UTILIZAÇÃO DE AMBIENTES PARALELOS NO PROCESSO DE APRENDIZADO DE ALGORITMOS DE BUSCA DE CAMINHO EM TEMPO REAL

VINICIUS MARQUES TERRA

UTILIZAÇÃO DE AMBIENTES PARALELOS NO PROCESSO DE APRENDIZADO DE ALGORITMOS DE BUSCA DE CAMINHO EM TEMPO REAL

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

Orientador: Luiz Chaimowicz
Co-orientador: Renato Ferreira

Belo Horizonte

Junho de 2010

© 2010, Vinicius Marques Terra.
Todos os direitos reservados.

Terra, Vinicius Marques
G323u Utilização de ambientes paralelos no processo de aprendizado de algoritmos de busca de caminho em tempo real / Vinicius Marques Terra. — Belo Horizonte, 2010
xxii, 53 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Orientador: Luiz Chaimowicz
Co-orientador: Renato Ferreira

1. Computação - Teses. 2. Inteligência Artificial - Teses. 3. Programação paralela - Teses. I. Orientador. II. Co-orientador. III. 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

Utilização de ambientes paralelos no processo de aprendizado de algoritmos de busca de caminho em tempo real.

VINICIUS MARQUES TERRA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

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

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

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

PROF. WAGNER MEIRA JÚNIOR
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 30 de junho de 2010.

Agradecimentos

O autor agradece à família e aos amigos pelo suporte, aos professores e colegas pelas colaborações.

Agradece também ao Conselho Nacional de Desenvolvimento Científico e Tecnológico pelo auxílio financeiro que contribuiu para a realização desta pesquisa.

O autor agradece ainda ao Georgia Institute of Technology, ao Sony-Toshiba-IBM Center of Competence, e à US National Science Fundation, pelo uso dos recursos Cell Broadband Engine, que contribuíram para esta pesquisa.

Resumo

A constante evolução dos jogos eletrônicos possibilita a criação de ambientes cada vez mais realistas e imersivos, sendo necessário que a interação entre jogo e jogador também avance no sentido do realismo, onde a Inteligência Artificial é uma das áreas responsáveis por prover tal interação. Além disso, é notável a evolução dos recursos presentes nos videogames, como a incorporação de processadores multinúcleo nos mesmos, o que muda o paradigma de programação de jogos. Uma das etapas no projeto da Inteligência Artificial para jogos é a movimentação das entidades deste, onde a principal técnica utilizada é a busca de caminho. Embora em diversas situações algoritmos tradicionais como o A* resolvam este problema sem comprometimento de performance, o aumento significativo do tamanho e da complexidade dos mapas, bem como a presença massiva de entidades em um mesmo ambiente faz com que a utilização destes algoritmos afete o desempenho dos jogos. Este problema é resolvido por algoritmos de busca de caminho em tempo real, onde o tempo de busca não aumenta com o tamanho dos ambientes. Tais algoritmos possuem um componente de aprendizado, que evita mínimos locais e aprimora os resultados das futuras buscas, visando chegar ao caminho de custo ótimo, processo que recebe o nome de convergência. Nesse trabalho, apresentamos uma estratégia de paralelização que visa diminuir tempo do processo de convergência, mantendo as restrições de tempo real presentes neste tipo de busca de caminho. A técnica de paralelização criada consiste na utilização de buscas auxiliares sem a restrição de tempo real, onde todas as buscas compartilham o aprendizado adquirido com a busca principal. A avaliação experimental, realizada na arquitetura Cell BE, mostra que, mesmo com o custo adicional necessário para a coordenação das buscas, a redução no tempo total para a convergência é significativa, ocorrendo ganhos tanto nas buscas em ambientes com menor quantidade de mínimos locais quanto em buscas maiores, onde a melhora do desempenho é ainda melhor.

Palavras-chave: busca de caminho em tempo real, programação paralela, agentes inteligentes.

Abstract

The constant evolution of electronic games enables the creation of increasingly realistic and immersive environments. Along with this evolution, it is necessary that the interaction between game and player also go towards realism, where Artificial Intelligence is one of the areas responsible for providing such interaction. Moreover, the evolution of the hardware present in video-games, such as multi-core processors, is remarkable. The presence of parallel environments changes the paradigm of game programming, and such changes also apply to Artificial Intelligence algorithms. One of the steps in the design of artificial intelligence for games is the movement of entities inside a game. The main technique used to control the movement of these entities is the path-finding, which consist in finding a best cost path between two points. Although in most cases traditional algorithms such as A* solve this problem without compromising performance, the increase in the size and complexity of the maps and also the presence of a massive number of entities in the same environment makes the use of these algorithms affect the game performance. This problem is solved by the real-time search algorithms, where the search occurs in a limited area and does not increase with the size of problem. Real-time search algorithms have a learning component, which avoids local minima and improve the results for future searches, in order to reach the minimum cost path. This process is named convergence. In this work, we present a parallelization strategy that aims to reduce the time of convergence, keeping the real time constraints of this type of search. The parallelization technique consist on the use of auxiliary searches without real-time restrictions, where all the searches share the learning acquired with the main search. The empirical evaluation, performed on Cell Broadband Engine Architecture, shows that even with the additional cost required for the coordination of searches, the reduction in the time to convergence is significant, showing gains both in searches occurring in environments with fewer local minima and in larger searches, where performance improvement is even better.

Keywords: real-time path-finding, parallel computing, intelligent agents.

Lista de Figuras

2.1 Uma iteração do A*. 7
2.2 Uma iteração do LRTA*. 11
2.3 Uma iteração do LSS-LRTA* com $d = 2$. 13
2.4 A arquitetura Cell Broadband Engine [Xia & Prasanna, 2008]. 15

3.1 Estrutura básica de uma tarefa. 18
3.2 Ciclo de criação de tarefas: (a) momento de acionamento da etapa de criação, (b) escolha dos estados base para criação das tarefas. 19
3.3 O processo de convergência de uma busca utilizando o LSS-LRTA*. 20
3.4 Uma busca LSS-LRTA* após processo de convergência. 21
3.5 Distribuidor de tarefas. 22
3.6 Processamento das tarefas nos SPEs. 24
3.7 Abordagem para montagem de tarefas com tabela Hash. 25
3.8 Abordagem para montagem de tarefas com matriz esparsa. 27
3.9 Abordagem para montagem de tarefas com autonomia para os SPEs. . . . 28

4.1 Tempo de execução total até a convergência da solução para versão do algoritmo com tabela Hash: (a) buscas em mapas com poucos mínimos locais, (b) buscas em mapas com muitos mínimos locais. A busca à esquerda de cada par corresponde à execução sequencial, e a busca à direita à versão paralela. 34
4.2 Porcentagem de ocupação da tabela Hash em relação ao tamanho do mapa para as versões sequencial e paralela das buscas. 34
4.3 Tempo de execução total até a convergência da solução para versão do algoritmo com matriz esparsa: (a) buscas em mapas com poucos mínimos locais, (b) buscas em mapas com muitos mínimos locais. A busca à esquerda de cada par corresponde à execução sequencial, e a busca à direita à versão paralela. 35

4.4 Tempo de execução total até a convergência da solução para versão do algoritmo com autonomia para os SPEs: (a) buscas em mapas com poucos mínimos locais, (b) buscas em mapas com muitos mínimos locais. A busca à esquerda de cada par corresponde à execução sequencial, e a busca à direita à versão paralela. 36
4.5 *Throughput* do PPE(a) e dos SPEs(b). 38
4.6 Tempo total de execução por número de SPEs utilizados. 39
4.7 Total de tentativas até a convergência: (a) escala aritmética, (b) escala logarítmica. 41

A.1 Mapas utilizados nos experimentos: buscas 1 a 7. 52
A.2 Mapas utilizados nos experimentos: buscas 8 a 14. 53

Lista de Tabelas

4.1 Tempo total (s) para a convergência. Buscas 1 à 7. 36
4.2 Tempo total (s) para a convergência. Buscas 8 à 14. 37
4.3 Número de tentativas para a convergência. Buscas 1 à 7. 40
4.4 Número de tentativas para a convergência. Buscas 8 à 14. 40

Lista de Algoritmos

2.1 O algoritmo A*. 6

2.2 O algoritmo LRTA*. 11

2.3 O algoritmo LSS-LRTA*. 12

3.1 O algoritmo principal do PPE. 23

3.2 O algoritmo do SPE. 24

3.3 O algoritmo do SPE com modificação de autonomia. 29

Lista de Siglas

A* A-Star

AIDA* Asynchronous Parallel IDA*

Cell BE, CBEA Cell Broadband Engine

D* D-Star

DMA Direct memory access

DRAM Dynamic random access memory

GPU Graphics Processing Unit

IDA* Iterative-Deepening A*

LPA* Lifelong Planning A*

LRTA* Learning Real-Time A*

LS Local Storage

LSS-LRTA* Local Search Space LRTA*

NPC Non-player character

PPE Power Processor Element

PR LRTS Path Refinement Learning Real-time Search

PRA* Parallel Retracting A*

SIMD Single Instruction, Multiple Data

SPE Synergistic Processing Element

Sumário

Agradecimentos **vii**

Resumo **ix**

Abstract **xi**

Lista de Figuras **xiii**

Lista de Tabelas **xv**

Lista de Algoritmos **xvii**

Lista de Siglas **xix**

1 Introdução **1**

1.1 Objetivos e contribuições deste trabalho 3
1.2 Organização do Trabalho 4

2 Referencial teórico **5**

2.1 Algoritmos de busca de caminho 5
2.1.1 A* 5
2.1.2 Variações do A* 7
2.2 Paralelização de algoritmos de busca 8
2.3 Algoritmos de busca de caminho em tempo real 9
2.3.1 LRTA* 10
2.3.2 LSS-LRTA* 12
2.3.3 Outros algoritmos 13
2.4 A Arquitetura Cell Broadband Engine 14
2.5 Considerações 16

3 Estratégia de paralelização **17**
3.1 Fluxo de execução no PPE 18
3.1.1 Estrutura básica de uma tarefa 18
3.1.2 Gerência de tarefas 19
3.1.3 Sincronização com SPEs 22
3.1.4 O Algoritmo do PPE 22
3.2 Fluxo de execução no SPE 23
3.3 Montagem das tarefas 25
3.3.1 Compartilhamento da tabela Hash de heurísticas 25
3.3.2 Utilizando matriz esparsa de heurísticas aprendidas 26
3.3.3 Redução do custo de sincronização com os SPEs 27
3.4 Resumo 30

4 Avaliação experimental **31**
4.1 Tempo de Execução 32
4.1.1 Paralelização utilizando compartilhamento da tabela Hash de heurísticas 33
4.1.2 Paralelização utilizando matriz esparsa de heurísticas aprendidas 35
4.1.3 Paralelização com redução do custo de sincronização dos SPEs . 35
4.2 Paralelismo de tarefas 37
4.3 Variação do número de SPEs 38
4.4 Intercalando planejamento e execução 39

5 Conclusões **43**
5.1 Sumário dos resultados 43
5.2 Limitações e trabalhos futuros 44

Referências Bibliográficas **47**

Apêndice A Mapas utilizados na avaliação experimental **51**

Capítulo 1

Introdução

Os frequentes avanços da Computação Gráfica, bem como de simulações de física, possibilitam aos desenvolvedores de jogos eletrônicos a criação ambientes cada vez mais realistas e imersivos. Atualmente, os jogadores aumentam suas exigências na medida em que os jogos evoluem. Dessa forma, entreter um jogador apenas com gráficos e movimentos realistas não é mais suficiente. Assim como estas características presentes nos jogos avançam, é necessário que a interação entre jogo e jogador também evolua. Um dos responsáveis por tal interação entre jogo e jogador é a Inteligência Artificial presente no mesmo. Portanto, aprimorar a inteligência artificial dos jogos é um dos atuais desafios para se criar jogos mais imersivos.

Juntamente com este avanço dos jogos, as tecnologias de *hardware* incorporadas nos videogames estão cada vez mais avançadas. Para serem capazes de gerar ambientes cada vez mais realistas, os videogames apresentam componentes de última geração, como unidades de processamento gráfico (GPUs) poderosas e processadores multinúcleo. Como nestes dispositivos a GPU está sempre ocupada com uma extensa carga de processamento gráfico, os processadores de propósito geral ficam responsáveis por outras cargas de processamento, como por exemplo os algoritmos de inteligência artificial.

Para que os jogos explorem ao máximo o potencial de *hardware* dos computadores e consoles, deve-se levar em conta todas características que estes apresentam. Dessa forma, a incorporação de processadores multinúcleo nos videogames muda o paradigma de programação de jogos. Para aproveitar o máximo de recursos, esta mudança de paradigma deve ser aplicada aos algoritmos de inteligência artificial.

Na inteligência artificial, um agente pode ser considerado uma entidade capaz de perceber seu ambiente por meio de sensores e de agir sobre esse ambiente por intermédio de atuadores [Russell & Norvig, 2003]. Em jogos eletrônicos, o conceito de agente pode

ser aplicado às entidades que não são controladas pelo jogador, no caso os *non-player characters* (NPCs).

Um dos grandes desafios no projeto de uma inteligência artificial realista em jogos eletrônicos é a movimentação de NPCs. Neste contexto, um dos principais problemas é a busca de caminho. Este problema consiste em encontrar o caminho mais eficiente, dado um ponto de partida e um objetivo, evitando obstáculos, inimigos, e minimizando custos (como tempo, distância, combustível, dinheiro, equipamento etc.). Nos jogos, a representação geralmente utilizada para os ambientes é a discretização em grafos. Dessa forma, os principais algoritmos utilizados em jogos para busca de caminho são baseados nos algoritmos de busca em grafos.

Algoritmos de busca em grafo clássicos, como o A* [Hart et al., 1968], são amplamente utilizados em diversos jogos, resolvendo este problema muitas vezes sem comprometimento de performance. Em todos os algoritmos deste tipo, o caminho completo deve ser computado antes que o agente execute a sua primeira ação. Consequentemente, o tempo de planejamento aumenta juntamente com o tamanho do problema, ou seja, não é constante.

Para se manter o compromisso dos jogos com a interatividade, se faz necessário que estes apresentem respostas rápidas para quaisquer requisições feitas. Utiliza-se a expressão tempo real para se referir a tal restrição no tempo de execução.

O avanço dos jogos faz com que os ambientes se tornem cada vez maiores e mais complexos, com uma quantidade cada vez maior de agentes atuando nestes ambientes. Jogos atuais começam a apresentar situações com mapas de gigantescas proporções e com milhares de agentes agindo simultaneamente. Dessa forma, como o custo computacional da busca de caminho aumenta juntamente com o tamanho e complexidade do problema, executar uma busca para cada um dos agentes pode prejudicar severamente o desempenho do jogo, quebrando o compromisso deste com respostas em tempo real.

Além disso, jogos também tendem a apresentar ambientes cada vez mais dinâmicos, que podem sofrer modificações em suas estruturas a qualquer momento, como a destruição de passagens, criação de barreiras, etc. Assim, a movimentação dos agentes deve levar tais modificações em consideração, evitando ações que fujam do realismo.

Métodos de busca de caminho em tempo real, como o LRTA* [Korf, 1990], resolvem os problemas descritos acima, relacionados à respostas em tempo real e à dinamicidade dos ambientes. Em vez de computar a solução completa, estes algoritmos computam apenas algumas ações por vez para que o agente comece a realizá-las, mantendo assim, o compromisso com respostas em tempo real. Dessa forma, a quantidade de planejamento por ação nestes métodos independe do tamanho do problema.

O fato de simplesmente impor um limite constante no tempo de planejamento faz

com que estes algoritmos percam completude, pois não garante que uma solução será encontrada. Tais algoritmos possuem também um mecanismo de aprendizado, que visa evitar mínimos locais para as buscas futuras, e é responsável por reconhecer modificações no ambiente. Segundo Korf [1990], o mecanismo de aprendizado garante que a realização de uma mesma busca sucessivamente resulte no caminho de custo mínimo. Este processo recebe o nome de convergência. Esta situação ocorre frequentemente em jogos, como, por exemplo, numa coleta de recursos em que um agente percorre o mesmo caminho diversas vezes, ou mesmo situações onde diversos agentes percorrem um mesmo caminho ou caminhos com trechos em comum.

O compromisso com respostas em tempo real faz com que o limite de tempo disponível para estes métodos de busca seja pequeno. Quanto menor o tempo disponível, menor será a região do mapa que a busca em tempo real explora por vez. Como consequência disto, maior será a sub-otimalidade das buscas, o que torna a convergência um processo demorado.

1.1 Objetivos e contribuições deste trabalho

O objetivo deste trabalho é propôr uma estratégia de paralelização que mantenha as restrições de tempo dos algoritmos de busca de caminho em tempo real e ao mesmo tempo diminua o tempo do processo de convergência. Além disso, este trabalho visa construir a paralelização proposta em uma arquitetura multinúcleo existente, presente em um console de última geração, no caso, a arquitetura Cell Broadband Engine, do Playstation^{TM} 3. Apesar da estratégia proposta poder ser adaptada para outras arquiteturas, como GPUs, este trabalho não entra em questões a respeito da utilização destas para processamento de propósito geral, visto que tal uso não se aplica a jogos comerciais, onde o processamento gráfico necessário impossibilita a utilização destas para outras cargas de trabalho.

