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

APLICAÇÃO DE PLANEJAMENTO EM JOGOS DE ESTRATÉGIA

Bruno Nepomuceno Luiz

Setembro de 2008

Bruno Nepomuceno Luiz

Aplicação de Planejamento em Jogos de Estratégia

Faculdade de Computação
Universidade Federal de Uberlândia
2008

Bruno Nepomuceno Luiz

Aplicação de Planejamento em Jogos de Estratégia

Submetido em atendimento parcial dos requisitos para a obtenção do grau de Mestre em Ciência da Computação junto à Universidade Federal de Uberlândia, Minas Gerais.

© Todos os direitos reservados

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada **"Aplicação de Planejamento em Jogos de Estratégia"** por **Bruno Nepomuceno Luiz** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 08 de Setembro de 2008

Orientador: ______________________________________

Prof. Dr. Carlos Roberto Lopes
Universidade Federal de Uberlândia UFU/MG

Banca Examinadora: ______________________________________

Prof. Dr. Luiz Chaimowicz
Universidade Federal de Minas Gerais UFMG/MG

____

Profa. Dra. Rita Maria da Silva Julia Universidade
Federal de Uberlândia UFU/MG

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Data: Setembro de 2008

Autor:	**Bruno Nepmouceno Luiz**
Título:	**Aplicação de Planejamento em Jogos de Estratégia**
Faculdade:	**Faculdade de Computação**
Grau:	**Mestre** Convocação**: Setembro** Ano**: 2008**

A Universidade Federal de Uberlândia possui permissão para distribuir e ter cópias desse documento para propósitos exclusivamente acadêmicos, desde que a autoria seja devidamente divulgada.

____

Autor

O AUTOR RESERVA OS DIREITOS DE PUBLICAÇÃO E ESSE OCUMENTO NÃO PODE SER IMPRESSO OU REPRODUZIDO DE OUTRA FORMA, SEJA NA TOTALIDADE OU EM PARTES SEM A PERMISSÃO ESCRITA DO AUTOR.

Dedico este trabalho a meus amados pais José Luiz dos Anjos e Consuelo Nepomuceno

Agradecimentos

A jornada foi longa e, pelo caminho, muitas pessoas contribuíram para que este trabalho fosse realizado. Correções ortográficas, sugestões, críticas, apoio moral, edição de imagens, companhia nas madrugadas regadas a coca-cola, sanduíches e muito código em C#; muitos foram os tipos de contribuições.

Na certeza de que, provavelmente me esquecerei de alguém, desde já, me desculpo e deixo registrado: acima de qualquer coisa, o sentimento de gratidão e o reconhecimento de seu auxílio estará sempre gravado em algum lugar de minha memória, mesmo que não esteja explicitamente presente nestas palavras.

Ainda assim, agradeço:

primeiramente, a meu pai José Luiz e a minha mãe Consuelo, pessoas ímpares que jamais pouparam esforços para que eu tivesse sempre a melhor educação; mestres que, no dia a dia, ensinaram-me os mais belos e corretos princípios que hoje possuo; pai e mãe sempre presentes; fonte eterna de amor, carinho e atenção para com seus três filhos;

a Camila - minha namorada a quem tanto amo - não somente pelos mapas editados no *Photoshop*, mas por estar sempre ao meu lado quando precisei, por ter paciência nos momentos em que tive que me ausentar para trabalhar, estudar e produzir esta pesquisa, por sempre acreditar em mim e principalmente por me fazer tão feliz em todo esse tempo em que estivemos juntos;

a meu orientador, professor e doutor Carlos; não tenho palavras para descrever minha gratidão por todos os ensinamentos, desde os primeiros anos de minha graduação até os últimos momentos de meu Mestrado; por sua paciência quase infinita e por sempre acreditar em mim, serei eternamente grato;

a meus queridos irmãos Marcos e Lysa, pessoas sempre muito carinhosas e de enormes corações; pessoas que sempre que me viam abatido, tinham simples mas sinceras palavras de apoio e incentivo, cúmplices de uma bela vida proporcionada por nossos pais; obrigado por todos esses anos de companheirismo, fraternidade e amor;

