldades inerentes a paralelização de algoritmos de busca heurística, o espaço de estados é divido em subregiões chamadas de *nblocks*. Conseqüentemente, diferentemente do A* que explora sempre o nó mais promissor primeiro, ele explora primeiramente o *nblock* que oferece as melhores perspectivas. Essa é explicação da origem de seu nome.

A forma de dividir o espaço de estados é denominada *Parallel Structured Duplicate Detection* (PSDD) e sua intenção é evitar a necessidade de sincronização para cada expansão de nós. Uma função de abstração dependente do domínio é empregada para realizar a separação do espaço de estados. Ela mapeia vários nós do

[11] O PRA* também inclui um esquema para reduzir a utilização de memória: os nós com heurística menos promissora são removidos da estrutura de dados *open*. Portanto, pertence também à classe *memory-concerned*.

[12] O *15-puzzle* assim como o *8-puzzle* são instâncias do *n-puzzle*. O domínio do *n-puzzle* é comumente empregado para avaliação experimental de algoritmos de busca heurística e é tratado na subseção 3.2.2.

grafo que representa o espaço de estados a um único nó do grafo abstrato (que corresponde a um *nblock*). O *nblock* de um nó é dito ser a sua imagem. Se dois vértices que possuem como imagem diferentes *nblocks* estão conectados por uma aresta, haverá também no grafo abstrato uma aresta entre as suas imagens. Portanto, a função de mapeamento define um grafo abstrato.

Cada *nblock* possui suas próprias estruturas de dados *open* e *closed*. A idéia central do PSDD é atribuir a cada um dos processadores envolvidos na computação *nblocks* que possam ser explorados paralelamente e independentemente. Ou seja, a escolha dos *nblocks* a serem processados é feita para que não exista interferência durante sua exploração. Quando a busca é realizada por um processador na porção do grafo delimitada por um *nblock*, não necessariamente somente esse nó abstrato será afetado. A expansão de nós do *nblock* pode produzir vértices cuja imagem seja qualquer um dos seus sucessores no grafo abstrato. O *duplicate detection scope* de um vértice abstrato $a$ é o conjunto de sucessores de $a$ no grafo abstrato. Além do próprio $a$, são os únicos *nblocks* cujas estruturas de dados *open* e *closed* precisam ser alteradas durante a exploração de $a$.

Com a finalidade de facilitar a seleção de *nblocks* com *duplicate detection scope* disjunto, mantém-se um campo $\sigma(a)$ para cada *nblock* $a$. Ele armazena o número de *nblocks* sucessores de $a$ que estão sendo utilizados por algum processador (de modo direto ou indireto). Basta, então, selecionar aqueles cujo o $\sigma$-*value* seja zero. O PSDD utiliza um único mecanismo de sincronização para manipulação do grafo abstrato com a finalidade de garantir que a aquisição e liberação dos *nblocks* não será efetuada simultaneamente por mais de um processador. Quando essas operações são realizadas, os $\sigma$-*values* dos nós abstratos cujos *duplicate detection scopes* interferem com o *duplicate detection scope* do *nblock* em questão devem ser alterados. Esse conjunto de nós é chamado de *interference scope* de um *nblock*.

Ambos conceitos são ilustrados através da figura 2.4 para *grids* de conectividade quatro. O *grid* exibido (duas vezes) nessa figura representa o grafo abstrato. Suas células são *nblocks*. O mapeamento das células do *grid* original de conectividade quatro foi feito sobrepondo a ele o *grid* mostrado (que possui uma resolução menor). Ao realizar essa sobreposição, várias células do *grid* original são envolvidas por um *nblock* (que é uma célula do *grid* de resolução menor). Como o *grid* original possui conectividade quatro, cada célula do *grid* que representa o grafo abstrato também possui quatro sucessores.

Ao explorar o *nblock* $a$, as quatros células destacadas na parte esquerda da figura poderão ser alteradas. No lado direito da figura, são evidenciados os nós abstratos que fazem parte do *interference scope* de $a$. Dois *nblocks* foram designados

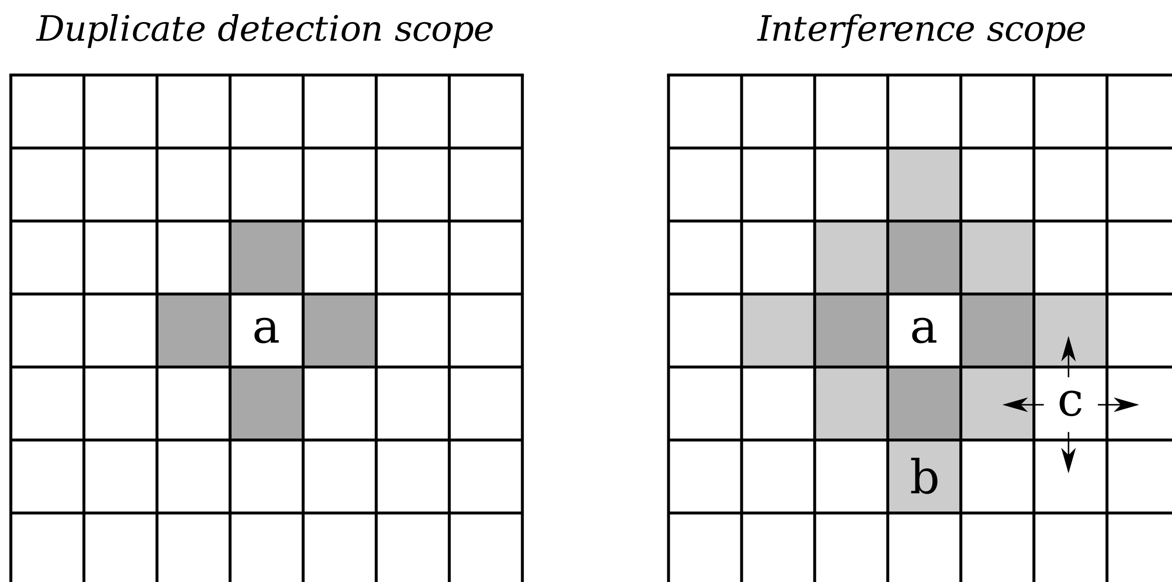



Figura 2.4: O *duplicate detection scope* e o *interference scope* do *nblock* $a$.

pelas letras $b$ e $c$. O primeiro, que faz parte do *interference scope* de $a$, não pode ser explorado enquanto $a$ é explorado, pois nesse caso mais de um processador poderia alterar, simultaneamente, o mesmo *nblock*. Já a exploração de $c$ pode ser realizada enquanto $a$ é explorado, pois nenhum dos nós abstratos no *duplicate detection scope* de $c$ (denotado por setas) faz parte do *duplicate detection scope* de $a$.

Ao começar a computação, o PBNF atribui a $g(s)$ o valor 0 e coloca esse nó na estrutura de dados *open* do *nblock* para o qual ele foi mapeado. O *nblock* em questão torna-se disponível para exploração e é inserido em uma fila de prioridades de *nblocks* onde a importância de cada elemento é medida através do menor $f$*-value* de sua estrutura de dados *open*. A preferência é daqueles com a menor prioridade. Nesse instante, os processadores disponíveis podem iniciar a exploração do espaço de estados. As operações executadas por cada um deles são detalhadas a seguir.

O processador tenta alocar para si um *nblock* afim de explorá-lo. As condições para que um *nblock* $a$ seja selecionável são: a presença de pelo menos um vértice em sua estrutura de dados *open*, ser zero o valor de $\sigma(a)$ e $a$ não estar alocado a nenhum processador. Se não for possível, aguarda até a liberação de um com essas características. Caso contrário, realiza a exploração de maneira semelhante ao A*. A expansão de um nó pode gerar sucessores mapeados a diferentes *nblocks* que devem ser inseridos em suas respectivas estruturas de dados *open*. Como não existe uma sincronização por camadas (ou seja, o algoritmo não expande os nós obedecendo a ordem crescente dos $f$*-values*) na expansão dos nós, pode ser necessário reabrir alguns deles (tirá-los da estrutura de dados *closed* e inseri-los novamente em *open*)

se o custo do novo caminho até ele for inferior ao já registrado. Nesse aspecto, portanto, o PBNF é semelhante a versão do A* para heurística somente admissível. A expansão de nós nessa subregião do espaço prossegue até que se esgotem os candidatos disponíveis ou até que seja possível obter um *nblock* mais promissor. A fim de evitar acesso excessivo à estrutura de dados que representa o grafo abstrato, sempre que possível, é necessária a expansão de um número mínimo de nós antes de tentar trocar de *nblock*.

A primeira solução encontrada, pela inexistência de uma sincronização por camadas, pode não ser a solução ótima. Para garantir a otimalidade, o algoritmo prossegue a exploração do espaço de estados até que não haja mais nós cujo $f$*-value* é menor que o custo da solução atual. Logo, a partir da primeira resposta encontrada o algoritmo inicia a poda dos vértices. Se a heurística empregada for pelo menos admissível, o PBNF garante sempre encontrar a solução ótima caso o problema tenha alguma solução.

### 2.3.5 Classe *anytime*

A quinta classe apresentada neste trabalho reúne os algoritmos de busca heurística *anytime*. Zilberstein [1996] descreve várias características desejáveis em algoritmos *anytime*. Porém, destaca a possibilidade do retorno de soluções num instante qualquer da computação. As soluções geradas inicialmente não necessariamente são ótimas, mas seu custo converge para o valor ótimo ao longo do tempo. Os algoritmos desta classe, tipicamente, calculam a primeira solução (não ótima na maioria da vezes) de forma rápida com o intuito de diminuir a fase inicial na qual não há solução disponível. Ao encontrar uma solução, a computação prossegue para refiná-la, deixando-a cada vez mais próxima do valor ótimo. Portanto, quanto mais tempo é empregado pelo algoritmo, melhores serão os resultados por ele calculados.

Alguns trabalhos, no contexto de busca heurística, empregam os termos *anytime* e tempo-real de forma indistinta. Existem semelhanças entre os conceitos. Uma delas é o aumento na qualidade da solução quando há uma disponibilidade maior de tempo. Além disso, em ambos existe a possibilidade de sacrificar a qualidade para obter uma resposta no tempo desejado. Outra semelhança é que os dois são úteis em domínios onde é melhor agir rapidamente do que atuar de maneira ótima. Fica evidente a presença de um compromisso entre qualidade e tempo nas duas classes.

Acredita-se, entretanto, pelo menos no contexto em questão, que as semelhanças não superam as diferenças. Uma delas é a capacidade dos algoritmos *any-*

*time* [Zilberstein, 1996] de quantificar a qualidade da solução que está sendo retornada. Algoritmos de tempo-real, de modo geral, não dispõe desse recurso. Além disso, os algoritmos de tempo-real restringem o planejamento a parte do grafo para conseguirem atender às restrições de tempo impostas e não garantidamente retornam soluções. Outra distinção é a limitação de parte dos algoritmos *anytime* que, apesar do nome, possuem uma fase inicial na qual não há solução disponível e não são capazes de atender à restrições de tempo. A última diferença é o intercalamento entre planejamento e execução existente nos algoritmos de tempo-real. Na maioria das vezes, ele não é encontrado nos algoritmos *anytime*. Com base em todas essas considerações, decidiu-se criar duas classes.

A utilização de algoritmos pertencentes a classe *anytime* é conveniente em aplicações onde há incerteza e/ou variação nas restrições de tempo. Uma resposta estará disponível assim que a fase inicial, de curta duração e responsável por calcular a primeira solução, terminar. Se mais tempo for provido ao algoritmo, ele será explorado na melhoria da qualidade da solução já computada. A aplicação, por conseguinte, possuirá uma flexibilidade para lidar com alterações no prazo para deliberação. Também disporá de recursos para conhecer a qualidade da solução atualmente disponível e empregar o compromisso entre tempo de execução e qualidade da solução a seu favor.

Um modo de calcular a primeira solução rapidamente envolve inflar artificialmente a heurística multiplicando-a por um fator ($w$) maior do que 1. Essa estratégia é explorada pelos algoritmos *Anytime Weighted A** (AWA*) [Hansen & Zhou, 2007] (detalhado na subseção 2.3.5.1) e *Anytime Repairing A** (ARA*) [Likhachev et al., 2003a,b]. O algoritmo ARA* difere do primeiro, pois, para aprimorar a qualidade das soluções computadas, novas buscas são repetidas com $w$ decrescendo até 1. Quando $w$ é igual a 1, a solução ótima é calculada (supondo que a heurística é pelo menos admissível). Além disso, propõe o reuso dos resultados de buscas anteriores de modo semelhante ao reaproveitamento realizado por alguns algoritmos da classe incremental (subseção 2.3.2).

Uma vantagem do ARA*, com relação ao AWA*, é um controle maior do limite da não otimalidade através dos vários valores de $w$ adotados. Likhachev et al. [2003b] relatam também resultados de experimentos onde o ARA* foi capaz de reduzir os custos computados de maneira mais gradual e rápida.

van den Berg et al. [2011] propõe o algoritmo *Anytime Nonparametric A** (ANA*). Seu grande diferencial, principalmente com relação ao ARA*, é a ausência dos parâmetros $w$ e do seu decremento providos pelo usuário. Ao invés de utilizar os *$f$-values* para ordenar os nós candidatos a expansão; o ANA* utiliza o

$e(x) = \frac{g(t)-g(x)}{h(x)}$ de forma decrescente, onde $x$ é um nó do grafo e $g(t)$ é o custo do caminho de $s$ até $t$ encontrado pela última busca realizada. Na primeira busca, quando ainda não há solução, utiliza os *h-values* de forma crescente. Com base na comparação, através de avaliações empíricas realizadas com o ARA*, os autores afirmam que normalmente o ANA* encontra a solução inicial mais rápido e gasta menos tempo entre o aperfeiçoamento delas.

Zhou & Hansen [2002b] mostraram que algoritmos *anytime* podem resolver o problema do alinhamento de múltiplas seqüências de DNA de forma satisfatória. Como já discutido neste trabalho, o espaço de estados é muito grande devido ao fator de ramificação elevado. O cálculo da solução ótima exige, pois, a utilização de algoritmos mais sofisticados do que o A*. Os algoritmos *anytime* possuem uma característica útil para esses problemas. Eles computam rapidamente uma solução (normalmente não ótima) que estabelece um limite superior e auxilia na poda de nós. Como a computação prossegue para garantir a solução ótima, novos limites superiores são calculados, cada vez mais próximos do valor desejado, permitindo a poda de mais nós candidatos a expansão ao longo do processo.

Uma outra aplicação, que explora melhor as propriedades específicas dos algoritmos desta classe, está na área de jogos digitais. Hawes [2003] propõe uma arquitetura voltada para agentes que atuam em ambientes com características semelhantes as de jogos digitais: dinâmico e parcialmente observável. Um dos componentes dessa arquitetura é um planejador *anytime* que emprega algoritmos semelhantes aos descritos anteriormente. Ele oferece a flexibilidade na geração de planejamentos quanto o tempo e a qualidade necessários para atuação nesse contexto. Segundo o autor, algumas vantagens dessa abordagem são: o agente gerencia explicitamente os recursos empregados durante o planejamento e é capaz de responder a mudanças de forma imediata. Nota-se que ambas relacionam-se diretamente com o dinamismo do ambiente onde um determinado objetivo poder deixar de ser prioritário ou sua importância para o agente aumentar ainda mais. A utilidade do resultado de um planejamento pode ainda variar com o tempo. Ou seja, se muito tempo for utilizado para determinar uma solução, ela pode ser inútil quando estiver terminada. Todas essas considerações foram levadas em conta para proposição da arquitetura. O conceito proposto foi implementado e avaliado através de um agente para um jogo comercial.

#### 2.3.5.1 Anytime Weighted A*

O *Anytime Weighted A** (AWA*) [Hansen & Zhou, 2007] é uma versão *anytime* do algoritmo *Weighted A**[13]. Inicialmente, ele computa uma solução não necessariamente ótima de maneira rápida devido a utilização dos $f'$*-values*. A busca, então, continua e soluções com um custo cada vez menor são encontradas. Quando a solução ótima é determinada, a computação termina. A partir da primeira solução, o usuário já dispõe de uma resposta e pode optar por usá-la ou aguardar mais tempo até que uma solução com a qualidade desejada esteja disponível. A escolha de $w$ permite a ele também empregar convenientemente o compromisso entre tempo de execução e qualidade da solução.

O algoritmo 2.4 apresenta uma versão do AWA* que assume o uso de uma heurística admissível. Para cada nó do grafo, o algoritmo mantém, além dos valores já empregados pelo A*, um $f'$*-value* computado de modo semelhante ao $f$*-value*, mas multiplicando o valor da heurística por $w$. Enquanto o algoritmo não é interrompido e *open* não está vazia; o vértice do grafo nessa estrutura de dados com menor $f'$*-value* é selecionado para expansão. Ao expandi-lo, pode ser necessário reabrir seus sucessores presentes em *closed* em razão da utilização dos $f'$*-values* como prioridade. A condição para que isso ocorra é a melhoria do *g-value* do nó que será reaberto. Portanto, o AWA*, para computar a solução ótima, realiza normalmente mais expansões que o A* (considerando a adoção por ambos da mesma heurística).

Uma solução encontrada é um limite superior para o custo do menor caminho de $s$ até $t$ e é empregada no critério da poda para evitar expansões desnecessárias. Se o AWA* dispuser de tempo suficiente e a heurística adotada for admissível, ele sempre encontrará a melhor solução. A razão pela qual isso é verdade está diretamente relacionada com o emprego simultâneo no algoritmo dos $f'$*-values* e dos $f$*-values*. Somente os $f$*-values* são usados como critério de poda. O algoritmo é capaz de computar também a diferença máxima do custo da solução retornada e da resposta ótima. Hansen & Zhou [2007] apresentam a prova formal da admissibilidade do AWA* e do limite superior do erro reportado.

[13]A principal diferença do algoritmo *Weighted A** para o A* é a utilização dos $f'$*-values* no lugar dos $f$*-values* para seleção dos nós que serão expandidos. Se $x$ é um vértice do grafo, $f'(x)$ é igual a $g(x) + w \times h(x)$ ($w$ é um valor maior do 1 definido pelo usuário). Essa alteração faz com que uma solução não necessariamente ótima seja encontrada num tempo muito inferior ao necessário para calcular a solução ótima.

1: $g(s) \leftarrow 0$
2: $f(s) \leftarrow g(s) + h(s)$
3: $f'(s) \leftarrow g(s) + w \times h(s)$
4: parent$(s) \leftarrow NULL$
5: $g(t) \leftarrow \infty$
6: open $\leftarrow \{s\}$
7: closed $\leftarrow \emptyset$
8: **while** open $\neq \emptyset \wedge \neg$interrupted **do**
9: $x \leftarrow$ argmin$\{f'(n) \mid n \in$ open$\}$
10: open $\leftarrow$ open$\setminus\{x\}$
11: closed $\leftarrow$ closed $\cup \{x\}$
12: **if** $x \neq t \wedge f(x) < g(t)$ **then**
13: **for all** arestas $(x, y)$ do grafo $\mid g(x) + d(x, y) < g(t)$ **do**
14: **if** $y \in$ open $\cup$ closed **then**
15: **if** $g(x) + d(x, y) < g(y)$ **then**
16: $g(y) \leftarrow g(x) + d(x, y)$
17: $f(y) \leftarrow g(y) + h(y)$
18: $f'(y) \leftarrow g(y) + w \times h(y)$
19: parent$(y) \leftarrow x$
20: **if** $y \in$ closed **then**
21: closed $\leftarrow$ closed$\setminus\{y\}$
22: open $\leftarrow$ open $\cup \{y\}$
23: **else**
24: $g(y) \leftarrow g(x) + d(x, y)$
25: $f(y) \leftarrow g(y) + h(y)$
26: $f'(y) \leftarrow g(y) + w \times h(y)$
27: parent$(y) \leftarrow x$
28: open $\leftarrow$ open $\cup \{y\}$
29: **if** $g(t) = \infty$ **then**
30: **if** open $\neq \emptyset$ **then**
31: **return** solução não encontrada {a primeira busca não foi concluída}
32: **else**
33: **return** não existe solução
34: **else**
35: **if** open $\neq \emptyset$ **then**
36: **print** $g(t) - \min\{f(n) \mid n \in$ open$\}$ {erro máximo}
37: **return** solução encontrada {não há garantia da otimalidade}
38: **else**
39: **return** solução ótima encontrada

Algoritmo 2.4: O algoritmo AWA* para heurística admissível.

### 2.3.6 Classe tempo-real

Os algoritmos desta classe são também referenciados na literatura como busca local ou busca centrada no agente [Koenig, 2001]. Reúne os algoritmos que podem realizar busca heurística na presença de restrições de tempo. Para conseguir atender a essas limitações, normalmente reduzem a busca a uma porção restrita do grafo em torno do agente e, conseqüentemente, não garantem a otimalidade das soluções. Além disso, diminuem a soma dos tempos de planejamento e execução com relação a métodos que primeiro determinam o plano para então o executarem [Koenig, 2001].

O uso dos algoritmo de busca heurística de tempo-real é conveniente em aplicações onde é necessário intercalar planejamento e execução. Nesses casos, é melhor agir rapidamente do que atuar de maneira ótima. Sua utilização também permite o tratamento de domínios com informação incompleta, pois na medida em que o plano é executado adquire-se mais informações sobre o ambiente e replaneja-se as ações.

O *Real-Time A** (RTA*) e *Learning Real-Time A** (LRTA*) [Korf, 1990] foram dois dos primeiros algoritmos de busca heurística de tempo-real propostos. Devido a sua relevância, o RTA* será detalhado na subseção 2.3.6.1. A diferença para o LRTA* está na forma como os *h-values* são modificados para permitir que sejam reaproveitados se o problema (mesmo espaço de estados e meta) for resolvido múltiplas vezes. O termo "*learning*" deve-se à convergência dos *h-values* (para os respectivos $h^*$*-values*) quando o problema é resolvido várias vezes. O algoritmo, portanto, "aprende" o caminho ótimo após diversas resoluções.

Como já mencionado, há um intercalamento de planejamento e execução. Abordagens tradicionais (RTA* e LRTA*, por exemplo), em cada uma dessas etapas, realizam uma nova busca beneficiando-se da atualização dos *h-values*. O algoritmo *Time-Bounded A** (TBA*) [Björnsson et al., 2009] propõe preservar as estruturas de dados *open* e *closed* durante as etapas da busca como forma de evitar a expansão do mesmo nó diversas vezes. Omitindo alguns detalhes, ele procede como o A* mas controla o número de expansões efetuadas para garantir a satisfação das restrições de tempo. As ações a serem exercidas pelo agente são determinadas com base no estado mais promissor na estrutura de dados *open*. Dessa forma, foi capaz de computar soluções com a mesma qualidade das calculadas pelo LRTA* empregando um tempo substancialmente inferior.

Koenig & Likhachev [2006b] empregam a mesma técnica utilizada no algoritmo AA* (subseção 2.3.2) para atualizar os *h-values* no algoritmo *Real-Time Adap-*

*tive A** (RTAA*). A contribuição, portanto, é uma nova forma de alterar os *h-values* que resulta num ganho de desempenho com relação ao LRTA* (segundo experimentos realizados pelos autores). De modo simplificado, o algoritmo A* é empregado como uma espécie de subrotina. Limita-se o número de expansões para garantir um comportamento de tempo-real. A execução do A* é interrompida se a meta foi encontrada ou se o tempo disponível foi esgotado. A heurística dos nós em *closed* é aprimorada e o agente executa ações para levá-lo à meta ou ao estado mais promissor de *open*.

Conforme explicado anteriormente, o algoritmo LRTA* "aprende" o caminho ótimo após várias resoluções do mesmo problema. Ou seja, os *h-values* convergem para os respectivos $h^*$*-values* quando o problema é resolvido diversas vezes. As atualizações realizadas ao longo desse processo nos *h-values* são importantes. Elas permitem que as computações futuras calculem soluções de melhor qualidade, demandando um número menor de iterações. Marques et al. [2011] propõe a utilização de abordagens paralelas[14] para diminuir o tempo necessário a essa convergência. De modo simplificado, emprega-se buscas auxiliares, executadas em paralelo, sem a restrição de tempo-real. Todas elas compartilham o aprendizado adquirido com a busca principal. Essa é a única busca que respeita as restrições de tempo impostas pela aplicação. Nos experimentos realizados pelos autores, houve uma expressiva diminuição no tempo total para a convergência.

Nos jogos digitais, os recursos computacionais são compartilhados por vários módulos do jogo: gráfico, sons, simulação, inteligência artificial e outros. Estilos de jogos digitais com características de tempo-real como *Real-time Strategy* (RTS) mantém a simulação a uma velocidade constante ao longo do tempo independentemente das ações dos jogadores. Uma das ações mais comuns nesse estilo de jogo é a movimentação de unidades pelo cenário. O jogador maneja as unidades através de comandos de alto nível e elas movimentam-se para o ponto escolhido desviando-se dos obstáculos existentes. Como o ambiente do jogo é dinâmico e parcialmente observável, pode ser necessário ajustar o planejamento durante sua execução. Alguns trabalhos [Koenig & Likhachev, 2006b; Bulitko et al., 2010] discutem a adoção de algoritmos de busca heurística de tempo-real para esse contexto. Infelizmente, como a maioria dos jogos são produtos comerciais, não é possível dizer quais abordagens utilizam ou compará-las com essas propostas. Mas ainda sim são uma boa motivação.

Ainda no contexto de jogos digitais, Pizzi et al. [2010] empregaram o algoritmo

[14]Esse trabalho também pertence à classe paralela (subseção 2.3.4).

RTA* para geração automática de soluções para fases (níveis) em forma de *storyboards*. O objetivo foi criar uma ferramenta de auxílio aos envolvidos na criação de jogos. Para a utilização do sistema é necessário primeiramente formular um problema de busca que representa o nível do jogo. A primeira etapa é a descrição das ações que podem ser realizadas pelo jogador e suas precondições, o estado inicial e o objetivo. Então, um planejador de tempo real que utiliza o RTA* gera a solução. A intercalação entre planejamento e execução é interessante nesse domínio, pois o ambiente é dinâmico. A proposta foi implementada e testada em um jogo comercial.

#### 2.3.6.1 RTA*

O *Real-Time A** (RTA*) [Korf, 1990] é um dos algoritmos de busca heurística de tempo real precursores. O objetivo do trabalho foi aplicar técnicas comuns na busca com adversários em algoritmos de busca para um único agente. Destacam-se: o horizonte de busca limitado e a realização das ações em intervalos de tempo constantes. A possibilidade de ajustar o tamanho do horizonte ou mesmo o número de nós expandidos antes que a decisão seja tomada permite que o algoritmo atenda a restrições de tempo severas exigidas por aplicações de tempo-real.

Várias modificações e alterações podem ser incorporadas ao RTA*. Com base nelas, surgiram novos algoritmos de busca heurística de tempo real. Restringe-se, no entanto, as discussões ao escopo da versão original que também é a mais simples.

Como já mencionado, o RTA* intercala planejamento e execução. A cada ciclo do algoritmo, calcula-se qual é a ação que levará a $t$ com o menor custo. Para limitar o tempo necessário ao cômputo dessa resposta, uma pequena porção do grafo ao redor do nó que representa o estado atual é explorado. Para os nós que estão na fronteira dessa exploração, a heurística é empregada para estimar o custo do caminho mais curto até $t$. Ou seja, para cada um desses vértices (denotado por $x$) o RTA* procede de forma semelhante ao A*: combina o custo do menor caminho partindo do nó que representa o estado atual até $x$ e a estimativa do custo do caminho ótimo de $x$ até $t$ através de uma soma.

Os custos são, então, propagados de baixo para cima na árvore de busca até que cada um dos vizinhos diretos do nó que representa o estado atual contenha a estimativa do custo do caminho ótima dele até $t$. A heurística do vértice no qual encontra-se o agente é atualizada com o custo da segunda melhor ação, e executa-se a ação correspondente ao vizinho com a menor estimativa. A importância de realizar essa atualização é evitar que o agente efetue os mesmos passos infinitamente. Ainda sim, isso poderá acontecer se $t$ não for alcançável.

Claramente o RTA* não é admissível, pois o uso de informação limitada nem sempre leva às melhores decisões. No entanto, ele é completo se o grafo é finito, com arestas de custo positivo, a heurística retorna somente valores finitos e $t$ pode ser atingido a partir de qualquer nó. Korf [1990] trata essa questão formalmente.

### 2.3.7 Considerações finais

Nesta seção, discutiu-se uma possível classificação dos algoritmos de busca heurística baseados no A*. Ela é composta de seis classes não excludentes entre si: bidirecional, incremental, *memory-concerned*, paralela, *anytime* e tempo-real. A fim de resumir graficamente toda a explicação, apresenta-se agora duas figuras. A figura 2.5 contém um esquema com todas as seis classes e os seus respectivos subtipos. Já a figura 2.6 expõe exemplos de extensões do A*, as classes as quais pertencem e a interferência que sofreram de outros algoritmos. Nessa figura, os algoritmos são organizados cronologicamente de acordo com a data na qual foram propostos.

Por se tratar de um algoritmo de muita relevância à Inteligência Artificial e proposto há mais de quatro décadas, o número de extensões do A* é grande. Não foi a intenção desta seção, pois, esgotar toda a literatura. Na escolha dos trabalhos contemplados, tiveram preferência as contribuições que inauguraram novas frentes de pesquisa e os algoritmos que apresentam alguma vantagem sobre o A*. Além disso, algumas extensões do A* não constam nesta classificação. Dentre elas, destacam-se duas abordagens expostas a seguir. O principal motivo pelo qual não foram incluídas na classificação foi o pequeno número de trabalhos a elas relacionados. Como as idéias foram introduzidas recentemente, poucos autores atestam o seu funcionamento. Além disso, a parcela de algoritmos cobertos por essas estratégias é pouco significativa se comparada com a respectiva quantia das outras seis classes que compõe a classificação.

Uma das abordagens não incluída na classificação é a hierárquica. A idéia central dessa estratégia é abstrair o grafo original através de um grafo menor e mais simples para reduzir o tempo de execução do A*. Cria-se, pois, dois níveis: um global (com menos detalhes) e um outro local (com toda a informação originalmente disponível). O grafo original é divido em subregiões para gerar o nível global. Vários vértices desse grafo são representados por um único nó do grafo do nível global, que delimita uma subregião do grafo original. Ao computar um caminho, emprega-se os grafos dos dois níveis em conjunto. A maior parcela da computação concentra-se no grafo do nível global que, por omitir vários detalhes, permite uma redução no tempo de execução. Um dos exemplos de algoritmo que adota essa abordagem é o

*Hierarchical Path-Finding A** (HPA*) [Botea et al., 2004].

A outra estratégia que também não foi contemplada na classificação trata do planejamento de ações para vários agentes cooperativos. Os desafios dessas aplicações são compartilhados por vários sistemas que também são distribuídos. São eles: disponibilidade apenas de informação parcial e existência de algum nível de incerteza (na comunicação, nas ações e/ou no conhecimento). Szer & Charpillet [2005] propõe utilizar o *Multi-agent A** (MAA*), uma extensão do A*, para resolver os *Decentralized partially-observable Markov decision problems* (DEC-POMDPs). Omitindo alguns detalhes, eles modelam o problema do planejamento no contexto de múltiplos agentes como um DEC-POMDP e o resolvem eficientemente através de uma busca heurística.