Algumas contribuições deste trabalho são listadas a seguir:

- **Paralelização de algoritmos de busca em tempo real:** é o objetivo principal deste trabalho, realizar a diminuição do tempo total de convergência de uma busca até a solução de custo ótimo, mantendo o compromisso com tempo real.

- **Política de classificação dos estados mais promissores:** este mecanismo, baseado no comportamento de colônias de formigas, faz com que as buscas explorem mais as regiões próximas do caminho ótimo, acelerando o processo de convergência. Tal classificador é utilizado para ordenação das tarefas a serem processadas em paralelo.

- **Metodologia para criação e distribuição de tarefas na arquitetura Cell:** criado após estudar e entender as características e os problemas da arquitetura Cell Broadband Engine, este mecanismo consiste em um distribuidor de tarefas específico para este processador, com sincronização entre o núcleos.

1.2 Organização do Trabalho

O restante deste trabalho foi dividido em 4 capítulos, organizados da seguinte forma:

- **Capítulo 2 [Referencial teórico]:** descreve trabalhos das áreas com as quais este trabalho se relaciona: busca de caminho, paralelização de busca de caminho e busca de caminho em tempo real. Além disso, apresenta uma breve descrição da arquitetura utilizada na paralelização.

- **Capítulo 3 [Estratégia de Paralelização]:** propõe uma estratégia de paralelização que visa acelerar a convergência até a solução de custo ótimo, mantendo as características da busca em tempo real.

- **Capítulo 4 [Avaliação experimental]:** discute os resultados dos experimentos realizados, utilizando as implementações das abordagens criadas para a estratégia de paralelização proposta.

- **Capítulo 5 [Conclusões]:** apresenta um resumo dos principais resultados, limitações e considerações sobre trabalhos futuros.

Capítulo 2

Referencial teórico

Este capítulo apresenta alguns dos principais trabalhos relacionados e/ou utilizados neste trabalho, que são divididos nas seguintes áreas: algoritmos de busca de caminho (*offline*), paralelização de algoritmos de busca e algoritmos de busca de caminho em tempo real. Além destes trabalhos relacionados, a última seção deste capítulo apresenta uma breve descrição do ambiente escolhido para a paralelização, a arquitetura Cell Broadband Engine (Cell BE). O entendimento das características dessa arquitetura é necessário para a compreensão de detalhes relevantes levados em conta na criação da estratégia de paralelização.

2.1 Algoritmos de busca de caminho

Esta seção apresenta alguns algoritmos de busca de caminho. Em todos estes algoritmos, o caminho completo deve ser computado antes que o primeiro movimento seja executado pelo agente. Consequentemente, a quantidade de planejamento por ação não possui uma delimitação constante e aumenta proporcionalmente ao tamanho do problema. Assim, este tipo de busca, também conhecido como *full-search* ou *offline-search*, não ocorre em tempo real.

2.1.1 A*

O A* (Lê-se: A-estrela) [Hart et al., 1968] é um algoritmo amplamente utilizado para busca de caminho. Sua aplicação vai desde encontrar rotas de deslocamento entre localidades à resolução de problemas como quebra-cabeças. É um algoritmo de busca em grafo bastante flexível, que geralmente encontra a solução ótima de forma mais eficiente que outras buscas.

```
1   inicializa listas: O ← ∅ e C ← ∅
2   adiciona s_i em O
3   while O ≠ ∅ do
4     escolhe o estado s de O com o menor f
5     retira s de O e adiciona em C
6     if s = s_g return sucesso
7     encontra os estados s' sucessores de s
8     for each s'
9       calcula a distância d entre s e s'
10      if s' ∉ O adiciona s' em O
11      else if g(s') > g(s) + d then g(s') = g(s) + d
12    end for
13  end while
14  return falha
```

Algoritmo 2.1: O algoritmo A*.

O Algoritmo 2.1 apresenta o A*. Este algoritmo utiliza uma busca por heurística (*best-first*) para encontrar o caminho de melhor custo, dado um estado inicial s_i e um estado objetivo s_g. Entende-se por heurística como sendo uma estimativa de distância de um estado qualquer até o estado objetivo s_g. O A* utiliza dois conjuntos: um conjunto O (lista aberta, do inglês *open list*) para armazenar os estados expandidos, e um conjunto C (lista fechada, do inglês *closed list*) para armazenar os estados visitados. Uma função f é usada para determinar a ordem em que o estados do grafo serão visitados. Esta função é composta pela soma de dois elementos: a distância g do estado inicial s_i até o estado corrente s, e uma heurística h admissível para estimar a distância de s até s_g. Uma heurística é dita ser admissível se o valor estimado nunca for maior que o custo real da distância medida. Este método, se utilizado com uma heurística admissível, garante que a solução de custo ótimo seja encontrada.

A Figura 2.1 mostra uma iteração da execução do A*. Após as inicializações (linhas 1-2 do Algoritmo 2.1), o algoritmo escolhe o estado que minimiza a função f (linha 4, Figura 2.1(a)). Escolhido o estado, o mesmo é expandido (linha 5). São então visitados os estados sucessores deste e, para cada um, é calculada a distância g para atualizar a função f (linhas 7-12, Figura 2.1(b)). É escolhido então o estado da lista de visitados que minimiza a função f, e este é então expandido (Figura 2.1(c)). O ciclo se repete até que o estado objetivo seja expandido (linha 6), ou que não existam mais estados a serem expandidos (linha 14).

A complexidade computacional do A* depende da heurística usada. No pior caso, o número de estados expandidos é exponencial em relação ao tamanho da solução (o

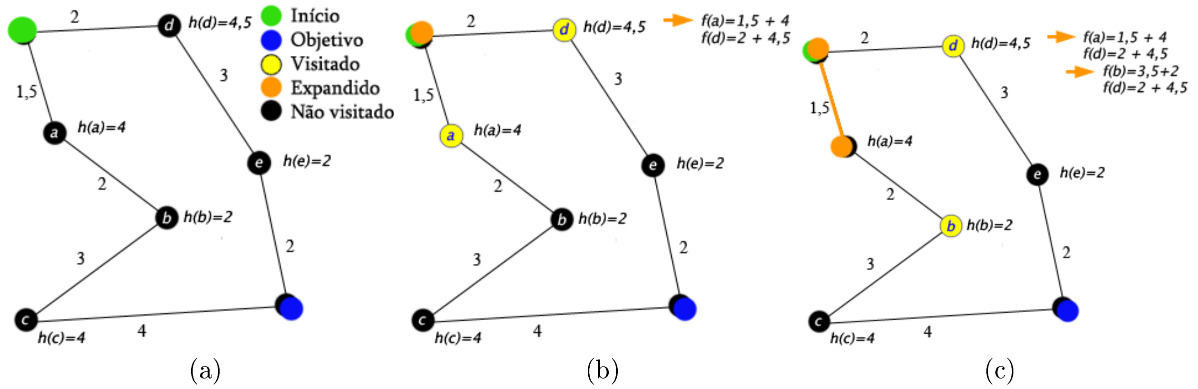


Figura 2.1: Uma iteração do A*.

caminho de custo ótimo). Mas este custo pode ser polinomial [Russell & Norvig, 2003], se o espaço de busca for organizado em forma de árvore, se houver apenas um estado objetivo, e se o erro da heurística não crescer mais que o logaritmo do custo real, ou em outras palavras, se a heurística obedecer à seguinte condição:

$$|h(x) - h^*(x)| = O(log\, h^*(x)),$$

Onde x é um estado qualquer, $h(x)$ a heurística para este estado, e $h^*(x)$ uma heurística perfeita, equivalente ao custo real.

2.1.2 Variações do A*

Nesta seção apresentamos algumas variações do A* que apresentam características que contribuiram para formular o conceito base deste trabalho. Para uma análise adicional sobre desempenho geral e melhorias em algoritmos de busca, consulte [Rios & Chaimowicz, 2010].

O *Iterative-Deepening A** (IDA*) [Korf, 1985] é uma variação do algoritmo de busca A*. Este método consiste em realizar uma série de buscas em profundidade independentes, cada uma com o limite de custo aumentado por uma quantidade mínima. No começo, este limite de custo é associado ao valor heurístico do estado inicial s_i, e então, a cada iteração, este limite é aumentado para o valor mínimo que excedeu o limite anterior.

Além do IDA*, outros métodos de busca podem ser citados, como os que geram o caminho inicial sob a suposição de que as áreas desconhecidas do mapa não possuem estados ocupados (pressuposto de espaço livre [Koenig et al., 2003]). Nestes métodos, o agente segue o caminho planejado até que o estado objetivo ou um estado ocupado (por

exemplo, uma parede) seja atingido. Neste último caso, o A* é chamado novamente para gerar um novo caminho, da posição corrente até o objetivo.

Visando aumentar a eficiência, diversos métodos para reutilizar a informação nos episódios de replanejamento foram sugeridos. Dentre os algoritmos com esta característica, podemos citar o D* [Stentz, 1995] e o *Lifelong Planning A** (LPA*) [Koenig & Likhachev, 2002]. Tais algoritmos reutilizam algumas informações da busca anterior para acelerar os episódios de replanejamento seguintes.

Outros métodos de busca de caminho utilizam ponderação da função heurística para reduzir o número de estados expandidos nas buscas subsequentes. A solução é geralmente encontrada em um menor tempo, mas a mesma é sub-ótima, podendo esta ser melhorada pela realização de buscas adicionais. Este aprimoramento pode ser feito reutilizando a lista aberta nas buscas adicionais [Hansen & Zhou, 2007] ou executando estas buscas em um túnel induzido pela solução sub-ótima [Furcy, 2006].

2.2 Paralelização de algoritmos de busca

Esta seção discute alguns trabalhos relacionados à paralelização de algoritmos de busca.

Kornfeld [1981] propõe uma paralelização de busca por heurística baseada em duas técnicas de programação: propagação de restrições[Barták, 1999] e teste de hipóteses. O princípio desta paralelização se baseia no fato de que sistemas baseados em propagação de restrições podem não ser capazes de escolher uma solução em casos onde mais de uma solução é possível, ou, até mesmo em casos que uma única solução exista, a propagação de restrições pode não ser capaz de encontrá-la. O método proposto consiste em um sistema composto destas duas técnicas, onde a propagação de restrições é utilizada para podar o espaço de busca, ainda permitindo que o teste de hipóteses continue a busca onde a propagação de restrições não é capaz de continuar. O teste de hipótese é realizado também de maneira paralela, onde se executa mais de um teste simultaneamente.

Reinefeld & Schnecke [1994] se baseiam no fato de que buscas por heurística não trabalham bem em situações onde a heurística possui um alto fator de ramificação e valores iniciais pobres, como por exemplo no problema das 15 peças (*fifteen puzzle*). Eles criam então o AIDA* (*Asynchronous Parallel IDA**), um método de busca massivamente paralelo, baseado no IDA*. O AIDA * baseia-se em um esquema de particionamento de dados, onde as diferentes partes do espaço de busca são processadas de forma assíncrona pela rotina sequencial mais rápida disponível, sendo que estas são executadas em paralelo.

PRA* (*Parallel Retracting A**) [Evett et al., 1995] é um algoritmo designado para executar em uma arquitetura SIMD (*Single Instruction Multiple Data*) com memória limitada. Este método consiste em uma paralelização de uma variação do A*, denominada *Retracting A**. Esta variação apresenta uma técnica para economia de memória que consiste na retração de estados já expandidos pelo algoritmo, escolhendo aqueles estados cuja heurística é menos promissora. A paralelização no PRA* ocorre distribuindo os estados gerados pela busca entre os processadores, diferentemente do A* e do próprio RA*, onde se tem um acompanhamento global e centralizado sobre estes estados. Em seguida, cada processador escolhe o estado local mais promissor, realizando sua expansão. Dessa forma, diversas expansões de estados são realizadas em paralelo.

Powley & Korf [1989] apresentam um método que combina ordenação dos estados que possuem um mesmo valor de $g + h$ com busca paralela em janela para encontrar soluções sub-ótimas aceitáveis. Este método consiste em realizar diversas buscas IDA* simultaneamente, cada uma em uma janela de busca, com limites de custos diferentes, onde a informação de ordenação dos estados é compartilhada pelos diferentes processos.

Além destes algoritmos, podemos citar outros como o algoritmo de Korf et al. [2005], que consiste em utilizar estratégias de divisão e conquista para encontrar um passo intermediário e obter os caminhos da solução e da origem até o passo intermediário e assim recursivamente até reconstruir todos os passos; e o algoritmo de Schulte [2000], que propõe uma paralelização de diversos métodos de busca em memória distribuída, separando explicitamente concorrência, distribuição e busca, resultando num algoritmo que tem como recurso distribuir a busca por diferentes sub-árvores.

2.3 Algoritmos de busca de caminho em tempo real

Esta seção apresenta alguns dos algoritmos de busca de caminho em tempo real existentes. É importante relembrar que a expressão tempo real refere-se à restrição de tempo imposta na execução deste tipo de algoritmo. Algumas vezes estes algoritmos serão referidos neste trabalho simplesmente como busca em tempo real.

Algoritmos deste tipo apresentam duas etapas iniciais: planejamento e aprendizado. O planejamento consiste em escolher os nós a serem expandidos e visitados, definindo um trecho do caminho. A etapa de aprendizado consiste na atualização dos valores heurísticos dos estados, conforme propriedades descritas à seguir.

Os algoritmos de busca em tempo real possuem as seguintes propriedades básicas:

O **espaço de busca local** é onde ocorre a busca, procurando a melhor movimentação à partir do estado atual. Define o conjunto de estados cujos valores heurísticos

são acessados na etapa de planejamento.

O **espaço de aprendizado local** é o conjunto de estados cujos valores heurísticos são atualizados. As heurísticas são atualizadas para prevenir *loops* e evitar mínimos locais.

A **regra de aprendizado** é uma regra utilizada para atualizar os valores heurísticos dos estados no espaço de aprendizado local.

Dessa forma, o aspecto geral de um algoritmo de busca em tempo real é: primeiro realiza-se a etapa de planejamento, de acordo com o espaço de busca local; em seguida, a etapa de aprendizado atualiza os estados no espaço de aprendizado local de acordo com a regra de aprendizado. Uma terceira etapa está relacionada à movimentação do agente, podendo ocorrer em paralelo com as próximas execuções das duas etapas anteriores. Nesta etapa, é utilizada uma **estratégia de controle** para realizar a movimentação do agente.

Um detalhe importante deste tipo de algoritmo é que, devido ao fato de realizarem a busca em um espaço limitado, casos sem solução (ex.: o estado destino é inatingível) não podem ser detectados, impedindo a detecção de falha na execução. Dessa forma, como a convergência para um caminho de custo mínimo nunca é atingida, o algoritmo não chega a uma situação de equilíbrio. Entende-se por situação de equilíbrio como sendo aquela após o término do processo de convergência, onde os trechos que formam o caminho de menor custo são sempre retornados pelo algoritmo. É importante observar que o conceito de laço infinito não se aplica neste caso, já que o algoritmo sempre retorna um trecho de caminho quando requisitado. O que acontece nesta situação é uma repetição de sequências de trechos apresentadas pelo algoritmo após a etapa de aprendizado terminar sem que se chegue no estado objetivo, fazendo com que o NPC transite repetidamente pelos mesmos trechos, caracterizando um comportamento alusivo ao de um laço infinito.

2.3.1 LRTA*

O algoritmo *Learning Real-Time A** (LRTA*) [Korf, 1990] é o primeiro e provavelmente mais conhecido método de busca em tempo real. O espaço de busca local deste algoritmo corresponde aos estados sucessores à profundidade 1 do estado corrente, e o espaço de aprendizado local é o próprio estado corrente.