a minha tia Terezinha, que sem hesitar, mesmo que de última hora, aceitou meu pedido de verificar a ortografia, gramática e normas técnicas para produção de trabalhos científicos, e o fez com tanta dedicação;

a meu tio Luis André, pela ajuda nas correções ortográficas e gramaticais do artigo enviado para o SBGames;

a meu amigo Augusto, companheiro de maratonas de programação, parceiro nos times *CodeSdruxulators* e *CodeSdruxulators2007*, uma pessoa muito dedicada, inteligente e que presenciou praticamente todo o processo de implementação dos algoritmos desenvolvidos e documentados nesta dissertação, ajudando sempre que possível;

a meu amigo Cássio, pelas inúmeras conversas sobre computação, metodologias de ensino, sistemas de avaliação, por nossas conversas sobre a vida; por estar sempre disposto a me ajudar, mesmo quando o assunto não era sua especialidade;

a Jeff Orkin, um pesquisador ocupadíssimo, mas que, prontamente, respondeu todos os meus emails sobre dúvidas relacionadas com a implementação do jogo F.E.A.R. e sobre o sistema de representação simbólica usado em seu trabalho.

Resumo

Com a indústria de jogos eletrônicos em constante crescimento e com um consumidor cada vez mais exigente, *designers* e programadores enfrentam, cada vez, maiores desafios para criar jogos realísticos. Nesse sentido, um tópico que vem sendo trabalhado bastante é a inteligência dos NPCs (*Non Player Characters*). Neste trabalho foi enfatizado o uso de técnicas de planejamento para controlar a Inteligência Artificial dos NPCs. A pesquisa documentada nesta dissertação foi guiada pela verificação de duas hipóteses: a viabilidade de utilização de um planejador baseado no algoritmo A* e em um modelo de representação de conhecimentos semelhante a do sistema STRIPS em jogos de estratégia; a possibilidade de uso desse planejador e de uma abordagem multiagente baseada em um sistema de leilões onde este último teria como função identificar o agente mais apto para cumprir uma determinada meta. O termo "jogos de estratégia" foi usado para não gerar confusão com relação aos jogos RTS tradicionais. O planejador e o sistema multiagente foram implementados e ínumeros testes realizados. Os resultados foram satisfatórios e as hipóteses comprovadas.

Palavras-chave: planejamento, jogos de estratégia, algoritmos de busca

Abstract

With a digital game industry in constant ascendant and a more demanding consumer, designers and programmers have faced even bigger challenges to create realistic games. In this way, people are working a lot in the NPCs (Non Player Characters) intelligence. In this research, it was emphasized the use of planning to control the Artificial Intelligence of the NPCs. The work documented here was guided by the verification of two hypotheses: the viability of using a planner based on the A* algorithm and on a knowledge representation model similar to STRIPS system in strategic games, and the possibility of using this planner in a Multiagent System approach based on an auction system used to identify agents more appropriated to fulfill a goal. The term "strategic games" was chosen to avoid misunderstandings about the traditional RTS games. The planner and the Multiagent System have been implemented and several tests were made. The results were satisfactory and the hypotheses proved.

Keywords: planning, strategic games, search algorithms

Sumário

Resumo vii
Abstract viii
Lista de Tabelas xi
Lista de Figuras xii
Lista de Siglas xiii
1. Introdução 1
  1.1 Objetivos 3
  1.2 Justificativa 4
  1.3 Hipóteses 5
2. Trabalhos Relacionados 6
  2.1 GOAP 6
  2.2 Planejamento HTN aplicado a jogos eletrônicos 8
  2.3 Planejamento Multiagente Baseado em Leilões e Reparação de Planos 9
  2.4 Planejamento para Produção de Recursos em Jogos do Tipo RTS 11
  2.5 Uso de Campos Potenciais em Jogos do Tipo RTS 12
3. Conceitos 14
  3.1 Jogos RTS 14
  3.2 Algoritmos de Busca 15
    3.2.1 Busca em largura 16
    3.2.2 Busca em profundidade 17
    3.2.3 Dijkstra 18
    3.2.4 A* 20
  3.3 Planejamento 24
    3.3.1 Linguagem de representação 25
    3.3.2 Planejamento por busca 30
    3.3.3 POP (Planejamento de ordem-parcial) 32
    3.3.4 Planejamento usando grafos 32
    3.3.5 Planejamento usando Lógica Proposiconal 35
  3.4 Sistemas Multiagentes 36
    3.4.1 Agente 36
    3.4.2 Sistemas Multiagente 38