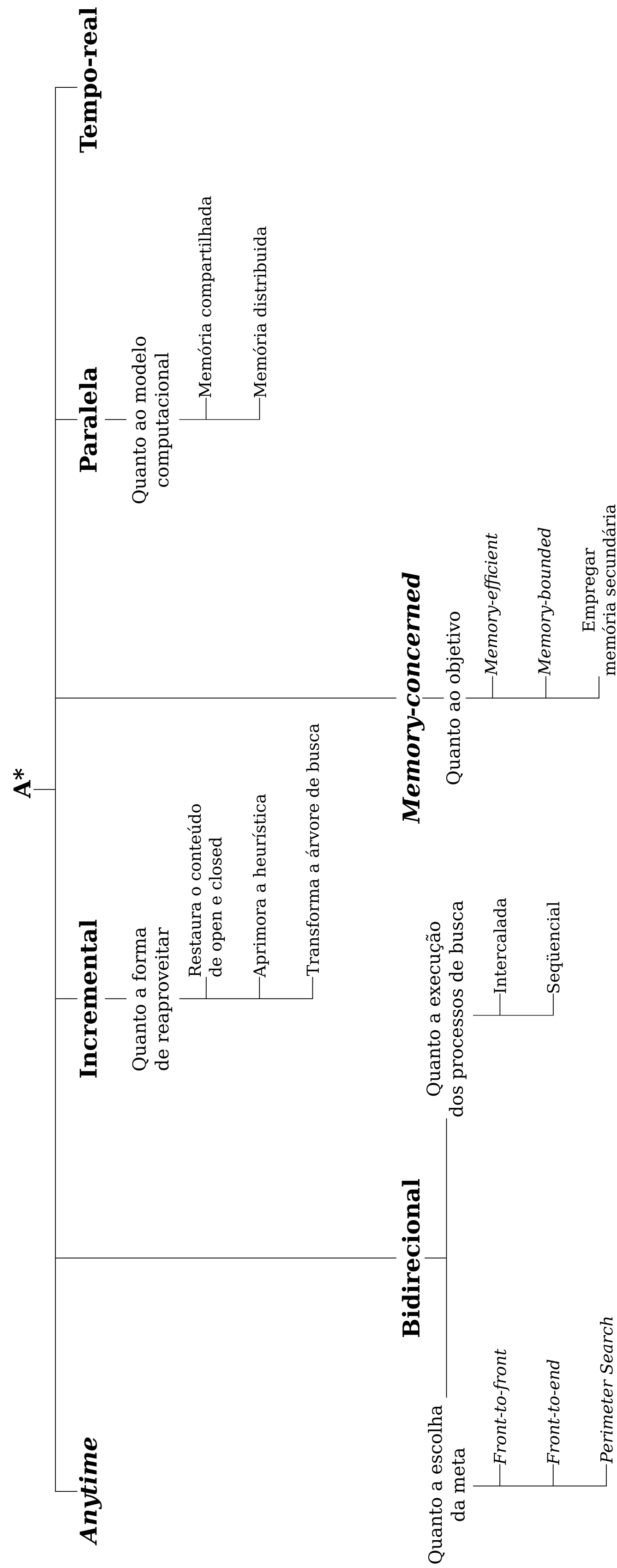



Figura 2.5: As seis classes que compõe a classificação dos algoritmos baseados no A* e os seus respectivos subtipos.

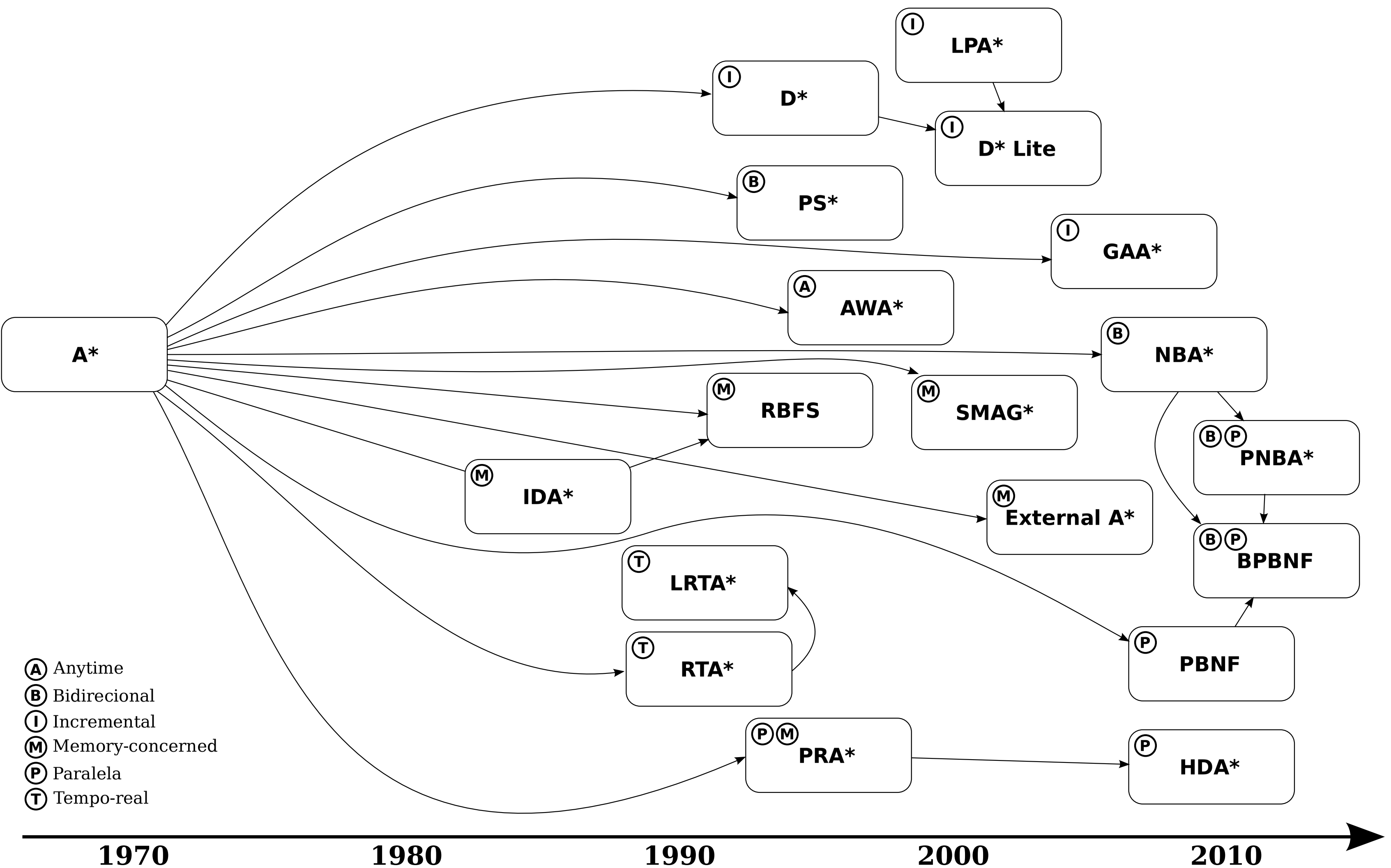



Figura 2.6: Exemplos de extensões do A* e as classes as quais pertencem. A interferência que os algoritmos sofreram é denotada através das setas, que apontam para o algoritmo influenciado. Os algoritmos estão organizados cronologicamente de acordo com a data na qual foram propostos.

## 2.4 Busca heurística bidirecional paralela

Outros trabalhos já investigaram as possibilidades de combinar as abordagens bidirecional e paralela na extensão do algoritmo de busca heurística A*. Nesta seção, serão discutidas as mais relevantes, comparando-as, sempre que possível, com os algoritmos apresentados nesta dissertação.

A maioria das versões bidirecionais paralelas do A* foram propostas para ambientes de memória distribuída. Uma exceção é o trabalho de Sohn [1993]. O *Parallel Bidirectional A* Search* (PBiA*S) explora o paralelismo existente na expansão dos nós. Como um algoritmo de busca bidirecional, mantém duas fronteiras. Os processadores são divididos em dois grupos que atuam na exploração das fronteiras de busca. A cada iteração, os nós mais promissores de cada fronteira são selecionados e expandidos paralelamente. Uma única estrutura de dados *closed* é compartilhada pelas duas buscas e, quando um nó já fechado pelo processo de busca oposto é encontrado, a computação termina. Os autores reportam um expressivo *speedup* nos domínios avaliados, principalmente no *8-puzzle*.

No entanto, deve-se avaliar mais cautelosamente esses resultados. Primeiramente, ao contrário dos dois algoritmos apresentados nos próximos capítulos, não há garantia da otimalidade da solução; pois a computação é interrompida quando as duas fronteiras se encontram. Por se tratar de um algoritmo não admissível, não faz sentido compará-lo ao A* nesse contexto. O *speedup* foi calculado utilizando a versão do PBiA*S com apenas um processador, não sendo possível afirmar que ele reduz o tempo de execução com relação ao A*. Além disso, abordagens simples para expansão de nós em paralelo, como discutido na subseção 2.3.4, podem piorar o desempenho com relação a versões seqüenciais devido a necessidade excessiva de sincronização. Os domínios onde o tempo para expansão de um vértice é muito pequeno se comparado a outras operações realizadas estão mais suscetíveis a esse problema. Uma vez que não são expostos detalhes da implementação, não é possível compreender como essa dificuldade foi superada, limitando as contribuições do estudo.

Uma abordagem utilizada para paralelização de algoritmos de busca heurística bidirecional emprega as chamadas ilhas (do inglês *islands*). As ilhas são nós do grafo que se comportam como metas secundárias e dividem o espaço de estados [Nelson & Toptsis, 1992]. Nesse contexto, são raízes das buscas que partem em cada um dos dois sentidos. Não é necessário que todas as ilhas participem do caminho que liga $s$ até $t$. No entanto, se no momento de eleger as ilhas não houver garantias da sua presença no menor caminho que conecte o nó inicial à meta, os

algoritmos não serão admissíveis.

O algoritmo *Parallel Bidirectional A** (PBA*) [Nelson & Toptsis, 1992] explora o paralelismo provido por essa divisão do espaço de estados. Para cada uma das ilhas, duas buscas são conduzidas: uma na direção de $t$ e a outra na direção de $s$. Partindo de $s$ e $t$, o grafo também é explorado nas duas direções. Todas essas buscas podem ser conduzidas paralelamente por diferente elementos computacionais. A computação termina quando um caminho de $s$ até $t$ é estabelecido de algum modo (não é necessário que envolva todas as ilhas). Dois pontos são críticos no algoritmo: detecção de nós explorados por duas buscas e seleção das ilhas. Embora haja um *speedup* significativo com relação à versão não paralela e ainda mais expressivo com relação ao A*; não há garantias da solução ótima.

Al-Ayyoub [2005] introduz uma forma genérica de paralelizar abordagens bidirecionais semelhantes ao *Perimeter Search* (subseção 2.3.1) para ambientes computacionais de memória distribuída. Como no *Perimeter Search*, é necessário gerar o perímetro antes de iniciar a busca. A proposta é gerar dois perímetros ao redor de $s$ e $t$ empregando uma busca em largura. De acordo com o número de elementos computacionais disponíveis, pode-se, além de realizar a construção dos perímetros ao mesmo tempo, dividir o trabalho da geração de um perímetro.

Os elementos computacionais são repartidos em dois grupos e cada um deles realizará a busca (com o uso do A*) em uma direção. Os nós dos perímetros são distribuídos de acordo com a direção na qual será praticada a exploração partindo desses vértices. Cada entidade mantém suas próprias estruturas de dados *open* e *closed*. O autor admite que isso pode levar à duplicação desnecessária do trabalho; mas explica que, devido às características do ambiente computacional, o esforço necessário para evitar esse problema é grande.

Como no *Perimeter Search*, a heurística é calculada considerando o custo estimado do caminho ótimo até os nós no perímetro oposto e o custo do menor caminho dos nós do perímetro oposto até sua raiz. Para garantir o cálculo da melhor solução, a computação só pode terminar quando os elementos computacionais em conjunto concluírem não ser praticável uma melhoria na solução. O elemento computacional que encontra uma solução dissemina seu custo para que cada um compare esse valor com o menor $f$*-value* existente em sua estrutura de dados *open*. Se para todos os elementos computacionais, o menor $f$*-value* de *open* é maior ou igual ao custo da solução informada e a heurística é pelo menos admissível, a computação pode cessar.

Um avaliação empírica foi conduzida no domínio do *15-puzzle*. Nos resultados obtidos, o algoritmo foi mais rápido do que a versão seqüencial correspondente.

Também foram realizadas comparações com uma abordagem unidirecional equivalente. O tempo de execução da versão bidirecional foi menor do que o tempo de execução da versão unidirecional especialmente para configurações mais difíceis (cujo número de movimentos para solução ótima é maior). Outra vantagem da estratégia bidirecional foi a capacidade de solucionar configurações que não puderam ser resolvidas por alternativas unidirecionais por causa da quantidade de memória necessária. Infelizmente, não foi realizada uma comparação com o A* tradicional diretamente.

Em suma, essas abordagens diferem bastante dos algoritmos introduzidos nos próximos capítulos deste trabalho; pois não garantem a otimalidade da solução computada e/ou são voltados para ambientes computacionais de memória distribuída. Por conseguinte, não foram comparadas diretamente com os novos algoritmos propostos nas avaliações experimentais realizadas.

# Capítulo 3

# PNBA*

Na seção 2.3, discutiu-se várias formas de diminuir o tempo de execução do A*. Uma delas é através da paralelização onde mais de um elemento computacional é empregado ao mesmo tempo na busca; diminuindo, conseqüentemente, o tempo de execução total. Outro modo envolve a utilização de abordagens bidirecionais através das quais a exploração do grafo é divida entre dois processos de busca. A redução no tempo de execução existe quando a soma do tamanho (número de nós) das duas árvores de busca é menor do que o tamanho de uma única árvore de busca de uma abordagem unidirecional.

Como a causa da redução nessas estratégias é diferente, é natural que surja a pergunta: elas podem ser combinadas para reduzir ainda mais o tempo de execução do A* sem violar a admissibilidade? A resposta é sim de acordo com este trabalho que introduz dois novos algoritmos de busca heurística bidirecional paralela: o PNBA* e BPBNF. Neste capítulo, será apresentado o PNBA*. Ele é a paralelização de um algoritmo de busca heurística bidirecional baseado no A*.

Este capítulo é constituído de duas seções. A primeira descreve o algoritmo PNBA*. A seção seguinte detalha a metodologia e expõe os resultados da avaliação empírica realizada. Ela também discute os resultados desses experimentos, promovidos a fim de medir o desempenho do PNBA* e de compará-lo com outros algoritmos. Os três diferentes domínios empregados também são explicados.

## 3.1 Descrição do algoritmo

O *Parallel New Bidirectional A** (PNBA*) [Rios & Chaimowicz, 2011] é uma implementação paralela do NBA* (subseção 2.3.1.1) para ambientes computacionais de memória compartilhada e explora as oportunidades de paralelização existentes nele. A

idéia é executar os dois processos de busca em paralelo ao invés de alternar as ações correspondentes aos seus turnos. O favorecimento à paralelização deve-se ao pequeno conjunto de variáveis consultadas por todos processos de busca: os $g_p$*-values*, $h_p$*-values*, $F_p$, $\mathcal{M}$ e $\mathcal{L}$. Somente as duas últimas podem ser alteradas por ambos.

O algoritmo 3.1 apresenta o pseudocódigo do PNBA*. A primeira parte inicia as variáveis do algoritmo. O trecho seguinte apresenta as ações do processo de busca 1. Elas são executadas paralelamente as atividades do processo de busca 2 (que podem ser obtidas através da substituição adequada dos subscritos: de 1 para 2 e vice-versa). O último trecho é executado quando os dois processos de busca são concluídos. Sua função é retornar a solução da busca.

O código é muito semelhante à versão do NBA* introduzida na subseção 2.3.1.1 e assume que as atribuições a tipos primitivos da arquitetura são realizadas de forma atômica. A atomicidade das atribuições implica que, se forem realizadas leituras durante a atribuição a uma variável, ou será recuperado o valor antigo ou o novo. Não é possível, portanto, a leitura de uma combinação dos valores antigo e novo. Essa necessidade justifica-se através da observação de que os $g_p$*-values*, $\mathcal{L}$ e $F_p$ podem ser lidos enquanto o outro processo de busca os atualiza.

A maioria das outras diferenças está relacionada com as variáveis compartilhadas. Uma delas é a adição de *finished*, comum aos dois processos de busca, cuja função é indicar quando a computação deve terminar. A estrutura de dados que representa $\mathcal{M}$ é também compartilha e deve ser capaz de lidar com acesso concorrente[1]. A atualização da variável $\mathcal{L}$ (linhas 27-32) é outra mudança introduzida no algoritmo. Primeiramente, uma verificação é feita fora da seção de exclusão mútua. Em seguida, se há alguma chance de existir um valor melhor para $\mathcal{L}$, a comparação e atualização são feitas dentro da seção de exclusão mútua para garantir a atomicidade da operação (do ponto de vista dos processos de busca).

Outra alteração importante foi o deslocamento da atribuição $\mathcal{M} \leftarrow \mathcal{M} \backslash \{x\}$ (linha 33), originalmente presente no NBA* após a linha 19. Sem essa mudança, o PNBA* poderia ignorar a conexão entre nós dependendo da seqüência da execução paralela das sentenças. Por exemplo, suponha a existência de um grafo com apenas dois nós $s$ e $t$, o nó inicial e o final respectivamente. Considerando a presença de uma aresta entre eles e que o conteúdo da linha 33 está logo após a linha 19 (como no algoritmo original), o processo de busca 1 remove $s$ de $open_1$, depois de $\mathcal{M}$ e sua execução é suspensa. O processo de busca 2 realiza operações equivalentes com o

[1]Os detalhes desse tratamento foram omitidos com intuito de facilitar o entendimento e de tornar mais sucinto o pseudocódigo. Na subseção 3.2.3, discute-se uma possível abordagem para essa questão.

```
1: {Procedimento para iniciar variáveis do PNBA* (serial):}
2: M ← ∅
3: L ← ∞
4: finished ← false
5: for all nós x do grafo que será explorado do
6:    g1(x) ← ∞
7:    g2(x) ← ∞
8:    M ← M ∪ {x}
9: for all p ∈ {1,2} do
10:    gp(sp) ← 0 {prepara o nó inicial e a fronteira do processo de busca p}
11:    fp(sp) ← gp(sp) + hp(sp)
12:    open_p ← {sp}
13:    Fp ← fp(sp)
14:
15: {Código executado pelo processo de busca 1 (paralelo):}
16: while ¬finished do
17:    x ← argmin{f1(n) | n ∈ open_1}
18:    open_1 ← open_1\{x}
19:    if x ∈ M then
20:       if (f1(x) < L) ∧ (g1(x) + F2 − h2(x) < L) then
21:          for all arestas (x,y) do grafo explorado pelo processo de busca 1 do
22:             if (y ∈ M) ∧ (g1(y) > g1(x) + d1(x,y)) then
23:                g1(y) ← g1(x) + d1(x,y)
24:                f1(y) ← g1(y) + h1(y)
25:                open_1 ← open_1 ∪ {y}
26:                parent_1(y) ← x
27:                if g1(y) + g2(y) < L then
28:                   lock
29:                   if g1(y) + g2(y) < L then
30:                      L ← g1(y) + g2(y)
31:                      meeting_node ← y
32:                   unlock
33:       M ← M\{x}
34:    if open_1 ≠ ∅ then
35:       F1 ← min{f1(n) | n ∈ open_1}
36:    else
37:       F1 ← ∞
38:       finished ← true
39:
40: {Após o término de ambos processos de busca (serial):}
41: if L = ∞ then
42:    return não existe solução
43: else
44:    return solução ótima encontrada
```

Algoritmo 3.1: O algoritmo PNBA* para uma heurística consistente.

nó $t$, expande $t$ gerando $s$ mas não é capaz de atualizar $g_2(s)$ e $\mathcal{L}$ porque $s \notin \mathcal{M}$. O processo de busca 1 retoma sua execução, expande $s$ gerando $t$, mas não é capaz de alterar $g_1(t)$ e $\mathcal{L}$ porque $t \notin \mathcal{M}$.

O deslocamento da sentença $\mathcal{M} \leftarrow \mathcal{M} \backslash \{x\}$ foi suficiente para que essa interação paralela entre os dois processos de busca não comprometesse o algoritmo. Antes de colocar um nó em $\mathcal{M}$ e, conseqüentemente, forçar o outro processo de busca a ignorá-lo, todos os seus sucessores/predecessores são gerados. Assim, para atingir um nó fechado por um processo de busca, o processo de busca oposto terá necessariamente passado pelos sucessores/predecessores desse nó que já terão sido gerados. A desvantagem trazida é a possibilidade da expansão desnecessária de nós que, no entanto, não viola a otimalidade da solução como será discutido a seguir.

Apesar de não serem apresentadas neste trabalho as provas da admissibilidade do PNBA*, sustenta-se a existência dessa propriedade no algoritmo em razão dos testes realizados (incluindo a avaliação empírica) e da argumentação aqui exposta. Primeiramente, é importante notar que as provas da admissibilidade do NBA* não fazem nenhuma suposição acerca da freqüência com que as ações de cada turno são executadas. Ou seja, um eventual desbalanceamento na execução dos processos de busca não causa a perda da otimalidade da solução.

A atenção, portanto, deve estar voltada para as variáveis compartilhadas pelos dois processos. Com o paralelismo, pode haver a utilização de uma informação desatualizada. As variáveis $F_p$, $\mathcal{L}$ e $\mathcal{M}$ são empregadas pelo PNBA* para reduzir o esforço computacional da busca. Na linha 20 uma expressão com $F_p$ e $\mathcal{L}$ decide se um nó deve ser expandido. A leitura de um valor desatualizado e o seu posterior uso nessa expressão levam apenas a expansão desnecessária de vértices. A explicação está na monotonicidade crescente e decrescente - respectivamente - dos valores dessas variáveis ao longo da computação. A primeira é conseqüência da consistência da heurística e a outra garantida pela forma como os novos valores são atribuídos a $\mathcal{L}$. Logo, se é possível podar um nó com os valores atuais de $F_p$ e $\mathcal{L}$ também será no futuro para todos os valores das atualizações.

O conjunto $\mathcal{M}$ evita a expansão e a inserção de nós nas estruturas de dados $open_p$. Após iniciar as variáveis, o número de vértices do grafo em $\mathcal{M}$ é monotonicamente decrescente. Ou seja, a partir desse momento nenhum dos processos de busca podem adicionar nós em $\mathcal{M}$, somente removê-los. Como nos casos anteriores, a utilização de $\mathcal{M}$ desatualizado causará apenas expansões desnecessárias e não prejudiciais à admissibilidade do PNBA*.

Essas características asseguram que o PNBA* não vai podar nós que não devam ser podados. Mas o oposto acontece, vértices que deveriam ser podados são

expandidos. Os efeitos possíveis de uma expansão são (linhas 21-32): atualização dos $g_p$-*values* dos sucessores de um nó, colocação dos sucessores em $open_p$ e alteração de $\mathcal{L}$. Tanto os $g_p$-*values* quanto $\mathcal{L}$ só são alterados se o novo valor é estritamente menor que o antigo. Não há, pois, como prejudicar o cálculo já efetuado. Se o vértice sucessor é adicionado a $open_p$, ele estará sujeito a essas mesmas regras caso seja expandido. Portanto, o único efeito da expansão de um nó que deveria ser podado é o aumento desnecessário do tempo de execução.

Por fim, os $g_p$-*values* desatualizados podem ser empregados nos cálculos de $\mathcal{L}$. No entanto, em algum momento antes que a computação termine, o valor de $\mathcal{L}$ será corrigido, pois uma alteração nos $g_p$-*values* é sempre seguida de uma modificação em $\mathcal{L}$. Logo, se uma mudança em $g_1(x)$ é sucedida por uma atribuição a $\mathcal{L}$ de um novo valor calculado através de $g_2(x)$ que estava prestes a ser atualizado; o processo de busca 2 armazenará em $\mathcal{L}$ o valor apropriado, porque ele terá disponível (após a atribuição a $g_2(x)$) os valores mais atuais de $g_1(x)$ e $g_2(x)$.

## 3.2 Avaliação empírica

Com o intuito de avaliar empiricamente o PNBA* e compará-lo com os algoritmos A* e NBA*, foram conduzidos experimentos em três domínios cuja utilização no contexto de busca heurística é muito comum: *pathfinding* em *grids* de conectividade quatro com arestas de custo uniforme e não uniforme e no *15-puzzle*. Eles são descritos respectivamente nas duas próximas subseções (3.2.1 e 3.2.2). A subseção 3.2.3 descreve os experimentos e expõe procedimentos realizados. Ao final, a subseção 3.2.4 apresentada os resultados e a sua discussão.

### 3.2.1 *Pathfinding* em *grids*

O *grid*[2] é uma representação discreta de um ambiente bidimensional. É formado por células com o mesmo tamanho. As células comumente possuem associadas a elas atributos relacionados com a porção do ambiente que representam. A variação na resolução do *grid* (número de células adotadas) permite controlar a quantidade de detalhes e, conseqüentemente, a quantidade de informação necessária para representá-lo. Exemplos de utilização encontram-se em áreas como Jogos Digitais, Computação Gráfica e Robótica.

[2]Uma possível tradução de *grid* é grade. No entanto, como o termo em inglês está bem estabelecido na Ciência da Computação e sua tradução pode confundir, optou-se por não traduzi-lo. O mesmo vale para o nome do domínio.

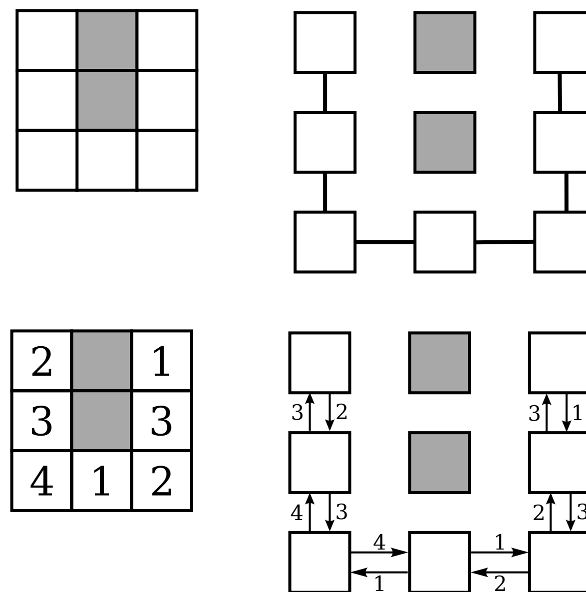



Figura 3.1: Dois *grids* de conectividade quatro com arestas de custo uniforme e não uniforme. À direita de cada um deles estão seus respectivos grafos. As células cuja cor de fundo é cinza estão bloqueadas.

No contexto do domínio do *pathfinding* em *grids*, algumas células estão bloqueadas, e o agente não pode se deslocar através delas. O restante possui um custo associado, que pode ser uniforme (igual para todas as células) ou não uniforme. A partir de uma célula, o agente pode se mover para quatro ou oito células vizinhas (que não estejam bloqueadas). Se o *grid* possui conectividade quatro, as direções permitidas são: para cima, para baixo, para esquerda e para direita. No caso de conectividade oito, as diagonais também são válidas.

Logo, o *grid* define um grafo onde as células são os nós e a conexão entre elas as arestas (figura 3.1). A presença de uma aresta entre dois nós representa a possibilidade do agente transitar entre os nós (que constituem as células) por ela conectados. Neste trabalho, particularmente, o custo de deslocamento entre as células está representado nas arestas. Ou seja, é como se o agente "pagasse" o custo ao utilizá-las. Dado uma célula inicial e final, o objetivo é computar o caminho com menor custo que leve o agente da célula inicial até a final.

### 3.2.2 *n-puzzle*

O *n-puzzle* consiste de peças numeradas de 1 até $n$ dispostas em um quadrado cujo lado é de tamanho $\sqrt{n+1}$. Uma das posições fica vazia e é através dela que pode-se deslocar as peças em quatro direções (para cima, para baixo, para esquerda e para

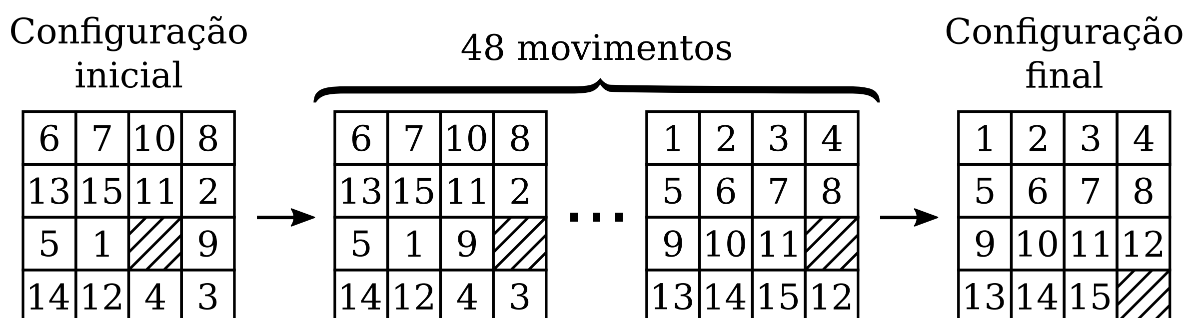



Figura 3.2: À direita o estado final do *15-puzzle* e à esquerda uma configuração cujo número mínimo de movimentos necessários para transformá-la no estado final é 48. Entre elas são mostradas duas configurações intermediárias que surgem em conseqüência dos movimentos das peças necessários para realizar a transformação.

direita) de acordo com a posição da peça. Como $\sqrt{n+1}$ deve ser um número inteiro, as instâncias de *n-puzzle* disponíveis são: *8-puzzle*, *15-puzzle* (figura 3.2), *24-puzzle*, ... Recebe-se as peças desorganizadas (dispostas em uma configuração aleatória) e o objetivo e ajustá-las com o menor número de movimentos de maneira a obter o estado final (no qual estão ordenadas).

Esse problema vem sendo utilizado na literatura da área de Inteligência Artificial como forma de avaliar o desempenho de algoritmos de busca heurística. Algumas justificativas possíveis [Ratner & Warmuth, 1986] são:

1. Não existe um algoritmo que encontre a solução ótima eficientemente, ou seja, sem a necessidade de grandes recursos computacionais como o A* e algoritmos nele baseados;

2. O problema é fácil de descrever e de manipular;

3. É um bom representante de uma classe de problemas onde o objetivo é encontrar o menor caminho entre dois nós de um grafo de estados;

4. O tamanho do espaço de estados é exponencial em $\sqrt{n+1}$ - tamanho do lado do quadrado no qual estão inseridas as peças;

5. O espaço de estados é definido implicitamente por um conjunto simples de regras.

No entanto, a razão mais notória para utilização do *n-puzzle* na avaliação do desempenho de algoritmos de busca heurística é que encontrar o menor número de movimentos necessários para obtenção da configuração final é *NP-Difícil* [Ratner & Warmuth, 1986, 1990]. Mais precisamente, o problema de decisão "existe uma

solução que transforma a configuração inicial na configuração final e requer menos do que $k$ movimentos ?" é *NP-Completo*.