A ideia chave deste método reside em intercalar planejamento e aprendizado. A Figura 2.2 mostra uma iteração do LRTA*, que é apresentado no Algoritmo 2.2. Cada estado s tem um valor heurístico associado a este. Primeiramente, o estado inicial é expandido. Então, o LRTA* visita os vizinhos imediatos, para poder escolher o estado

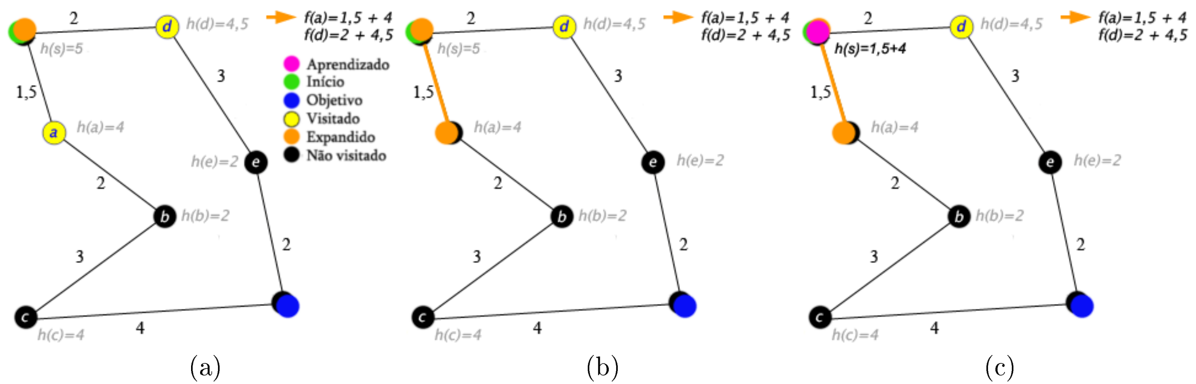


Figura 2.2: Uma iteração do LRTA*.

sucessor (linhas 4-5 do Algoritmo 2.2, Figura 2.2(a)). O algoritmo escolhe o sucessor que minimiza a função f (Figura 2.2(b)), isto é, a soma do custo g de se mover até um sucessor s' e o custo estimado h deste sucessor até o estado objetivo s_g. O LRTA* então atualiza o valor h do estado corrente para aproximar melhor sua estimativa até s_g (regra de aprendizado - linha 6, Figura 2.2(c)). Finalmente, move-se até o sucessor s' escolhido (estratégia de controle - linha 7).

---

1 inicializa heurísticas: $h \leftarrow h_0$
2 reinicia estado corrente: $s \leftarrow s_i$
3 **while** $s \notin s_g$ **do**
4   gera estados sucessores a uma ação do estado s
5   encontra o estado sucessor s' com o menor $f = g + h$
6   atualiza $h(s)$ para $f(s')$ se $f(s')$ for maior
7    $s \leftarrow s'$
8 **end while**

Algoritmo 2.2: O algoritmo LRTA*.

É importante observar que mesmo com os valores heurísticos aumentando, a heurística sempre se mantém admissível, pois o aumento destes valores se deve ao custo g, que não é uma estimativa, e sim um custo real. Qualquer execução do LRTA* que se inicia em s_i e termina em s_g é designada tentativa (do inglês *trial*). Outra propriedade importante deste algoritmo descrita por Korf [1990] é que se o LRTA* é redefinido para o estado inicial sempre que chega ao objetivo, mantendo-se os valores heurísticos que foram atualizados em uma tentativa para a outra, este encontrará um caminho mínimo de s_i até s_g. A esta sequência de tentativas até que se chegue ao caminho mínimo é dado o nome de convergência. Segundo Korf [1990], o LRTA* resolve situações de empate sistematicamente para cada estado de acordo com uma ordem arbitrária

que é selecionada inicialmente. Se o desempate ocorre sistematicamente, isto é, se a mesma escolha prevalece para todas as tentativas, a convergência acontece quando não existirem mais alterações em qualquer valor heurístico durante uma tentativa. Esta propriedade então é usada para detectar a convergência.

2.3.2 LSS-LRTA*

O método de busca por heurística em tempo real *Local Search Space LRTA** (LSS-LRTA*) [Koenig & Sun, 2009] é uma versão do LRTA* que utiliza o A* para determinar o espaço de busca local e que possui um menor tempo de convergência.

Existem duas questões importantes a serem levadas em consideração no algoritmo de Korf [1990]. Uma se relaciona à quantidade de estados que devem estar presentes no espaço de busca local. O LRTA* é frequentemente utilizado considerando-se o espaço de busca local como sendo apenas os estados sucessores do estado corrente. Embora variações desse algoritmo com um espaço de busca local maior foram previamente sugeridas, estas versões normalmente não satisfazem restrições de tempo real, já que elas determinam o tamanho deste espaço de busca local levando em conta outras considerações, tal como o tamanho da depressão (mínimo local) relacionada aos valores heurísticos no entorno do estado corrente, ou mesmo o tamanho da área do mapa já explorada [Koenig & Sun, 2009]. Outra importante questão é saber quais estados devem ter seus valores heurísticos atualizados, ou o tamanho do espaço de aprendizado local. A versão original do LRTA* atualiza apenas o valor heurístico do estado corrente. Dessa forma, o aprendizado não ocorre rapidamente [Russell & Wefald, 1991].

O LSS-LRTA* aborda as duas questões descritas acima: utiliza o A* para determinar o espaço de busca local obedecendo restrições de tempo real, e atualiza os valores heurísticos de todos os estados presentes neste espaço, ou seja, o espaço de aprendizado local é o próprio espaço de busca local. Dessa forma, o aprendizado ocorre mais rapidamente, e com isso o tempo total para convergência diminui.

O Algoritmo 2.3 apresenta os passos do LSS-LRTA*, e a Figura 2.3 mostra uma iteração do algoritmo. Após as inicializações (linhas 1-2, Figura 2.3(a)), o algoritmo utiliza o A* (linha 4), que realiza uma busca do estado corrente s para s_g até que d estados sejam expandidos (profundidade da busca, do inglês *lookahead*) ou que s_g esteja para ser expandido. Os estados expandidos formam o espaço de busca local. O algoritmo então escolhe o estado s' com o menor custo total f (Figura 2.3(b)). Em seguida, o LSS-LRTA* realiza uma busca Dijkstra [1959] para atualizar os valores heurísticos de todos os estados presentes no espaço de busca local (linha 6, Figura 2.3(c)). O algoritmo de Dijkstra substitui os valores heurísticos de todos os estados

1 inicializa heurísticas: $h \leftarrow h_0$
2 reinicia estado corrente: $s \leftarrow s_i$
3 **while** $s \notin s_g$ **do**
4 realiza uma busca A* de s para s_g até que d estados sejam expandidos, gerando uma fronteira
5 escolhe o estado de fronteira s' com o menor $f = g(s, s') + h(s', s_g)$
6 utiliza o algoritmo de Dijkstra para substituir o valor h de cada estado expandido
7 $s \leftarrow s'$
8 **end while**

Algoritmo 2.3: O algoritmo LSS-LRTA*.

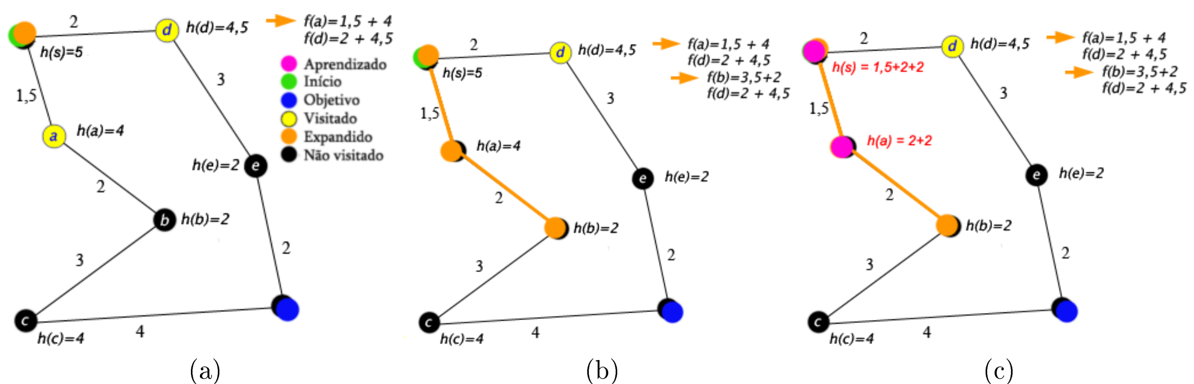


Figura 2.3: Uma iteração do LSS-LRTA* com $d = 2$.

pela soma da distância g até o estado fronteira escolhido s' e o valor heurístico de s'. Finalmente, o LSS-LRTA* move o agente ao longo do caminho encontrado pelo A* até s' (linha 7). O processo se repete até que se chegue no estado s_g.

Tanto o A* quanto o Dijkstra expandem os estados apenas uma vez e são portanto eficientes [Koenig & Sun, 2009]. A ideia de utilizar Dijkstra para atualizar os valores heurísticos é para que o espaços de busca e aprendizado mantenham-se localmente coerentes, e então ocorra o máximo de propagação possível dos valores heurísticos dos estados de fronteira para os estados dentro do espaço de aprendizado local. O princípio da coerência local baseia-se na ideia de que, se para cada estado em um espaço é mantido um valor heurístico que representa a melhor estimativa do custo até o objetivo, então as decisões localmente ótimas podem ser baseadas nos valores dos estados vizinhos [Pemberton & Korf, 1992]. A utilização do A* se deve ao fato de sua performance superar a de uma busca em largura simples, como comprovado experimentalmente por Koenig & Sun [2009].

2.3.3 Outros algoritmos

Além dos métodos de busca em tempo real apresentados, podemos citar o LRTA*(k) [Hernández & Meseguer, 2005], que é uma variação do LRTA* apresentando uma metodologia de propagação para a atualização dos valores heurísticos denominada propagação limitada. Neste algoritmo, o espaço de aprendizado local consiste no caminho já percorrido pela busca, delimitado pelo parâmetro k ou pelo término do método de propagação.

Outros algoritmos de busca em tempo real importantes são os métodos que utilizam o conceito de abstração de grafos. PR LRTS (*Path Refinement Learning Real-time Search*) [Bulitko et al., 2007] utiliza um mecanismo que pré computa abstrações baseadas em cliques de grafos. Estas abstrações resultam em diferentes níveis de busca, onde os níveis superiores definem o conjunto de estados (*clusters*) que farão parte das buscas nos níveis inferiores. Um processo denominado refinamento cria então um corredor de estados para se realizar a busca. D LRTA* [Bulitko et al., 2008] utiliza este mesmo mecanismo de níveis de abstração para tentar reduzir o tempo do processo de aprendizado, acrescentando uma técnica para definir automaticamente a profundidade da busca (*lookahead*) e *subgoals* dentro de cada *cluster*.

O kNN LRTA* [Bulitko & Bjornsson, 2009] é um algoritmo que, em vez de utilizar abstrações de grafos, se baseia no método de classificação kNN (*k nearest neighbors*) para construir um banco de dados de *subgoals*. Durante a busca, este método verifica no banco de dados os casos mais similares ao problema em questão e sugere um *subgoal* para a busca LRTA*.

2.4 A Arquitetura Cell Broadband Engine

Esta seção apresenta uma descrição da arquitetura utilizada neste trabalho para a execução dos algoritmos utilizando a estratégia de paralelização proposta. A arquitetura Cell Broadband Engine (Cell BE, CBEA ou simplesmente Cell) consiste em um processador com características únicas voltadas para o processamento distribuído e aplicações ricas em mídia. Inclusive, a primeira das grandes aplicações comerciais do Cell foi no Playstation® 3, um dos principais consoles da última geração. Ele foi criado para cobrir a lacuna presente entre processadores convencionais e processadores específicos (como GPUs).

A Figura 2.4 mostra a arquitetura Cell BE. Esta arquitetura define um multiprocessador de chip único, com nove processadores operando em uma memória compartilhada e coerente. A característica mais notável do Cell BE é que, embora todos os

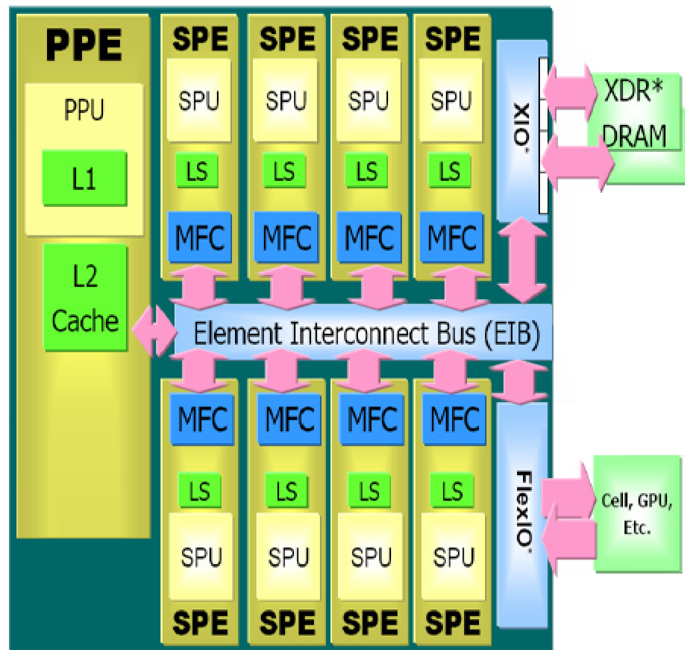


Figura 2.4: A arquitetura Cell Broadband Engine [Xia & Prasanna, 2008].

processadores compartilhem o espaço de endereçamento efetivo que inclui a memória principal, sua função é especializada em dois tipos: O Power Processor Element (PPE) e o Synergistic Processing Element (SPE). O Cell BE possui um PPE e oito SPEs.

O primeiro tipo de processador, o PPE, é um núcleo de processamento *multithreaded* baseado na tecnologia Power Architecture^{TM} de 64-bits, totalmente compatível com as especificações desta tecnologia. O PPE pode rodar sistemas operacionais e aplicativos tanto de 32-bits quanto de 64-bits.

O segundo tipo de processador, o SPE, é um processador RISC com organização SIMD 128-bits de alta performance, otimizado para rodar aplicações de computação intensiva. Os SPEs são processadores independentes, cada um executando seu próprio programa. Cada SPE consiste em uma unidade de processamento (Synergistic Processor Unit - SPU) e um controlador de fluxo de memória. A geração atual dos processadores Cell contém 256KB de memória interna para instruções e dados, chamada de Local Storage (LS). A LS não opera como uma cache convencional, já que ela não é transparente para o programa e não contém estruturas de *hardware* para predizer qual dado carregar. Cada SPE tem acesso completo à memória compartilhada (DRAM), realizando operações de DMA (*direct memory access*) através de um barramento chamado Element Interconnect Bus.

Além das operações de DMA, existe um outro mecanismo de comunicação similar

a caixas de correio, chamado de Mailboxes. Este mecanismo é utilizado para controlar a comunicação entre um SPE e o PPE ou outros dispositivos. As Mailboxes guardam mensagens de 32-bits. Cada SPE tem duas Mailboxes para enviar mensagens e uma para receber mensagens.

Segundo Gschwind et al. [2006], existe, de certa forma, uma dependência mútua entre o PPE e os SPEs. A combinação destes dois elementos trabalhando em harmonia produz um efeito maior do que cada um trabalhando sozinho. O SPE depende do PPE para rodar o sistema operacional e, em muitos casos, a *thread* de nível superior de um aplicativo. Já o PPE depende dos SPEs para fornecer a maior parte do desempenho da aplicação.

Devido às características deste processador, como ausência de cache e o tamanho limitado da LS dos SPEs, diversos cuidados devem ser levados em consideração ao se implementar uma aplicação nesta arquitetura. Deve se manter um equilíbrio entre código-fonte e estruturas de dados, bem como um maior controle do tamanho da pilha de execução para evitar problemas de estouro. Além disso, a ausência de cache demanda uma atenção para evitar que os SPEs fiquem ociosos enquanto aguardam pelo término das operações de DMA.

2.5 Considerações

Conforme visto neste capítulo, a arquitetura Cell BE foi criada para ser uma ponte entre os processadores convencionais e específicos de sua geração. Além disto, esta arquitetura se encontra presente em um console atual de última geração. Por estas razões, foi escolhida a arquitetura Cell Broadband Engine para implementação e execução da estratégia de paralelização proposta.

Além disso, o fato do LSS-LRTA* permitir um controle do tamanho dos espaços de busca e aprendizado local facilita a implementação da paralelização proposta na arquitetura Cell BE, levando em consideração as características desta arquitetura, descritas na Seção 2.4 deste capítulo. Dessa forma, para todas as implementações e experimentos realizados neste trabalho, foi escolhido o algoritmo de busca em tempo real LSS-LRTA*[Koenig & Sun, 2009]. É importante ressaltar que a estratégia de paralelização proposta pode ser utilizada em conjunto com qualquer outro algoritmo de busca de caminho em tempo real, visto que todos possuem as mesmas propriedades básicas, descritas na Seção 2.3 deste capítulo.

Capítulo 3

Estratégia de paralelização

A característica principal dos algoritmos de busca em tempo real é o compromisso temporal: os algoritmos devem fornecer uma resposta respeitando um limite de ações pré-estabelecido, diretamente relacionado a um intervalo de tempo. Considerando esta restrição imposta, o princípio básico da estratégia de paralelização proposta é manter tal restrição em uma busca principal, e efetuar trechos complementares a esta busca em paralelo, utilizando núcleos de processamento auxiliares. Estas buscas auxiliares não necessitam atender à restrição temporal, e portanto podem possuir um maior limite de ações a serem executadas. Todas estas buscas, inclusive a busca principal, compartilham o mesmo aprendizado adquirido (aprimoração dos valores heurísticos). Assim, o ganho que se espera com a paralelização é um número menor de tentativas para que se chegue ao custo ótimo, ou seja, a diminuição do tempo de convergência.