4. Project Hoshimi ... 40
5. *AStarPlanner* ... 45
5.1 O Algoritmo ... 48
5.2 A Implementação e Arquitetura dos agentes ... 52
5.2.1 Implementação do planejador ... 53
5.3 Testes Realizados ... 54
5.3.1 Teste de desempenho ... 54
5.3.2 Testes de Qualidade dos Planos Gerados ... 58
6. Multiagente *AStarPlanner* ... 61
6.1 O Algoritmo ... 61
6.2 *MultiAStarPlanner* no *Project Hoshimi* ... 63
6.3 Testes realizados ... 65
6.3.1 Testes com 3, 5 e 7 nanorobôs ... 67
6.3.2 Testes com 20 nanorobôs ... 71
6.3.3 Testes com pequenos turnos ... 73
7. Conclusão ... 75
7.1 Considerações Finais ... 75
7.2 Trabalhos Futuros ... 78
Referências ... 79
Anexo A: Mapas usados nos testes dos algoritmos *AStarPlanner* e *MultiAStarPlanner* ... 82

Lista de Tabelas

Tabela 3-1: Comparação entre STRIPS e ADL. 28

Tabela 5-1: Tabela comparativa de resultados dos times *CodeSdruxulators* e *CodeSdruxulators2007*. 56

Tabela 5-2: Tempo médio de geração de planos. 57

Tabela 5-3: Testes de qualidade de planos gerados.. 58

Tabela 6-1: Símbolos e ações que os possuem na lista de efeitos. 64

Tabela 6-2: Testes *MultiAStarPlanner*/Simple*AStarPlanner* com rota otimizada............. 68

Tabela 6-3: Testes *MultiAStarPlanner*/Simple*AStarPlanner* com rota aleatória................ 70

Tabela 6-4: *MultiAStarPlanner*/SimpleStarPlanner com 20 nanorobôs c/ rota otimizada.. 71

Tabela 6-5: *MultiAStarPlanner*/SimpleStarPlanner com 20 nanorobôs c/ rota aleatória.... 72

Tabela 6-6: Testes com turnos de 33 milissegundos e rotas otimizadas. 73

Tabela 6-7: Testes com turnos de 33 milissegundos e rotas aleatórias. 74

Lista de Figuras

Figura 3-1: Grafo direcionado(a) e grafo não direcionado (b). 16
Figura 3-2: Busca em largura. 17
Figura 3-3: Busca em profundidade. 18
Figura 3-4: Psedocódigo do algoritmo de Dijkstra. 19
Figura 3-5: Pseudo-código do algoritmo A*. 21
Figura 3-6: Exemplo A* - estado inicial. 22
Figura 3-7: Exemplo A* - sucessores de n. 23
Figura 3-8: Exemplo A* - caminho gerado. 24
Figura 3-9: Mundo dos blocos. 25
Figura 3-10: Representação do problema "ter e comer o bolo". 33
Figura 3-11: Planning Graph do problema "ter e comer o bolo". 34
Figura 3-12: Infra-estrutura básica de um Sistema Multiagente (WOOLDRIDGE, 2002, p106). 39
Figura 4-1: Pontos AZN, HP e de Navegação. 41
Figura 4-2: Densidade das células. 42
Figura 4-3: Fluxos sanguíneos (*bloodstreams*). 43
Figura 5-1: Pseudocódigo do *AStarPlanner.* 50
Figura 5-2: Pseudocódigo do Método TratarPrecondições. 51
Figura 5-3: Ordem para percorrer pontos do tipo HP. 56
Figura 6-1: Pseudocódigo do algoritmo *MultiAStarPlanner.* 62
Figura 6-2: Mapa A4 (à esquerda) e SC3 (à direita). 68
Figura 6-3: Rotas A e B. 69
Figura A-1: Mapa A1. 82
Figura A-2: Mapa A2. 83
Figura A-3: Mapa A3. 84
Figura A-4: Mapa A4. 85
Figura A-5: Mapa Mashroom D. 86
Figura A-6: Mapa SC3. 87
Figura A-7: Mapa SC4. 88