### 3.2.3 Metodologia

O *15-puzzle* é um tipo de problema cuja representação explícita do espaço de estados não cabe inteiramente na memória. Ao longo da computação, apenas a parcela necessária foi gerada e armazenada. Já para os outros dois domínios (*pathfinding* em *grids)*, assumiu-se caber inteiramente na memória toda representação do espaço de estados. Portanto, possuíam um grafo explícito associado a eles nas implementações efetuadas. Essa diferenciação é importante porque impactou diretamente na implementação e representa as duas situações mais comuns.

As implementações foram realizadas através da linguagem de programação C++ (compilador g++ versão 4.4.5) com auxílio da biblioteca *pthreads*. As técnicas de programação empregadas foram as mesmas em todos os algoritmos para garantir que as comparações realizadas sejam válidas. Os experimentos foram efetuados em uma máquina com processador Intel(R) Core(TM) i7 2,67GHz, com 12 gigabytes de memória principal e com sistema operacional Ubuntu (*kernel* Linux versão 2.6.35). O recurso do *hyperthreading* foi desabilitado para evitar o uso dos processadores virtuais. Assim, o sistema operacional dispunha de quatro processadores reais. Para tornar eficiente a alocação de memória em paralelo foi empregada a biblioteca TC-Malloc [Ghemawat & Menage, 2012].

Em todos domínios, uma *flag* foi mantida para indicar a presença de um nó em $\mathcal{M}$ (NBA* e PNBA*) ou em *closed* (A*). Como não houve uma representação explícita do espaço de estados no *15-puzzle*, uma tabela *hash* de dois níveis foi empregada para garantir a existência de apenas uma representação de cada estado durante a execução. Um *heap* binário implementou as estruturas de dados *open* e $open_p$ dos algoritmos.

A utilização de mais de um nível na tabela *hash* justifica-se pela necessidade de evitar contenções nos algoritmos paralelos. A tabela *hash* de dois níveis possuía um número fixo de subtabelas independentes. A função de mapeamento decidia qual das subtabelas seria utilizada e qual era a posição (*bucket*) do elemento nela. Para esses algoritmos, a estrutura de dados em discussão também implementou atomicamente as operações *has*, *get* e *insertIfAbsent*. Suas funções são, respectivamente, verificar a presença de um elemento, recuperar um elemento e inserir um elemento se ainda não estiver presente. O uso de dois níveis e de *buckets* visou evitar o bloqueio completo do acesso à estrutura durante essas operações. Para os algoritmos

que não necessitam de acesso concorrente, a mesma implementação foi empregada mas sem o custo dos mecanismos de sincronização.

Para avaliar os algoritmos no domínio do *15-puzzle*; uma base de configurações, organizada pelo comprimento (número de movimentos) da solução ótima, foi criada. Os algoritmos solucionaram configurações cujo comprimento variou de 48 até 58 em incrementos de dois. Para cada comprimento, computou-se a média aritmética e o desvio padrão para as 50 execuções com diferentes configurações (as mesmas entre os algoritmos).

No domínio do *pathfinding* em *grids*, uma série de *grids* quadrados foram gerados aleatoriamente. Uma célula era bloqueada com uma probabilidade de 33% e as células inicial e final foram posicionadas sempre nos cantos superior esquerdo e inferior direito respectivamente. A figura 3.3 traz um pequeno exemplo. O esforço para deslocar-se entre as células dos labirintos de custo não uniforme foram randomicamente escolhidos do intervalo fechado de 1 a 8. As dimensões do labirinto variaram de 6000 a 9000 em incrementos de 1000. Os dados coletados para cada tamanho também foram resultados da solução de 50 labirintos (aqueles sem solução foram ignorados até esse número ser atingido).

Em todos os domínios foi empregada uma heurística consistente: a distância de *Manhattan*. Os empates (nós com o mesmo $f$*-value*) foram resolvidos em favor daqueles com maior $g$*-value* - uma estratégia considerada satisfatória [Koenig & Likhachev, 2006b]. As métricas colhidas foram o tempo de relógio correspondente a execução e o número de expansões realizadas por cada algoritmo. A execução do algoritmo foi divida em três fases cujos tempos foram coletados separadamente: inicialização, execução e finalização. Elas correspondem respectivamente às atividades realizadas para iniciar as estruturas de dados e variáveis do algoritmo, à execução do laço principal da busca e às ações para liberar os recursos adquiridos. Nos problemas onde o espaço de estados foi representado explicitamente, a fase de inicialização foi o momento no qual os algoritmos construíram uma representação explícita do grafo adequada à busca que realizariam. Em algoritmos paralelos, utilizou-se nessa fase o mesmo número de processadores da fase de execução para realizar as tarefas de maneira mais rápida. No *15-puzzle*, a fase de finalização foi utilizada predominantemente para liberar a memória adquirida ao longo da busca para denotar a porção do grafo explorado. Nos outros domínios, liberou-se os recursos empregados na representação explícita do espaço de estados nessa fase.

Para facilitar a comparação, realizou-se o cálculo da média aritmética e do desvio padrão utilizando os valores absolutos e relativos. Os valores relativos foram obtidos para cada execução dos algoritmos para um mesmo problema considerando o

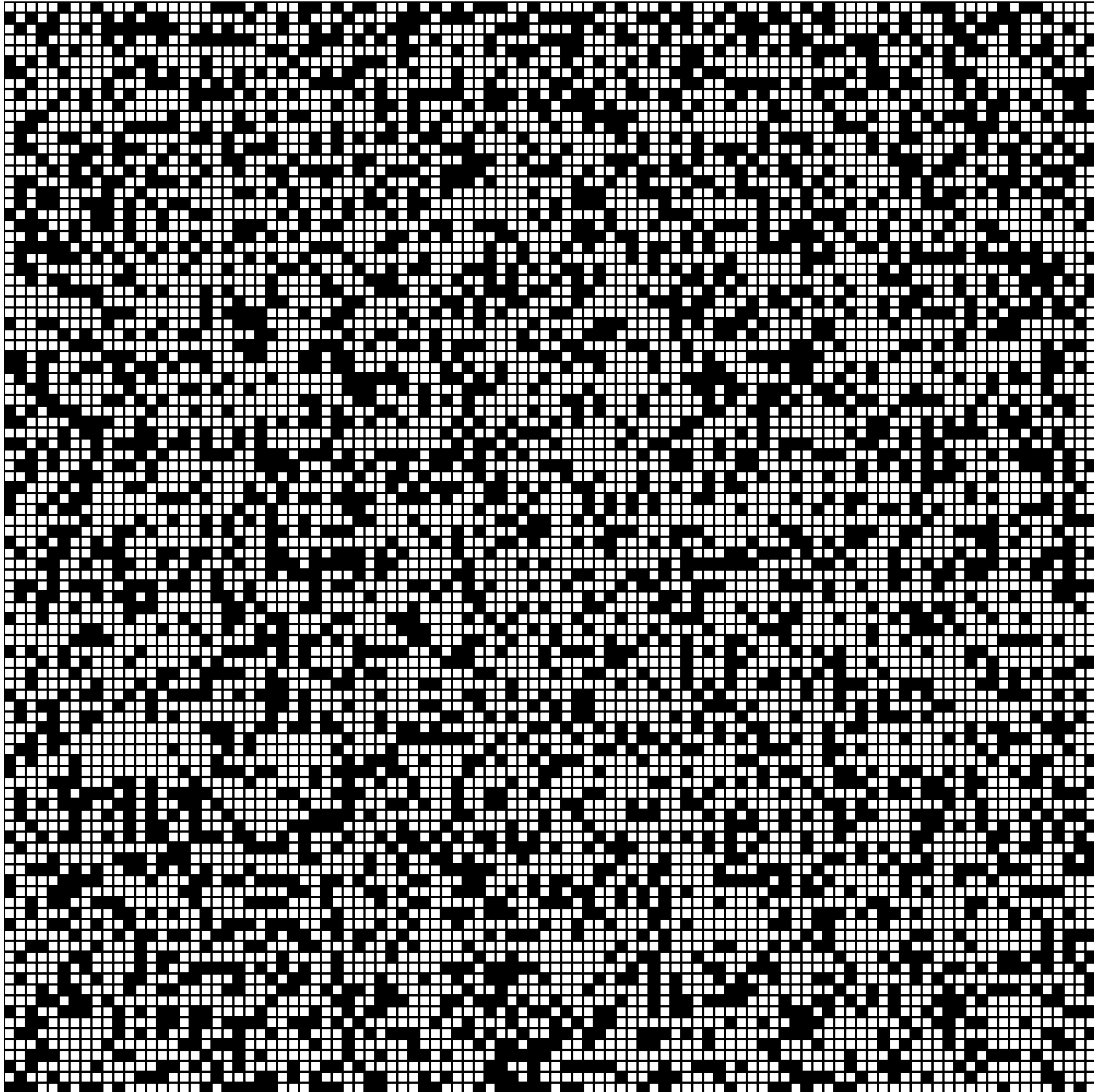


Figura 3.3: Exemplo de *grid* gerado automaticamente para os experimentos. Tanto sua largura quanto o seu comprimento são 100 e aproximadamente 33% de suas células estão bloqueadas (cor preta).

número de expansões e tempo de execução total do PNBA* como 100%.

### 3.2.4 Resultados e discussão

As figuras 3.4 e 3.5 mostram, respectivamente, o tempo de execução e o número de expansões dos experimentos realizados no domínio do *pathfinding* em *grids* com arestas de custo uniforme para várias dimensões de *grid*. Os dados também são apresentados nas tabelas 3.1 e 3.2. Nelas o desvio padrão está entre parênteses e nos gráficos é exibido com barras.

Claramente, o PNBA* possui o menor tempo de execução nesse domínio que é aproximadamente duas vezes mais rápido do que o NBA*. Em todos os *grids* resolvidos, ele foi o mais rápido. O NBA* apresentou o maior tempo na fase de inicialização por causa do tamanho da representação dos nós do grafo. Para os algoritmos NBA* e PNBA*, é necessário manter, além das informações empregadas pelo A*, informações de cada um dos dois processos de busca. Essas demandaram um trabalho extra no momento de criar e iniciar a representação do espaço de estados que seria explorado. No PNBA* isso foi atenuado através da paralelização dessa fase.

Na fase de execução, o A* demandou um tempo muito maior, pois expandiu muito mais nós do que o NBA* e o PNBA*. O número de nós expandidos pelos algoritmos bidirecionais foi muito próximo. A pequena diferença existente pode ser explicada pela variação aleatória gerada na execução em paralelo dos processos de busca. A desigualdade no tempo dessa fase dos algoritmos bidirecionais deve-se também a execução concorrente. Apesar de expandir praticamento o mesmo número de nós do que o NBA*, o PNBA* o fez utilizando mais de um processador para reduzir o tempo necessário. O tempo da fase de finalização representou uma porção insignificante do tempo total para todos algoritmos avaliados nesse domínio.

A paralelização obteve nesse domínio os resultados desejados. O PNBA* gastou aproximadamente metade do tempo do NBA* para calcular o menor caminho nos *grids* empregados. O número de expansões, todavia, manteve-se praticamente igual para esses algoritmos. Isso evidencia o sucesso da estratégia de paralelização adotada no PNBA*.

No domínio do *pathfinding* em *grids* com arestas de custo não uniforme, a superioridade do PNBA* também fica evidente. Os gráficos com o tempo total e o número de expansões estão respectivamente nas figuras 3.6 e 3.7. Os dados também são exibidos nas tabelas 3.1 e 3.2.

A primeira vista, ao comparar os tempos totais de execução com os respec-

tivos valores do domínio anterior, o desempenho do PNBA* em relação ao NBA* parece ser ainda melhor nesse contexto. Mas na realidade o *speedup* na fase de execução relativo ao NBA* foi praticamente o mesmo em ambos domínios para todas dimensões de *grid*. No entanto, como a duração da fase de execução foi maior nesse contexto, a sua influência no tempo total também aumentou. Dessa forma, o *speedup* do PNBA* em relação ao NBA* calculado com os tempos totais obtidos para esse domínio aproximou-se, mais do que no domínio anterior, do respectivo *speedup* computado considerando os tempos da fase execução. Conseqüentemente, a diferença entre os valores totais (do PNBA* e do NBA*) acentuou-se nesse domínio.

O *speedup* do PNBA* em relação ao NBA* calculado com os tempos da fase execução no *pathfinding* em *grids* (com arestas de custo uniforme e não uniforme) foi super linear. A razão central foi o aumento da disponibilidade de recursos e a forma como foram empregados. Com a utilização de dois processadores, a memória *cache* disponível ao PNBA* dobrou em relação à mesma quantia empregada pelo algoritmo NBA*. Além disso, a forma como esse recurso foi explorado no PNBA* favoreceu a ocorrência do *speedup* super linear. Como cada processador atuou em uma tarefa específica (execução de um processo de busca), a localidade de referência foi maior. Conseqüentemente, como o funcionamento de toda hierarquia de memória baseia-se nesse princípio, houve uma diminuição expressiva no tempo de execução.

Novamente, o número de expansões efetuadas pelos algoritmos bidirecionais foi equivalente. No entanto, a comparação dessas quantias com as respectivas do domínio anterior mostra um acréscimo em todos algoritmos avaliados. A causa dessa variação foi a dificuldade oferecida por esse problema. Como o custo das arestas não era uniforme, a heurística, para garantir a admissibilidade e consistência computou as estimativas levando em consideração apenas o menor peso. A conseqüência foi que seu cálculo subestimou o custo real por um fator maior.

Nota-se, entretanto, que a variação no número de expansões não foi proporcional entre os algoritmos. O valor do PNBA* e do NBA* aproximou-se consideravelmente do número de expansões realizadas pelo A*, se o domínio anterior for utilizado como referência. Ou seja, a busca bidirecional foi mais afetada pelo uso de uma heurística pobre.

As figuras 3.8 e 3.9 mostram, respectivamente, o tempo de execução e o número de expansões dos experimentos realizados no domínio do *15-puzzle*. Como nos outros domínios, os dados também são apresentados em tabelas (3.3 e 3.4) para facilitar a compreensão.

O desvio padrão encontrado foi muito elevado, principalmente para os valores absolutos. A causa foi a variação do erro total dos valores gerados pela heurística

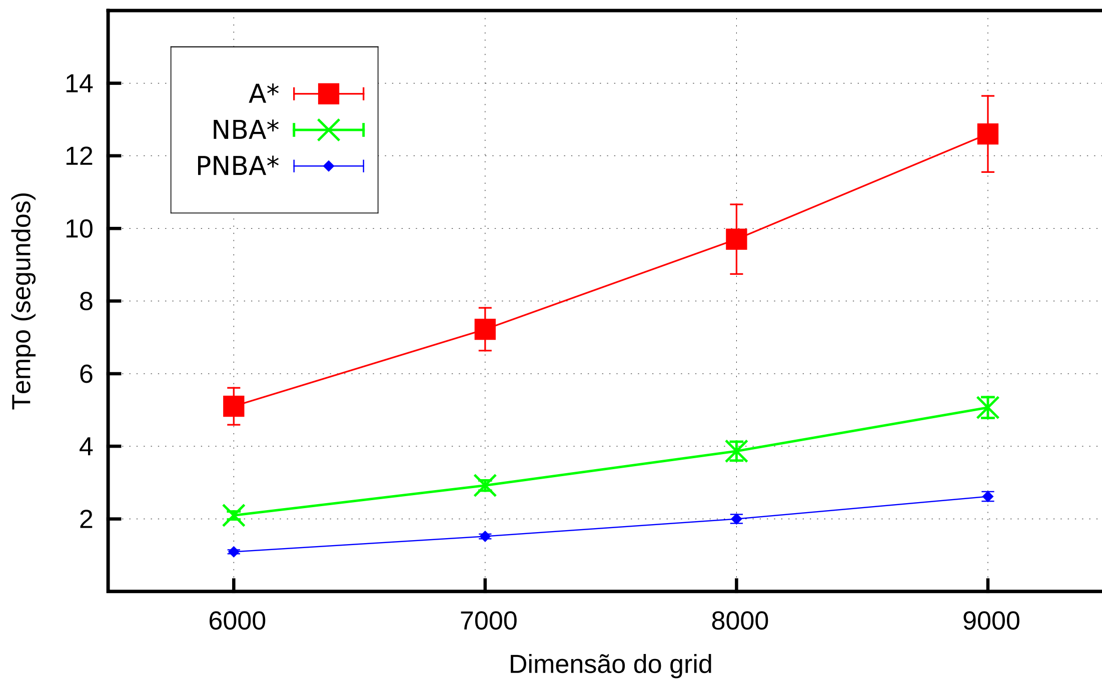
para resolução das configurações do *15-puzzle*. Algumas delas foram favorecidas já que uma parcela maior das estimativas computadas possuía um erro inferior. Assim, apesar da solução ótima dessas configurações possuir o mesmo comprimento; o esforço computacional para resolvê-las variou expressivamente.

Na grande maioria das vezes, o PNBA* resolveu as configurações do *15-puzzle* utilizando um tempo inferior. O pior resultado foi para configurações cuja solução ótima era composta de 48 passos (como mostra a coluna "# vezes foi o mais rápido" da tabela 3.4). Conforme a dificuldade média aumentou através do incremento do número de passos da solução de menor custo, o tempo de execução do PNBA* diminuiu em relação aos dois outros algoritmos. No entanto, a diferença entre PNBA* e NBA* foi muito inferior a encontrada nos domínios discutidos anteriormente. A razão principal foi a distinção das implementações. Como já explicado, nesse domínio a representação do espaço de estados foi gerada dinamicamente ao longo da computação. Para garantir uma representação única para cada estado, uma tabela *hash* foi empregada. Durante o tratamento do acesso concorrente, fez-se necessário algumas vezes o bloqueio da execução de um processo de busca até a conclusão da operação realizada pelo outro. A implementação da tabela *hash* com acesso concorrente mostrou-se crítica nesse caso.

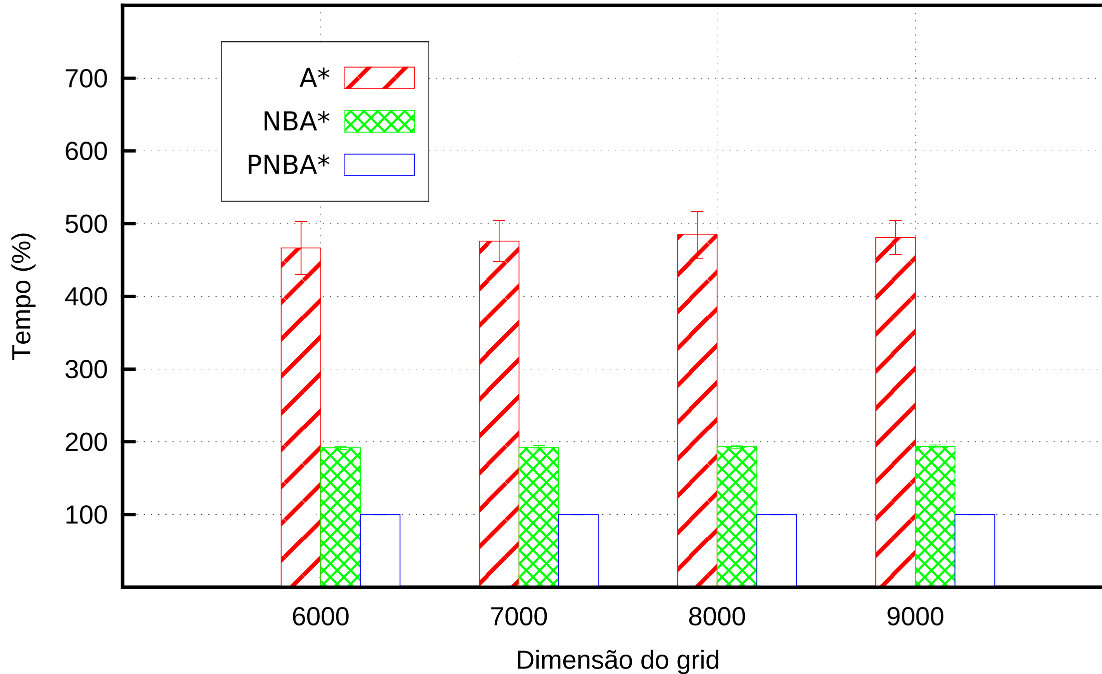
Há também outras variações com relação aos domínios já expostos. Uma delas é a duração da fase de finalização que foi muito maior para todos os algoritmos avaliados. Nessa fase, foi feita a liberação dos recursos adquiridos durante a execução para representar o espaço de estados. No entanto, diferentemente dos casos anteriores, como a aquisição foi feita de forma fragmentada o mesmo ocorreu com a liberação causando esse aumento no tempo de execução. Também foi possível perceber que a diferença entre o número de expansões do A* e dos algoritmos bidirecionais foi muito superior a encontrada nos experimentos realizados no domínio do *pathfinding* em *grids*.

Consoante aos resultados dos experimentos anteriores, o número de expansões do NBA* e PNBA* foi muito semelhante. A redução do tempo de execução novamente foi conseqüência da paralelização do algoritmo.

Os resultados em todos domínios avaliados demonstram claramente os benefícios do PNBA*: ampliação das vantagens da busca heurística bidirecional através da condução dos dois processos de busca em paralelo. O efeito foi uma redução no tempo de execução total. As alterações no NBA* que levaram à proposição do PNBA*, portanto, foram eficazes. Em dois dos três domínios, esse algoritmo empregou, para computar as soluções ótimas, aproximadamente a metade do tempo de execução utilizado pelo NBA*.




(a) Tempo total em segundos para os vários tamanhos de *grids* quadrados.

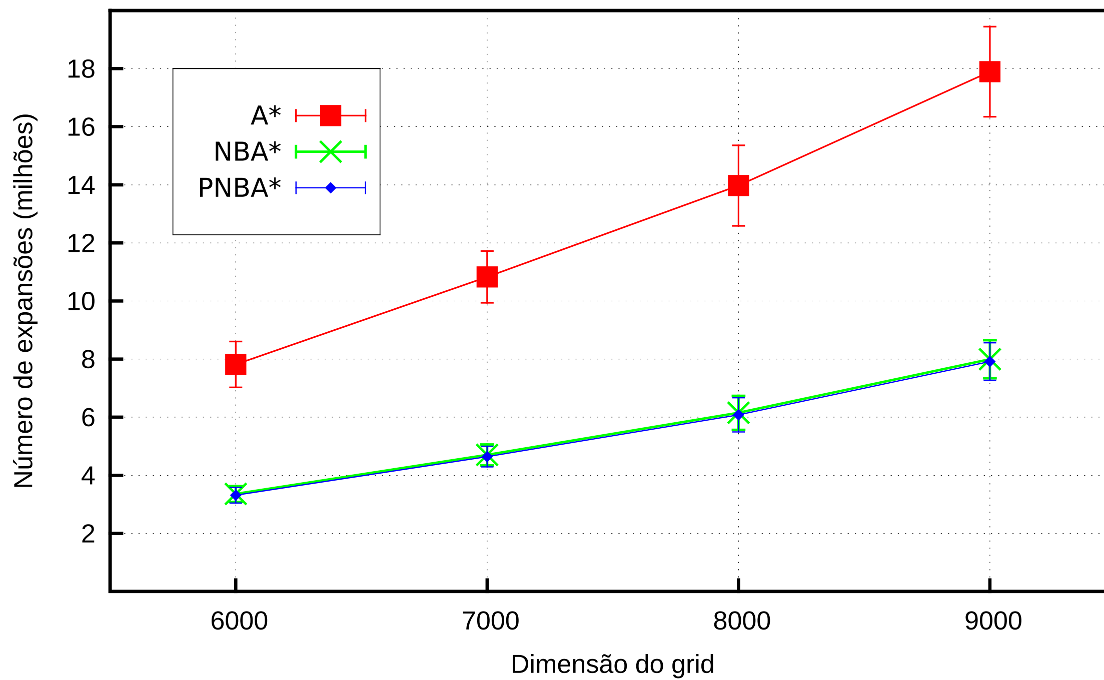



(b) Tempo total percentual para os vários tamanhos de *grids* quadrados.

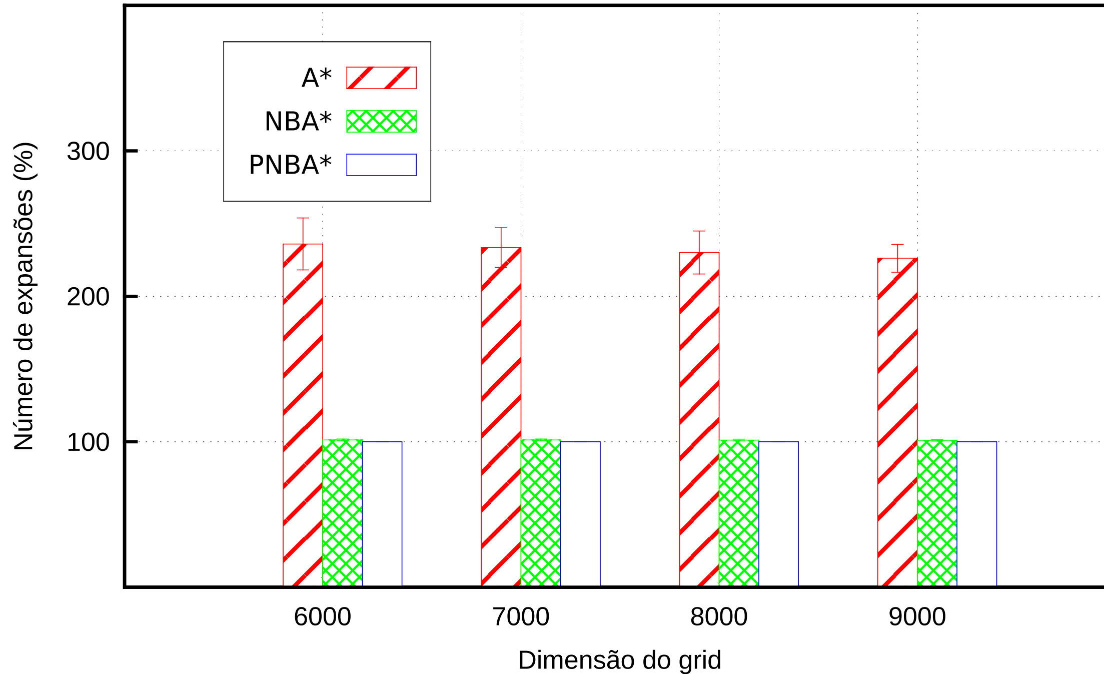
Figura 3.4: Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme.

Tabela 3.1: Tempo absoluto para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme.

| Dimensão do *grid* | Algoritmo | Tempo absoluto (segundos) | | | |
|---|---|---|---|---|---|
| | | Inicialização | Execução | Finalização | Total |
| *Pathfinding* em *grids* com arestas de custo uniforme | | | | | |
| 6000 | A* | 0,51069 (0,00515) | 4,58807 (0,50658) | 0,00113 (0,00002) | 5,09988 (0,50727) |
| | NBA* | 0,81681 (0,00891) | 1,27670 (0,11193) | 0,00078 (0,00031) | 2,09430 (0,11108) |
| | PNBA* | 0,48725 (0,00828) | 0,60450 (0,05175) | 0,00081 (0,00014) | 1,09255 (0,05265) |
| 7000 | A* | 0,69682 (0,00816) | 6,52496 (0,59005) | 0,00086 (0,00029) | 7,22264 (0,59064) |
| | NBA* | 1,11209 (0,01114) | 1,80502 (0,14022) | 0,00114 (0,00031) | 2,91826 (0,14131) |
| | PNBA* | 0,66317 (0,01161) | 0,85212 (0,06622) | 0,00102 (0,00024) | 1,51631 (0,06502) |
| 8000 | A* | 0,90490 (0,00915) | 8,79521 (0,95982) | 0,00104 (0,00031) | 9,70115 (0,95882) |
| | NBA* | 1,44360 (0,01583) | 2,42021 (0,26057) | 0,00125 (0,00034) | 3,86506 (0,26105) |
| | PNBA* | 0,85335 (0,01291) | 1,14557 (0,12483) | 0,00107 (0,00014) | 1,99999 (0,12202) |
| 9000 | A* | 1,13352 (0,01022) | 11,46479 (1,04969) | 0,00112 (0,00032) | 12,59943 (1,04924) |
| | NBA* | 1,82546 (0,01557) | 3,23862 (0,28705) | 0,00140 (0,00025) | 5,06548 (0,28624) |
| | PNBA* | 1,08471 (0,01596) | 1,53178 (0,13236) | 0,00118 (0,00012) | 2,61767 (0,13225) |
| *Pathfinding* em *grids* com arestas de custo não uniforme | | | | | |
| 6000 | A* | 0,51015 (0,00458) | 13,02088 (0,07102) | 0,00117 (0,00002) | 13,53219 (0,07032) |
| | NBA* | 0,81822 (0,00883) | 8,63457 (0,06235) | 0,00067 (0,00027) | 9,45346 (0,06369) |
| | PNBA* | 0,48886 (0,00694) | 3,99189 (0,02713) | 0,00070 (0,00010) | 4,48146 (0,02752) |
| 7000 | A* | 0,69652 (0,00820) | 18,37828 (0,04726) | 0,00084 (0,00028) | 19,07564 (0,04842) |
| | NBA* | 1,11862 (0,01378) | 12,18805 (0,13551) | 0,00114 (0,00028) | 13,30781 (0,13662) |
| | PNBA* | 0,66324 (0,01380) | 5,63532 (0,04051) | 0,00097 (0,00025) | 6,29953 (0,04478) |
| 8000 | A* | 0,91039 (0,01016) | 25,02783 (0,06436) | 0,00104 (0,00033) | 25,93925 (0,06547) |
| | NBA* | 1,44757 (0,01689) | 16,50990 (0,12897) | 0,00122 (0,00027) | 17,95871 (0,12758) |
| | PNBA* | 0,85469 (0,01534) | 7,63804 (0,05230) | 0,00107 (0,00026) | 8,49380 (0,05542) |
| 9000 | A* | 1,13640 (0,01205) | 32,70581 (0,07762) | 0,00116 (0,00029) | 33,84337 (0,07772) |
| | NBA* | 1,82601 (0,01613) | 21,64307 (0,14791) | 0,00124 (0,00029) | 23,47031 (0,15148) |
| | PNBA* | 1,08155 (0,01445) | 9,97486 (0,06397) | 0,00120 (0,00020) | 11,05760 (0,06586) |




(a) Número de expansões (milhões) para os vários tamanhos de *grids* quadrados.