A representação de grafos utilizada neste trabalho é na forma de grade bidimensional. Nesta representação, cada vértice da grade corresponde a um estado no espaço de busca, sendo que cada estado pode ser transitável ou ocupado por uma parede. Neste último caso, o agente é impossibilitado de se mover para o estado. Se o agente puder se mover entre quaisquer dois estados vizinhos, então existe uma aresta entre esses vértices. As arestas podem apresentar custos diferenciados, representando, por exemplo, diferentes tipos de chão, como terra, asfalto, água, etc. Para fins de simplificação, neste trabalho foram considerados apenas dois custos: para movimentos cardinais, o custo das arestas é igual a 1, e para movimentos diagonais, o custo é igual a $\sqrt{2}$. A heurística inicial utilizada é a heurística admissível chamada distância *octile*, definida como o caminho mínimo entre duas localidades, considerando que todos os estados são transitáveis. Esta função heurística é uma variação da distância de *Manhatan*, incluindo o caso de movimentos diagonais.

No restante deste capítulo detalhamos a estratégia de paralelização explicando

as variações desta que foram criadas. A Seção 3.1 descreve o fluxo de execução do algoritmo do núcleo principal. A Seção 3.2 descreve o fluxo de execução do algoritmo dos núcleos auxiliares. A Seção 3.3, última deste capítulo, aborda as três variações criadas para montagem das tarefas a serem processadas.

3.1 Fluxo de execução no PPE

Esta seção apresenta os detalhes do algoritmo principal executado no PPE. De uma forma geral, o algoritmo consiste em, primeiramente, realizar um trecho da busca principal, para então serem criadas tarefas correspondentes aos trechos de buscas auxiliares, com o objetivo de acelerar o processo de aprendizado. Em seguida, estas tarefas são distribuídas aos SPEs em uma etapa de sincronização.

A Seção 3.1.1 apresenta a estrutura de uma tarefa, a Seção 3.1.2 detalha os componentes do algoritmo relacionados à criação e ordenação das tarefas, a Seção 3.1.3 descreve a sincronização com os SPEs para a distribuição das tarefas, e a Seção 3.1.4 apresenta uma visão mais detalhada do algoritmo.

3.1.1 Estrutura básica de uma tarefa

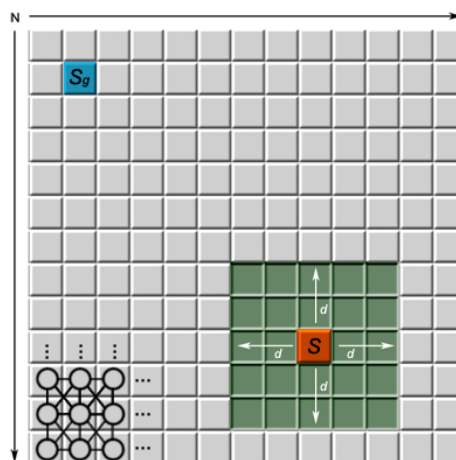


Figura 3.1: Estrutura básica de uma tarefa.

A Figura 3.1 apresenta a estrutura básica de uma tarefa. A grande área em cinza claro representa um mapa de dimensões $N \times N$. O estado s representa o estado corrente, que é o parâmetro base na criação de uma tarefa. Já o estado s_g representa o estado objetivo (o *goal*). A área mais escura ao redor do estado corrente (proporcional à distância d, o *lookahead* da busca) delimita o sub-grafo (ou *tile*) no qual o trecho de

busca será realizado. Uma tarefa é composta por duas estruturas de dados associadas a este *tile*, sendo uma para representar um subconjunto de estados do mapa, e outra para representar os valores heurísticos destes estados. A tarefa ainda contém a informação extra do endereço de memória para escrita dos valores heurísticos atualizados pelas buscas dos SPEs.

3.1.2 Gerência de tarefas

A gerência de tarefas no PPE é a parte do algoritmo responsável pela criação e ordenação das tarefas a serem processadas nos SPEs. A seguir, descrevemos detalhes destas etapas.

3.1.2.1 Criação de tarefas

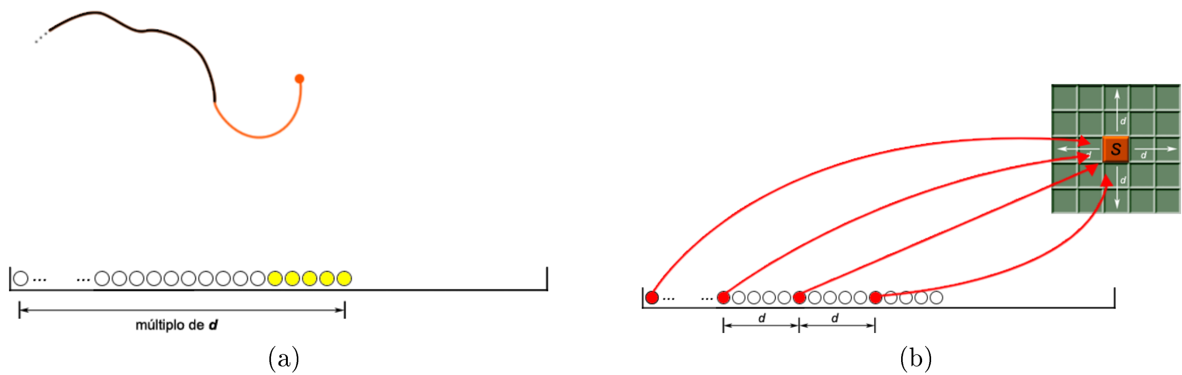


Figura 3.2: Ciclo de criação de tarefas: (a) momento de acionamento da etapa de criação, (b) escolha dos estados base para criação das tarefas.

A etapa de criação de tarefas se dá da seguinte forma: considera-se que o *lookahead* d das buscas auxiliares (executadas no SPE) seja um múltiplo maior ou igual ao *lookahead* da busca principal (executada no PPE). Dessa forma, sempre que a busca principal acumula um trecho total de caminho já percorrido que seja múltiplo de d (Figura 3.2(a)), são criadas tarefas considerando os estados presentes neste caminho percorrido, separados pela distância d, à partir do estado inicial s_i até o último estado múltiplo de d no caminho percorrido (Figura 3.2(b)). Para entender melhor, imagine uma situação em que o *lookahead* da busca principal é igual a 5 e o das buscas auxiliares seja $d = 15$. Suponha que a busca principal ainda não atingiu s_g, e que a mesma já percorreu um caminho com um tamanho $n \times 15$. Serão criadas então tarefas para os estados de índices 0 (s_i), 15, 30, 45, 60, ..., $(n-1) \times 15$. O parâmetro s é o único armazenado nesta etapa. A montagem das tarefas é detalhada na Seção 3.3.

3.1.2.2 Ordenação das tarefas

O processo de convergência de busca LSS-LRTA* não ocorre de maneira regular. Em outras palavras, o custo de uma tentativa futura não será necessariamente menor que o custo das tentativas já executadas. A Figura 3.3 demonstra a irregularidade característica do processo de convergência de uma busca LSS-LRTA*. A visualização dos custos foi limitada ao valor máximo de 2000 para facilitar a visualização dos demais valores.

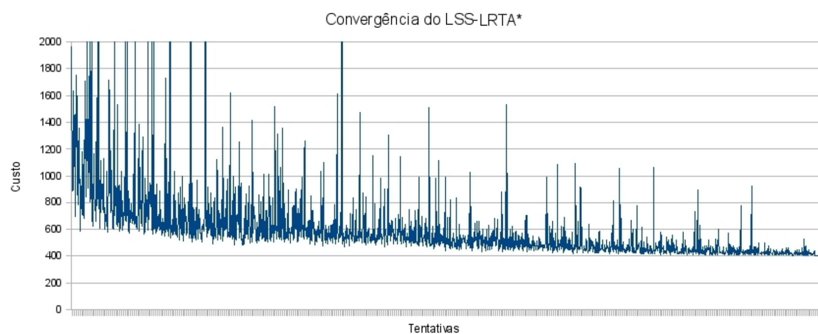
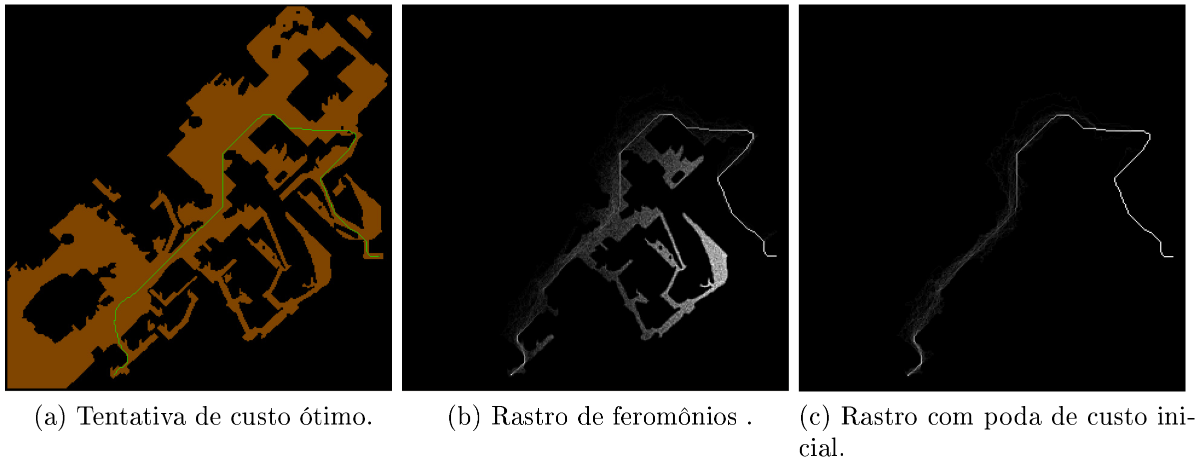


Figura 3.3: O processo de convergência de uma busca utilizando o LSS-LRTA*.

Levando em consideração essa irregularidade, este trabalho propôs uma maneira de premiar a regularidade das tentativas. O princípio proposto é inspirado no comportamento de colônias de formigas, onde ocorre uma cooperação através da estigmergia [Holldobler & Wilson, 1990] - forma de comunicação utilizada por formigas, onde não há uma comunicação direta, e sim através de marcas deixadas no ambiente na forma de feromônios (hormônios presentes nas formigas). Este comportamento está presente em diversos Sistemas de Rede de Sensores Sem Fio (RSSF), que seguem os princípios da swarm intelligence [Kennedy et al., 2001], onde a inteligência do sistema não se localiza nos agentes individuais, mas no produto das suas interações, exibindo um comportamento complexo. Adequando ao contexto do problema, a ideia é prover uma comunicação implícita entre uma tentativa e outra, seguindo estas propriedades. Basicamente, quando uma tentativa se completa, o custo desta é comparado a um custo previamente armazenado (o melhor custo até o momento). Se o custo da tentativa for menor que o custo armazenado, o primeiro passa a ser então o novo melhor custo, e todos os estados presentes no caminho percorrido por esta tentativa são acrescidos de feromônio. Este princípio auxilia a restringir o tamanho do espaço onde a busca é realizada, isto é, restringe os estados do mapa que serão explorados.

A Figura 3.4 ilustra os rastros deixados pelas tentativas no processo de convergência de uma busca com o LSS-LRTA*. Pode-se notar que a região no entorno da



(a) Tentativa de custo ótimo. (b) Rastro de feromônios . (c) Rastro com poda de custo inicial.

Figura 3.4: Uma busca LSS-LRTA* após processo de convergência.

tentativa com custo ótimo possui uma quantidade maior de feromônios, com destaque para trechos do próprio caminho ótimo. Ainda observando a Figura 3.4(b), nota-se que algumas regiões de mínimos locais possuem uma quantidade considerável de feromônios. Este fenômeno ocorre devido à imprecisão do valor inicial para o melhor custo previamente armazenado, atribuído inicialmente como infinito. Para contornar este problema, uma última otimização realizada foi atribuir um valor inicial para este custo. Esta alteração faz com que tentativas iniciais de custos muito elevados sejam descartadas, realizando uma espécie de poda. A Figura 3.4(c) mostra os rastros da mesma busca quando se utiliza um valor de custo inicial igual a três vezes o valor heurístico entre s_i e s_g.

Esta técnica motivou uma pequena alteração na criação de tarefas, descrita na Seção 3.1.2.1: sempre que o custo do caminho já percorrido ultrapassa o melhor custo armazenado, a criação de tarefas é interrompida para tal tentativa. Nesse caso, como já se sabe que o custo da tentativa ultrapassou o melhor custo armazenado, este caminho não é tão interessante para exploração pois, por não ser o caminho ótimo, seu maior custo provavelmente se deve a trechos percorridos em áreas de mínimos locais, o que acarretaria uma concentração de tarefas criadas nestas áreas.

Finalmente, a criação desta técnica também possibilitou formular um método de ordenação para as tarefas distribuídas aos SPEs. Para tal, um heap de prioridades foi criado para o armazenamento das mesmas, ordenado pela quantidade de feromônios do estado s que compõe a tarefa. Um segundo parâmetro de ordenação foi definido para a situação em que duas tarefas possuíssem a mesma quantidade de feromônios. Nesse caso, a prioridade é da tarefa mais antiga. Esta técnica apresentou performance superior à utilização de uma simples fila FIFO para a ordenação das tarefas.

3.1.3 Sincronização com SPEs

A etapa de sincronização com SPEs consiste na comunicação e distribuição de novas tarefas. Toda a comunicação entre PPE e SPEs é feita utilizando Mailboxes (ver Seção 2.4). Foi criado um distribuidor de tarefas, apresentado na Figura 3.5. Este distribuidor foi inspirado no emissor de tarefas de Xia & Prasanna [2008], criado para resolver problemas de inferência exata utilizando a arquitetura Cell BE.

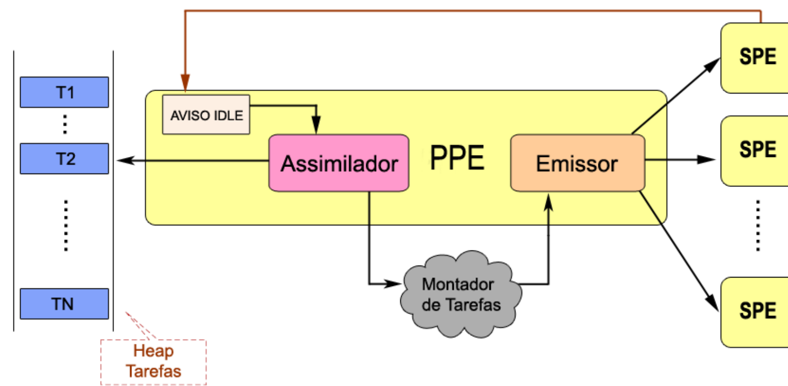


Figura 3.5: Distribuidor de tarefas.

O PPE verifica a caixa de saída de cada SPE para saber se este está ocioso, e caso esteja, envia um elemento do heap para a etapa de montagem de tarefas (descrita na Seção 3.3). Após a etapa de montagem ser concluída, a tarefa é emitida para o SPE.

3.1.4 O Algoritmo do PPE

O fluxo básico de execução no PPE é apresentado no Algoritmo 3.1. O comportamento do algoritmo é semelhante ao ciclo do LSS-LRTA*, acrescido das etapas de gerência de tarefas e sincronização com SPEs. Após as inicializações (linhas 1-2), o algoritmo entra no ciclo mais interno (linha 4), onde realiza um trecho da busca principal (linha 5) delimitado pelo *lookahead* desta busca. Em seguida, a etapa de criação de novas tarefas descrita em 3.1.2.1 é acionada (linha 6) seguida da etapa de sincronização com os SPEs descrita em 3.1.3 (linha 7). O ciclo interno termina ao chegar em s_g, e então o mecanismo de atribuição de feromônios é acionado. O ciclo externo se repete até o fim do processo de convergência.

```
1  inicializa heurísticas: h ← h_0
2  reinicia estado corrente: s ← s_i
3  while existirem alterações dos valores em h
4     while s ∉ s_g do
5        realiza uma iteração do LSS-LRTA*(s, s_g, d)
6        cria novas tarefas
7        sincronização com SPEs
8     end while
9     atualiza feromônios dos estados presentes na última tentativa
10 end while
```

Algoritmo 3.1: O algoritmo principal do PPE.

3.2 Fluxo de execução no SPE

Nesta seção, apresentamos o algoritmo executado pelos SPEs. Devido à limitação da memória destes processadores, foram necessários diversos cuidados na construção deste algoritmo. Uma característica dos SPEs é que a memória local destes deve conter o código a ser executado, bem como as variáveis e métodos estáticos, e também a pilha de execução. As primeiras modificações foram realizadas no algoritmo LSS-LRTA*.