Lista de Siglas

ADL	*Action Description Language*
AIPS	*Artificial Intelligence Planning System*
F.E.A.R.	*First Encounter Assault Recon*
FPS	*First Person Shooter*
FSM	*Finite State Machines*
GOAP	*Goal Oriented Action Planning*
HP	*Hoshimi Point*
HTN	*Hierarchical Task-Network*
IA	Inteligência Artificial
MIT	*Massachusetts Institute of Technology*
NPC	*Non Player Character*
PDDL	*Planning Domain Definition Language*
RBS	*Rule Based Systems*
RPG	*Role Playing Game*
RTS	*Real Time Strategy*
SDK	*Software Developer Kit*
STRIPS	*STanford Research Institute Problem Solver*
UT	*Unreal Tournament*

1.Introdução

Segundo o NPD Group, a indústria de jogos eletrônicos cresce consideravelmente a cada ano (NPD, 2008). Em 2007, um único jogo – *Halo 3* – gerou, nas 24 horas posteriores ao seu lançamento, um faturamento de cerca de 170 milhões de dólares, apenas nos Estados Unidos (HALO, 2008).

Paralelamente ao crescimento em quantidade, pode-se notar um avanço na qualidade gráfica e realismo. No início, na época do Atari 2600, encontravam-se personagens representados por pequenos cubos - como no jogo *Aventure* (ADVENTURE, 1979); em jogos de corrida como Enduro (ENDURO, 1993), o máximo da interação entre os jogadores era a possibilidade de movimentar um modelo de carro, desprovido de grandes detalhes gráficos, de um lado para o outro e a inteligência dos oponentes era praticamente inexistente.

Hoje em dia, com a nova geração de vídeo games e placas gráficas, chegou-se a um nível de detalhes inimaginável para os jogadores de Atari 2600. Em jogos como *Gears of War* (GEARS, 2006), os modelos dos personagens, assim como seu sistema de movimentos, é praticamente perfeito. Os atuais jogos de corrida são considerados simuladores, pela grande quantidade de recursos que oferecem e pela fidelidade à realidade. Em *Forza Motorsport 2* (FORZA, 2007), pode-se alterar, por exemplo, calibragens dos pneus, suspensão, sistema de frenagem, curva de potência do motor, diferencial. O sistema de física trabalha detalhes como desgaste e temperatura de pneus, traçado e relevo das pistas e aumento de velocidade devido à ausência de ar frontal quando um carro se encontra imediatamente atrás de outro.

Outro fator que vem sendo trabalhado bastante é a inteligência dos NPCs (*Non Player Characters*). Um NPC é um personagem que não é controlado pelo jogador, podendo ser, por exemplo, um monstro, um carro de corrida, um lutador ou um atirador de elite. Em *Doom II* (DOOM, 1994), os monstros apareciam sempre nas mesmas posições e apresentavam sempre o mesmo comportamento e métodos de ataque. Hoje em dia, em jogos como *F.E.A.R.* (F.E.A.R, 2005), NPC’s são conduzidos por metas e, dependendo da situação e ambiente, atuam de uma forma ou outra. Em Bioshock (BIOSHOCK, 2007), o

jogador pode atear fogo em seus oponentes. Estes, quando em chamas, procuram por água para tentar sobreviver.