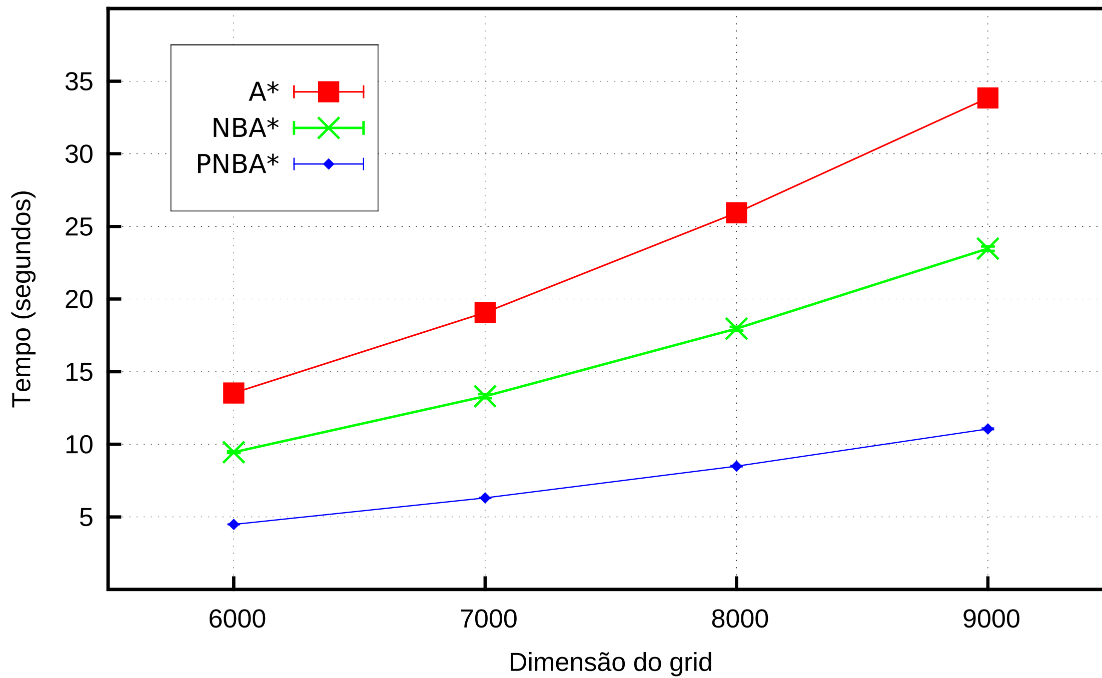



(b) Número de expansões percentual para os vários tamanhos de *grids* quadrados.

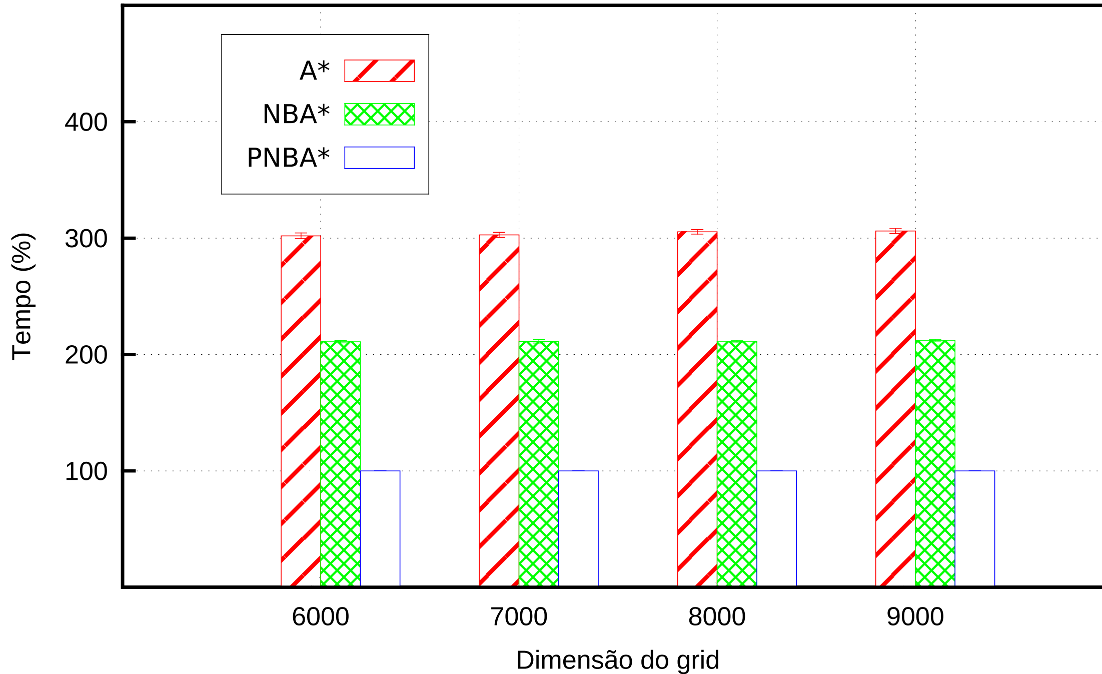
Figura 3.5: Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme.

Tabela 3.2: Número de expansões e tempo total relativo para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme.

| Dimensão do *grid* | Algoritmo | # expansões | | Tempo total relativo (%) | # vezes foi o mais rápido |
|---|---|---|---|---|---|
| | | Relativo (%) | Absoluto (milhões) | | |
| *Pathfinding* em *grids* com arestas de custo uniforme | | | | | |
| 6000 | A* | 236,00 (17,90) | 7,81624 (0,78886) | 466,42 (36,39) | 0 |
| | NBA* | 101,26 (0,60) | 3,35712 (0,27234) | 191,65 (2,09) | 0 |
| | PNBA* | 100,00 (0,00) | 3,31515 (0,26652) | 100,00 (0,00) | 50 |
| 7000 | A* | 233,51 (13,65) | 10,82923 (0,89239) | 475,98 (28,37) | 0 |
| | NBA* | 101,28 (0,62) | 4,70220 (0,36019) | 192,42 (2,41) | 0 |
| | PNBA* | 100,00 (0,00) | 4,64295 (0,35472) | 100,00 (0,00) | 50 |
| 8000 | A* | 230,15 (14,76) | 13,97318 (1,38395) | 484,59 (32,13) | 0 |
| | NBA* | 101,17 (0,59) | 6,15117 (0,58447) | 193,19 (2,12) | 0 |
| | PNBA* | 100,00 (0,00) | 6,08134 (0,58758) | 100,00 (0,00) | 50 |
| 9000 | A* | 226,16 (9,59) | 17,89415 (1,54900) | 480,85 (23,45) | 0 |
| | NBA* | 101,00 (0,50) | 7,99570 (0,65457) | 193,46 (2,04) | 0 |
| | PNBA* | 100,00 (0,00) | 7,91630 (0,64242) | 100,00 (0,00) | 50 |
| *Pathfinding* em *grids* com arestas de custo não uniforme | | | | | |
| 6000 | A* | 135,21 (0,75) | 23,35051 (0,00470) | 301,97 (2,42) | 0 |
| | NBA* | 100,05 (0,00) | 17,27788 (0,09625) | 210,95 (1,05) | 0 |
| | PNBA* | 100,00 (0,00) | 17,26995 (0,09634) | 100,00 (0,00) | 50 |
| 7000 | A* | 135,15 (0,80) | 31,78390 (0,00536) | 302,82 (2,12) | 0 |
| | NBA* | 100,04 (0,00) | 23,52717 (0,13921) | 211,25 (1,49) | 0 |
| | PNBA* | 100,00 (0,00) | 23,51777 (0,13932) | 100,00 (0,00) | 50 |
| 8000 | A* | 135,06 (0,84) | 41,51355 (0,00592) | 305,40 (1,98) | 0 |
| | NBA* | 100,03 (0,00) | 30,74873 (0,19040) | 211,43 (0,77) | 0 |
| | PNBA* | 100,00 (0,00) | 30,73813 (0,19063) | 100,00 (0,00) | 50 |
| 9000 | A* | 135,00 (0,70) | 52,54284 (0,00552) | 306,08 (2,00) | 0 |
| | NBA* | 100,03 (0,00) | 38,93348 (0,20238) | 212,26 (0,79) | 0 |
| | PNBA* | 100,00 (0,00) | 38,92189 (0,20227) | 100,00 (0,00) | 50 |

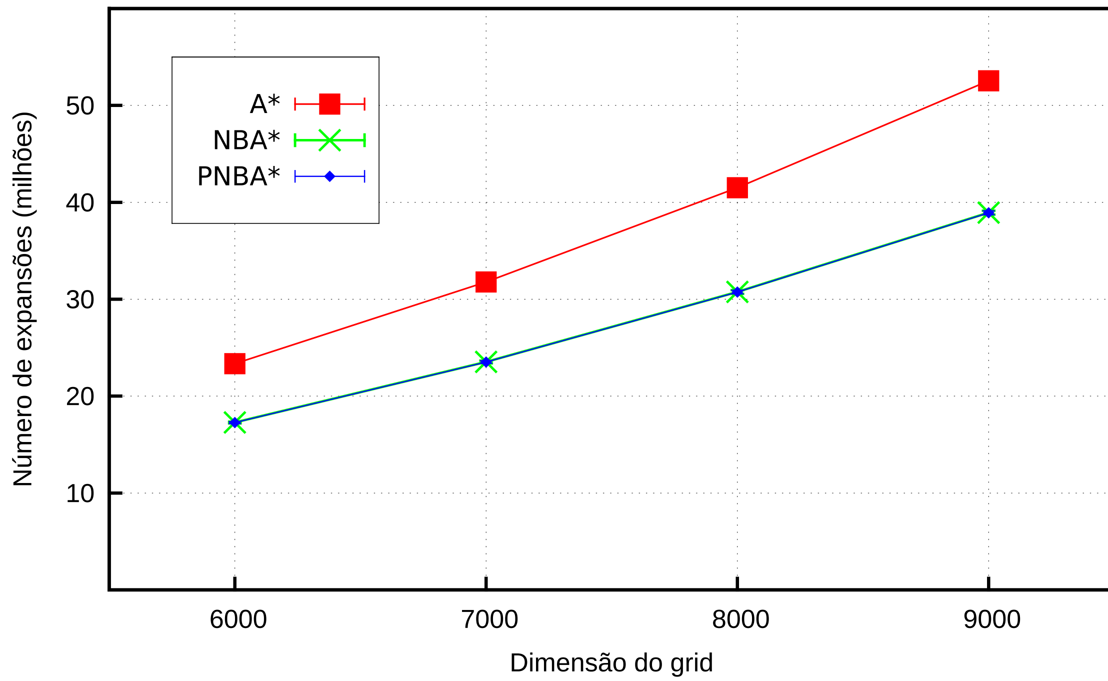



(a) Tempo total em segundos para os vários tamanhos de *grids* quadrados.

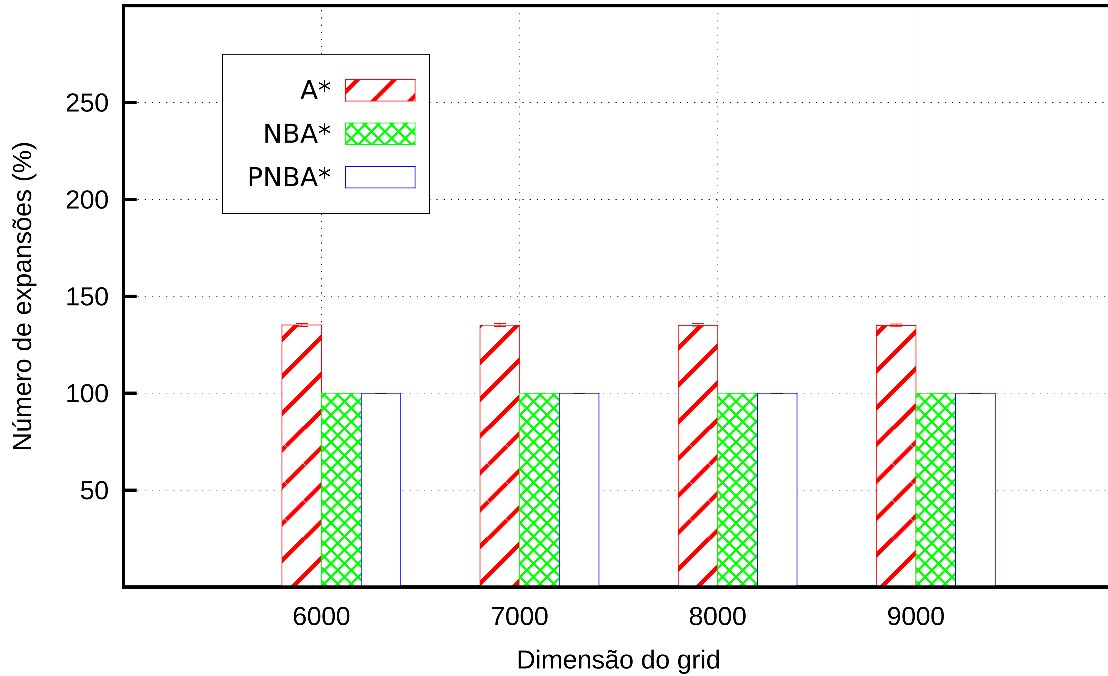



(b) Tempo total percentual para os vários tamanhos de *grids* quadrados.

Figura 3.6: Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme.

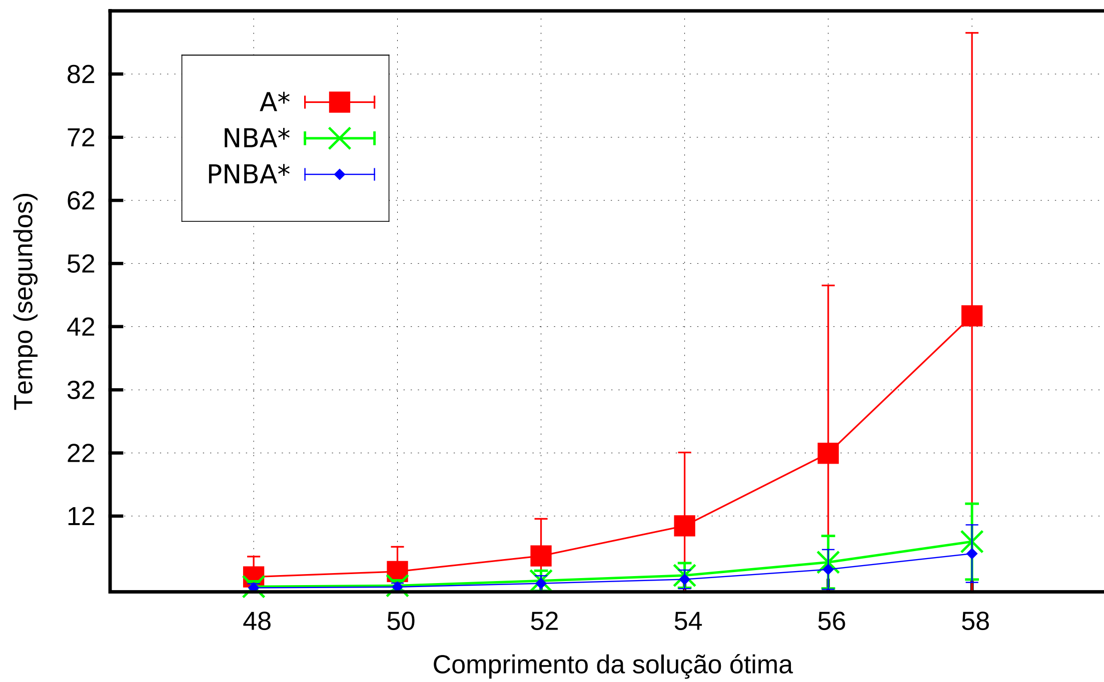



(a) Número de expansões (milhões) para os vários tamanhos de *grids* quadrados.

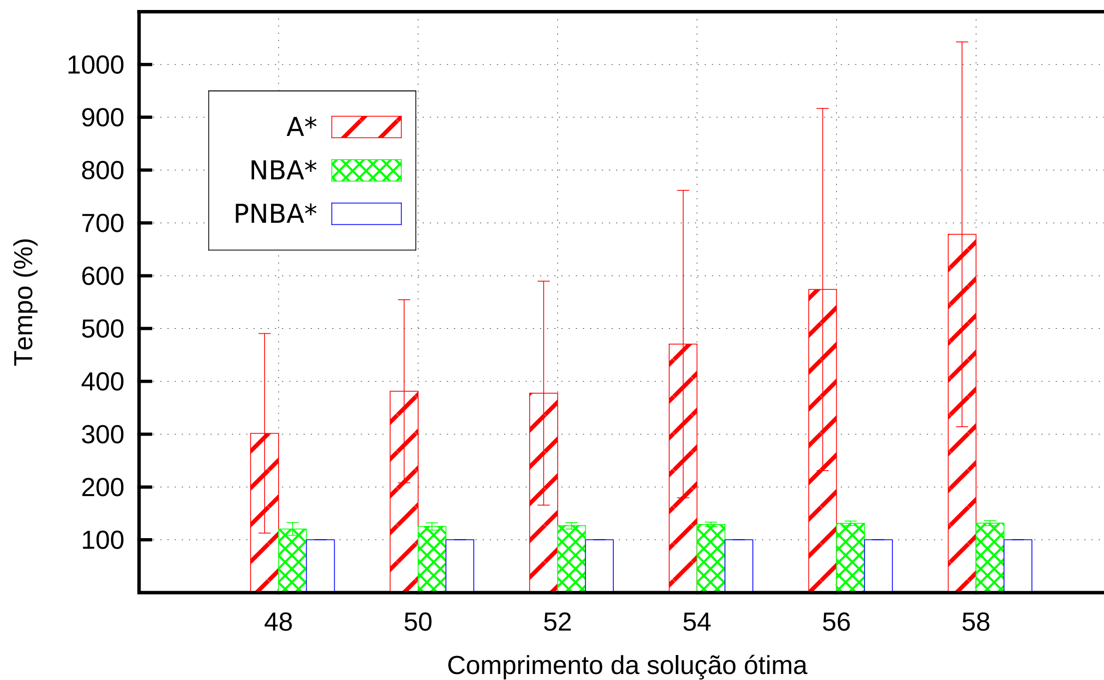



(b) Número de expansões percentual para os vários tamanhos de *grids* quadrados.

Figura 3.7: Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme.




(a) Tempo total em segundos para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.

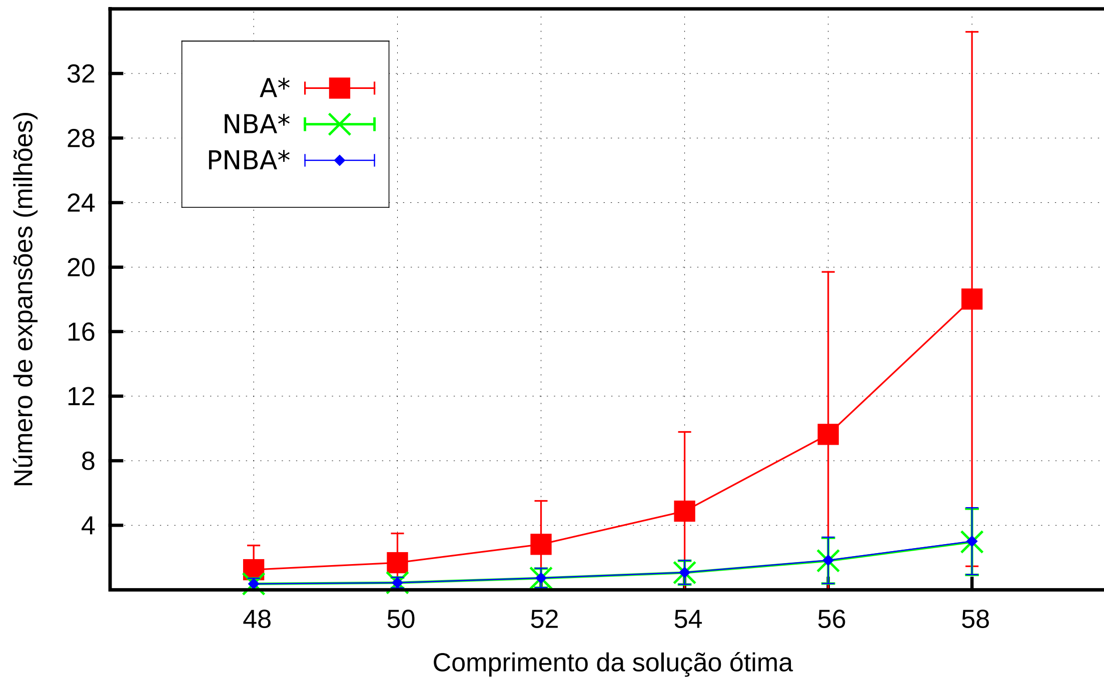



(b) Tempo total percentual para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.

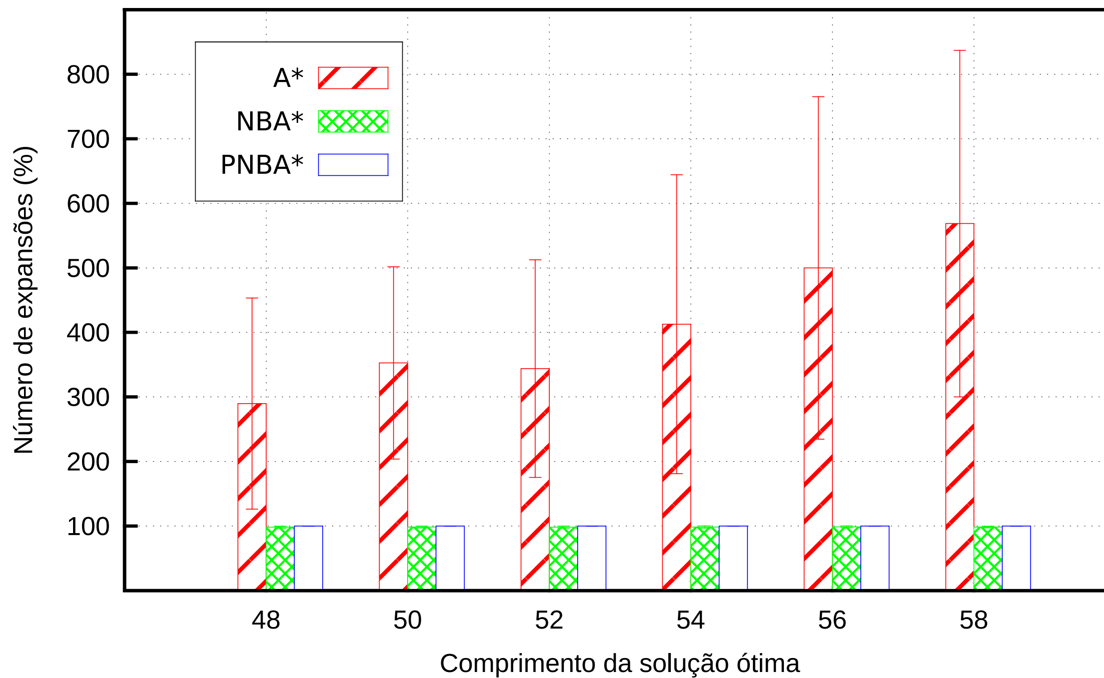
Figura 3.8: Tempo total absoluto e relativo no domínio do *15-puzzle*.

Tabela 3.3: Tempo absoluto para o domínio do *15-puzzle*.

| Comprimento da solução ótima | Algoritmo | Tempo absoluto (segundos) | | | |
|---|---|---|---|---|---|
| | | Inicialização | Execução | Finalização | Total |
| 48 | A* | 0,00006 (0,00000) | 2,10160 (2,79854) | 0,27353 (0,41942) | 2,37519 (3,21409) |
| | NBA* | 0,00006 (0,00000) | 0,75175 (0,77932) | 0,07265 (0,07385) | 0,82446 (0,85309) |
| | PNBA* | 0,00008 (0,00001) | 0,57354 (0,57683) | 0,07478 (0,07229) | 0,64840 (0,64898) |
| 50 | A* | 0,00006 (0,00000) | 2,83321 (3,37844) | 0,38655 (0,54523) | 3,21982 (3,92049) |
| | NBA* | 0,00007 (0,00001) | 0,90292 (0,75539) | 0,08792 (0,07121) | 0,99091 (0,82643) |
| | PNBA* | 0,00008 (0,00000) | 0,68246 (0,55509) | 0,08894 (0,06874) | 0,77147 (0,62363) |
| 52 | A* | 0,00006 (0,00000) | 4,97654 (5,01774) | 0,71992 (0,85201) | 5,69652 (5,86265) |
| | NBA* | 0,00007 (0,00000) | 1,58619 (1,45252) | 0,15334 (0,14529) | 1,73960 (1,59712) |
| | PNBA* | 0,00008 (0,00001) | 1,18727 (1,05867) | 0,15386 (0,13976) | 1,34122 (1,19789) |
| 54 | A* | 0,00006 (0,00001) | 8,99863 (9,73540) | 1,46745 (1,87727) | 10,46614 (11,60521) |
| | NBA* | 0,00007 (0,00000) | 2,37750 (1,77619) | 0,23046 (0,17820) | 2,60802 (1,95363) |
| | PNBA* | 0,00008 (0,00000) | 1,76183 (1,29703) | 0,23000 (0,17207) | 1,99191 (1,46827) |
| 56 | A* | 0,00006 (0,00001) | 18,54786 (22,02585) | 3,38516 (4,58028) | 21,93309 (26,59985) |
| | NBA* | 0,00007 (0,00000) | 4,23768 (3,67726) | 0,45179 (0,48537) | 4,68953 (4,15789) |
| | PNBA* | 0,00009 (0,00001) | 3,10902 (2,68062) | 0,44395 (0,47417) | 3,55306 (3,15070) |
| 58 | A* | 0,00006 (0,00001) | 36,46153 (36,78236) | 7,24735 (8,04592) | 43,70891 (44,81459) |
| | NBA* | 0,00007 (0,00001) | 7,10524 (5,22747) | 0,84894 (0,78185) | 7,95424 (6,00057) |
| | PNBA* | 0,00009 (0,00001) | 5,20672 (3,79743) | 0,83880 (0,77572) | 6,04561 (4,56321) |




(a) Número de expansões (milhões) para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.




(b) Número de expansões percentual para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.

Figura 3.9: Número de expansões absoluto e relativo no domínio do *15-puzzle*.

Tabela 3.4: Número de expansões e tempo total relativo para o domínio do *15-puzzle*.

| Comprimento da solução ótima | Algoritmo | # expansões | | Tempo total relativo (%) | # vezes foi o mais rápido |
|---|---|---|---|---|---|
| | | Relativo (%) | Absoluto (milhões) | | |
| 48 | A* | 289,74 (163,53) | 1,24769 (1,50820) | 301,62 (188,81) | 5 |
| | NBA* | 98,21 (1,61) | 0,36938 (0,34090) | 120,50 (12,10) | 2 |
| | PNBA* | 100,00 (0,00) | 0,37440 (0,34519) | 100,00 (0,00) | 43 |
| 50 | A* | 352,76 (148,90) | 1,68216 (1,81372) | 381,22 (173,42) | 0 |
| | NBA* | 98,48 (1,48) | 0,44145 (0,32757) | 125,29 (6,92) | 1 |
| | PNBA* | 100,00 (0,00) | 0,44675 (0,32810) | 100,00 (0,00) | 49 |
| 52 | A* | 343,89 (168,65) | 2,82133 (2,69316) | 377,56 (212,03) | 0 |
| | NBA* | 98,22 (1,83) | 0,72916 (0,59402) | 126,56 (6,28) | 0 |
| | PNBA* | 100,00 (0,00) | 0,73934 (0,59693) | 100,00 (0,00) | 50 |
| 54 | A* | 412,69 (231,50) | 4,87865 (4,91104) | 470,53 (291,12) | 0 |
| | NBA* | 98,59 (1,82) | 1,06972 (0,73469) | 129,17 (4,20) | 1 |
| | PNBA* | 100,00 (0,00) | 1,08287 (0,74142) | 100,00 (0,00) | 49 |
| 56 | A* | 499,92 (265,23) | 9,63091 (10,07592) | 573,88 (342,83) | 0 |
| | NBA* | 98,85 (1,49) | 1,80873 (1,41138) | 131,09 (4,28) | 0 |
| | PNBA* | 100,00 (0,00) | 1,82961 (1,43426) | 100,00 (0,00) | 50 |
| 58 | A* | 568,55 (268,56) | 18,02112 (16,55817) | 678,36 (364,31) | 0 |
| | NBA* | 98,57 (1,24) | 2,97607 (2,04802) | 131,55 (4,71) | 0 |
| | PNBA* | 100,00 (0,00) | 3,01757 (2,07243) | 100,00 (0,00) | 50 |

# Capítulo 4

# BPBNF

Uma das limitações do algoritmo PNBA*, proposto no capítulo anterior, é o emprego de apenas dois processadores na busca. Neste capítulo, será discutido o *Bidirectional Parallel Best-NBlock-First* (BPBNF): um algoritmo de busca heurística bidirecional paralela que emprega vários processadores e o paradigma bidirecional de modo semelhante ao utilizado nos algoritmos NBA* e PNBA*.

Este capítulo é constituído de duas seções. A primeira descreve o algoritmo BPBNF. A seção seguinte detalha a metodologia e expõe os resultados da avaliação empírica realizada. Ela também discute os resultados desses experimentos, promovidos a fim de medir o desempenho do BPBNF e compará-lo com outros algoritmos.

## 4.1 Descrição do algoritmo

O BPBNF é uma forma de empregar mais de dois processadores a uma busca bidirecional semelhante à do PNBA* e também um modo de melhorar o tempo de execução do algoritmo PBNF (algoritmo de busca heurística paralela no qual ele se baseia). Logo, aplicou-se os conceitos da busca bidirecional - algoritmos NBA* (subseção 2.3.1.1) e PNBA* (capítulo 3) - no algoritmo PBNF (apresentado na subseção 2.3.4.1 deste documento). A modificação central para aplicar o paradigma bidirecional ao PBNF envolveu permitir que o espaço de busca definido por um *nblock* fosse explorado simultaneamente por dois processadores, cada um em uma direção distinta. Ou seja, como na busca bidirecional há duas árvores de busca cujas raízes são $s$ e $t$.

Os *nblocks* do BPBNF possuem dois *duplicate detection scope* independentes, cada um referente a um dos processos de busca. Dessa forma, para facilitar a seleção de *nblocks* para exploração, é necessário utilizar os $\sigma_p$-*values*. A semântica é

a mesma do PBNF, mas agora está restrita a um processo de busca. O nó abstrato também passa a contar com as estruturas de dados $open_p$[1].

No PBNF há uma única tarefa associada com cada *nblock*: explorá-lo. Como existem vários *nblocks*, estão disponíveis várias tarefas. Qualquer um dos processadores pode alocar para si uma das atividades disponíveis e, ao longo da execução, ele pode realizar várias delas. No BPBNF existem duas tarefas para cada *nblock*, pois a busca é conduzida em duas direções distintas. Diferentemente do PNBA*, um processador pode atuar em qualquer um dos dois processos de busca ao selecionar as tarefas existentes. Ou seja, a troca do *nblock* explorado e, possivelmente, do processo de busca contemplado ocorre dinamicamente ao longo da execução.

O BPBNF seleciona sempre as tarefas mais promissoras primeiramente. Também, como no PBNF, a prioridade de uma tarefa é o menor $f_p$*-value* da estrutura de dados $open_p$ do *nblock* da tarefa em questão ($p$ é o processo de busca da atividade). Quanto menor é o $f_p$*-value* de $open_p$, maior é a importância da atividade. Portanto, o nome "*best nblock first*" é válido para o BPBNF.

Como no algoritmo PNBA*, é necessário manter para os vértices do BPBNF os $f_p$*-values*, $g_p$*-values* e $h_p$*-values*. Há também uma variável $\mathcal{L}$ que armazena o custo da melhor solução calculada até determinado momento da computação. O modo de sua atualização é o mesmo utilizado no PNBA*. É necessário, pois, empregar dois mecanismos de sincronização diferentes no algoritmo em discussão. Um deles é para atualização de $\mathcal{L}$ e o outro para manipulação do grafo abstrato. A necessidade dessa variável justifica-se pela incorporação, no BPBNF, do mesmo critério de poda do NBA* e do PNBA*. Outra semelhança com os algoritmos mencionados é o critério para finalização. Quando não existem mais nós para expansão em algum dos processos de busca, a computação é encerrada. Em outras palavras, quando todas as estruturas de dados $open_p$ estão vazias para qualquer um dos valores de $p$, é chegado o momento de interromper a busca.