No intuito de reduzir o tamanho do código, eliminando bibliotecas extras, foram retiradas as estruturas de vetores dinâmicos e as tabelas Hash, substituídas por vetores e matrizes estáticas, de tamanho fixo relacionado ao tamanho do *tile* da tarefa. O objetivo desta modificação era para se obter um melhor controle do espaço ocupado pelas estruturas de dados. Estas estruturas modificadas estavam presentes nos conjuntos característicos do algoritmo de busca: a lista aberta (composta por um heap de estados e de uma tabela Hash para indexar as posições dos estados no heap) e a lista fechada (composta por um Hash de estados). Uma implementação preliminar da lista aberta substituía o heap de estados presente nesta por um vetor ordenado. Esta modificação fez com que o resultado das buscas fossem diferentes da mesma busca sendo realizada no PPE, com os mesmos parâmetros de entrada. O motivo desta diferença é a maneira como são realizados os desempates na escolha do próximo estado a ser expandido. Uma modificação foi realizada para que o algoritmo retornasse os mesmos resultados da busca principal. A classe que implementa a lista aberta no algoritmo LSS-LRTA* do PPE, utilizando conceitos de orientação a objetos e também sobrecarga de operadores, foi adaptada para o SPE. Mas esta nova abordagem gerou outro problema: o estouro da pilha de execução. A solução encontrada então foi retirar tal classe, e transformar a lista aberta em um vetor estático, atrelando a este vetor um heap de índices de estados, e não mais de estados. Esta solução não mais apresentou

problemas de estouro da pilha de execução.

1 sincronização com PPE para receber tarefa no *buffer* b_0
2 sincronização com PPE para receber tarefa no *buffer* b_1
3 **while** $tarefas \neq fim$ **do**
4 aguarda preenchimento de b_0
5 realiza uma iteração do **LSS-LRTA***(s_0, s_g, d)
6 escreve novos valores h no endereço de saída e_0
7 sincronização com PPE para receber tarefa no *buffer* b_0
8 aguarda preenchimento de b_1
9 realiza uma iteração do **LSS-LRTA***(s_1, s_g, d)
10 escreve novos valores h no endereço de saída e_1
11 sincronização com PPE para receber tarefa no *buffer* b_1
12 **end while**

Algoritmo 3.2: O algoritmo do SPE.

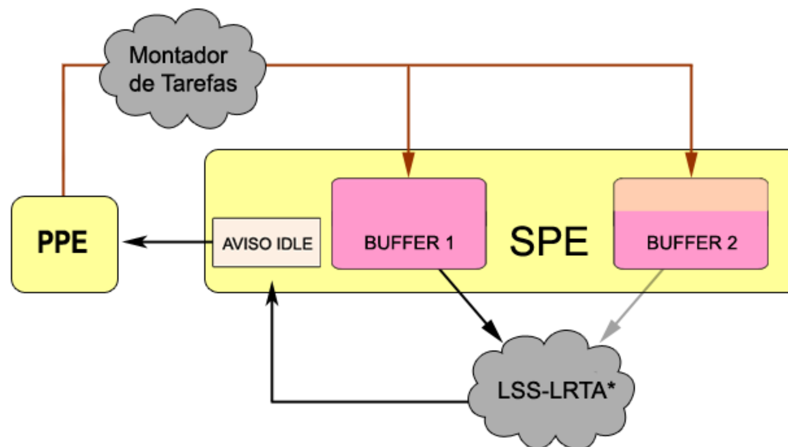


Figura 3.6: Processamento das tarefas nos SPEs.

O Algoritmo 3.2 apresenta o fluxo de execução no SPE, também mostrado no diagrama apresentado pela Figura 3.6. Como os SPEs não possuem cache, são necessárias operações DMA explícitas para transferência de dados. Por esta característica, foram criados dois *buffers* de tarefas, com o objetivo de esconder a latência das transferências. Após requisitar tarefas para os dois *buffers* (linhas 1-2), o algoritmo entra em seu ciclo principal de execução. Assim que o primeiro *buffer* é preenchido com a tarefa (linha 4), o trecho da busca é executado (linha 5) e os valores heurísticos atualizados pela busca são enviados de volta (linha 6). O algoritmo pede uma nova tarefa para o primeiro *buffer* (linha 7), e então passa a aguardar o preenchimento da tarefa enviada para o segundo *buffer* (linha 8). Em seguida, o trecho de busca relacionado à tarefa presente

no segundo *buffer* é executado (linha 9), e os valores heurísticos atualizados por esta busca são enviados de volta (linha 10). O algoritmo então pede uma nova tarefa para o segundo *buffer*. O ciclo se repete até que a tarefa contendo o código que indica o fim da execução seja recebido em um dos *buffers*. Maiores detalhes sobre o envio dos valores heurísticos atualizados pelas buscas são esclarecidos na Seção 3.3.

3.3 Montagem das tarefas

A montagem das tarefas consiste em preencher os componentes que formam a estrutura da mesma, descritos na Seção 3.1.1. Este trabalho apresenta três metodologias diferentes para montagem das tarefas. Uma primeira abordagem utilizando tabela Hash para armazenar as heurísticas aprendidas, uma outra abordagem substituindo a tabela Hash por uma matriz esparsa, e uma terceira abordagem onde a montagem das tarefas é realizada tanto pelo PPE quanto pelos SPEs.

3.3.1 Compartilhamento da tabela Hash de heurísticas

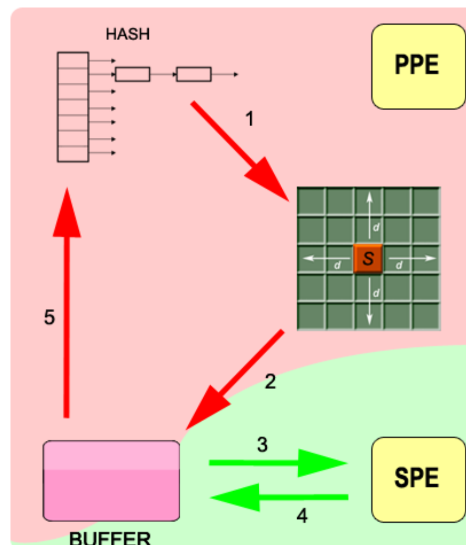


Figura 3.7: Abordagem para montagem de tarefas com tabela Hash.

A abordagem descrita nesta seção consiste em manter a propriedade original do algoritmo LSS-LRTA* em que a tabela Hash de heurísticas é compartilhada entre uma tentativa e outra. Dessa maneira, Hash passa a ser também compartilhado pelas buscas dos SPEs. A montagem das tarefas ocorre da seguinte maneira: O PPE reserva espaços fixos na memória para representar os *buffers* de entrada e saída relacionados a cada SPE. Assim que o montador de tarefas recebe o parâmetro s e a informação de qual

buffer utilizar, este passa a preencher o *buffer* designado com o *tile* que representa a região do mapa sob a qual vai ser realizada a busca (passos 1 e 2 da Figura 3.7). Os valores heurísticos dos estados nesta região também são lidos da tabela Hash e escritos no *tile* correspondente às heurísticas. O parâmetro da tarefa que armazena o endereço de saída é preenchido com o endereço do *buffer* relacionado ao SPE que pede a tarefa. O SPE então copia os dados da tarefa do *buffer* de entrada (passo 3). Quando a tarefa termina de ser processada, os valores heurísticos atualizados são escritos no *buffer* indicado pelo endereço de saída (passo 4). Quando uma nova tarefa para este mesmo SPE é requisitada, os valores do *buffer* de saída são lidos e inseridos no Hash pelo PPE (passo 5) antes que a montagem da nova tarefa inicie.

3.3.2 Utilizando matriz esparsa de heurísticas aprendidas

A abordagem apresentada na seção anterior requer certo esforço computacional para preenchimento dos buffers. Conforme será mostrado mo Capítulo 4, esta carga extra de processamento gera um custo adicional considerável. Foi proposta então uma nova abordagem visando eliminar este custo adicional. O primeiro passo foi eliminar o preenchimento dos *buffers*. Para que isso fosse possível, foi necessário trocar a tabela Hash de valores heurísticos por uma matriz. Se na abordagem anterior a tabela Hash possuísse uma porcentagem de ocupação pequena, esta matriz seria considerada esparsa. Neste caso, o que ocorre então é um *trade-off* entre memória e tempo de processamento, pois o tempo gasto pelo PPE com preenchimento dos *buffers* não existe mais e, em contrapartida, a matriz ocupa um espaço em memória maior que a tabela Hash.

Com essas modificações, tanto a estrutura quanto montagem das tarefas passa a ocorrer de maneira diferente. As tarefas não mais possuem os *tiles* que representam a região do mapa e os valores heurísticos dos estados. Agora elas possuem a informação de onde começa o *tile* e do tamanho da linha da matriz que representa o mapa. Essa informação do tamanho da linha é necessária para que o SPE requisite as operações de transferência DMA, onde cada requisição corresponde a um bloco contínuo na memória. Como não existem mais *buffers* fixos no PPE, os *tiles* passam a ser uma região não contínua qualquer da matriz. A partir do parâmetro recebido s, o montador calcula o início da região da busca com base no *lookahead* d (passo 1 da Figura 3.8). Lembrando que esse parâmetro calculado na verdade é decomposto em dois: um para o início do *tile* na matriz que representa o mapa, e outro para o início do *tile* na matriz com as heurísticas dos estados. Em seguida, o SPE preenche o *tile* da tarefa no *buffer* (passo 2), linha a linha, onde cada linha do *tile* se distancia das outras pelo valor de

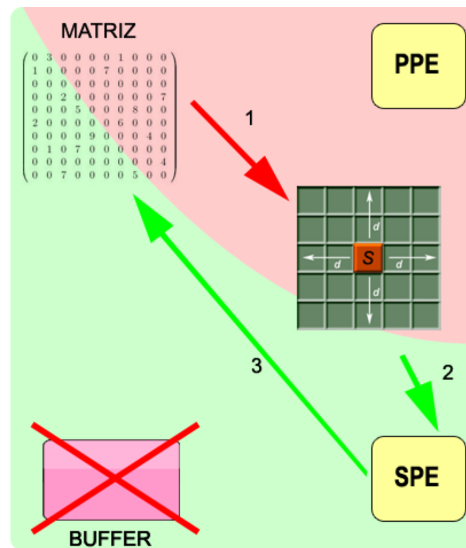


Figura 3.8: Abordagem para montagem de tarefas com matriz esparsa.

uma linha do mapa. Finalmente, os valores heurísticos atualizados após o trecho de busca ter sido realizado no SPE são escritos (passo 3) na mesma região da matriz de heurísticas passada como entrada.

3.3.3 Redução do custo de sincronização com os SPEs

A terceira e última abordagem para montagem das tarefas visa reduzir o custo adicional da sincronização do PPE com os SPEs. Como será mostrado no Capítulo 4, o algoritmo do PPE utilizando a abordagem descrita anteriormente gasta mais tempo na etapa de sincronização do que na busca de caminho. Esta carga extra prejudica o compromisso do algoritmo com a restrição de tempo real. Para manutenção deste compromisso, seria necessária uma redução considerável do *lookahead* da busca. Visando resolver este problema, a solução seria reduzir o tempo gasto pelo PPE com a etapa de sincronização.

A primeira ideia foi a de se utilizar um segundo fluxo de execução, ou *thread*, no PPE, que seria responsável pela sincronização com os outros SPEs. Esta ideia surgiu em consequência do algoritmo de busca possuir muitas instruções condicionais, o que poderia gerar certa quantidade de endereços não resolvidos, resultando em "bolhas"(ou "buracos") no *pipeline* de execução [Hennessy et al., 2003]. Dessa forma, como o núcleo do PPE é *multithreaded*, uma segunda *thread* poderia preencher estas "bolhas"no *pipeline* com suas instruções.

O problema desta abordagem é que, mesmo estabelecendo prioridade para a *thread* da busca, ainda existiria concorrência de recursos computacionais entre outras *threads*. Relembrando o ciclo de execução de algoritmos de busca de caminho em

tempo real, onde são intercaladas etapas de planejamento e ação, esta segunda *thread* poderia competir em recursos com a etapa de ação. Outra maneira de se enxergar tal problema é considerar que tal *thread* estaria "burlando"o compromisso com respostas em tempo real, visto que o algoritmo de busca retorna de sua execução quando ainda existem recursos computacionais sendo consumidos pela parte de sincronização. Um exemplo prático desta situação é o ciclo de um jogo (*game-loop*), composto por diversas etapas, incluindo as de planejamento/ação, onde algoritmos de Inteligência Artificial constituem apenas algumas dessas etapas. Esta segunda *thread* para a sincronização com os SPEs entraria em concorrência com o restante do *game-loop*, prejudicando o desempenho geral do jogo.

Uma segunda hipótese considerada seria transferir a etapa de sincronização para um SPE. Esta hipótese não se mostrou interessante devido à quantidade limitada de memória dos SPEs. Essa limitação faz com que o heap de tarefas tenha uma capacidade consideravelmente limitada. Mesmo se o heap fosse mantido no PPE, o SPE responsável pela sincronização necessitaria de comunicação constante com o PPE para a gerência do heap de tarefas, o que não reduziria o custo. A sincronização seria simplesmente concentrada em apenas um SPE, sendo este uma ponte de comunicação com os demais SPEs.

Figura 3.9: Abordagem para montagem de tarefas com autonomia para os SPEs.

A terceira hipótese surgiu analisando o modelo de tarefa definido. Cada tarefa corresponde a um trecho de busca, e não a uma busca completa. Dessa forma, os SPEs estão sempre requisitando novos trechos, gerando um alto custo de sincronização. Dessa análise veio a ideia de permitir maior autonomia por parte dos SPEs: à partir de

uma tarefa recebida, estes passariam a realizar a busca até que a mesma chegasse no estado objetivo s_g. Nesta abordagem, os SPEs ficariam responsáveis pela montagem das tarefas subsequentes à tarefa enviada pelo PPE, ou seja, a primeira tarefa de uma busca é montada pelo PPE, e as outras são montadas pelo próprios SPEs até que a busca chegue ao estado objetivo (passo 1 da Figura 3.9). Os passos subsequentes (2 e 3) permanecem iguais aos da abordagem anterior.

```
1   sincronização com PPE para receber tarefa no buffer b0
2   sincronização com PPE para receber tarefa no buffer b1
3   while tarefas ≠ fim do
5      aguarda preenchimento de b0
6      realiza uma iteração do LSS-LRTA*(s0, sg, d)
7      escreve novos valores h no endereço de saída e0
8      if s0 = sg
9         sincronização com PPE para receber tarefa no buffer b0
10     end if
11     aguarda preenchimento de b1
12     realiza uma iteração do LSS-LRTA*(s1, sg, d)
13     escreve novos valores h no endereço de saída e1
14     if s1 = sg
15        sincronização com PPE para receber tarefa no buffer b1
16     end if
17  end while
```

Algoritmo 3.3: O algoritmo do SPE com modificação de autonomia.

O Algoritmo 3.3 mostra o fluxo de execução dentro dos SPEs após a alteração de autonomia para os mesmos. A diferença deste algoritmo para o apresentado na Seção 3.2 são os passos que verificam se o estado s é igual a s_g (linhas 8 e 14). A sincronização com o PPE para requisição de uma nova tarefa (linhas 9 e 15) só é feita quando tais condicionais são satisfeitas. Caso contrário, o próprio SPE realiza a montagem da nova tarefa. Para localizar o endereço na memória compartilhada dos novos *tiles* que irão compor próxima tarefa, o algoritmo do SPE faz o deslocamento do endereço anterior, utilizando as diferenças de coordenadas entre o antigo e o novo valor de s (antes e após a etapa de busca, respectivamente).

O fato de dar mais autonomia aos SPEs, além de tornar esta etapa um processo assíncrono, diminui o controle do PPE sobre a região a ser explorada e a ter os valores heurísticos do estados atualizados. Essa diminuição do controle pode fazer com que os SPEs realizem suas buscas em regiões comuns, ocorrendo uma maior sobreposição de tarefas e prejudicando o processo de convergência. Uma possível alternativa para contornar tal situação seria fazer com que os SPEs compartilhassem suas listas fechadas,

de forma a evitar a exploração de uma mesma região. Mesmo sem levar em conta a memória limitada destes processadores, este compartilhamento exigiria uma etapa extra de sincronização entre os SPEs, o que poderia tornar o processo tão lento quanto aquele com a presença da etapa sincronização com o PPE.

De fato, esse aumento na sobreposição de tarefas pode até ocorrer, o que diminuiria o espalhamento dos estados do mapa com valores heurísticos atualizados, aumentando o tempo gasto pelo PPE com as buscas e, consequentemente, o tempo de convergência. Entretanto, os resultados empíricos apresentados no Capítulo 4 derrubam esta hipótese, demonstrando que o ganho com a redução do tempo de sincronização é muito mais significativo do que possíveis aumentos no tempo gasto pelo PPE com a busca de caminho devido ao aumento de sobreposições de tarefas.