Essa evolução não aconteceu somente pelo desejo dos desenvolvedores de jogos em criar algo mais realista, mas é resultado, também, de um jogador muito mais exigente. Antigamente, os jogadores contentavam-se em atirar em quadradinhos no espaço, hoje em dia, querem espaçonaves bem detalhadas, com diversos tipos de armas, escudos, sistemas de propulsão diferenciados. As pessoas não se importavam com o fato do monstro se encontrar sempre na mesma posição. Ultimamente, desejam que os NPC's trabalhem em equipe e apresentem comportamentos inteligentes, desenvolvendo questões como ataque coordenado, e que decidam pela retirada de um combate quando em uma situação de desvantagem.

Com isso, as antigas técnicas para determinar o comportamento de NPCs baseadas em FSM (*Finite State Machines*) e RBS (*Rule Based Systems*) começam a ser substituídas. Ao invés de percorrer um grafo predefinido de transições, um NPC controlado por um algoritmo de planejamento procura uma seqüência de ações que satisfaça uma meta (ORKIN, 2005).

O uso de técnicas de planejamento torna um NPC mais flexível para gerenciar situações inesperadas. Além disso, a utilização de metas e ações cria uma estrutura que facilita a manutenção, compartilhamento e reuso de código. Este é o foco principal do trabalho desenvolvido e documentado nesta dissertação: criar algoritmos de planejamento que possam ser aplicados em jogos eletrônicos e que façam com que personagens não controlados pelo jogador apresentem comportamentos inteligentes.

É importante ressaltar que o domínio de jogos eletrônicos é um excelente ambiente de testes para pesquisas na área de Inteligência Artificial. Hawes (2003) descreve duas características de mundos virtuais presentes nos jogos que justificam tal afirmação. A primeira é que o mundo de um jogo eletrônico é interativo, de tal forma que diversos personagens (controlados ou não pelo jogador) estão sempre interagindo e modificando o mesmo ambiente. Isto que dizer que, do ponto de vista do personagem, o estado do mundo muda não somente pela realização de suas ações, mas, também, pelas ações dos outros personagens. Além disso, o mundo deve responder, imediatamente, a uma ação de um personagem, ou seja, o tempo é totalmente relevante. Neste sentido, pode ser considerado um sistema de tempo-real.

Deve-se ainda considerar a complexidade e semelhança destes ambientes com o mundo real. Conforme mencionado anteriormente, os ambientes são tão complexos e fiéis à realidade que são até classificados como simuladores. Neste sentido, são ótimos para testes de algoritmos que tratam o gerenciamento de recursos, colaboração, planejamento e definição de rotas.

Por fim, deve-se deixar claro que, mesmo sendo os jogos eletrônicos ótimos ambientes para testes, muitas vezes não foram utilizados devido a limitações de *hardware*. Isto porque algoritmos usados para implementar as principais técnicas de IA (Inteligência Artificial), são, em geral, computacionalmente caros, ou seja, demandam muito poder de processamento. Contudo, com a evolução do *hardware* responsável por executar o código dos jogos, pode-se notar, cada vez mais, a exploração de técnicas de IA para tornar os jogos mais realistas.

1.1 Objetivos

O principal objetivo deste trabalho é criar algoritmos de planejamento e verificar a sua viabilidade em jogos eletrônicos envolvendo formulação de estratégias e táticas. Os jogos eletrônicos que apresentam estas características são classificados como jogos de estratégia em tempo real ou jogos RTS (*Real-Time Strategy Games*). Desta forma, este trabalho usa os jogos de estratégia como ambientes em que os algoritmos desenvolvidos possam ser inseridos e gerar os resultados desejados.