Para utilizar o critério de poda do NBA* e do PNBA* no BPBNF, é necessário atualizar as variáveis $F_p$. Porém, essa operação não é trivial, porque a fronteira de cada um dos processos de busca pode estar distribuída nos vários *nblocks* existentes que por sua vez podem estar sendo explorados por diversos processadores ao mesmo tempo. Das operações realizadas pelo PNBA*, ela foi sem dúvidas a mais difícil de ser incorporada ao BPBNF.

A relação entre a freqüência das atualizações das variáveis $F_p$ e o tempo de execução do algoritmo sugere a existência de um equilíbrio. Quanto mais próximos

[1]Na implementação realizada, a adoção das estruturas de dados *closed* não é necessária. Mais detalhes serão expostos na subseção 4.2.1.

os valores de $F_p$ estão de $F_p^*$, maior será a acurácia da informação disponível para poda de nós. Conseqüentemente, menos expansões serão realizadas pelo algoritmo. Entretanto, atualizações muito freqüentes levam a um aumento da necessidade de sincronizar e comprometem o paralelismo do algoritmo.

A atualização das variáveis $F_p$ nunca é efetuada por mais de um processador ao mesmo tempo. Ela é feita sempre que um processador adquire um nó abstrato para exploração, enquanto ele ainda está manipulando o grafo abstrato para garantir acesso exclusivo. Antes de explicar em detalhes como essa operação é realizada, cabe ressaltar uma característica dos valores de $F_p^*$ ao longo da computação: a monotonicidade crescente. Ou seja, os valores do menor $f_p$-*value* de todas as estruturas de dados $open_p$ são monotonicamente crescentes. A forma como os $g_p$-*values* são alterados (sempre incrementando outro $g_p$-*value*) e o uso de uma heurística consistente são as causas da existência dessa propriedade. Cabe mencionar, entretanto, que o menor $f_p$-*value* (considerando apenas a estrutura de dados $open_p$ de um único *nblock*) pode diminuir ao longo da execução.

No contexto de um processo de busca $p$, dois tipos de *nblock* interessam no cálculo de $F_p$. O primeiro compreende aqueles aptos a serem explorados, pois possuem pelo menos um nó em sua estrutura de dados $open_p$. O segundo inclui aqueles que estão bloqueados. Dois são os motivos possíveis: possuem $\sigma_p \neq 0$ ou estão alocados a um processador.

Encontrar o menor $f_p$-*value* dos *nblocks* do primeiro tipo é simples, pois eles não estão sendo explorados. A dificuldade está em realizar a mesma tarefa para os *nblocks* da segunda classe, já que a todo momento as suas estruturas de dados $open_p$ podem estar sendo alteradas.

Antes que um *nblock* seja bloqueado, o valor do seu menor $f_p$-*value* é armazenado separadamente para ser empregado posteriormente no cálculo em questão. Esse valor nunca é modificado por mais de um processador ao mesmo tempo. Assim, as atualizações sempre são efetuadas pelo processador que detém o acesso ao grafo abstrato. Quando um *nblock* $a$ deixa de ser explorado, vários nós abstratos poderão ser desbloqueados. Nesse momento, aqueles que possivelmente foram alterados (estão no *duplicate detection scope* de $a$) e deverão ainda permanecer bloqueados (porque estão no *interference scope* de outro *nblock*) terão seu menor $f_p$-*value* lido novamente e armazenado separadamente para consultas futuras.

A forma como o processo para atualização de $F_p$ é conduzido garante que as informações empregadas não poderão estar sendo modificadas. No momento de escolher o próximo valor de $F_p$, o processador basicamente seleciona o menor valor associado a cada *nblock*. Os *nblocks* consultados são apenas aqueles dos dois tipos

mencionados. No primeiro caso, é trivial recuperar a informação desejada. Se uma lista ordenada pelo menor $f_p$-*value* da estrutura de dados $open_p$ for mantida, o tempo necessário será mínimo. No caso dos vértices abstratos que estão bloqueados, o valor utilizado é aquele lido e armazenado previamente de modo separado. Também é possível manter uma lista ordenada desse tipo de *nblock* para reduzir o esforço computacional necessário.

Apesar de não serem apresentadas neste trabalho as provas da admissibilidade do BPBNF, sustenta-se a sua existência no algoritmo em questão em razão dos testes realizados (incluindo a avaliação empírica) e da argumentação aqui exposta. A sua admissibilidade apoia-se na presença dessa propriedade nos algoritmos inspiradores de sua criação: NBA*, PNBA* e PBNF. A diferença central do BPBNF para PNBA* é a execução de cada um dos processos de busca por mais de um processador sem respeitar a ordem crescente dos respectivos $f_p$-*values*, ou seja, o algoritmo é *"best-first"* apenas no nível dos *nblocks*. A argumentação agora exposta objetiva demostrar que essa alteração não viola a admissibilidade.

O critério de poda do NBA* e do PNBA* implementado no BPBNF é conservador, pois emprega os valores $F_p$ e $\mathcal{L}$ que são limites confiáveis computados pelo próprio algoritmo. Independente da ordem em que os nós são expandidos, se a heurística é consistente, nunca será possível encontrar uma solução que parta da raiz do processo de busca $p$ e tenha um custo menor do que $F_p^*$. A explicação é a monotonicidade crescente dos $f_p$-*values* e $F_p^*$ ao longo da computação (conseqüência da heurística consistente). Logo, como no PNBA*, a utilização de $F_p$ diferente do valor real $F_p^*$ (desde que sempre menor) acarreta apenas a expansão desnecessária de nós. Como já discutido no capítulo anterior, essas expansões não prejudicam a admissibilidade.

O fato de não ser respeitada a ordem crescente dos $f_p$-*values* do processo de busca $p$ ao longo da computação faz com que no momento da expansão de um nó o $g_p$-*value* possa ser diferente do respectivo $g_p^*$-*value*. A poda de um nó com $g_p$-*value* diferente do respectivo $g_p^*$-*value* não compromete a admissibilidade do algoritmo já que os critérios empregados na busca independem disso. Eventualmente, esse nó podado poderá ser expandido se um novo caminho até ele (com um custo inferior) for encontrado.

Esta é outra característica importante do BPBNF herdada do PBNF. Se um nó $y$ é alcançado através de um caminho que não é o menor, isso terá acontecido por que um de seus ancestrais ($x$ por exemplo) deixou de ser expandido. O nó $x$ continuará na fronteira e eventualmente será expandido se for possível encontrar uma solução melhor do que a atual através dele. Talvez também seja necessário expandir

novamente $y$ para propagar a todos seus sucessores o novo caminho de menor custo encontrado. Essa é a razão pela qual o PBNF e BPBNF podem expandir o mesmo nó diversas vezes.

Por fim, é importante reforçar algumas das vantagens do BPBNF. Com relação ao PNBA*, a melhoria introduzida pelo algoritmo em discussão é a possibilidade de empregar mais de dois processadores ao mesmo tempo na exploração do grafo que representa o espaço de estados. Já na comparação com o PBNF, em dois quesitos destaca-se a superioridade do BPBNF. O primeiro deles é o aumento do número de tarefas disponíveis para os processadores já que cada *nblock* está associada a duas. Outro questão é a utilização de um critério de poda mais sofisticado que reduz o número de expansões realizadas pelo algoritmo.

## 4.2 Avaliação empírica

### 4.2.1 Metodologia

Com o intuito de avaliar empiricamente o BPBNF e compará-lo com os algoritmos A*, PBNF e PNBA*, foram conduzidos experimentos em três domínios cuja utilização no contexto de busca heurística é muito comum: *pathfinding* em *grids* de conectividade quatro com arestas de custo uniforme e não uniforme e no *15-puzzle*. A metodologia empregada foi praticamente a mesma descrita na subseção 3.2.3. As poucas diferenças devem-se a questões de implementação particulares dos algoritmos PNBF e BPBNF e serão agora apresentadas.

Como já mencionado, o PSDD depende de uma função de abstração que define o grafo abstrato. No domínio do *pathfinding* em *grids*, a função empregada realizou o mapeamento através da utilização de um *grid* com resolução menor. Em cada célula desse *grid*, cabiam 2500 células do *grid* original (ou seja, o tamanho relativo dos lados da célula quadrada era 50). Cada uma dessas células correspondia a um *nblock*. Os sucessores do *nblock* no grafo abstrato eram as (até) quatro células vizinhas a ele sem considerar as diagonais. Já no domínio do *15-puzzle*, a função de abstração considerava a posição vazia e das peças um e dois para realizar o mapeamento de uma configuração. Dessa forma, existiam $16 \times 15 \times 14 = 3360$ *nblocks*. Os sucessores do *nblock* no grafo abstrato eram os (até) quatro nós abstratos gerados movendo-se as peças vizinhas a posição vazia.

Para evitar a troca constante de *nblocks* e, conseqüentemente, diminuir a realização de sincronizações necessárias para obter acesso ao grafo abstrato; determina-se a realização (sempre que possível) de um número mínimo de expansões. Esse

comportamento foi originalmente proposto por Burns et al. [2009], autores do algoritmo PNBF.

Assim, o número mínimo de expansões exigidas antes de tentar trocar de *nblock* foi 150 e 32 para os domínios do *pathfinding* em *grids* e *15-puzzle* respectivamente. Como a máquina usada nos experimentos dispunha de quatro processadores, os algoritmos PBNF e BPBNF foram executados com duas, três e quatro *threads*.

O PBNF manteve uma lista de *nblocks* aptos a serem explorados ordenada pelo menor $f$*-value* de cada um deles. A implementação foi realizada através de um *heap* binário. O BPBNF adotou quatro listas: duas com *nblocks* aptos a serem explorados ordenadas pelo menor $f_p$*-value* e duas para facilitar no cálculo de $F_p$ com os nós abstratos bloqueados. Logo, para cada processo de busca $p$ haviam duas listas. Todas elas também foram representadas através de *heaps* binários.

Tanto no PBNF quanto no BPBNF, optou-se por não representar as respectivas estruturas de dados *closed*, pois a condição para que um nó seja colocado na estrutura de dados *open* independe completamente (nesse caso) de *closed*. É mais racional, portanto, empregar a condição para que um nó seja reaberto no controle de expansões. Assim, na implementação desses algoritmos um nó gerado era sempre colocado na respectiva estrutura de dados *open* quando um caminho com um custo menor (do que o atual) até ele fosse encontrado. Para isso, foi necessário garantir uma única representação para cada estado. No domínio do *15-puzzle*, como apenas a parcela necessária da representação do espaço de estados foi gerada e armazenada, uma tabela *hash* de dois níveis (a mesma adota nos algoritmos avaliados na subseção 3.2.3) foi empregada para assegurar a observância dessa restrição.

## 4.2.2 Resultados e discussão

### 4.2.2.1 PBNF *versus* BPBNF - número de *threads*

O intuito dos primeiros experimentos realizados foi comparar o tempo de execução total dos algoritmos PBNF e BPBNF quando o número de *threads* disponíveis a cada um deles variava. A figura 4.1 mostra graficamente os resultados obtidos. O *speedup* não foi linear para nenhum deles, ou seja, a redução no tempo de execução não foi diretamente proporcional ao aumento do número de processadores. Duas foram as principais razões. A primeira foi o aumento do número de acessos ao grafo abstrato que causou uma elevação no tempo médio de espera (já que dois ou mais processadores não podem acessá-lo simultaneamente). O outra motivo foi a diminuição do número de *nblocks* disponíveis para exploração ao longo da computação.

Porém, para qualquer um dos dois algoritmos experimentados o acréscimo do número de *threads* levou a uma diminuição no tempo total de execução. Conseqüentemente, tanto o PBNF quanto o BPBNF com quatro *threads* foram os mais rápidos. Assim, no restante dos experimentos efetuados, sempre utilizou-se quatro *threads* nesses algoritmos.

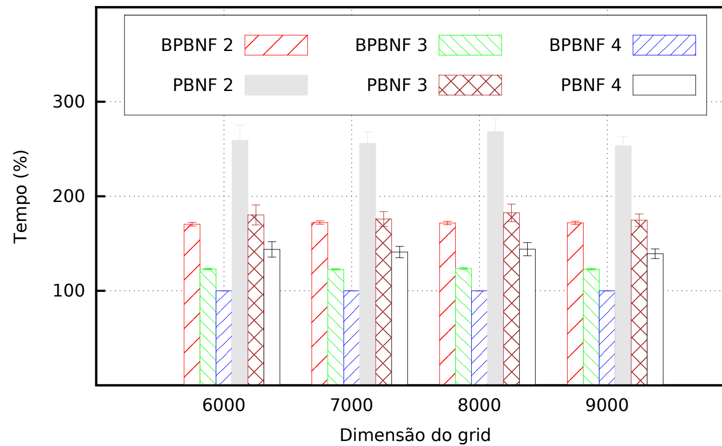
#### 4.2.2.2 A* *versus* PBNF *versus* BPBNF

As figuras 4.2 e 4.3 mostram respectivamente o tempo de execução e o número de expansões dos experimentos realizados no domínio do *pathfinding* em *grids* com arestas de custo uniforme para várias dimensões de *grid*. Os dados também são apresentados nas tabelas 4.1 e 4.2. Nelas o desvio padrão está entre parênteses e nos gráficos é exibido com barras.

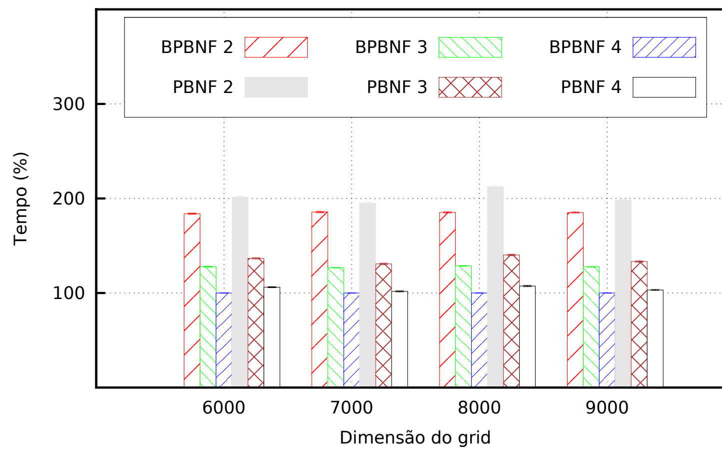
Claramente o BPBNF possui o menor tempo de execução nesse domínio. Em todos os *grids* resolvidos, ele foi o mais rápido. O A* apresentou o maior tempo na fase de inicialização como mostrado na tabela 4.1. Os dois algoritmos paralelos foram mais rápidos nessa fase devido a paralelização. Em conseqüência do tamanho da representação dos nós do grafo, o algoritmo BPBNF foi mais lento do que o PBNF nessa fase. Por ser um algoritmo bidirecional, é necessário manter informações além das empregadas pelo PBNF devido a utilização de dois processos de busca. Essas demandaram um trabalho extra no momento de criar e iniciar a representação do espaço de estados que seria explorado.

Considerando apenas a fase de execução, o BPBNF foi aproximadamente duas vezes mais rápido do que o PBNF. A razão principal é a diferença do número de expansões realizadas pelos algoritmos. A utilização do paradigma bidirecional no BPBNF permitiu uma expressiva diminuição do número de expansões em relação ao PBNF. O número de expansões do PBNF, quando comparado ao mesmo valor do BPBNF, foi superior ao dobro. Já se comparado ao respectivo valor do A*, nota-se um ligeiro aumento. A explicação são as expansões especulativas praticadas por esse algoritmo e as posteriores expansões necessárias para propagação da informação correta. Apesar disso, como a exploração do grafo é feita por mais de um processador, o tempo de execução total é muito inferior. O tempo da fase de finalização representou uma porção insignificante do tempo total para todos algoritmos avaliados nesse domínio.

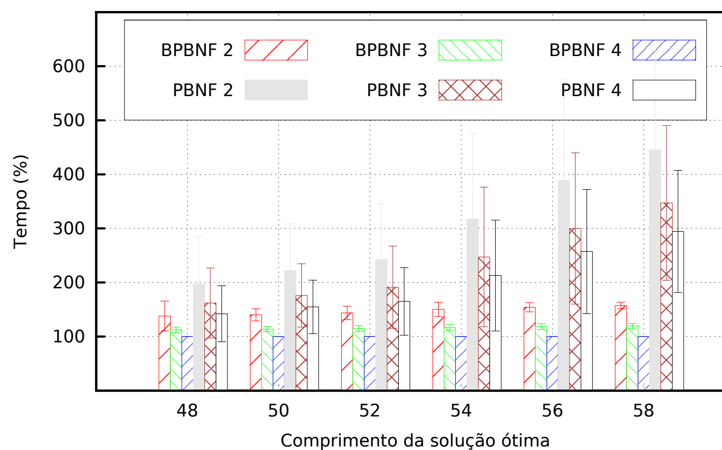
No domínio do *pathfinding* em *grids* com arestas de custo não uniforme, os benefícios trazidos pelo emprego do paradigma bidirecional no PBNF foram modestos. Os gráficos com o tempo total e o número de expansões estão respectivamente




(a) Tempo total percentual para os vários tamanhos de *grids* quadrados com arestas de custo uniforme.




(b) Tempo total percentual para os vários tamanhos de *grids* quadrados com arestas de custo não uniforme.




(c) Tempo total percentual para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.

Figura 4.1: Tempo total relativo para os algoritmos PBNF e BPBNF com duas, três e quatro *threads* nos três domínios avaliados.

nas figuras 4.4 e 4.5.

Em conformidade com o resultado apresentado no capítulo anterior, houve um acréscimo no número de expansões de todos algoritmos em comparação com as quantias respectivas do domínio anterior. A razão foi a dificuldade oferecida por esse domínio (necessidade do uso de uma heurística pobre). Nota-se também que a variação no número de expansões não foi proporcional entre os algoritmos. O crescimento do número de expansões nos algoritmos bidirecionais foi superior se o domínio anterior for utilizado como referência.

Conseqüentemente, apesar do BPBNF ter apresentado uma redução considerável no tempo da fase de execução, esse crescimento fez com que o ganho na fase de execução não fosse capaz de suprir o trabalho extra da fase de inicialização. Como nos algoritmos NBA* e PNBA*, o seu custo de inicialização é maior do que o respectivo valor de algoritmos unidirecionais, por causa do tamanho da representação dos nós do grafo. Para esses algoritmos bidirecionais, é necessário manter, além das informações empregadas pelos algoritmos unidirecionais, informações de cada um dos dois processos de busca. Essas demandaram um trabalho extra no momento de criar e iniciar a representação do espaço de estados que seria explorado. O tempo de execução total do BPBNF, portanto, foi praticamente o mesmo do PBNF. Ambos, porém, foram muito mais rápidos do que o A* em razão da paralelização (inclusive da fase de inicialização).

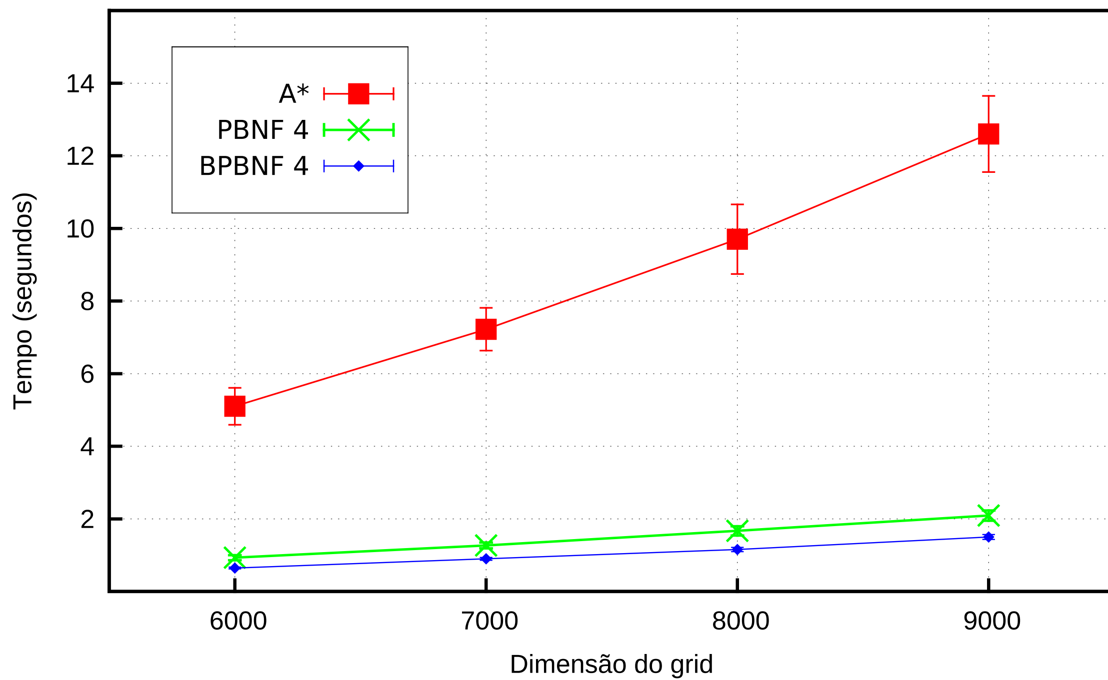
As figuras 4.6 e 4.7 mostram o tempo de execução e o número de expansões respectivamente dos experimentos realizados no domínio do *15-puzzle*. Como nos outros domínios, os dados também são apresentados em tabelas (4.3 e 4.4) para facilitar a compreensão.

O desvio padrão, como no capítulo anterior, foi grande nesse domínio pela mesmo motivo: a variação do erro total dos valores gerados pela heurística para resolução das configurações do *15-puzzle*. Na grande maioria das vezes, o BPBNF resolveu as configurações do *15-puzzle* utilizando um tempo inferior. Seu pior resultado foi para configurações cuja solução ótima era composta de 48 passos. Conforme a dificuldade das configurações foi ampliada, o desempenho do BPBNF melhorou em relação aos outros algoritmos.

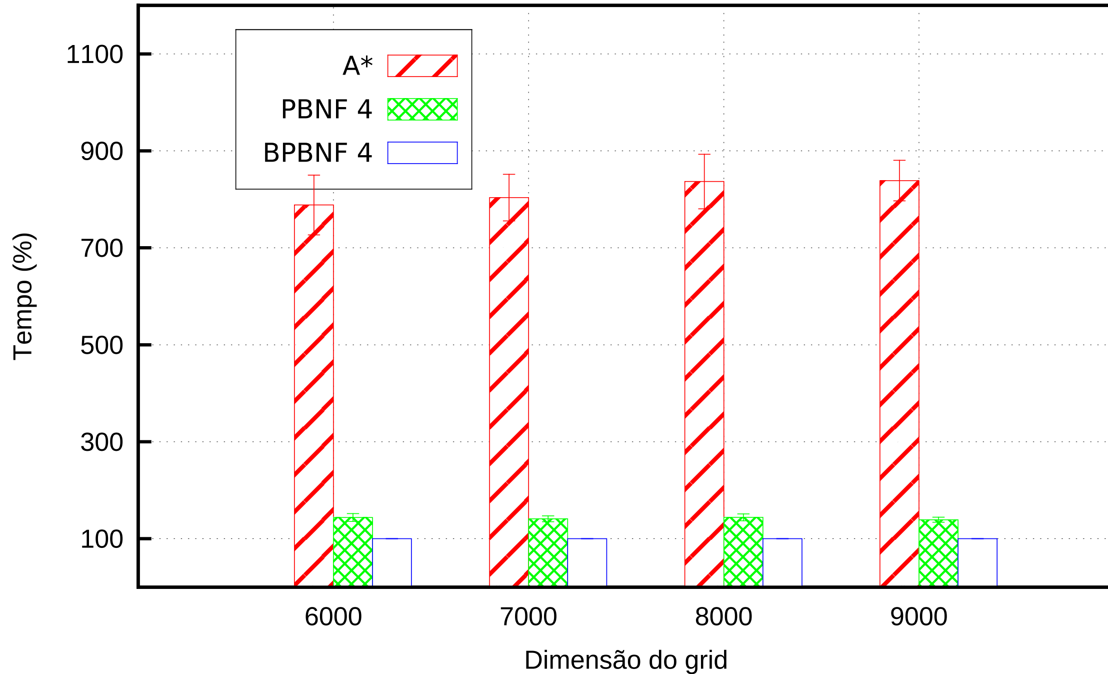
A taxa de crescimento do número de expansões dos algoritmos paralelos não foi a mesma. Para configurações cuja solução ótima era composta de 48 passos, o número de expansões do PBNF foi o dobro do mesmo valor do BPBNF. No entanto, quando 58 passos compunham a melhor solução das configurações, o número de expansões do BPBNF foi três vezes menor. Essa é a explicação para o aumento da diferença dos tempos de execução desses algoritmos em relação aos contextos

anteriores.

Os resultados dos experimentos evidenciaram uma clara superioridade do BPBNF em relação ao A*. Se comparado ao PBNF, em dois dos três domínios empregados também foi possível notar a superioridade do BPBNF. A redução do tempo de execução foi conseqüência da aplicação do paradigma bidirecional ao algoritmo. Portanto, a sua vantagem primordial é a aliança dos paradigmas bidirecional e paralelo, realizada de um modo que acumula os seus benefícios.




(a) Tempo total em segundos para os vários tamanhos de *grids* quadrados.

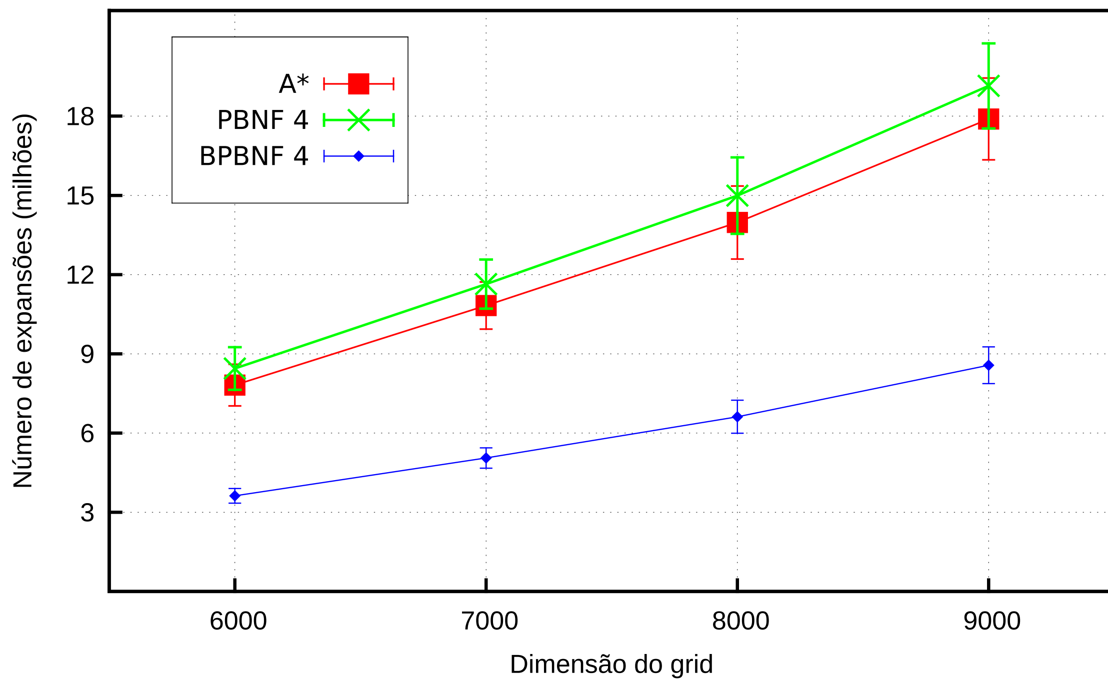



(b) Tempo total percentual para os vários tamanhos de *grids* quadrados.

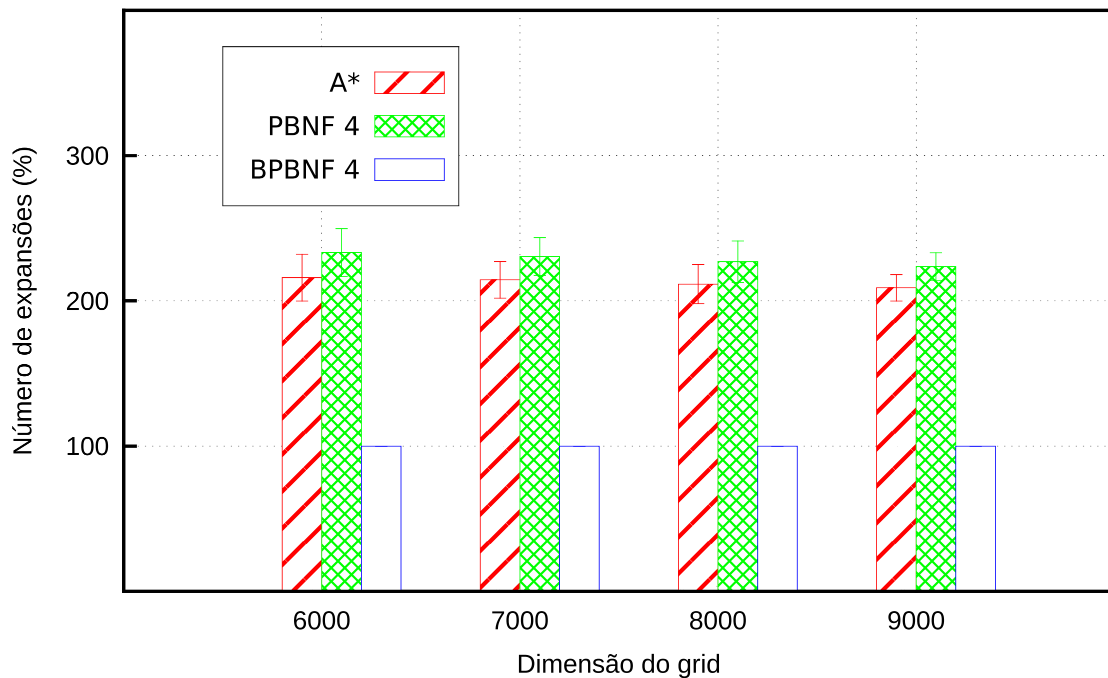
Figura 4.2: Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme.