3.4 Resumo

Este capítulo apresentou a estratégia de paralelização proposta nesse trabalho. A ideia básica é utilizar núcleos de processamento auxiliares para efetuar trechos de busca, compartilhando o aprendizado adquirido com a busca principal, visando diminuir o tempo de convergência. Foram apresentados os fluxos de execuções do algoritmo para o núcleo principal e para os núcleos auxiliares, além de mostrar a estrutura básica das tarefas, bem como são montadas, ordenadas e distribuídas. Também foram discutidas as melhorias acrescentadas na etapa de montagem de tarefas.

Capítulo 4

Avaliação experimental

Este capítulo apresenta os experimentos realizados para avaliação da estratégia de paralelização criada. Foi avaliado o tempo total de convergência da solução para o algoritmo paralelo utilizando os três métodos de montagem de tarefas, descritos na Seção 3.3. A avaliação do paralelismo de tarefas, medido em tarefas processadas por segundo, também é apresentada neste capítulo. Além disso, é também mostrada uma avaliação mais detalhada do tempo total de execução em relação ao número de SPEs utilizados, realizada para a paralelização que utiliza a redução de sincronização através da maior autonomia para os SPEs. Por fim, apresentamos uma avaliação, realizada em um ambiente simulado, da quantidade de tentativas para a convergência utilizando esta última paralelização.

Embora todas as buscas utilizadas nas avaliações mostrem uma diminuição no tempo total de execução, o comportamento especulativo dos métodos de busca de caminho faz com que o processo de convergência não seja regular para todas as buscas. Buscas com mais mínimos locais tendem a fazer com que os SPEs explorem mais uma mesma região do mapa, gerando uma maior sobreposição de tarefas em relação a outras buscas com menor quantidade de mínimos locais. Dessa forma, a natureza especulativa destas técnicas acarreta em anomalias de ganho (*speedup*), não sendo este uniforme para todas as buscas. Por esta razão, a avaliação de *speedup* por número de SPEs utilizados não foi utilizada neste trabalho.

As avaliações experimentais dos algoritmos implementados neste trabalho foram realizadas utilizando *blades* IBM BladeCenter QS20. Foi utilizado o *cluster* do Georgia Institute of Technology. Cada *blade* destas é composta por 2 processadores Cell BE 3.2GHz, com 1GB de memória principal (512MB por processador). Os grafos utilizados nestes experimentos são modelados a partir de ambientes presentes em jogos reais como

Baldur's Gate [1]. Para as buscas, são utilizados 13 mapas distintos, mostrados no apêndice A.

Conforme discutido no Capítulo 2, para todos os experimentos realizados neste trabalho o algoritmo de busca em tempo real escolhido foi o LSS-LRTA*[Koenig & Sun, 2009]. As buscas em paralelo executadas nos experimentos utilizam um *lookahead* igual a 5 para o PPE e 3 vezes maior nos SPEs ($d = 15$). A busca sequencial é executada somente no PPE, com um *lookahead* de tamanho 5.

A presença do paralelismo altera o comportamento deste algoritmo em comparação com buscas de caminho sequenciais. Como existem tarefas sendo executadas em paralelo, não se tem garantia da ordem de chegada das mesmas. Dessa forma, em situações onde as tarefas são relacionadas a regiões do mapa que se sobrepõem (ou simplesmente sobreposição de tarefas, conforme mencionado anteriormente) não se sabe qual tarefa irá atualizar os estados da região por último. Uma mesma busca pode então apresentar variações, o que pode ser considerado um comportamento de característica não determinística. Assim sendo, para todos os experimentos realizados, cada resultado exibido foi obtido pela média de 5 execuções.

O restante deste capítulo é organizado da seguinte forma: na Seção 4.1 mostramos os comparativos de tempo de execução da paralelização utilizando as três abordagens de montagem de tarefas, na Seção 4.2 apresentamos a medida de desempenho em tarefas processadas por segundo (*throughput*) para a paralelização com redução de custo de sincronização com os SPEs, a Seção 4.3 mostra o tempo de execução para esta última paralelização variando a quantidade de SPEs utilizados, e finalmente, na Seção 4.4 reproduzimos um ambiente típico de aplicação deste algoritmo para mostrar um comparativo entre o número de tentativas para a convergência executadas na versão sequencial e na versão paralela.

4.1 Tempo de Execução

Nesta seção analisamos o tempo total de execução do processo de convergência até a tentativa de custo ótimo. São demonstradas um total de 14 buscas aleatórias. As 7 primeiras buscas correspondem às buscas realizadas em mapas que não apresentam muitas ramificações de caminhos, por não possuírem muitas divisões de cômodos. Dessa forma, existe uma menor possibilidade de ocorrência de mínimos locais nas buscas realizadas nestes ambientes, e assim a convergência para as versões sequenciais das buscas ocorre em uma quantidade menor de tentativas. As outras 7 buscas foram

[1] http://www.bioware.com/games/baldurs_gate/

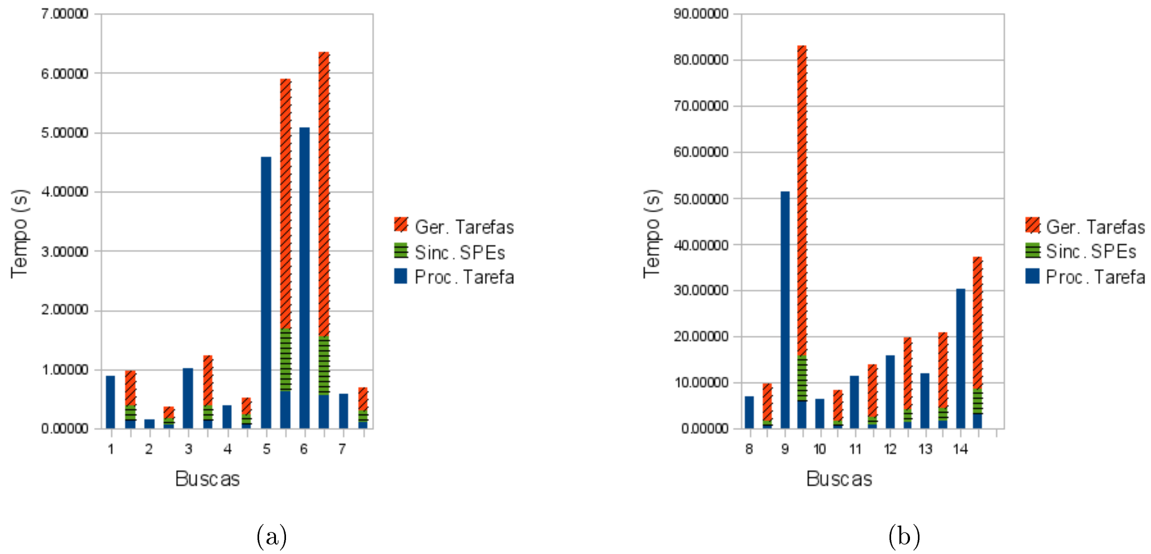
realizadas em mapas com regiões onde há uma maior quantidade de paredes e cômodos, com maior possibilidade de ramificações, e consequentemente maior probabilidade de ocorrência de mínimos locais nas buscas. A convergência na versão sequencial das buscas nesses ambientes ocorre em uma quantidade maior de tentativas, chegando a mais de 2000 tentativas para a busca de maior tempo. Em todos os resultados apresentados, as buscas estão numeradas de 1 a 14.

4.1.1 Paralelização utilizando compartilhamento da tabela Hash de heurísticas

Esta seção avalia o tempo total para a convergência do universo de buscas para a versão paralela do algoritmo que compartilha a tabela Hash de valores heurísticos. A Figura 4.1 apresenta os resultados, que estão organizados em pares de buscas, onde a busca à esquerda de cada par corresponde à execução sequencial, e a busca à direita de cada par à versão paralela. O tempo de execução na versão paralela está segmentado nas três etapas do algoritmo: a busca de caminho em si (processamento de tarefa), a sincronização do PPE com os SPEs para distribuição das tarefas, e a gerência de tarefas, que engloba a criação, montagem e ordenação destas. É importante ressaltar que a escala do eixo de tempo é diferente de (a) para (b). Como visto, o tempo de execução da versão paralela ultrapassa o tempo da versão sequencial em todas as buscas. Analisando a divisão de carga de trabalho da versão paralela, percebe-se que o tempo gasto com a busca (em azul) é menor que o tempo gasto com busca na versão sequencial. O *overhead* é causado pela sincronização com os SPEs e principalmente pela gerência de tarefas, onde, conforme discutido na Seção 3.3, o PPE é o responsável por preencher os *buffers* de tarefas dos SPEs.

4.1.1.1 Avaliação da ocupação de memória pelo Hash

A porcentagem de ocupação da tabela Hash de valores heurísticos compartilhados para estas buscas é mostrada na Figura 4.2. Observa-se que o percentual para as buscas em paralelo é semelhante ao percentual da versão sequencial. Para maioria das buscas, o percentual varia entre 5% e 20%, não atingindo a marca de 50% em nenhum caso. Conforme discutido anteriormente, a substituição desta tabela Hash por uma matriz resultaria então em uma matriz esparsa.



(a) (b)

Figura 4.1: Tempo de execução total até a convergência da solução para versão do algoritmo com tabela Hash: (a) buscas em mapas com poucos mínimos locais, (b) buscas em mapas com muitos mínimos locais. A busca à esquerda de cada par corresponde à execução sequencial, e a busca à direita à versão paralela.

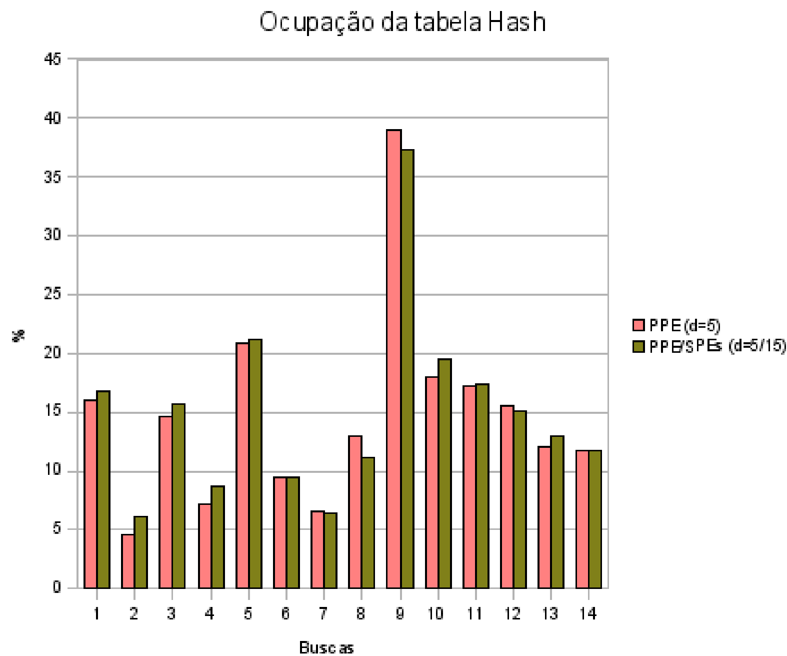


Figura 4.2: Porcentagem de ocupação da tabela Hash em relação ao tamanho do mapa para as versões sequencial e paralela das buscas.

4.1.2 Paralelização utilizando matriz esparsa de heurísticas aprendidas

Esta seção avalia o algoritmo em paralelo que utiliza uma matriz esparsa no lugar da tabela Hash para representar os valores heurísticos. Conforme descrito na Seção 3.3, essa abordagem retira do PPE a responsabilidade de preenchimento dos *buffers*. A Figura 4.3 mostra os resultados das execuções, organizados em pares de buscas, de maneira similar à apresentada na Figura 4.1.

Nesta versão, nota-se que o *overhead* causado pelo preenchimento dos *buffers* desaparece, e a etapa de gerência de tarefas passa a ter um custo quase imperceptível. O tempo total de execução para convergência das buscas em paralelo já se apresenta inferior ao tempo da versão sequencial. Mesmo assim, percebe-se ainda um *overhead* significativo, causado pela etapa de sincronização do PPE com os SPEs. Esta etapa consome aproximadamente 64% do tempo total.

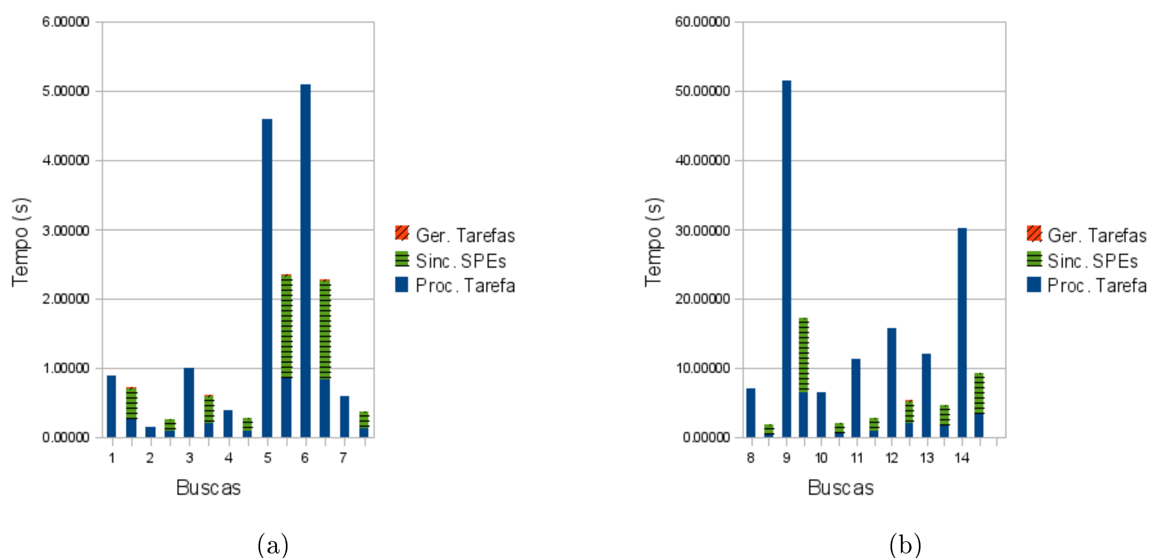


Figura 4.3: Tempo de execução total até a convergência da solução para versão do algoritmo com matriz esparsa: (a) buscas em mapas com poucos mínimos locais, (b) buscas em mapas com muitos mínimos locais. A busca à esquerda de cada par corresponde à execução sequencial, e a busca à direita à versão paralela.

4.1.3 Paralelização com redução do custo de sincronização dos SPEs

A avaliação do algoritmo com a modificação para maior autonomia dos SPEs visando reduzir o custo de sincronização é apresentada nesta seção. A Figura 4.4 apresenta os

resultados organizados em pares, de maneira similar aos apresentados na Figura 4.1. As Tabelas 4.1 e 4.2 mostram os valores, em segundos.

Nestes experimentos, é possível observar que o *overhead* causado tanto pelo preenchimento dos *buffers* quanto pela sincronização não é mais impactante. Enquanto no primeiro caso esse custo adicional não mais existe, para o caso da sincronização, mesmo ainda existindo uma pequena parcela, ocorre uma notável redução (lembrando que a sincronização ainda existe, sendo que um SPE faz a requisição de uma nova tarefa para o PPE apenas quando sua busca chega ao objetivo). O *overhead* de sincronização é ainda menor para buscas mais extensas, como observado na Figura 4.4(b) e na Tabela 4.2. Isto se justifica porque, em buscas maiores, os SPEs levam mais tempo para atingir o estado objetivo, e com isso requisitam menos tarefas para o PPE, diminuindo o tempo gasto com sincronização.

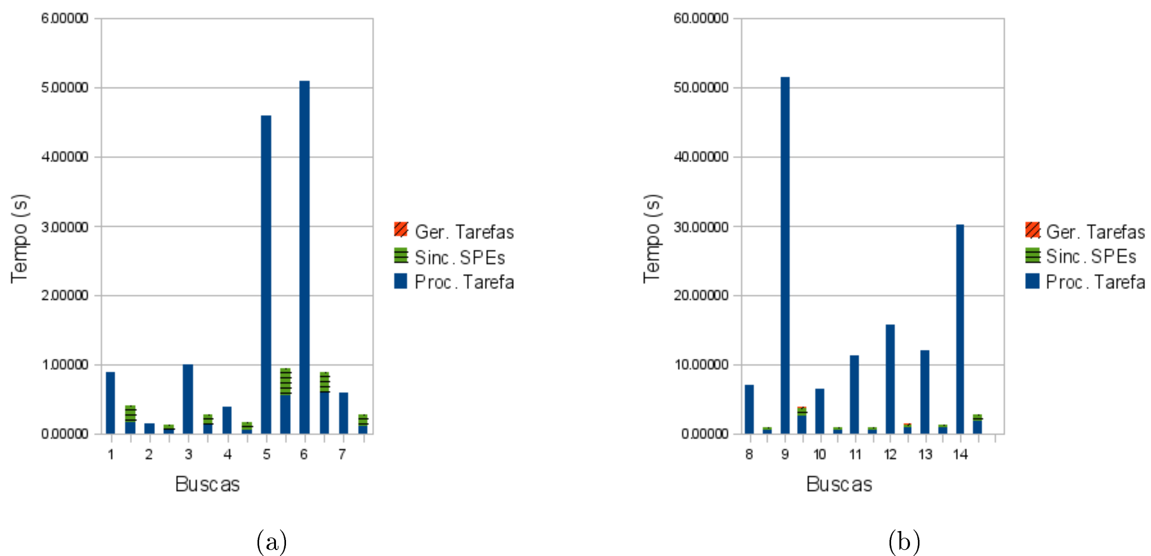


Figura 4.4: Tempo de execução total até a convergência da solução para versão do algoritmo com autonomia para os SPEs: (a) buscas em mapas com poucos mínimos locais, (b) buscas em mapas com muitos mínimos locais. A busca à esquerda de cada par corresponde à execução sequencial, e a busca à direita à versão paralela.