O termo "jogo RTS" será evitado em todo o trabalho para não causar confusões com os tradicionais jogos RTS como *Warcraft II* ou *Age of Empires*. De fato, o ambiente utilizado para a realização dos testes, corresponde a um jogo de estratégia não baseado em turnos, ou seja, planejamento e execução acontecem em tempo real. Assim, a melhor categoria para enquadrá-lo seria jogos RTS. Contudo, ele não apresenta alguns elementos comuns aos jogos RTS tradicionais como, por exemplo, um sistema de evolução tecnológica. Assim, doravante, o termo usado será "jogo de estratégia", evitando-se, inclusive o uso da expressão "jogo eletrônico" de estratégia. Esse ambiente de testes será apresentado no Capítulo 4.

O primeiro objetivo estipulado foi é a criação de um algoritmo de planejamento seguindo uma abordagem de busca. Como base para a criação deste algoritmo, propõe-se o uso do A*.

Além disso, outro objetivo é estender o uso do algoritmo de planejamento para um ambiente multiagente. Pensou-se na criação de um sistema de leilões para identificação dos melhores agentes para o cumprimento de metas. Isto é, dado uma meta, seria criado um sistema em que agentes fariam ofertas de tal forma que a oferta de menor custo definiria o agente responsável pela realização do plano.

1.2 Justificativa

A principal motivação para o desenvolvimento desse trabalho foi uma frase em um artigo de Jeff Orkin (2004) dizendo que a técnica que ele utilizou, provavelmente seria aplicável em jogos de outros tipos. "É fácil extrapolar como o uso dessas mesmas técnicas poderiam ser aplicadas a outros tipos de jogos, como, por exemplo, jogos do tipo RTS."

Em seu trabalho, ele também usou A* para gerar planos, contudo, em jogos do tipo FPS. Pretende-se aqui verificar se realmente é possível usar as técnicas apresentadas por Orkin em jogos de estratégia, uma vez que, neste tipo de jogo, imagina-se que os planos são bem maiores do que os planos gerados em jogos do tipo FPS, o que pode comprometer o desempenho.

Além disso, em um outro artigo, Orkin (2005) comenta, na seção trabalhos futuros, que gostaria de usar técnicas semelhantes para gerar planos para esquadrões de NPCs, ou seja, nada nesse sentido havia sido experimentado ainda. Desta forma, pensou-se em tentar estender o primeiro algoritmo criado para usá-lo em um sistema multiagente. Como base para definição dos agentes responsáveis pela execução de metas, optou-se por um sistema de leilões, haja vista que essa abordagem também nunca havia sido testada em um algoritmo de planejamento aplicado em jogos de estratégia.

Por fim, apesar de ser aplicado em um domínio específico, haveria uma preocupação em não restringir demais o algoritmo, de tal forma que ele pudesse ser aplicado em outras áreas.

1.3 Hipóteses

Em meio a esse cenário, consideram-se duas hipóteses como guia para o desenvolvimento do trabalho descrito nesta dissertação. A primeira hipótese está relacionada com a aplicação de um algoritmo de busca para a definição de planos em um jogo de estratégia de forma eficiente e eficaz. O algoritmo em questão é o A* (pronuncia-se A estrela ou A *star*). Essa idéia foi usada pelo pesquisador Jeff Orkin em (2004, 2005), mas em um outro domínio, ou seja, jogos eletrônicos do tipo FPS (*First Person Shooter*). À primeira vista, parece ser o mesmo domínio – jogos eletrônicos – todavia, essas duas categorias (RTS e FPS) apresentam características bem distintas. Em um jogo do tipo FPS, não existem tantas possibilidades para os NPCs. A idéia básica é eliminar o personagem do jogador. Em um jogo de estratégia, o importante é traçar a melhor estratégia para atingir objetivos.

Como segunda hipótese, procura-se verificar se é possível que um agente dotado de um planejador baseado no A* pode fazer parte de um sistema multiagente. A característica principal deste sistema consiste na identificação, por meio de um sistema de leilão, dos agentes mais aptos para realização de metas propostas.

A idéia de utilização do sistema de leilão foi inspirada no trabalho de Krogt (2005) em outro domínio. Na literatura, ninguém havia tentado usar esse tipo de sistema em jogos de estratégia. Esta é uma das propostas desta pesquisa.