Tabela 4.1: Tempo absoluto para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme.

| Dimensão do *grid* | Algoritmo | Tempo absoluto (segundos) | | | |
|---|---|---|---|---|---|
| | | Inicialização | Execução | Finalização | Total |
| *Pathfinding* em *grids* com arestas de custo uniforme | | | | | |
| 6000 | A* | 0,51069 (0,00515) | 4,58807 (0,50658) | 0,00113 (0,00002) | 5,09988 (0,50727) |
| | PBNF 4 | 0,18305 (0,00145) | 0,74238 (0,07039) | 0,00439 (0,00007) | 0,92983 (0,07036) |
| | BPBNF 4 | 0,30505 (0,00252) | 0,33581 (0,02581) | 0,00535 (0,00018) | 0,64621 (0,02545) |
| 7000 | A* | 0,69682 (0,00816) | 6,52496 (0,59005) | 0,00086 (0,00029) | 7,22264 (0,59064) |
| | PBNF 4 | 0,24902 (0,00156) | 1,01220 (0,08047) | 0,00570 (0,00022) | 1,26692 (0,08067) |
| | BPBNF 4 | 0,41086 (0,00311) | 0,47982 (0,03700) | 0,00748 (0,00056) | 0,89816 (0,03761) |
| 8000 | A* | 0,90490 (0,00915) | 8,79521 (0,95982) | 0,00104 (0,00031) | 9,70115 (0,95882) |
| | PBNF 4 | 0,32529 (0,00306) | 1,33624 (0,12846) | 0,00721 (0,00028) | 1,66874 (0,12844) |
| | BPBNF 4 | 0,53621 (0,00471) | 0,61149 (0,05800) | 0,00993 (0,00043) | 1,15763 (0,05808) |
| 9000 | A* | 1,13352 (0,01022) | 11,46479 (1,04969) | 0,00112 (0,00032) | 12,59943 (1,04924) |
| | PBNF 4 | 0,41031 (0,00381) | 1,67274 (0,14079) | 0,00722 (0,00033) | 2,09026 (0,14091) |
| | BPBNF 4 | 0,67905 (0,00690) | 0,80852 (0,06636) | 0,01291 (0,00102) | 1,50047 (0,06658) |
| *Pathfinding* em *grids* com arestas de custo não uniforme | | | | | |
| 6000 | A* | 0,51015 (0,00458) | 13,02088 (0,07102) | 0,00117 (0,00002) | 13,53219 (0,07032) |
| | PBNF 4 | 0,18355 (0,00146) | 1,72999 (0,00254) | 0,00447 (0,00012) | 1,91801 (0,00260) |
| | BPBNF 4 | 0,30564 (0,00294) | 1,49559 (0,00903) | 0,00532 (0,00006) | 1,80655 (0,00934) |
| 7000 | A* | 0,69652 (0,00820) | 18,37828 (0,04726) | 0,00084 (0,00028) | 19,07564 (0,04842) |
| | PBNF 4 | 0,24863 (0,00155) | 2,33525 (0,00304) | 0,00569 (0,00008) | 2,58957 (0,00348) |
| | BPBNF 4 | 0,41089 (0,00282) | 2,12709 (0,01395) | 0,00723 (0,00008) | 2,54521 (0,01435) |
| 8000 | A* | 0,91039 (0,01016) | 25,02783 (0,06436) | 0,00104 (0,00033) | 25,93925 (0,06547) |
| | PBNF 4 | 0,32504 (0,00249) | 3,18999 (0,00565) | 0,00711 (0,00016) | 3,52214 (0,00698) |
| | BPBNF 4 | 0,53707 (0,00438) | 2,73377 (0,01876) | 0,00950 (0,00010) | 3,28034 (0,01934) |
| 9000 | A* | 1,13640 (0,01205) | 32,70581 (0,07762) | 0,00116 (0,00029) | 33,84337 (0,07772) |
| | PBNF 4 | 0,41103 (0,00375) | 3,97562 (0,00473) | 0,00723 (0,00020) | 4,39388 (0,00602) |
| | BPBNF 4 | 0,68103 (0,00674) | 3,56332 (0,02004) | 0,01213 (0,00010) | 4,25648 (0,02058) |




(a) Número de expansões (milhões) para os vários tamanhos de *grids* quadrados.

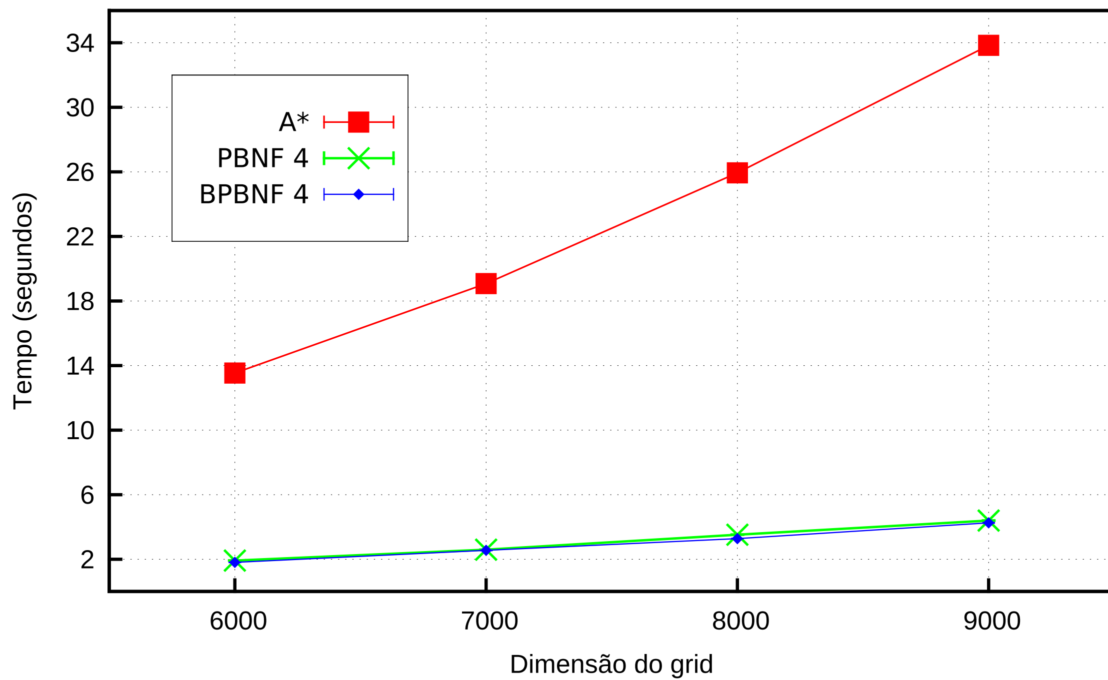



(b) Número de expansões percentual para os vários tamanhos de *grids* quadrados.

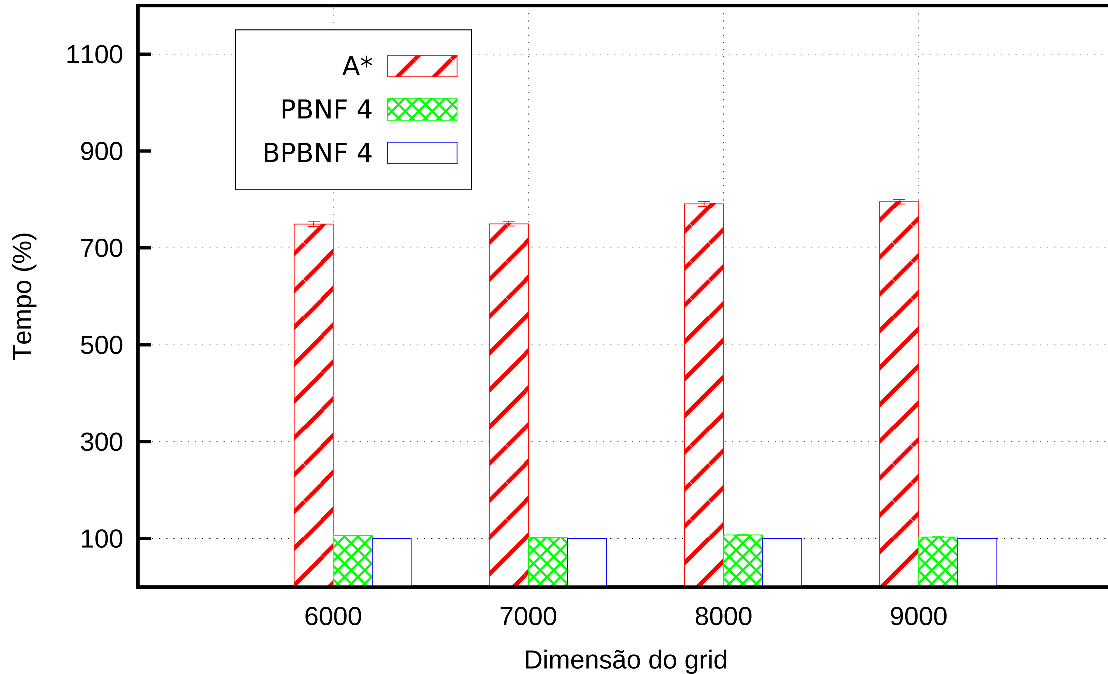
Figura 4.3: Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme.

Tabela 4.2: Número de expansões e tempo total relativo para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme.

| Dimensão do *grid* | Algoritmo | # expansões | | Tempo total relativo (%) | # vezes foi o mais rápido |
|---|---|---|---|---|---|
| | | Relativo (%) | Absoluto (milhões) | | |
| *Pathfinding* em *grids* com arestas de custo uniforme | | | | | |
| 6000 | A* | 215,99 (16,12) | 7,81624 (0,78886) | 788,30 (61,53) | 0 |
| | PBNF 4 | 233,33 (16,39) | 8,44185 (0,80647) | 143,82 (8,13) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 3,62079 (0,27582) | 100,00 (0,00) | 50 |
| 7000 | A* | 214,49 (12,58) | 10,82923 (0,89239) | 803,55 (47,94) | 0 |
| | PBNF 4 | 230,55 (13,01) | 11,63966 (0,93098) | 141,02 (6,03) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 5,05468 (0,38469) | 100,00 (0,00) | 50 |
| 8000 | A* | 211,52 (13,53) | 13,97318 (1,38395) | 836,78 (56,27) | 0 |
| | PBNF 4 | 226,95 (14,23) | 14,98986 (1,44588) | 144,07 (7,01) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 6,61564 (0,62519) | 100,00 (0,00) | 50 |
| 9000 | A* | 208,97 (9,11) | 17,89415 (1,54900) | 838,66 (41,90) | 0 |
| | PBNF 4 | 223,64 (9,42) | 19,14821 (1,61070) | 139,22 (5,07) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 8,56782 (0,69532) | 100,00 (0,00) | 50 |
| *Pathfinding* em *grids* com arestas de custo não uniforme | | | | | |
| 6000 | A* | 130,33 (0,75) | 23,35051 (0,00470) | 749,08 (4,94) | 0 |
| | PBNF 4 | 135,23 (0,77) | 24,22802 (0,01757) | 106,17 (0,57) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 17,91738 (0,10400) | 100,00 (0,00) | 50 |
| 7000 | A* | 130,30 (0,79) | 31,78390 (0,00536) | 749,49 (4,31) | 0 |
| | PBNF 4 | 135,11 (0,79) | 32,95661 (0,02140) | 101,75 (0,56) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 24,39284 (0,14728) | 100,00 (0,00) | 50 |
| 8000 | A* | 130,24 (0,83) | 41,51355 (0,00592) | 790,77 (4,95) | 0 |
| | PBNF 4 | 135,00 (0,84) | 43,02889 (0,02221) | 107,37 (0,69) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 31,87475 (0,20291) | 100,00 (0,00) | 50 |
| 9000 | A* | 130,21 (0,69) | 52,54284 (0,00552) | 795,12 (4,51) | 0 |
| | PBNF 4 | 134,92 (0,70) | 54,44489 (0,02050) | 103,23 (0,51) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 40,35309 (0,21296) | 100,00 (0,00) | 50 |

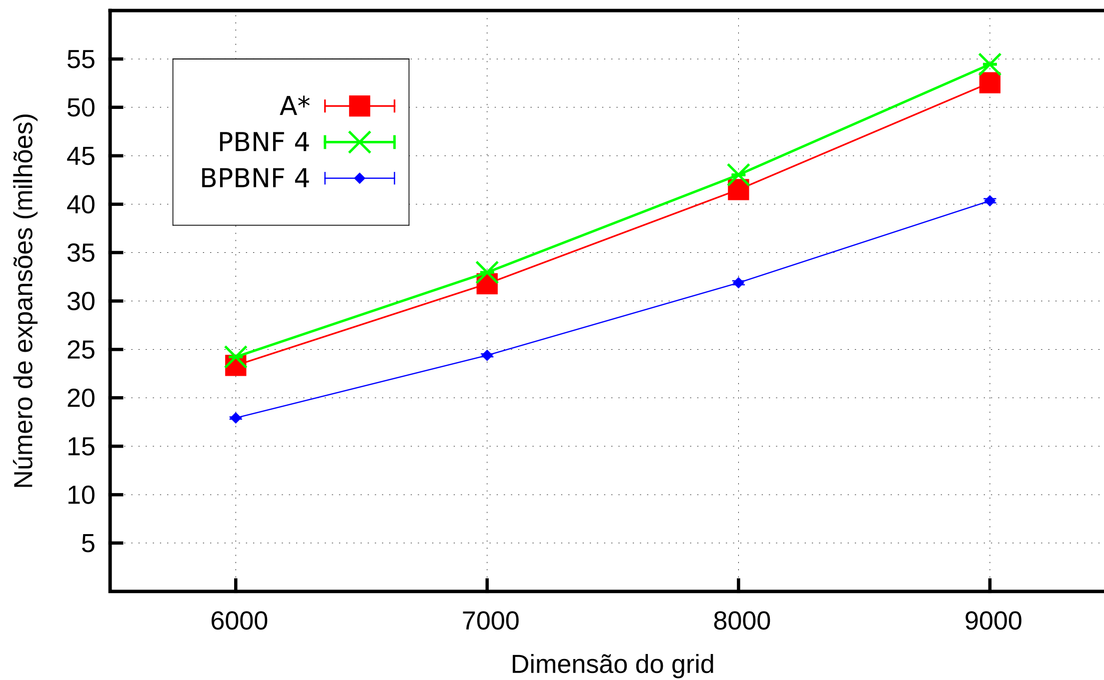



(a) Tempo total em segundos para os vários tamanhos de *grids* quadrados.

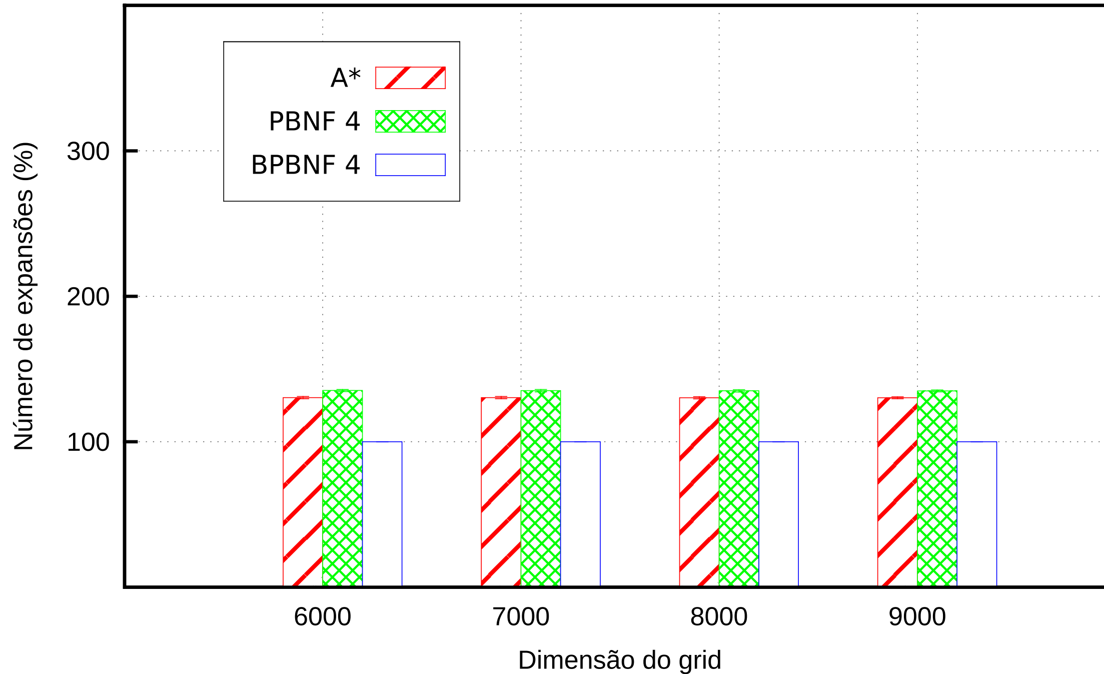



(b) Tempo total percentual para os vários tamanhos de *grids* quadrados.

Figura 4.4: Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme.

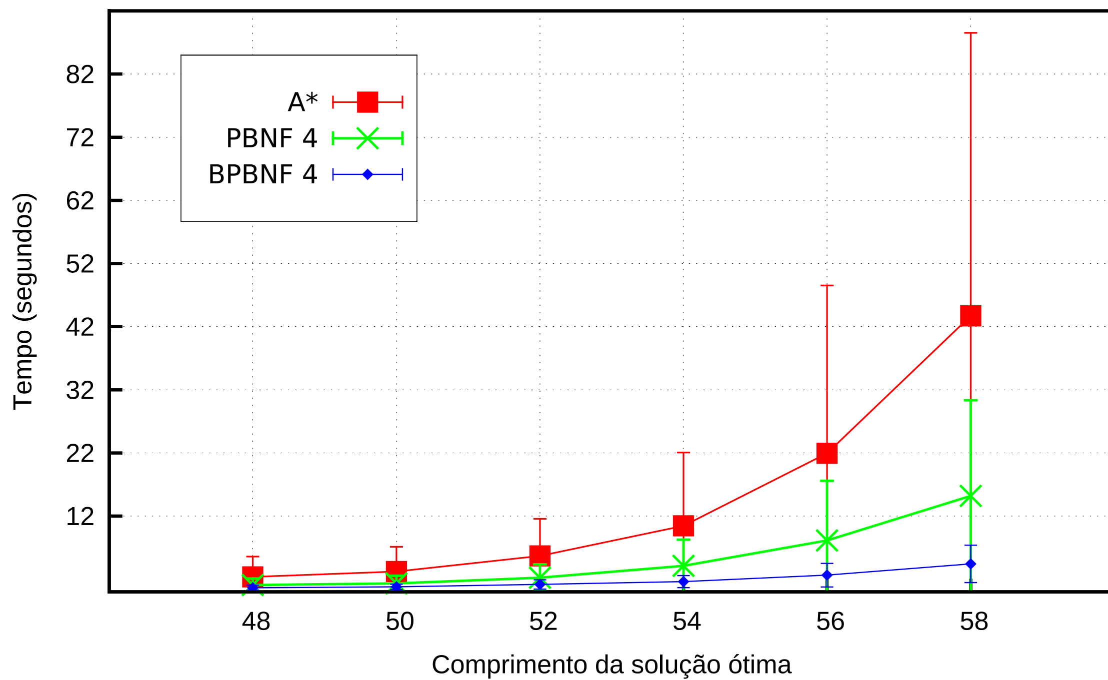



(a) Número de expansões (milhões) para os vários tamanhos de *grids* quadrados.

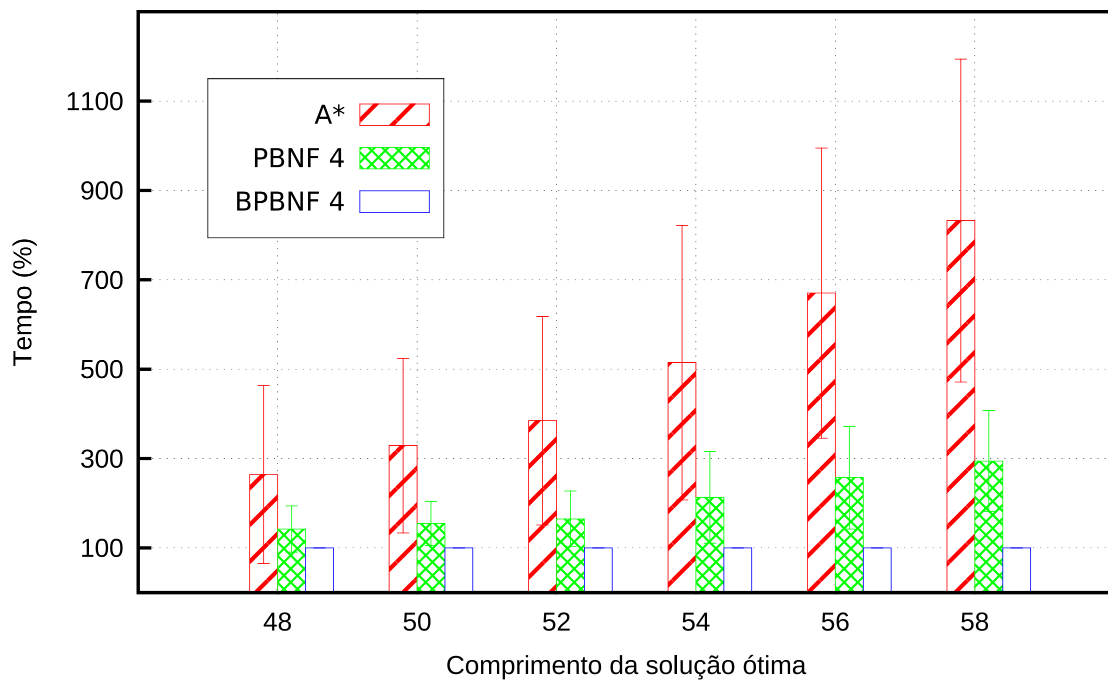



(b) Número de expansões percentual para os vários tamanhos de *grids* quadrados.

Figura 4.5: Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme.




(a) Tempo total em segundos para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.

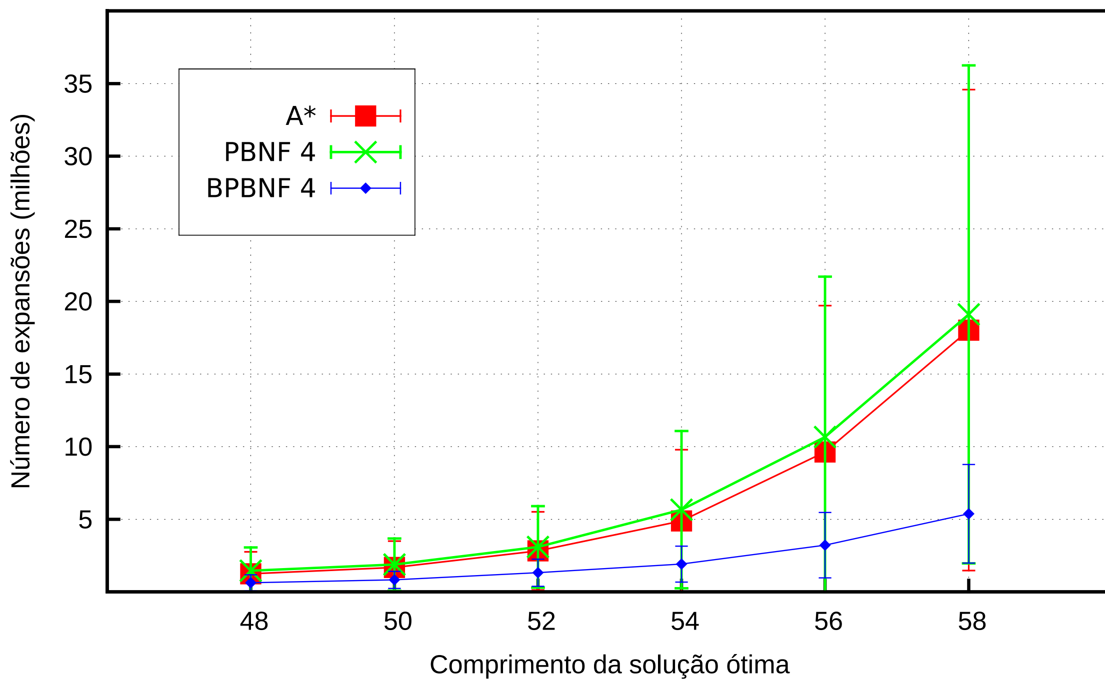



(b) Tempo total percentual para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.

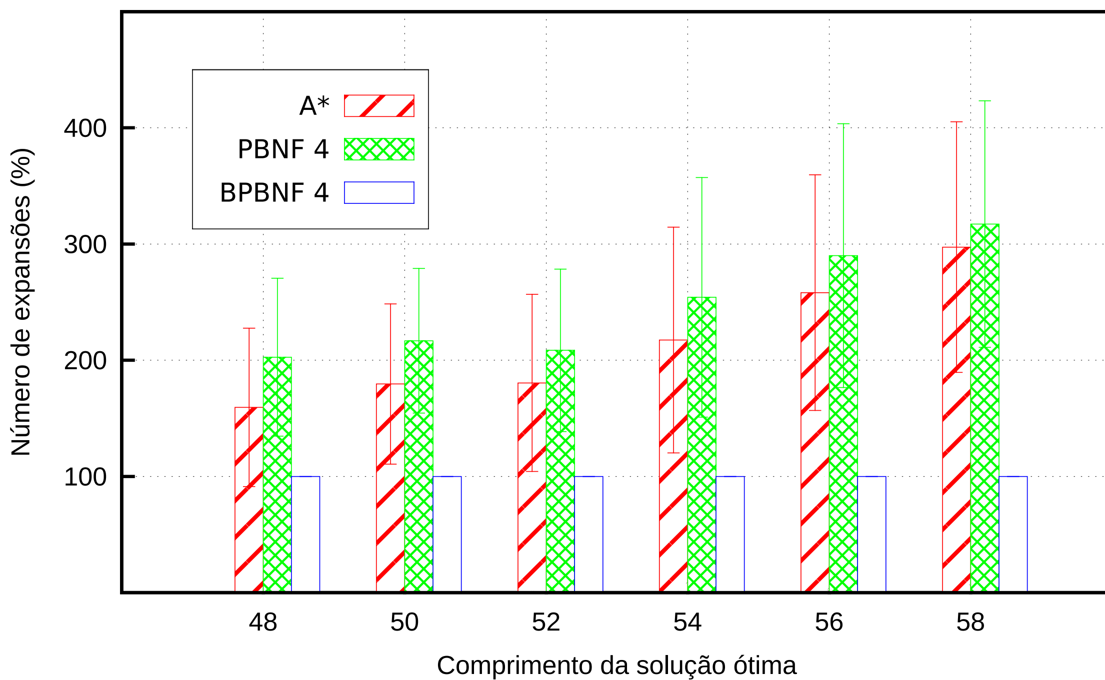
Figura 4.6: Tempo total absoluto e relativo no domínio do *15-puzzle*.

Tabela 4.3: Tempo absoluto para o domínio do *15-puzzle*.

| Comprimento da solução ótima | Algoritmo | Tempo absoluto (segundos) | | | |
|---|---|---|---|---|---|
| | | Inicialização | Execução | Finalização | Total |
| 48 | A* | 0,00006 (0,00000) | 2,10160 (2,79854) | 0,27353 (0,41942) | 2,37519 (3,21409) |
| | PBNF 4 | 0,12748 (0,00293) | 0,61869 (0,73963) | 0,33262 (0,30022) | 1,07879 (1,03959) |
| | BPBNF 4 | 0,14693 (0,00262) | 0,31968 (0,28816) | 0,18745 (0,10124) | 0,65406 (0,38944) |
| 50 | A* | 0,00006 (0,00000) | 2,83321 (3,37844) | 0,38655 (0,54523) | 3,21982 (3,92049) |
| | PBNF 4 | 0,12783 (0,00313) | 0,81192 (0,85358) | 0,40692 (0,34213) | 1,34667 (1,19517) |
| | BPBNF 4 | 0,14673 (0,00265) | 0,42111 (0,30926) | 0,22159 (0,10685) | 0,78943 (0,41633) |
| 52 | A* | 0,00006 (0,00000) | 4,97654 (5,01774) | 0,71992 (0,85201) | 5,69652 (5,86265) |
| | PBNF 4 | 0,12738 (0,00279) | 1,44766 (1,49262) | 0,66661 (0,60902) | 2,24166 (2,09971) |
| | BPBNF 4 | 0,14702 (0,00284) | 0,70823 (0,55217) | 0,31858 (0,18411) | 1,17384 (0,73625) |
| 54 | A* | 0,00006 (0,00001) | 8,99863 (9,73540) | 1,46745 (1,87727) | 10,46614 (11,60521) |
| | PBNF 4 | 0,12957 (0,00269) | 2,75849 (2,89813) | 1,23082 (1,24918) | 4,11887 (4,14645) |
| | BPBNF 4 | 0,14876 (0,00274) | 1,02916 (0,70445) | 0,43398 (0,24644) | 1,61189 (0,95089) |
| 56 | A* | 0,00006 (0,00001) | 18,54786 (22,02585) | 3,38516 (4,58028) | 21,93309 (26,59985) |
| | PBNF 4 | 0,12964 (0,00225) | 5,51645 (6,39898) | 2,48003 (3,08645) | 8,12612 (9,47537) |
| | BPBNF 4 | 0,14942 (0,00235) | 1,79663 (1,37224) | 0,70453 (0,49422) | 2,65059 (1,86634) |
| 58 | A* | 0,00006 (0,00001) | 36,46153 (36,78236) | 7,24735 (8,04592) | 43,70891 (44,81459) |
| | PBNF 4 | 0,12949 (0,00196) | 10,39419 (10,27288) | 4,66180 (4,89530) | 15,18548 (15,16216) |
| | BPBNF 4 | 0,14887 (0,00278) | 3,11034 (2,17789) | 1,17461 (0,78671) | 4,43382 (2,96510) |




(a) Número de expansões (milhões) para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.




(b) Número de expansões percentual para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.

Figura 4.7: Número de expansões absoluto e relativo no domínio do *15-puzzle*.

Tabela 4.4: Número de expansões e tempo total relativo para o domínio do *15-puzzle*.

| Comprimento da solução ótima | Algoritmo | # expansões | | Tempo total relativo (%) | # vezes foi o mais rápido |
|---|---|---|---|---|---|
| | | Relativo (%) | Absoluto (milhões) | | |
| 48 | A* | 159,37 (68,17) | 1,24769 (1,50820) | 263,96 (198,96) | 11 |
| | PBNF 4 | 202,61 (67,95) | 1,45042 (1,60321) | 142,26 (51,77) | 6 |
| | BPBNF 4 | 100,00 (0,00) | 0,62605 (0,54883) | 100,00 (0,00) | 33 |
| 50 | A* | 179,53 (68,97) | 1,68216 (1,81372) | 328,94 (195,40) | 3 |
| | PBNF 4 | 216,75 (62,28) | 1,88039 (1,79809) | 154,86 (49,36) | 5 |
| | BPBNF 4 | 100,00 (0,00) | 0,82380 (0,58792) | 100,00 (0,00) | 42 |
| 52 | A* | 180,48 (76,23) | 2,82133 (2,69316) | 384,67 (233,21) | 4 |
| | PBNF 4 | 208,53 (69,94) | 3,09433 (2,80652) | 165,02 (62,56) | 3 |
| | BPBNF 4 | 100,00 (0,00) | 1,32062 (0,93577) | 100,00 (0,00) | 43 |
| 54 | A* | 217,39 (97,10) | 4,87865 (4,91104) | 514,61 (307,08) | 2 |
| | PBNF 4 | 254,15 (103,05) | 5,65847 (5,41619) | 212,97 (102,50) | 1 |
| | BPBNF 4 | 100,00 (0,00) | 1,91242 (1,23836) | 100,00 (0,00) | 47 |
| 56 | A* | 258,18 (101,47) | 9,63091 (10,07592) | 670,22 (324,50) | 0 |
| | PBNF 4 | 290,08 (113,53) | 10,65678 (11,04655) | 257,27 (114,85) | 2 |
| | BPBNF 4 | 100,00 (0,00) | 3,21605 (2,25108) | 100,00 (0,00) | 48 |
| 58 | A* | 297,38 (107,89) | 18,02112 (16,55817) | 832,61 (361,12) | 0 |
| | PBNF 4 | 317,20 (106,06) | 19,10561 (17,14253) | 294,41 (112,82) | 1 |
| | BPBNF 4 | 100,00 (0,00) | 5,37210 (3,39483) | 100,00 (0,00) | 49 |

#### 4.2.2.3 PNBA* *versus* BPBNF

Esta subseção expõe a comparação dos dois algoritmos introduzidos neste trabalho: PNBA* (capítulo 3) e BPBNF. Optou-se por utilizar o BPBNF com quatro processadores já que o objetivo é comparar o melhor de cada um dos algoritmos. As figuras 4.8 e 4.9 mostram respectivamente o tempo de execução e o número de expansões dos experimentos realizados no domínio do *pathfinding* em *grids* com arestas de custo uniforme para várias dimensões de *grid*. Os dados também são apresentados nas tabelas 4.5 e 4.6. Os gráficos correspondentes ao domínio do *pathfinding* em *grids* com arestas de custo não uniforme estão nas figuras 4.10 e 4.11. As tabelas mencionadas anteriormente também apresentam os seus dados.