Algoritmo	busca 1	busca 2	busca 3	busca 4	busca 5	busca 6	busca 7
Processamento	0.16784	0.05527	0.13358	0.06895	0.56139	0.59220	0.12192
Sincronização	0.24206	0.08293	0.14753	0.09840	0.38515	0.29770	0.16081
Gerência	0.00075	0.00027	0.00063	0.00036	0.00153	0.00237	0.00060
Total Paralelo	0.41065	0.13847	0.28174	0.16771	0.94807	0.89227	0.28333
Sequencial	0.90097	0.15979	1.01299	0.39413	4.58590	5.08936	0.59745

Tabela 4.1: Tempo total (s) para a convergência. Buscas 1 à 7.

Algoritmo	busca 8	busca 9	busca 10	busca 11	busca 12	busca 13	busca 14
Processamento	0.52728	2.70891	0.59272	0.65958	1.02441	0.98362	1.87429
Sincronização	0.43355	1.13826	0.34435	0.30766	0.41957	0.37967	0.88401
Gerência	0.00294	0.01511	0.00311	0.00405	0.00544	0.00553	0.00712
Total Paralelo	0.96377	3.86228	0.94018	0.97129	1.44942	1.36882	2.76542
Sequencial	7.15000	51.41000	6.48000	11.42000	15.83590	12.05000	30.28000

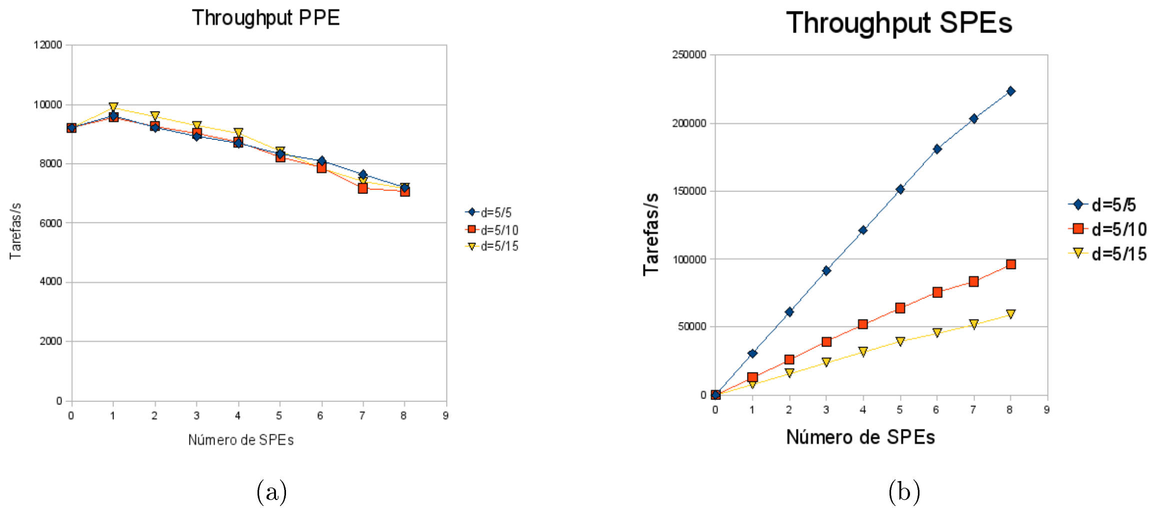
Tabela 4.2: Tempo total (s) para a convergência. Buscas 8 à 14.

4.2 Paralelismo de tarefas

Esta seção apresenta uma medida do paralelismo de tarefas, ou *throughput*, para a abordagem de paralelização com redução de custo de sincronização com os SPEs. O *throughput* é importante para se avaliar a escalabilidade da paralelização proposta. Em outras palavras, para se avaliar a relação do número total de tarefas processadas com o aumento do número de núcleos de processamento utilizados. Para esta avaliação, foi escolhida a busca de número 12 do universo de buscas. O motivo dessa escolha é o fato desta busca pertencer ao segundo grupo de buscas, onde o número de tentativas para convergência da versão sequencial é maior, o que permite um maior número de tarefas para serem processadas. Ainda nesta avaliação, foram executados três grupos de buscas: no primeiro grupo, tanto o *lookahead* do PPE quanto o dos SPEs é igual 5; no segundo grupo o *lookahead* dos SPEs vale o dobro do PPE (5 para o PPE e 10 para os SPEs) e no terceiro grupo o valor do *lookahead* é igual a 5 para o PPE e 15 para os SPEs. Estes grupos foram criados levando em conta que o *lookahead* está diretamente relacionado ao tempo de execução de uma tarefa, e consequentemente à quantidade total de tarefas processadas. Os valores relativos à quantidade zero de SPES correspondem à da versão sequencial do algoritmo de busca.

A Figura 4.5 mostra o *throughput* para o PPE(a) e para os SPEs(b). Como observado, o *throughput* no PPE decresce quando se aumenta o número de SPEs utilizados. Este comportamento ocorre porque quanto maior o número de SPEs, mais atualizações dos valores heurísticos ocorrerão, e em consequência disso menos tarefas serão necessárias para o PPE processar até a convergência. Na Figura 4.5(b),percebe-se que o *throughput* tem uma maior taxa de crescimento no grupo de buscas 5/5. Para os demais grupos, o aumento é atenuado na medida que o tamanho do *lookahead* dos SPEs aumenta. Esse fenômeno ocorre pois como em todos os três grupos a latência da transferência de memória já está oculta pelo processamento em 2 *buffers* nos SPEs (ver Seção 3.2), à medida que se aumenta o *lookahead* da tarefa o tempo de processamento desta também aumenta. Para o caso deste algoritmo, o menor *throughput* das buscas que possuem maior *lookahead* não implica em uma diminuição do desempenho, pois

uma busca de maior profundidade faz com que uma área maior da matriz de valores heurísticos seja atualizada, diminuindo o tempo total de convergência.



(a) (b)

Figura 4.5: *Throughput* do PPE(a) e dos SPEs(b).

4.3 Variação do número de SPEs

Esta seção analisa o tempo total para convergência de uma busca em relação à quantidade de SPEs utilizados. O *lookahead* da busca do PPE é igual à 5, e dos SPEs igual à 15. Para esta avaliação também foi escolhida a busca de número 12 do universo de buscas, devido ao maior tempo para convergência, o que minimiza os erros das medições.

Embora o *speedup* seja variável de uma busca para outra, conforme já dito no início deste capítulo, o comportamento do gráfico de tempo total de execução por número de SPEs apresenta uma característica comum a todas as buscas realizadas. Essa característica, observada na Figura 4.6, é que o tempo de convergência passa a ser semelhante à partir da utilização de 5 SPEs até a utilização de todos os 8 SPEs.

A criação das tarefas tem como objetivo evitar o máximo de colisão possível entre as áreas de busca. Mas o fato de dar mais autonomia aos SPEs faz com que esse controle fuja um pouco do PPE, de certa forma. Este comportamento acarreta uma maior sobreposição de tarefas dos SPEs. Tal sobreposição tende a aumentar na medida em que o número de SPEs utilizados cresce. Esta pode ser a justificativa para o fato dos tempos de execução serem semelhantes na utilização de 5 até 8 SPEs. Com

menos SPEs, o espalhamento das tarefas funciona de melhor forma, diminuindo as possibilidades de sobreposição.

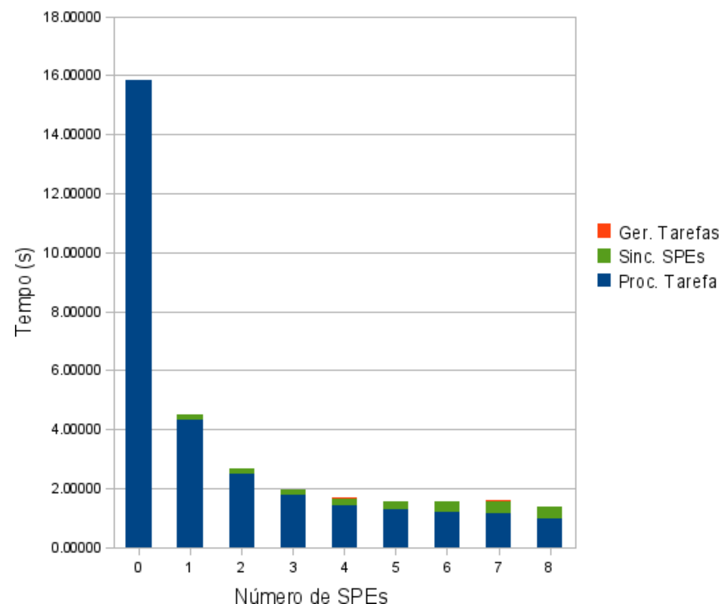


Figura 4.6: Tempo total de execução por número de SPEs utilizados.

Devido ao fenômeno descrito acima, a utilização de todos os SPEs pode não ser tão vantajosa. A redução do número de SPEs utilizados, além de prover uma economia de energia, libera recursos para outras aplicações. No caso de um jogo, os SPEs não utilizados poderiam ser aproveitados em outras partes do *game-loop*, por exemplo.

4.4 Intercalando planejamento e execução

Até aqui, todos os experimentos realizados neste capítulo para avaliar o desempenho apresentavam um comportamento de execução ininterrupta. Isto quer dizer que as chamadas do algoritmo são feitas de maneira sucessiva, sem nenhuma outra execução entre as chamadas. Entretanto, a utilização de algoritmos de busca de caminho em tempo real consiste em uma execução por etapas intercaladas, podendo até mesmo ocorrer o processamento de outras cargas não relacionadas à busca. Um exemplo deste tipo de execução é o ambiente de um jogo, onde a etapa de planejamento é intercalada com outras etapas, como por exemplo as ações a serem tomadas pelos NPCs após o planejamento. Um ciclo de execução destas etapas em um jogo recebe o nome de *game-loop*. Um detalhe importante é que o planejamento pode ser realizado para um grande trecho de caminho, sendo necessário mais do que um ciclo para que o NPC se mova por este trecho. Tal situação elimina a necessidade de se executar a etapa de planejamento

em cada ciclo, o que pode então aumentar o intervalo entre uma chamada e outra do algoritmo. Dessa forma, para se obter uma medida de desempenho do algoritmo mais próxima de uma situação real, modelamos as condições de tempo para serem similares a de um jogo, que intercala busca de caminho com outras etapas.

Para se estimar o tamanho do *game-loop*, avaliamos então algumas características dos jogos. A maioria dos jogos atuais para consoles utilizam uma taxa de atualização de 60Hz, o que significa dizer que são gerados 60 quadros por segundo. Dessa forma, o intervalo entre um quadro e o próximo é de 1/60 segundos. Considerando o fato de um ciclo do *game-loop* ser executado a cada quadro, esta aproximação é válida para estimar o tamanho do *game-loop*. Considere então uma situação em que o NPC se movimenta uma quantidade x de estados de um ciclo para outro. Na pior das hipóteses, esta quantidade será exatamente igual ao *lookahead* da busca, acarretando em uma requisição de busca de caminho por ciclo. Com base nessas informações, foi utilizado um intervalo de 1/60 segundos entre as requisições de buscas para o PPE. Este intervalo simula a execução das outras etapas do *game-loop*.

Algoritmo	busca 1	busca 2	busca 3	busca 4	busca 5	busca 6	busca 7
Sequencial	208	35	158	95	186	205	113
Paralelo	34	13	22	12	9	7	15

Tabela 4.3: Número de tentativas para a convergência. Buscas 1 à 7.

Algoritmo	busca 8	busca 9	busca 10	busca 11	busca 12	busca 13	busca 14
Sequencial	611	2058	727	1098	675	397	726
Paralelo	13	20	26	26	8	7	5

Tabela 4.4: Número de tentativas para a convergência. Buscas 8 à 14.

A Figura 4.7 mostra o número de tentativas até a convergência para as buscas realizadas no experimento neste cenário de simulação. Os gráficos são exibidos em escala aritmética, para uma melhor visão geral, e logarítmica, para facilitar a identificação dos valores. Os resultados também são mostrados nas Tabelas 4.3 e 4.4. Como visto, os ganhos são ainda mais significativos quando comparados a uma execução ininterrupta, em certos casos mais do que 100 vezes superior. Este ganho ocorre principalmente pela inserção do intervalo entre as requisições. Enquanto na versão sequencial o processador se ocupa com outras cargas de processamento, para então voltar ao processamento da busca somente na próxima chamada do algoritmo, na versão paralela o PPE se ocupa com estas outras cargas, enquanto os SPEs permanecem realizando suas tarefas em paralelo, trabalhando em segundo plano, mas com dedicação exclusiva. Um detalhe

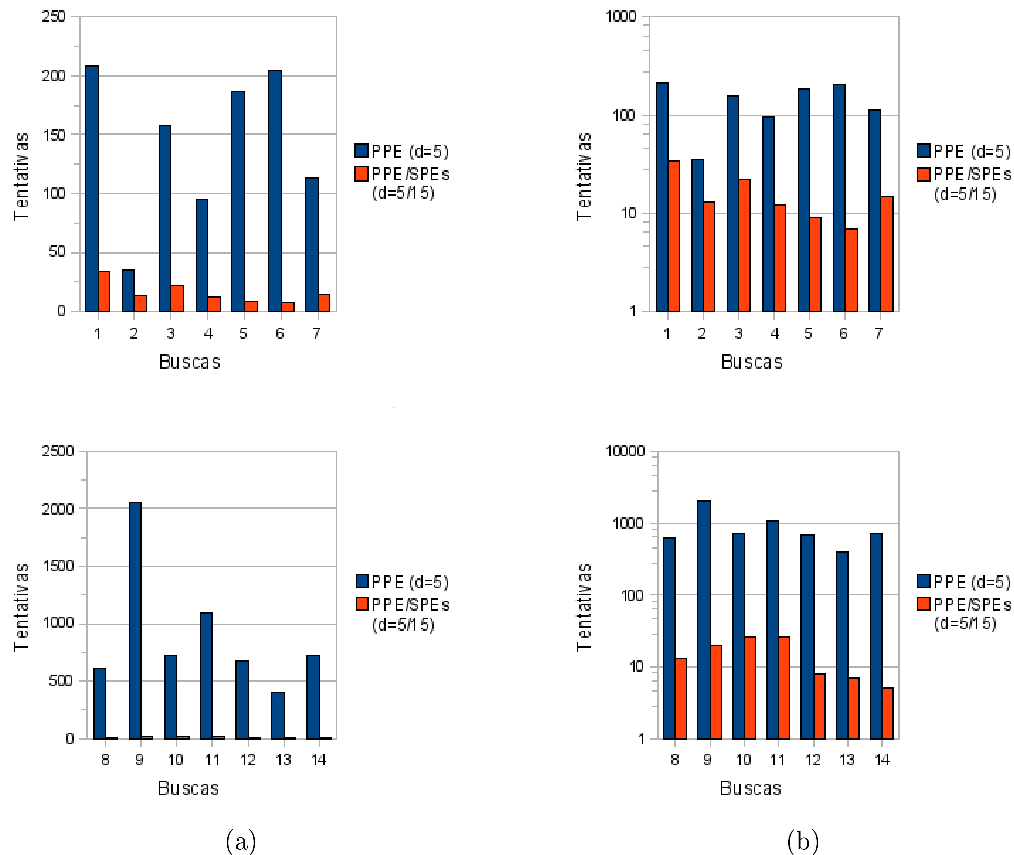


Figura 4.7: Total de tentativas até a convergência: (a) escala aritmética, (b) escala logarítmica.

relevante na última versão da paralelização é que a modificação para conceder maior autonomia aos SPEs tem influência positiva no desempenho, pois ela reduz a quantidade de sincronização com o PPE, diminuindo assim a possibilidade dos SPEs estarem ociosos, aguardando nova tarefa, enquanto o PPE se ocupa com outras cargas. É importante observar que este desempenho está próximo de uma situação real do que nas outras avaliações, sendo então o ganho mais representativo para se avaliar a paralelização realizada.

Capítulo 5

Conclusões

Neste capítulo apresentamos um resumo da contribuição desse trabalho, algumas limitações, bem como apontamos direções para trabalhos futuros.

5.1 Sumário dos resultados

Este trabalho estudou o problema relacionado ao tempo de convergência de uma busca de caminho em tempo real, ou seja, o processo de aprendizado do algoritmo de busca. O objetivo, de forma geral, relacionou-se à utilização de ambientes paralelos para acelerar a convergência, utilizando núcleos de processamento auxiliares que compartilham o mesmo aprendizado adquirido com a busca principal.

Para este trabalho, foram estudados diversos algoritmos de busca em tempo real, visando compreender o comportamento básico destes. Foi escolhida também uma arquitetura específica para a implementação, o Cell BE, presente em um console de última geração. Esta arquitetura foi estudada visando compreender suas características e limitações. Com base nesses estudos, foi então criada a estratégia de paralelização, onde o aprendizado é compartilhado entre as buscas, visando reduzir o tempo total de convergência.

Os primeiros resultados deste trabalho vêm de uma versão inicial da estratégia de paralelização proposta, adequada à uma arquitetura específica escolhida, no caso o processador Cell BE. Nesta primeira abordagem, que tentou preservar as características da busca original, mantendo a tabela Hash desta para armazenamento dos valores heurísticos aprendidos, o desempenho apresentado não se mostrou satisfatório. Nela, o tempo gasto pelo processador principal (o PPE) com o preenchimento dos *buffers* de tarefas dos processadores auxiliares (os SPEs) apresentou um custo adicional maior até que o tempo gasto propriamente com a busca de caminho.

Posteriormente, foi proposta uma segunda abordagem, onde se sugeria um *trade-off* entre memória e tempo de processamento, ocorrido pela substituição da tabela Hash por uma matriz esparsa. Esta modificação possibilitou uma queda do custo de gerenciamento de tarefas realizado pelo PPE. Tal queda se deu pela eliminação da responsabilidade de preenchimento dos *buffers* de tarefas pelo PPE.

Finalmente, foi feita uma modificação na estratégia paralelização visando diminuir o custo de sincronização entre o PPE e os SPEs. Esta abordagem, que permitia uma maior autonomia das buscas auxiliares, mostrou um desempenho satisfatório, sempre com tempos de execução inferiores à versão sequencial da mesma busca em tempo real, bem como em relação às buscas que utilizaram as estratégias de paralelização descritas anteriormente.

Este trabalho ainda apresentou um mecanismo para classificação de estados, baseados no comportamento das colônias de formigas. Este mecanismo reduz a área explorada pelas buscas, pois restringe esta às regiões mais próximas do caminho ótimo. Esta metodologia foi utilizada para ordenação das tarefas, onde as tarefas com maior prioridade são aquelas cujo estado inicial é classificado como mais promissor.

Analisando os resultados apresentados na Seção 4.4, onde o experimento simula uma situação de jogo, nota-se uma ordem de ganho considerável na versão paralela em detrimento da versão sequencial. De certa forma, este ganho está relacionado não somente à descentralização das tarefas, mas também ao assincronismo no processamento paralelo, que permite o processamento de vários trechos de buscas pelos SPEs até mesmo quando o PPE se ocupa com outros processamentos não relacionados à busca. Além disso, a forma na qual a estratégia de paralelização foi construída, orientada por tarefa, permite a execução de diversas buscas simultaneamente. Esta propriedade garante a escalabilidade da paralelização, o que torna possível a distribuição da extensa carga causada pelo aumento massivo do número de NPCs executando buscas simultaneamente, ou pelo aumento do tamanho e da complexidade dos mapas.

5.2 Limitações e trabalhos futuros

Algumas limitações e considerações sobre esse trabalho, que motivam a continuação do mesmo, são apresentadas à seguir:

1. Embora não comprovada empiricamente, existe uma possibilidade de que seguidas sobreposições de tarefas façam com que a busca em paralelo leve mais tempo para a convergência do que a busca sequencial. Esta possibilidade ocorreria quando, em situações de buscas por regiões com muitos mínimos locais, onde diversas

buscas auxiliares seriam realizadas numa mesma região, as buscas mais antigas atualizassem a matriz esparsa sempre após as buscas mais novas. Dessa forma, a convergência seria atrasada, existindo a remota possibilidade de até mesmo não ocorrer.

2. Também seria interessante avaliar a metodologia de ordenação das tarefas utilizando outros algoritmos, como buscas em tempo real que utilizem abstrações de grafos. O algoritmo de classificação de tarefas pode ser melhorado, considerando, por exemplo, os diversos níveis de abstração destas buscas.

3. A utilização desta estratégia de paralelização em outros contextos é uma possibilidade interessante. A substituição da tabela de heurísticas numa tabela de roteamento, por exemplo, poderia levar esta metodologia à aplicação em sistemas CDN (*Content Delivery Network*), onde o algoritmo seria utilizado para a construção de tabelas de roteamento dos espelhos de conteúdo em situações onde restrições de tempo real fossem exigidas. Outras áreas interessantes a serem levadas em consideração para aplicação desta estratégia são sistema de redes e sensores sem fio, redes *ad hoc*, par-a-par, ou sistemas distribuídos que exijam algum compromisso com tempo real.

Referências Bibliográficas

Barták, R. (1999). Constraint programming: In pursuit of the holy grail. In *Proceedings of WDS99 (invited lecture), Prague, June*. Citeseer.

Bulitko, V. & Bjornsson, Y. (2009). kNN LRTA*: Simple Subgoaling for Real-Time Search. *Artificial Intelligence for Interactive Digital Entertainment Conference.*

Bulitko, V.; Lustrek, M.; Schaeffer, J.; Bjornsson, Y. & Sigmundarson, S. (2008). Dynamic control in real-time heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 32:419 – 452.

Bulitko, V.; Sturtevant, N.; Lu, J. & Yau, T. (2007). Graph abstraction in real-time heuristic search. *J. Artif. Int. Res.*, 30(1):51--100.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269--271.

Evett, M.; Mahanti, A.; Nau, D.; Hendler, J. & Hendler, J. (1995). Pra*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 25:133--143.

Furcy, D. (2006). ITSA*: Iterative tunneling search with A*. In *Proceedings of the National Conference on Artificial Intelligence (AAAI), Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications, Boston, Massachusetts.*

Gschwind, M.; Hofstee, H. P.; Flachs, B.; Hopkins, M.; Watanabe, Y. & Yamazaki, T. (2006). Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10--24.

Hansen, E. & Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28(1):267--297.

Hart, P.; Nilsson, N. & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions*, 4(2):100–107.

Hennessy, J.; Patterson, D.; Goldberg, D. & Asanovic, K. (2003). *Computer architecture: a quantitative approach*. Morgan Kaufmann.

Hernández, C. & Meseguer, P. (2005). LRTA*(k). In *Proceedings of the 19th international joint conference on Artificial intelligence*, pp. 1238--1243. Morgan Kaufmann Publishers Inc.

Holldobler, B. & Wilson, E. (1990). *The ants*. Belknap Press.

Kennedy, J.; Eberhart, R. & Shi, Y. (2001). *Swarm intelligence*. Morgan Kaufmann Publishers.

Koenig, S. & Likhachev, M. (2002). Incremental A^*. *Advances in Neural Information Processing Systems*, 2:1539--1546.

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

Koenig, S.; Tovey, C. & Smirnov, Y. (2003). Performance bounds for planning in unknown terrain. *Artificial Intelligence*, 147(1-2):253--279.

Korf, R. (1985). Depth-first iterative-deepening* 1:: An optimal admissible tree search. *Artificial intelligence*, 27(1):97--109.

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

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

Kornfeld, W. A. (1981). The use of parallelism to implement a heuristic search. In *In IJCAI*, pp. 575--580.

Pemberton, J. & Korf, R. (1992). Making locally optimal decisions on graphs with cycles.

Powley, C. & Korf, R. (1989). Single agent parallel window search: A summary of results. In *Proceedings of the eleventh International Joint Conference on Artificial Intelligence*, pp. 36--41. Citeseer.

Reinefeld, E. & Schnecke, V. (1994). Aida* - asynchronous parallel ida*. In *Paderborn Center for Parallel Computing*, pp. 295--302.

Rios, L. & Chaimowicz, L. (2010). A survey and classification of a* based best-first heuristic search algorithms. In *Advances in Artificial Intelligence - SBIA 2010*, volume 6404 of *Lecture Notes in Computer Science*, pp. 253–262. Springer Berlin / Heidelberg.

Russell, S. & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Russell, S. & Wefald, E. (1991). *Do the right thing: studies in limited rationality*. The MIT Press.

Schulte, C. (2000). Parallel search made simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP*, pp. 41--57. Citeseer.

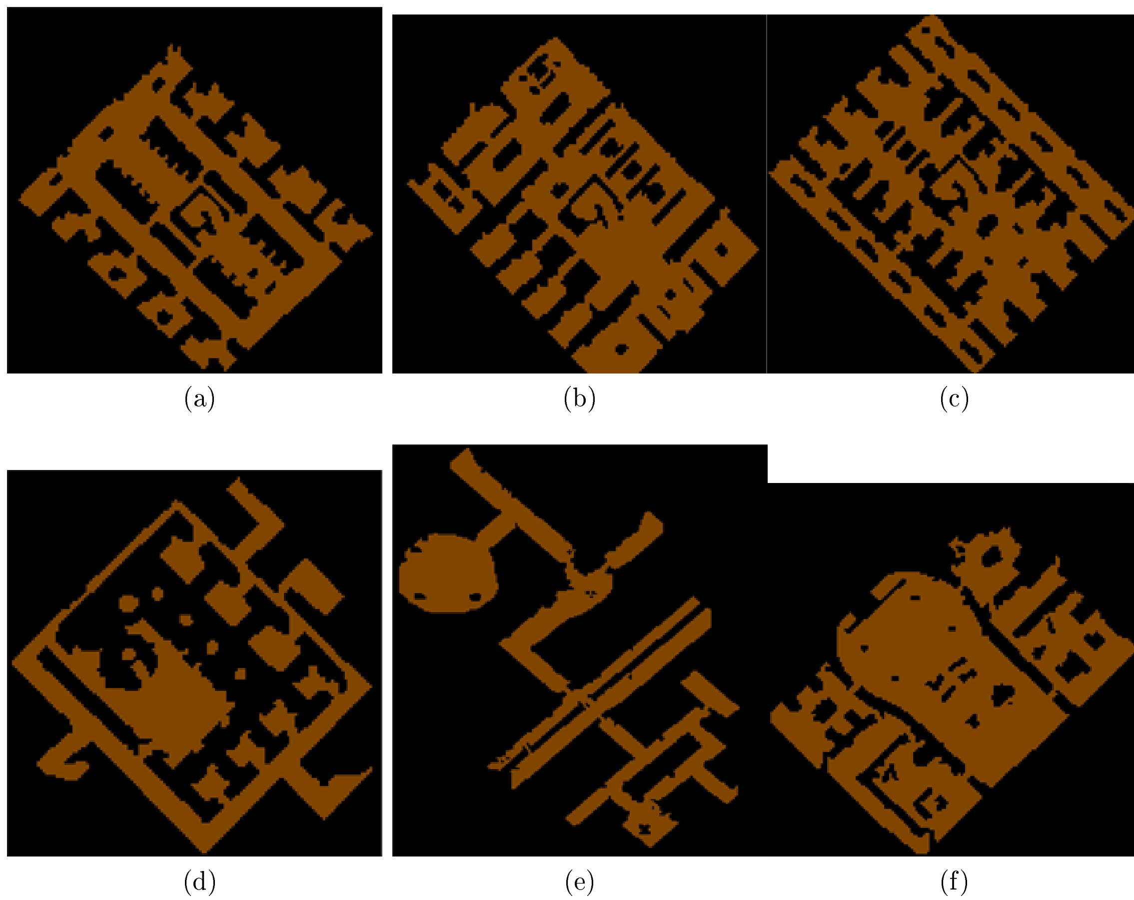
Stentz, A. (1995). The focussed D* algorithm for real-time replanning. In *International Joint Conference on Artificial Intelligence*, volume 14, pp. 1652--1659. Citeseer.

Xia, Y. & Prasanna, V. K. (2008). Parallel exact inference on the cell broadband engine processor. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1--12, Piscataway, NJ, USA. IEEE Press.

Apêndice A

Mapas utilizados na avaliação experimental

Os mapas são mostrados nas Figuras A.1 e A.2. As buscas 1 e 2 foram realizadas no mapa A.1(a), a busca 3 no mapa A.1(b), a 4 no mapa A.1(c), a 5 foi feita no mapa A.1(d), a 6 no mapa A.1(e) e a busca 7 no mapa A.1(f). A busca 7 foi realizada no mapa A.2(a), a busca 8 no mapa A.2(b), a 9 foi feita no mapa A.2(c), a busca 10 no mapa A.2(d), a 11 foi realizada no mapa A.2(e), a 12 no mapa A.2(f) e a busca 14 no mapa A.2(g).



(a) (b) (c)

(d) (e) (f)

Figura A.1: Mapas utilizados nos experimentos: buscas 1 a 7.

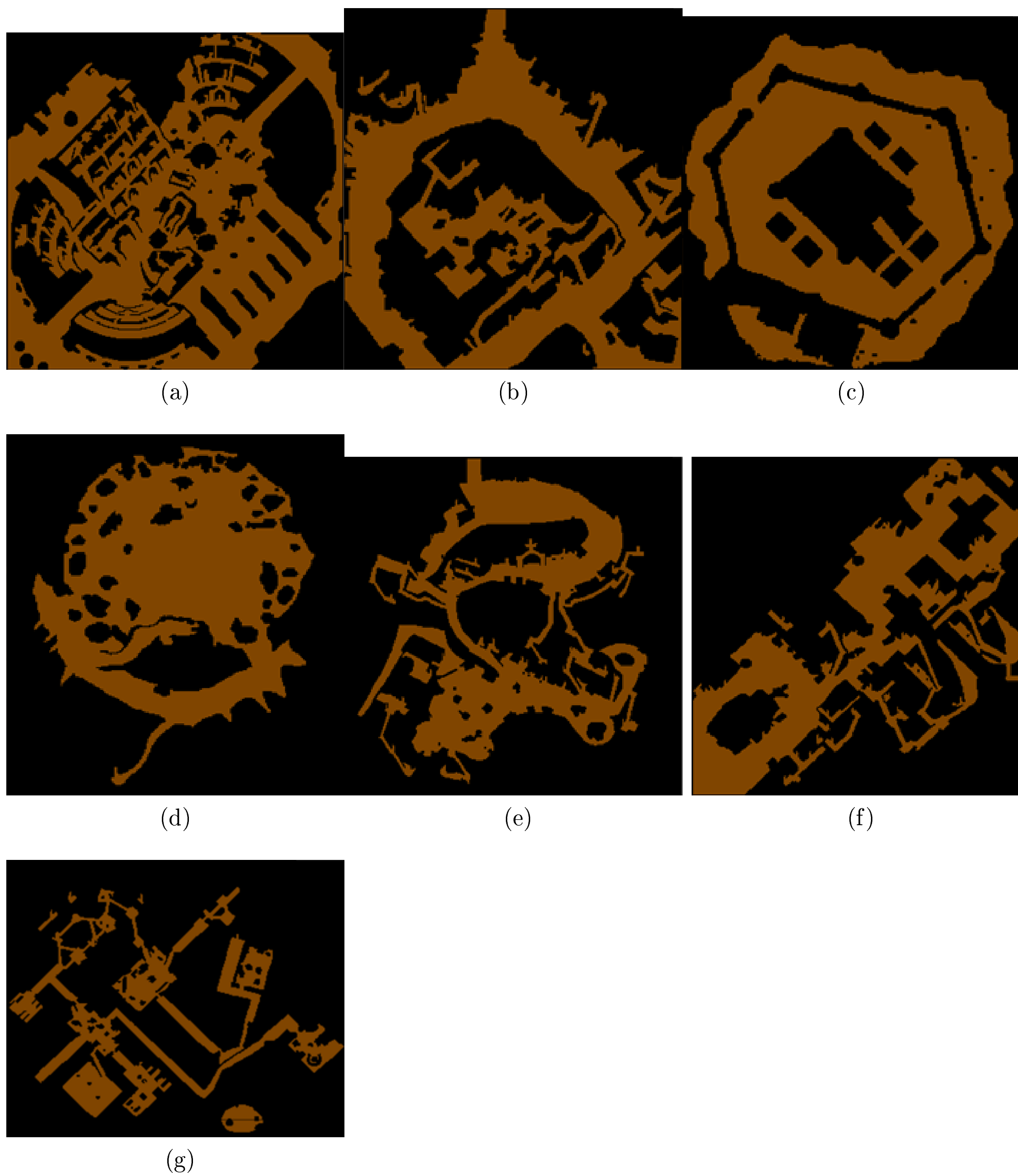

(a) (b) (c) (d) (e) (f) (g)

Figura A.2: Mapas utilizados nos experimentos: buscas 8 a 14.