Nesses domínios, a superioridade do BPBNF é evidente, principalmente em razão do uso de quatro processadores. Essa é, pois, uma das vantagens do BPBNF. Diferentemente do PNBA*, ele não está limitado a utilização de dois processadores. Além disso, outras características do BPBNF possivelmente também contribuíram para esse resultado com menor influência. A primeira delas é a divisão das fronteiras dos processos de busca em várias estruturas de dados $open_1$ e $open_2$. A outra é a possibilidade dos processadores atuarem, dinamicamente, na exploração da duas fronteiras de busca. Isso permite que foquem o esforço na exploração de porções ainda mais promissoras do espaço de estados.

O número de expansões por ele realizadas foi ligeiramente superior a mesma quantia do PNBA*. A causa foram as expansões especulativas praticadas pelo BPBNF e as posteriores expansões necessárias para propagação da informação correta. No domínio do *pathfinding* em *grids* com arestas de custo não uniforme, a diferença nos tempos de execução dos algoritmos foi maior do que no domínio com arestas de custo uniforme.

Houve uma variação desigual em ambos algoritmos diante do aumento do número de expansões necessárias ao cômputo da solução (conseqüência do uso de uma heurística pobre). Uma possível justificativa para esse acontecimento é o menor custo de operação assintótico do BPBNF devido ao emprego de vários *heaps* binários independentes na representação das fronteiras. Essa quantia varia de acordo com o número de nós do grafo que representa o espaço de estados (pois existem dois para cada *nblock*) enquanto no PNBA* é sempre dois.

As figuras 4.12 e 4.13 mostram respectivamente o tempo de execução e o número de expansões dos experimentos realizados no domínio do *15-puzzle*. Como nos outros domínios, os dados também são apresentados em tabelas (4.7 e 4.8) com o intuito facilitar a compreensão.

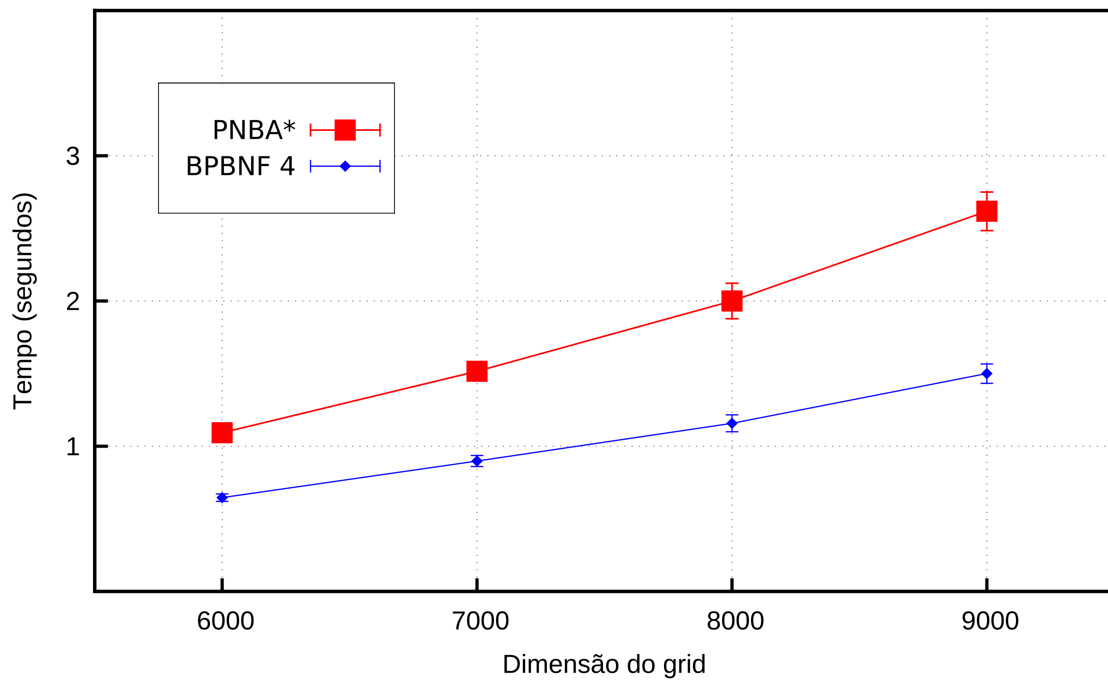
Para configurações cujo comprimento da solução ótima foi 48, o tempo de execução total do PNBA* foi superior. Nelas o tempo da fase de execução era uma parcela pequena do tempo total para o algoritmo BPBNF. Ou seja, o custo da paralelização superou os benefícios na resolução dessas configurações nesse algoritmo.

No entanto, conforme a dificuldade das configurações aumentou, a situação se inverteu, e o BPBNF passou a ser o mais veloz. A duração da fase de execução representou um percentual maior do tempo total de execução para esse algoritmo. Logo, como a fase de execução do BPBNF é mais rápida do que a fase de execução do PNBA* (para as mesmas configurações do *15-puzzle*), o algoritmo obteve um tempo de execução total inferior.

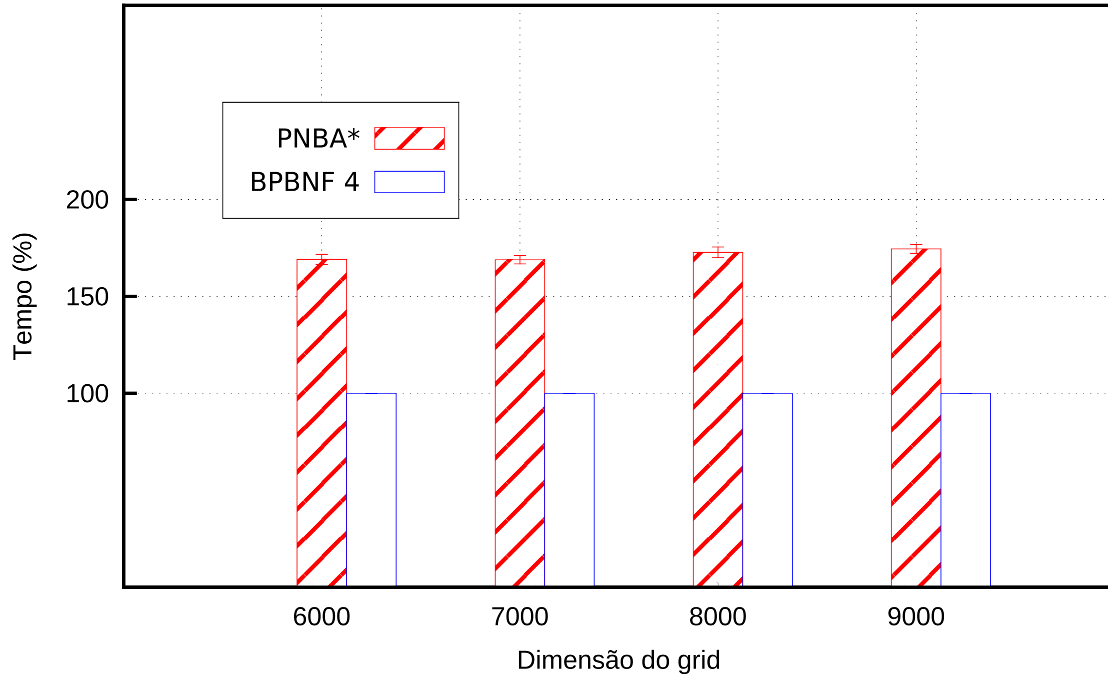
A utilização de mais processadores na busca foi, novamente, a principal causa dessa diferença. Além disso, a existência, para cada *nblock*, de dois *heaps* binários e de uma tabela *hash* também contribuiu para a menor duração da fase de execução do BPBNF. O fato de existirem várias instâncias da última estrutura de dados mencionada diminui as chances de duas *threads* estarem acessando a mesma. Quando isso acontece, há uma elevação do número de sincronizações e um aumento no tempo de resposta das operações.

Diferentemente dos domínios anteriores, o número de expansões realizadas pelo BPBNF foi muito superior ao mesmo valor do PNBA*. Possivelmente a realização das expansões especulativas e a não seleção do menor $f_p$-*value* global no momento de expandir os nós foram as causas. No entanto, os benefícios da modificação (emprego de mais processadores) superaram as adversidades. Para configurações com dificuldade suficientemente grande, o BPBNF foi mais rápido do que o PNBA* nesse domínio.

A superioridade do BPBNF (com quatro processadores) perante o PNBA* ficou evidente em dois dos três domínios avaliados. No contexto do *15-puzzle*, para configurações suficientemente difíceis, ele também obteve o menor tempo de execução total. Cabe ressaltar, porém, que esse aumento na velocidade vem acompanhado de um acréscimo na dificuldade de implementação do algoritmo. Ou seja, o fato de ser mais rápido faz com que também seja mais complexo. O mesmo ocorre com a maioria dos algoritmos discutidos neste trabalho. O contexto onde será aplicado definirá precisamente se os benefício da redução do tempo de execução total valem a complexidade do algoritmo.

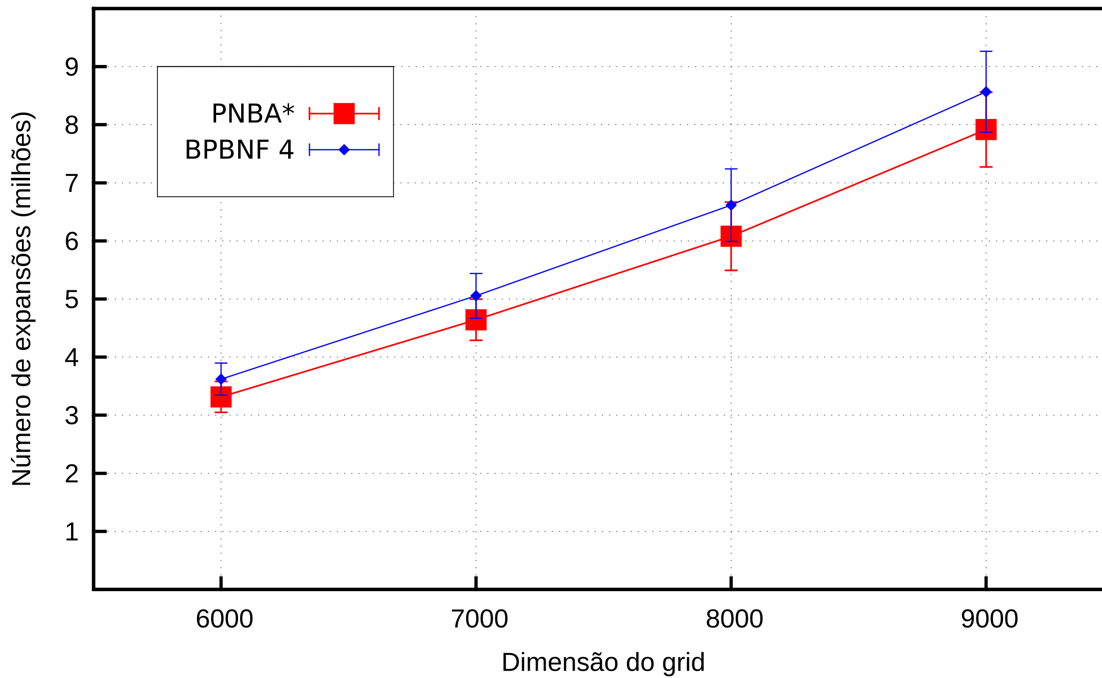



(a) Tempo total em segundos para os vários tamanhos de *grids* quadrados.

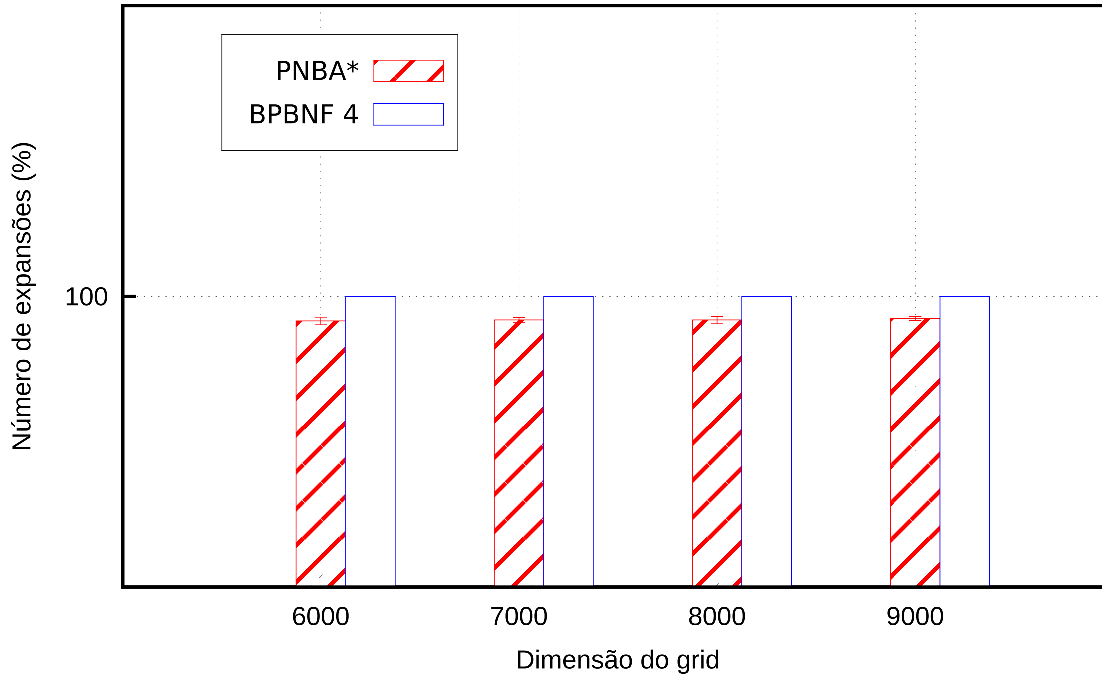



(b) Tempo total percentual para os vários tamanhos de *grids* quadrados.

Figura 4.8: Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme.




(a) Número de expansões (milhões) para os vários tamanhos de *grids* quadrados.

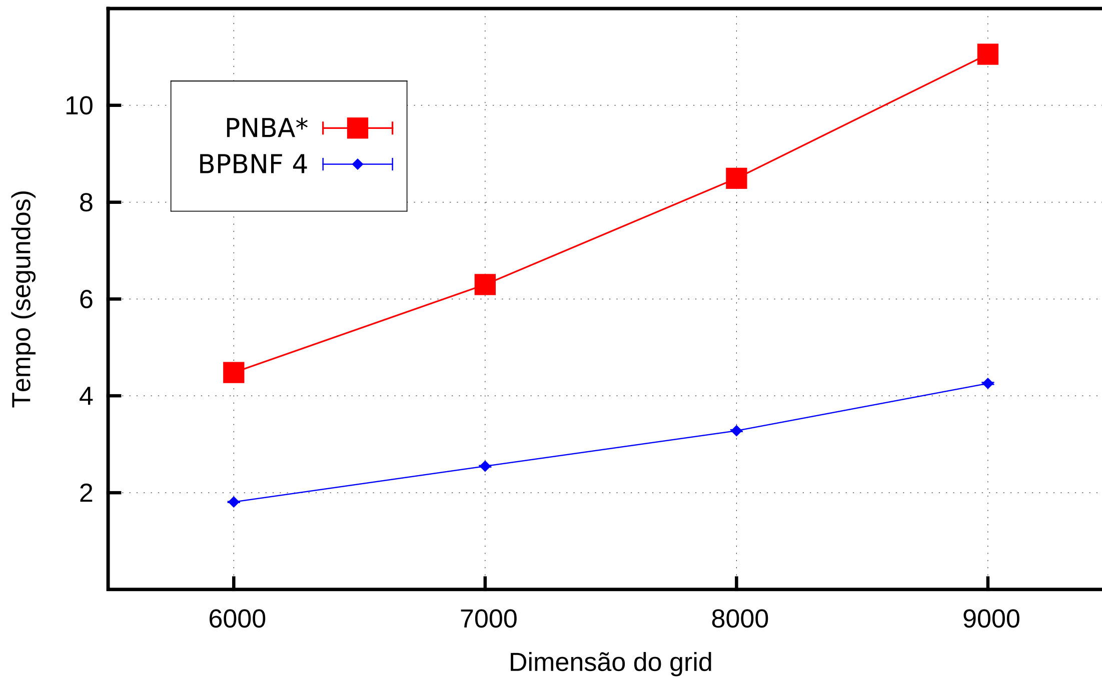



(b) Número de expansões percentual para os vários tamanhos de *grids* quadrados.

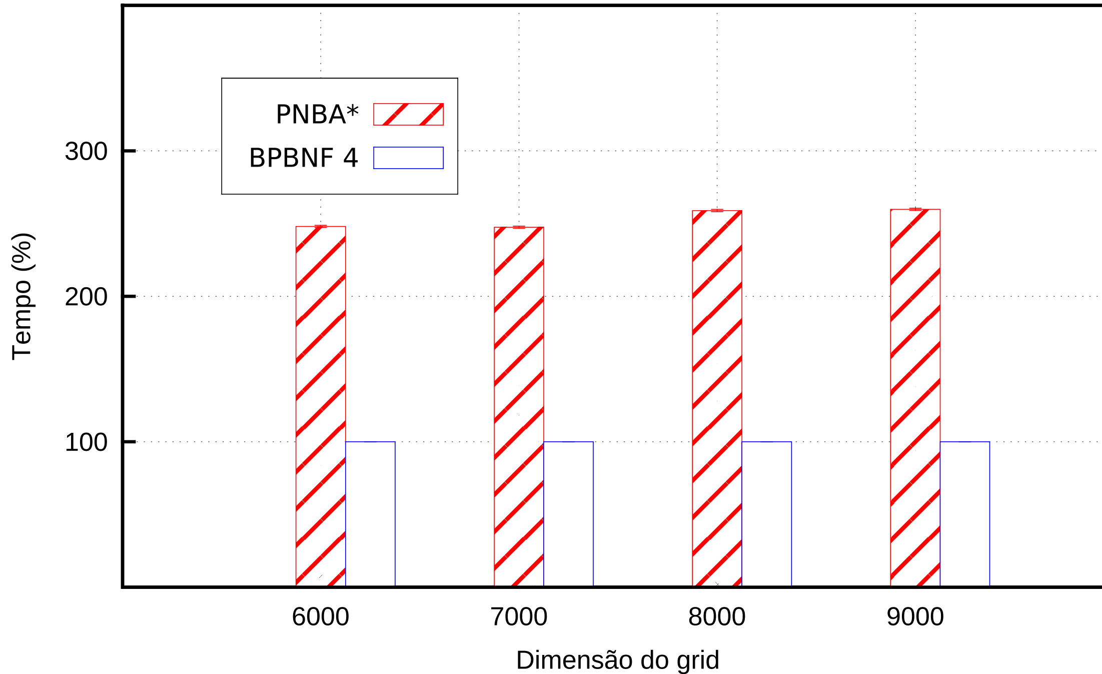
Figura 4.9: Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo uniforme.

Tabela 4.5: Tempo absoluto para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme.

| Dimensão do *grid* | Algoritmo | Tempo absoluto (segundos) | | | |
|---|---|---|---|---|---|
| | | Inicialização | Execução | Finalização | Total |
| *Pathfinding* em *grids* com arestas de custo uniforme | | | | | |
| 6000 | PNBA* | 0,48725 (0,00828) | 0,60450 (0,05175) | 0,00081 (0,00014) | 1,09255 (0,05265) |
| | BPBNF 4 | 0,30505 (0,00252) | 0,33581 (0,02581) | 0,00535 (0,00018) | 0,64621 (0,02545) |
| 7000 | PNBA* | 0,66317 (0,01161) | 0,85212 (0,06622) | 0,00102 (0,00024) | 1,51631 (0,06502) |
| | BPBNF 4 | 0,41086 (0,00311) | 0,47982 (0,03700) | 0,00748 (0,00056) | 0,89816 (0,03761) |
| 8000 | PNBA* | 0,85335 (0,01291) | 1,14557 (0,12483) | 0,00107 (0,00014) | 1,99999 (0,12202) |
| | BPBNF 4 | 0,53621 (0,00471) | 0,61149 (0,05800) | 0,00993 (0,00043) | 1,15763 (0,05808) |
| 9000 | PNBA* | 1,08471 (0,01596) | 1,53178 (0,13236) | 0,00118 (0,00012) | 2,61767 (0,13225) |
| | BPBNF 4 | 0,67905 (0,00690) | 0,80852 (0,06636) | 0,01291 (0,00102) | 1,50047 (0,06658) |
| *Pathfinding* em *grids* com arestas de custo não uniforme | | | | | |
| 6000 | PNBA* | 0,48886 (0,00694) | 3,99189 (0,02713) | 0,00070 (0,00010) | 4,48146 (0,02752) |
| | BPBNF 4 | 0,30564 (0,00294) | 1,49559 (0,00903) | 0,00532 (0,00006) | 1,80655 (0,00934) |
| 7000 | PNBA* | 0,66324 (0,01380) | 5,63532 (0,04051) | 0,00097 (0,00025) | 6,29953 (0,04478) |
| | BPBNF 4 | 0,41089 (0,00282) | 2,12709 (0,01395) | 0,00723 (0,00008) | 2,54521 (0,01435) |
| 8000 | PNBA* | 0,85469 (0,01534) | 7,63804 (0,05230) | 0,00107 (0,00026) | 8,49380 (0,05542) |
| | BPBNF 4 | 0,53707 (0,00438) | 2,73377 (0,01876) | 0,00950 (0,00010) | 3,28034 (0,01934) |
| 9000 | PNBA* | 1,08155 (0,01445) | 9,97486 (0,06397) | 0,00120 (0,00020) | 11,05760 (0,06586) |
| | BPBNF 4 | 0,68103 (0,00674) | 3,56332 (0,02004) | 0,01213 (0,00010) | 4,25648 (0,02058) |

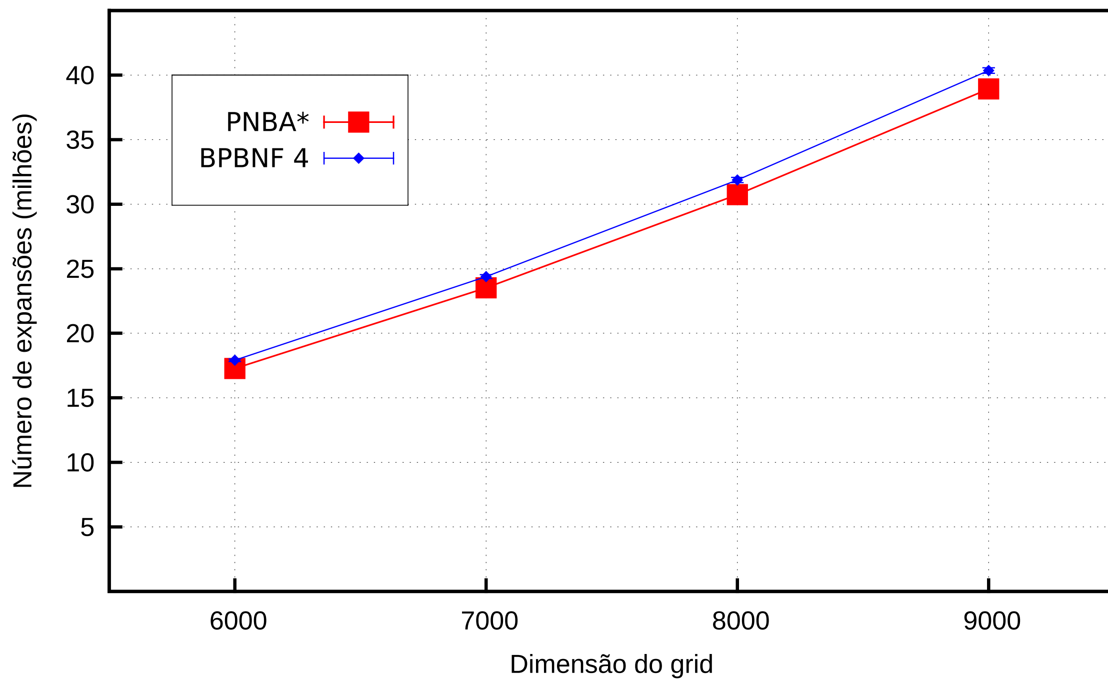



(a) Tempo total em segundos para os vários tamanhos de *grids* quadrados.

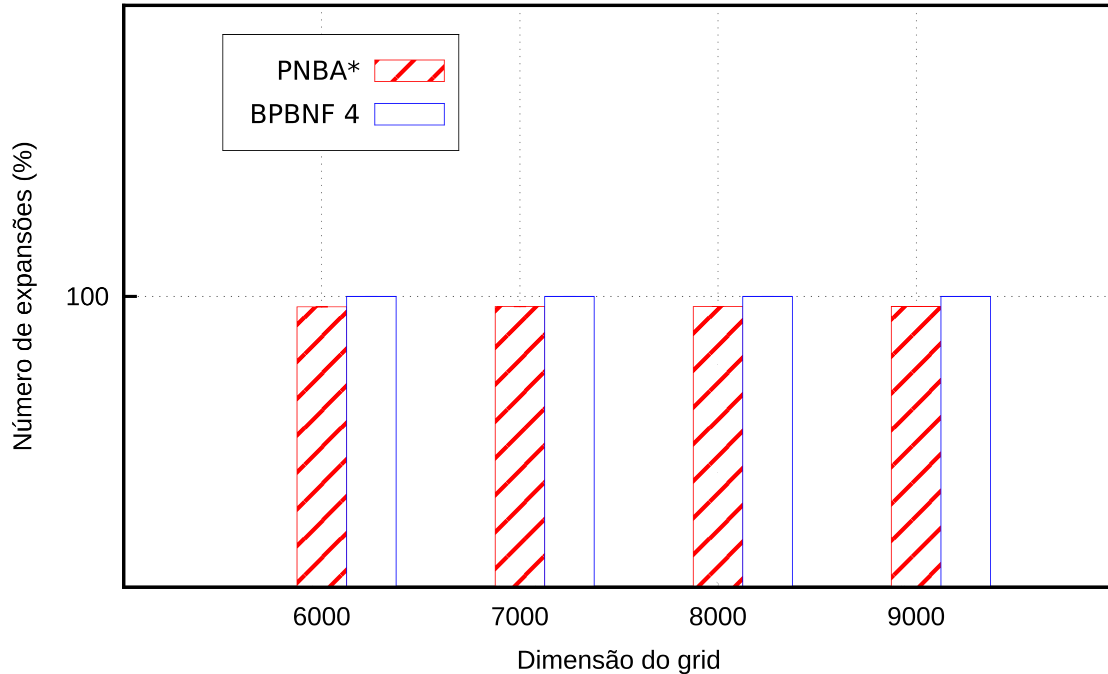



(b) Tempo total percentual para os vários tamanhos de *grids* quadrados.

Figura 4.10: Tempo total absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme.




(a) Número de expansões (milhões) para os vários tamanhos de *grids* quadrados.

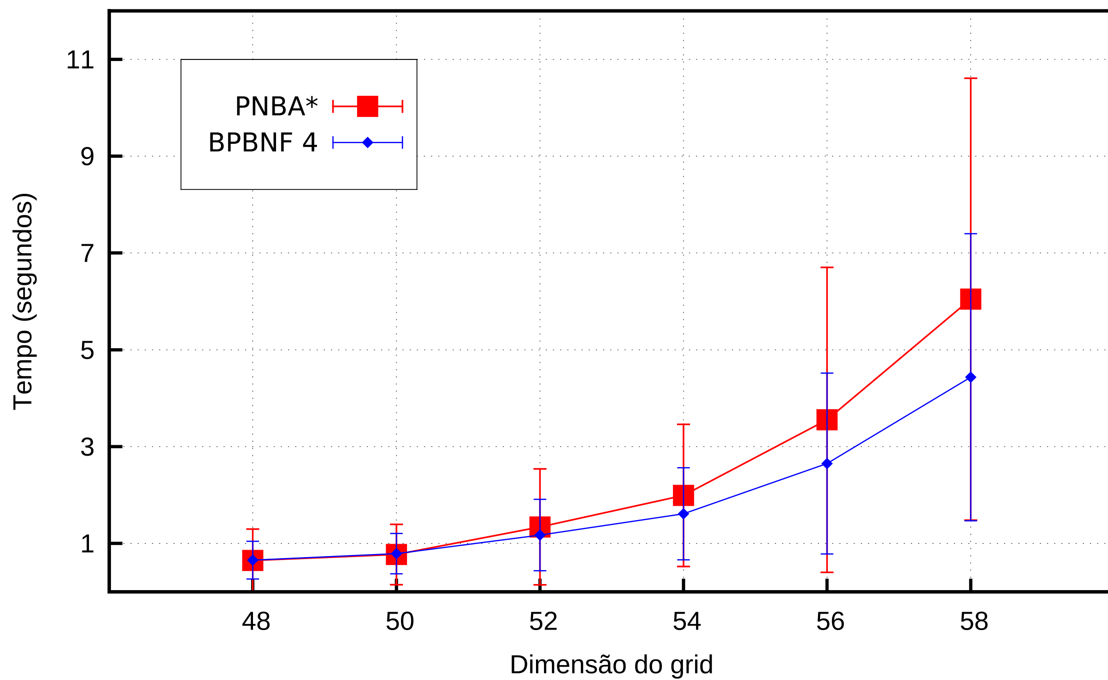



(b) Número de expansões percentual para os vários tamanhos de *grids* quadrados.

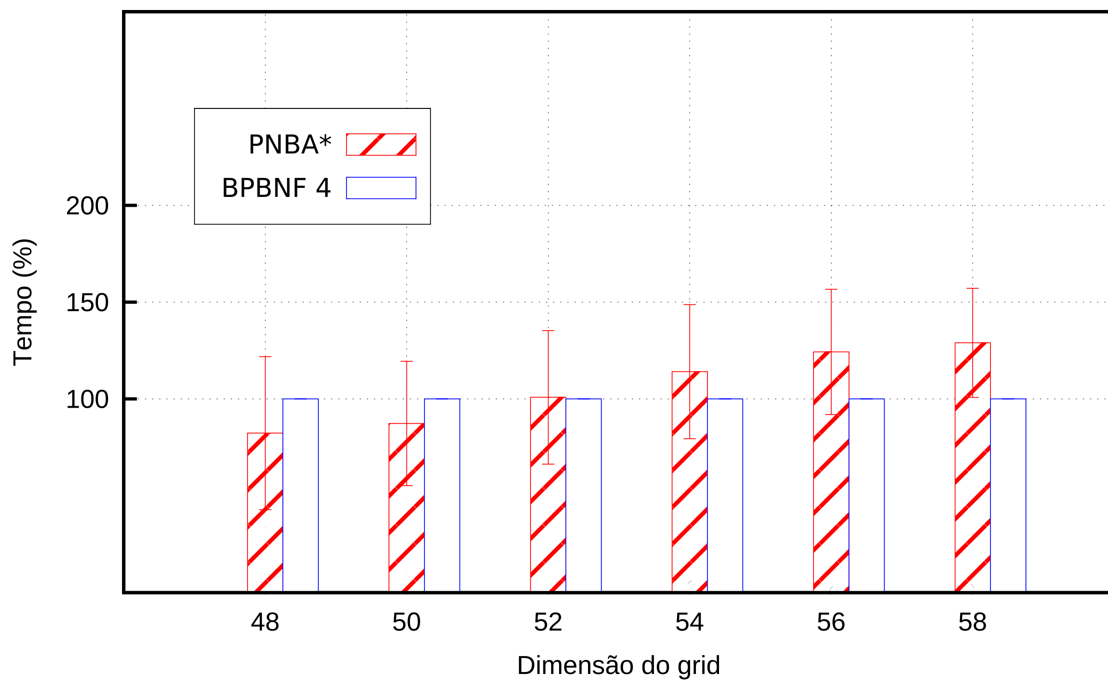
Figura 4.11: Número de expansões absoluto e relativo no domínio do *pathfinding* em *grids* com arestas de custo não uniforme.

Tabela 4.6: Número de expansões e tempo total relativo para o domínio do *pathfinding* em *grids* com arestas de custo uniforme e não uniforme.

| Dimensão do *grid* | Algoritmo | # expansões | | Tempo total relativo (%) | # vezes foi o mais rápido |
|---|---|---|---|---|---|
| | | Relativo (%) | Absoluto (milhões) | | |
| *Pathfinding* em *grids* com arestas de custo uniforme | | | | | |
| 6000 | PNBA* | 91,53 (1,09) | 3,31515 (0,26652) | 169,03 (2,62) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 3,62079 (0,27582) | 100,00 (0,00) | 50 |
| 7000 | PNBA* | 91,86 (0,93) | 4,64295 (0,35472) | 168,83 (2,14) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 5,05468 (0,38469) | 100,00 (0,00) | 50 |
| 8000 | PNBA* | 91,91 (1,15) | 6,08134 (0,58758) | 172,68 (2,79) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 6,61564 (0,62519) | 100,00 (0,00) | 50 |
| 9000 | PNBA* | 92,40 (0,72) | 7,91630 (0,64242) | 174,42 (2,25) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 8,56782 (0,69532) | 100,00 (0,00) | 50 |
| *Pathfinding* em *grids* com arestas de custo não uniforme | | | | | |
| 6000 | PNBA* | 96,39 (0,06) | 17,26995 (0,09634) | 248,07 (0,79) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 17,91738 (0,10400) | 100,00 (0,00) | 50 |
| 7000 | PNBA* | 96,41 (0,04) | 23,51777 (0,13932) | 247,50 (0,78) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 24,39284 (0,14728) | 100,00 (0,00) | 50 |
| 8000 | PNBA* | 96,43 (0,04) | 30,73813 (0,19063) | 258,93 (0,83) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 31,87475 (0,20291) | 100,00 (0,00) | 50 |
| 9000 | PNBA* | 96,45 (0,03) | 38,92189 (0,20227) | 259,78 (0,75) | 0 |
| | BPBNF 4 | 100,00 (0,00) | 40,35309 (0,21296) | 100,00 (0,00) | 50 |




(a) Tempo total em segundos para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.

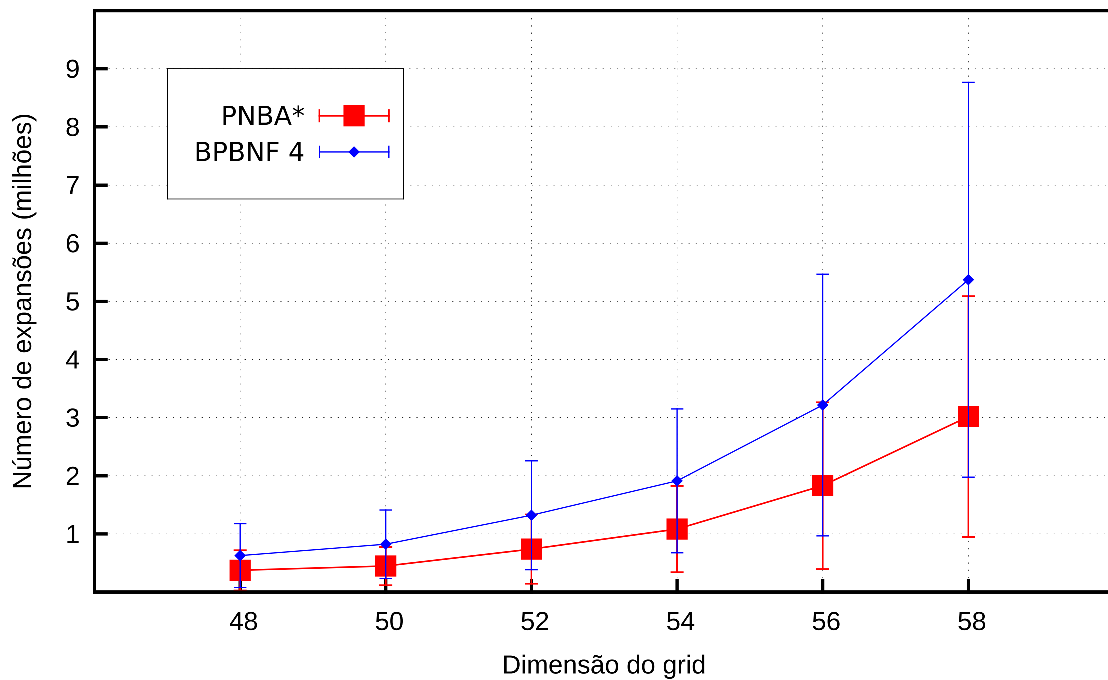



(b) Tempo total percentual para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.

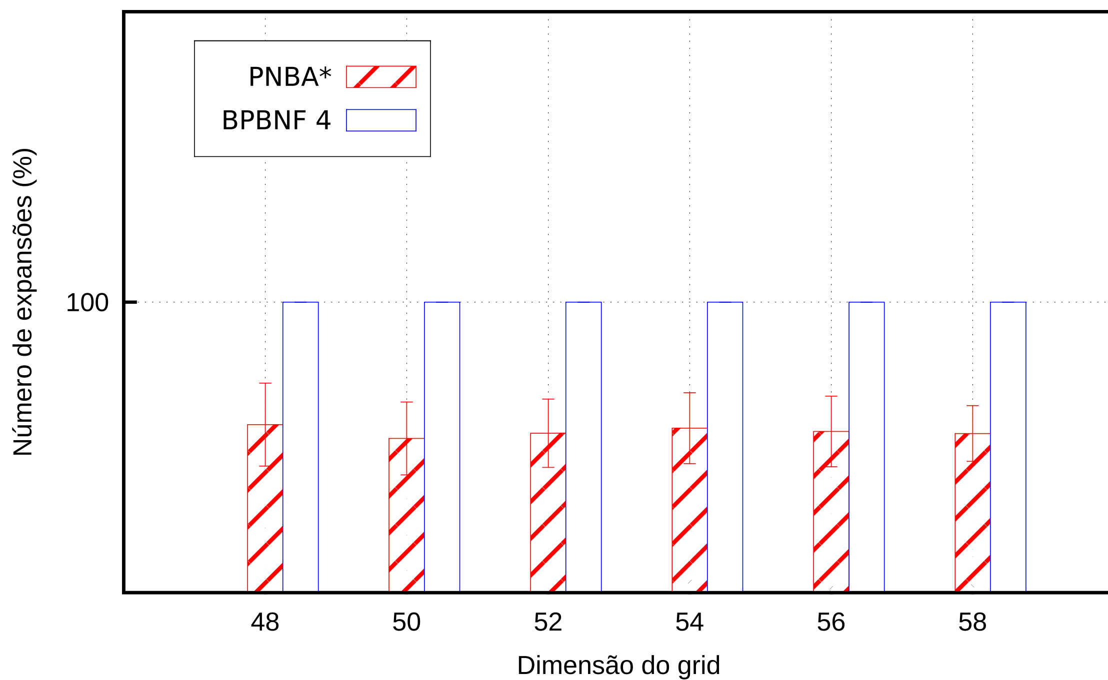
Figura 4.12: Tempo total absoluto e relativo no domínio do *15-puzzle*.

Tabela 4.7: Tempo absoluto para o domínio do *15-puzzle*.

| Comprimento da solução ótima | Algoritmo | Tempo absoluto (segundos) | | | |
|---|---|---|---|---|---|
| | | Inicialização | Execução | Finalização | Total |
| 48 | PNBA* | 0,00008 (0,00001) | 0,57354 (0,57683) | 0,07478 (0,07229) | 0,64840 (0,64898) |
| | BPBNF 4 | 0,14693 (0,00262) | 0,31968 (0,28816) | 0,18745 (0,10124) | 0,65406 (0,38944) |
| 50 | PNBA* | 0,00008 (0,00000) | 0,68246 (0,55509) | 0,08894 (0,06874) | 0,77147 (0,62363) |
| | BPBNF 4 | 0,14673 (0,00265) | 0,42111 (0,30926) | 0,22159 (0,10685) | 0,78943 (0,41633) |
| 52 | PNBA* | 0,00008 (0,00001) | 1,18727 (1,05867) | 0,15386 (0,13976) | 1,34122 (1,19789) |
| | BPBNF 4 | 0,14702 (0,00284) | 0,70823 (0,55217) | 0,31858 (0,18411) | 1,17384 (0,73625) |
| 54 | PNBA* | 0,00008 (0,00000) | 1,76183 (1,29703) | 0,23000 (0,17207) | 1,99191 (1,46827) |
| | BPBNF 4 | 0,14876 (0,00274) | 1,02916 (0,70445) | 0,43398 (0,24644) | 1,61189 (0,95089) |
| 56 | PNBA* | 0,00009 (0,00001) | 3,10902 (2,68062) | 0,44395 (0,47417) | 3,55306 (3,15070) |
| | BPBNF 4 | 0,14942 (0,00235) | 1,79663 (1,37224) | 0,70453 (0,49422) | 2,65059 (1,86634) |
| 58 | PNBA* | 0,00009 (0,00001) | 5,20672 (3,79743) | 0,83880 (0,77572) | 6,04561 (4,56321) |
| | BPBNF 4 | 0,14887 (0,00278) | 3,11034 (2,17789) | 1,17461 (0,78671) | 4,43382 (2,96510) |




(a) Número de expansões (milhões) para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.




(b) Número de expansões percentual para configurações do *15-puzzle* com solução ótima de diferentes comprimentos.

Figura 4.13: Número de expansões absoluto e relativo no domínio do *15-puzzle*.

Tabela 4.8: Número de expansões e tempo total relativo para o domínio do *15-puzzle*.

| Comprimento da solução ótima | Algoritmo | # expansões | | Tempo total relativo (%) | # vezes foi o mais rápido |
|---|---|---|---|---|---|
| | | Relativo (%) | Absoluto (milhões) | | |
| 48 | PNBA* | 57,84 (14,30) | 0,37440 (0,34519) | 82,35 (39,50) | 35 |
| | BPBNF 4 | 100,00 (0,00) | 0,62605 (0,54883) | 100,00 (0,00) | 15 |
| 50 | PNBA* | 53,10 (12,54) | 0,44675 (0,32810) | 87,36 (32,08) | 31 |
| | BPBNF 4 | 100,00 (0,00) | 0,82380 (0,58792) | 100,00 (0,00) | 19 |
| 52 | PNBA* | 54,87 (11,76) | 0,73934 (0,59693) | 100,84 (34,48) | 25 |
| | BPBNF 4 | 100,00 (0,00) | 1,32062 (0,93577) | 100,00 (0,00) | 25 |
| 54 | PNBA* | 56,62 (12,21) | 1,08287 (0,74142) | 114,12 (34,65) | 14 |
| | BPBNF 4 | 100,00 (0,00) | 1,91242 (1,23836) | 100,00 (0,00) | 36 |
| 56 | PNBA* | 55,48 (12,16) | 1,82961 (1,43426) | 124,28 (32,35) | 11 |
| | BPBNF 4 | 100,00 (0,00) | 3,21605 (2,25108) | 100,00 (0,00) | 39 |
| 58 | PNBA* | 54,79 (9,63) | 3,01757 (2,07243) | 128,98 (28,17) | 6 |
| | BPBNF 4 | 100,00 (0,00) | 5,37210 (3,39483) | 100,00 (0,00) | 44 |

# Capítulo 5

# Conclusão

Este trabalho investigou o uso em conjunto dos paradigmas bidirecional e paralelo no A*. O A* é um dos algoritmos de busca heurística mais importantes em Inteligência Artificial. Apesar da diminuição significativa no esforço computacional da busca proporcionada pela adoção da heurística, em muitos contextos isso não é suficiente para a aplicação do A*. No pior caso, a sua complexidade de tempo e de espaço são exponenciais.

A comunidade científica, com o intuito de encontrar formas de lidar melhor com essa questão, tem despendido um grande esforço. Conseqüentemente, várias extensões do algoritmo A* tem sido propostas. As contribuições deste trabalho estão diretamente relacionadas com essas extensões e são uma forma de organizar os principais algoritmos de busca baseados no A* e dois novos algoritmos de busca heurística bidirecional paralela.

A classificação das extensões do A* exposta neste trabalho é uma forma de organizar os principais algoritmos de busca baseados no A* que foram propostos na literatura. Ela é estruturada em seis classes (bidirecional, incremental, *memory-concerned*, paralela, *anytime* e tempo-real) não excludentes entre si (ou seja, um algoritmo pode pertencer a mais de uma delas). Trata-se de um instrumento importante, pois fornece uma visão geral das alternativas existentes e uma direção no caso de soluções mais específicas.

O primeiro algoritmo de busca heurística bidirecional paralela apresentado foi o PNBA*. Ele é uma implementação paralela do NBA* para ambientes computacionais de memória compartilhada. A idéia é executar os dois processos de busca em paralelo ao invés de alternar as ações correspondentes aos seus turnos. Em todos os domínios empregados, o PNBA* foi significativamente mais rápido do que o A* e o NBA*. O número de expansões efetuadas pelos algoritmos bidirecionais foi pra-

ticamente equivalente. A redução do tempo de execução em relação ao NBA* foi conseqüência da paralelização do algoritmo.

O BPBNF, algoritmo de busca heurística bidirecional paralela também introduzido neste trabalho, é uma forma de generalizar a idéia do algoritmo PNBA* para mais de dois processadores e também um modo de melhorar o tempo de execução do algoritmo PBNF (algoritmo de busca heurística paralela no qual ele se baseia). A modificação central para aplicar o paradigma bidirecional ao PBNF envolveu permitir que o espaço de busca definido por um *nblock* fosse explorado simultaneamente por dois processadores (cada um em uma direção distinta).

Quando os desempenhos dele e dos algoritmos A* e PBNF foram comparados empiricamente, evidenciou-se uma clara superioridade do BPBNF em relação ao A*. Se comparado ao PBNF, em dois dos três domínios empregados também foi possível notar a supremacia do BPBNF. Essa redução do tempo de execução foi conseqüência da aplicação do paradigma bidirecional ao algoritmo. Em relação ao PNBA*, a superioridade do BPBNF (empregando quatro processadores) ficou evidente em dois dos três domínios avaliados. No contexto do *15-puzzle*, para configurações suficientemente difíceis, ele também obteve o menor tempo de execução total.

Esses resultados atendem os objetivos deste trabalho e comprovam que a união de características de diferentes classes de extensões do A* pode ser benéfica. A utilização de dois processos de busca permitiu a paralelização do NBA* e também a "bidirecionalização" do PBNF. Os algoritmos de busca heurística bidirecional paralela combinaram os paradigmas bidirecional e paralelo de um modo que acumulou os seus benefícios. Associados, o paralelismo que permite expandir vários nós ao mesmo tempo e o paradigma bidirecional que reduz o número de expansões necessárias ao cômputo da solução ótima obtiveram melhores desempenhos do que as respectivas técnicas separadas. Esses algoritmos de busca heurística bidirecional paralela também obtiveram tempos de execução inferiores ao do A*. Logo, a combinação desses paradigmas é, igualmente, uma forma viável de reduzir as demandas computacionais do A*.

Entretanto, a implementação de ambos algoritmos é mais complexa do que a do A*, principalmente no caso do BPBNF. Ela requer a utilização de primitivas de sincronização e estruturas de dados capazes de lidar com acesso concorrente de modo inteligente. Também é necessário a utilização de um hardware mais específico, ou seja, é preciso que os computadores disponham de vários processadores que acessam uma memória compartilhada. Felizmente, a tendencia atual dos novos processadores (disseminação de máquinas *multicore*) atenua essa última restrição.

## 5.1 Trabalhos futuros

A seguir, apresentam-se uma lista de possíveis continuações deste trabalho. Algumas delas estão relacionadas com as suas limitações.

- **Aprimoramento da avaliação empírica**: os três domínios empregados na avaliação experimental cobrem várias situações. No entanto, domínios cujo o tempo de expansão de um nó é grande se comparado a outras operações realizadas pelos algoritmos não foram contemplados. O problema do caixeiro viajante seria uma opção, apesar de existirem diversas abordagens possivelmente melhores para solução desse problema. A intenção, pois, é através do problema do caixeiro viajante explorar características de outros domínios, onde a busca em um espaço de estados é uma solução viável, na avaliação dos algoritmos de busca heurística.

  Além disso, a coleta de mais métricas pode facilitar o entendimento/explicação de alguns comportamentos e resultados. Algumas possibilidades são: duração das duas fases de execução (no caso de algoritmos bidirecionais), número de nós podados, percentual de utilização dos processadores ao longo da computação, número de vezes que cada nó foi expandido, quantidade total de memória empregada e o total de nós gerados e visitados. Um dos questionamentos, por exemplo, a ser respondido é se a quantidade de memória total empregada pelos algoritmos bidirecionais é inferior à respectiva quantia dos algoritmos unidirecionais. Apesar de realizarem um número de expansões inferior, os algoritmos bidirecionais armazenam um número de atributos maior para cada nó.

- **Combinações de outras classes de extensões do A***: este trabalho explorou abordagens bidirecionais e paralelas através da combinação dessas classes. Algo semelhante já foi realizado entre as classes *memory-concerned* e paralela [Cook & Varnell, 1998]. No entanto, desconhece-se na literatura tentativas de combinar os paradigmas incremental e paralelo com o intuito de reduzir o tempo de execução total;

- **Provas da admissibilidade dos algoritmos PNBA* e BPBNF**: a admissibilidade dos dois algoritmos aqui introduzidos não foi formalmente justificada. Durante as discussões apenas apresentou-se uma argumentação cujo o foco foi a intuição. Uma das dificuldades para realização dessa tarefa é justamente o paralelismo dos algoritmos que demanda utilização de recursos mais com-

plexos nas argumentações. Uma versão preliminar com as provas da admissibilidade do PNBA* já está pronta;

# Referências Bibliográficas

Al-Ayyoub, A. E. (2005). Distributed Unidirectional and Bidirectional Heuristic Search: Algorithm Design and Empirical Assessment. *The Journal of Supercomputing*, 32(3):231--250.

Berenson, D.; Srinivasa, S. S.; Ferguson, D.; Collet, A. & Kuffner, J. J. (2009). Manipulation planning with Workspace Goal Regions. *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pp. 618--624.

Björnsson, Y.; Bulitko, V. & Sturtevant, N. (2009). TBA*: time-bounded A*. Em *Proceedings of the International Joint Conference on Artifical intelligence*, pp. 431--436.

Botea, A.; Müller, M. & Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of Game Development*, 1:7--28.

Bourg, D. M. & Seemann, G. (2004). *AI for Game Developers*. O'Reilly Media, Inc.

Bulitko, V.; Björnsson, Y.; Sturtevant, N. & Lawrence, R. (2010). Real-time Heuristic Search for Pathfinding in Video Games. Em *Artificial Intelligence for Computer Games*. Springer.

Burns, E.; Lemons, S.; Zhou, R. & Ruml, W. (2009). Best-First Heuristic Search for Multi-Core Machines. Em *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 449–455.

Choset, H.; Lynch, K. M.; Hutchinson, S.; Kantor, G. A.; Burgard, W.; Kavraki, L. E. & Thrun, S. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press.

Cook, D. J. & Varnell, R. C. (1998). Adaptive parallel iterative deepening search. *Journal of Artificial Intelligence Research*, 9:139--166.

de Champeaux, D. & Sint, L. (1977). An Improved Bidirectional Heuristic Search Algorithm. *Journal of the ACM*, 24(2):177--191.

Dijkstra (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269--271.

Dillenburg, J. F. & Nelson, P. C. (1994). Perimeter Search. *Artificial Intelligence*, 65(1):165--178.

Edelkamp, S.; Jabbar, S. & Schroedl, S. (2004). External A*. Em *Proceedings of the Annual German Conference on Artificial Intelligence*, volume 3238, pp. 226--240.

Edelkamp, S. & Schrödl, S. (2012). *Heuristic Search - Theory and Applications*. Morgan Kaufmann.

Evett, M. P.; Hendler, J. A.; Mahanti, A. & Nau, D. S. (1995). PRA*: Massively Parallel Heuristic Search. *J. Parallel Distrib. Comput*, 25(2):133--143.

Felner, A.; Kraus, S. & Korf, R. E. (2003). KBFS: K-Best-First Search. *Annals of Mathematics and Artificial Intelligence*, 39(1-2):19--39.

Ferguson, D. & Stentz, A. (2005). Field D*: An Interpolation-Based Path Planner and Replanner. Em *Proceedings of the International Symposium of Robotics Research*, pp. 1926--1931.

Ghemawat, S. & Menage, P. (2012). TCMalloc : Thread-Caching Malloc. Acessado dia 29 de janeiro de 2012 através do endereço: `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

Hansen, E. A. & Zhou, R. (2007). Anytime Heuristic Search. *Journal of Artificial Intelligence Research*, 28:267--297.

Hart, P. E.; Nilsson, N. J. & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100--107.

Hawes, N. A. (2003). *Anytime Deliberation For Computer Game Agents*. Tese de doutorado, University of Birmingham.

Hillier, F. S.; Lieberman, G. J.; Hillier, F. & Lieberman, G. (2004). *Introduction to Operations Research*. McGraw-Hill.

Hourtash, A. & Tarokh, M. (2001). Manipulator path planning by decomposition: algorithm and analysis. *IEEE Transactions on Robotics*, 17(6):842--856.

Ikeda, T.; Hsu, M.-Y.; Imai, H.; Nishimura, S.; Shimoura, H.; Hashimoto, T.; Tenmoku, K. & Mitoh, K. (1994). A fast algorithm for finding better routes by AI search techniques. Em *Proceedings of the Vehicle Navigation and Information Systems Conference*, pp. 291--296.

Kaindl, H. & Kainz, G. (1997). Bidirectional Heuristic Search Reconsidered. *Journal of Artificial Intelligence Research*, 7:283--317.

Kishimoto, A.; Fukunaga, A. S. & Botea, A. (2009). Scalable, Parallel Best-First Search for Optimal Sequential Planning. Em *Proceedings of the International Conference on Automated Planning and Scheduling*.

Klunder, G. A. & Post, H. N. (2006). The shortest path problem on large-scale real-road networks. *Networks*, 48(4):182--194.

Koenig, S. (2001). Agent-Centered Search. *AI Magazine*, 22(4).

Koenig, S. (2011). Dynamic Fringe-Saving A*. Acessado dia 24 de outubro de 2011 através do endereço: `http://idm-lab.org/bib/abstracts/Koen09e.html`.

Koenig, S. & Likhachev, M. (2002). Improved Fast Replanning for Robot Navigation in Unknown Terrain. Em *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 968--975.

Koenig, S. & Likhachev, M. (2006a). A New Principle for Incremental Heuristic Search: Theoretical Results. Em *Proceedings of the International Conference on Autonomous Planning and Scheduling*, pp. 402--405.

Koenig, S. & Likhachev, M. (2006b). Real-time adaptive A*. Em *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 281--288.

Koenig, S.; Likhachev, M. & Furcy, D. (2004a). Lifelong Planning A*. *Artificial Intelligence*, 155(1-2):93--146.

Koenig, S.; Likhachev, M.; Liu, Y. & Furcy, D. (2004b). Incremental Heuristic Search in Artificial Intelligence. *AI Magazine*, 25:99--112.

Korf, R. E. (1985). Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97--109.

Korf, R. E. (1990). Real-Time Heuristic Search. *Artificial Intelligence*, 42(2-3):189--211.

Korf, R. E. (1993). Linear-space best-first search. *Artificial Intelligence*, 62(1):41--78.

Korf, R. E. (1996). Artificial Intelligence Search Algorithms. Relatório técnico, University of California.

Korf, R. E. (2004). Best-First Frontier Search with Delayed Duplicate Detection. Em *Proceedings of the National Conference on Artificial Intelligence, Conference on Innovative Applications of Artificial Intelligence*, pp. 650--657.

Korf, R. E.; Zhang, W.; Thayer, I. & Hohwald, H. (2005). Frontier search. *Journal of the ACM*, 52(5):715--748.

Likhachev, M.; Ferguson, D. I.; Gordon, G. J.; Stentz, A. & Thrun, S. (2005). Anytime Dynamic A*: An Anytime, Replanning Algorithm. Em *Proceedings of the International Conference on Automated Planning and Scheduling*, pp. 262--271.

Likhachev, M.; Gordon, G. & Thrun, S. (2003a). ARA*: Formal Analysis. Relatório técnico, Carnegie Mellon University.

Likhachev, M.; Gordon, G. J. & Thrun, S. (2003b). ARA*: Anytime A* with Provable Bounds on Sub-Optimality. Em *Proceedings of the Neural Information Processing Systems Conference*.

Marques, V.; Chaimowicz, L. & Ferreira, R. (2011). Speeding Up Learning in Real-Time Search through Parallel Computing. Em *Proceedings of the International Symposium on Computer Architecture and High Performance Computing*, pp. 176--182.

Millington, I. (2006). *Artificial Intelligence for Games*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Nelson, P. C. & Toptsis, A. A. (1992). Unidirectional and Bidirectional Search Algorithms. *IEEE Software*, 9(2):77--83.

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers, San Francisco.

Orkin, J. (2003). Applying Goal-Oriented Planning for Games. Em *AI Game Programming Wisdom 2*. Charles River Media.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pijls, W. & Post, H. (2009). Yet another bidirectional algorithm for shortest paths. Relatório técnico EI 2009-10, Erasmus University Rotterdam, Econometric Institute.

Pijls, W. & Post, H. (2010). Note on "A new bidirectional algorithm for shortest paths". *European Journal of Operational Research*, 207(2):1140--1141.

Pizzi, D.; Lugrin, J.-L.; Whittaker, A. & Cavazza, M. (2010). Automatic Generation of Game Level Solutions as Storyboards. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(3):149--161.

Pohl, I. (1969). Bi-directional and heuristic search in path problems. Relatório técnico 104, SLAC (Stanford Linear Accelerator Center), Stanford, California.

Politowski, G. & Pohl, I. (1984). D-Node Retargeting in Bidirectional Heuristic Search. Em *Proceedings the National Conference on Artificial Intelligence*, pp. 274--277.

Qian, K.; Nymeyer, A. & Susanto, S. (2005). Abstraction-Guided Model Checking Using Symbolic IDA* and Heuristic Synthesis. Em *Proceedings of the International Formal Techniques for Networked and Distributed Systems Conference*, volume 3731, pp. 275--289.

Ratner, D. & Warmuth, M. (1986). Finding a Shortest Solution for the $N \times N$ Extension of the 15-PUZZLE is Intractable. Em *Proceedings of the National Conference on Artificial Intelligence*, pp. 168--172.

Ratner, D. & Warmuth, M. (1990). The $(n^2 - 1)$-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10:111--137.

Rios, L. H. O. & Chaimowicz, L. (2009). trAIns: An Artificial Inteligence for OpenTTD. Proceedings of SBGames, pp. 52--63. IEEE Computer Society.

Rios, L. H. O. & Chaimowicz, L. (2010). A Survey and Classification of A* Based Best-First Heuristic Search Algorithms. Em *Proceedings of SBIA*, volume 6404, pp. 253--262. Springer.

Rios, L. H. O. & Chaimowicz, L. (2011). PNBA*: A Parallel Bidirectional Heuristic Search Algorithm. Em *Proceedings of CSBC 2011 - ENIA*, Natal (RN).

Russell, S. (1992). Efficient Memory-Bounded Search Methods. Em *Proceedings of the European Conference on Artificial Intelligence*, pp. 1--5.

Russell, S. J. & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education.

Sohn, A. (1993). Parallel Bidirectional A* Search on a Symmetry Multiprocessor. Em *Proceedings of the Symposium on Parallel and Distributed Systems*, pp. 19--22.

Stentz, A. (1994). Optimal and Efficient Path Planning for Partially-Known Environments. Em *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 3310--3317.

Stentz, A. (1995). The Focussed D* Algorithm for Real-Time Replanning. Em *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1652--1659.

Sun, X. & Koenig, S. (2007). The Fringe-Saving A* Search Algorithm - A Feasibility Study. Em *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 2391--2397.

Sun, X.; Koenig, S. & Yeoh, W. (2008). Generalized Adaptive A*. Em *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 469--476. IFAAMAS.

Sun, X.; Yeoh, W. & Koenig, S. (2009). Dynamic fringe-saving A*. Em *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 891--898.

Sun, X.; Yeoh, W. & Koenig, S. (2010). Moving target D* Lite. Em *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, pp. 67--74.

Szer, D. & Charpillet, F. (2005). MAA*: A heuristic search algorithm for solving decentralized POMDPs. Em *In Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, pp. 576--583.

van den Berg, J.; Shah, R.; Huang, A. & Goldberg, K. Y. (2011). ANA*: Anytime Nonparametric A*. Em *Proceedings of AAAI Conference on Artificial Intelligence*.

Vempaty, N. R.; Kumar, V. & Korf, R. E. (1991). Depth-First Versus Best-First Search. Em *AAAI*, pp. 434--440.

Vidal, V.; Bordeaux, L. & Hamadi, Y. (2010). Adaptive K-Parallel Best-First Search: A Simple but Efficient Algorithm for Multi-Core Domain-Independent Planning. Em *Proceedings of the Annual Symposium on Combinatorial Search*.

Wang, L. & Jiang, T. (1994). On the complexity of multiple sequence alignment. *Journal of computational biology : a journal of computational molecular cell biology*, (4):337--348.

Yoo, H. & Lafortune, S. (1989). An Intelligent Search Method for Query Optimization by Semijoins. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):226--237.

Yoshizumi, T.; Miura, T. & Ishida, T. (2000). A* with Partial Expansion for Large Branching Factor Problems. Em *Proceedings of the National Conference on Artificial Intelligence and Conference on Innovative Applications of Artificial Intelligence*, pp. 923--929.

Zhou, R. & Hansen, E. A. (2002a). Memory-Bounded A* Graph Search. Em *Proceedings of the International Florida Artificial Intelligence Research Society Conference*, pp. 203--209. AAAI Press.

Zhou, R. & Hansen, E. A. (2002b). Multiple Sequence Alignment Using Anytime A*. Em *Proceedings of the National Conference on Artificial Intelligence and Conference on Innovative Applications of Artificial Intelligence*, pp. 975--977.

Zilberstein, S. (1996). Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, 17(3):73--83.