Os demais capítulos desta dissertação estão estruturados da seguinte forma: o Capítulo 2 apresenta um conjunto de trabalhos relacionados com o tema proposto; o Capítulo 3 descreve os principais conceitos relacionados com o trabalho realizado; o Capítulo 4 apresenta o ambiente usado para testar a qualidade e desempenho dos algoritmos propostos nos capítulos 5 e 6; o Capítulo 5 apresenta o algoritmo de planejamento *AStarPlanner*; o Capítulo 6 descreve o *MultiAStarPlanner* e os testes com ele realizados; o Capítulo 7 apresenta a conclusão, considerações finais e sugestões para trabalhos futuros.

2. Trabalhos Relacionados

Conforme citado, com a evolução do *hardware* dos vídeo games e dos computadores, técnicas de IA começaram a ser aplicadas nos jogos mais recentes. Este capítulo descreve o trabalho de dois pesquisadores – Jeff Orkin (2004, 2005) e Hector Munoz (2004, 2005) - que utilizaram IA em jogos de computador e também fatores de uma pesquisa sobre Replanejamento. Apesar de o tema principal deste último trabalho ser a questão do Replanejamento, o ponto importante e relacionado com essa dissertação é a utilização de um sistema de leilão em um ambiente multiagente.

Além destes trabalhos, destaca-se também a pesquisa de Chan (CHAN et. al, 2007) sobre a geração de planos para produção de recursos em jogos do tipo RTS e de Hagelback e Johansson sobre o uso campos potenciais em jogos do tipo RTS.

2.1 GOAP

Segundo Orkin (2005), atualmente os desenvolvedores de jogos estão enfrentando novos desafios relacionados com escala de desenvolvimento. As equipes cresceram drasticamente, o escopo dos jogos vem se tornando cada vez maior, o tempo de produção, cada vez menor, e tem-se uma tendência por jogos com finais cada vez mais abertos e variados. Nesse sentido, cada equipe responsável por uma área diferente da criação de um jogo, deve ter em mente o objetivo de criar códigos reutilizáveis. A equipe responsável pela IA não pode ser diferente.

Nesse contexto, as máquinas de estados finitos, apesar de largamente utilizadas, não são a melhor solução em termos de reutilização de código para modelar o comportamento de NPC's. Isso porque cada máquina de estado é específica para um determinado comportamento, de tal forma que, a incorporação ou exclusão de um novo estado pode alterar completamente a estrutura da máquina como um todo.

A opção de Orkin foi usar o formalismo de um sistema GOAP (*Goal Oriented Action Planning*), regressivo e de tempo-real. Esse tipo de sistema impõe uma estrutura modular de arquitetura que facilita o compartilhamento de comportamento entre NPC's.

Nesse tipo de sistema, NPC's são modelados em forma de agentes que trabalham com metas e um conjunto de ações. Não existe nenhum tipo de mapeamento explícito entre metas e ações. Cada agente tenta satisfazer as metas mais relevantes em um dado momento. Para cada meta, um algoritmo de planejamento gera a melhor seqüência de ações para atingi-la. É importante ressaltar, que o planejamento acontece em tempo real, o que favorece o gerenciamento de situações inesperadas.

O algoritmo de planejamento segue uma abordagem baseada em uma busca regressiva. A idéia básica foi transpor o A*, comumente usado em jogos em problemas do tipo *pathfinding* (identificar a melhor rota entre um ponto origem e outro ponto destino), para o domínio de ações e metas. Nesse caso, tem-se como origem o estado corrente do mundo e o destino, a meta a ser atingida. Desta forma, cada nó seria representado por um estado do mundo resultante da aplicação de uma ação.

O estado do mundo é descrito em uma estrutura de dados implementada como um vetor de símbolos estruturado na forma de par-valor. Os símbolos incluem propriedades básicas de um NPC de um jogo do tipo FPS, como, por exemplo, posição, indicação se a arma está carregada, se o alvo (um outro NPC ou personagem do jogador) está apontando sua arma para NPC e se o alvo está morto. É importantes ressaltar que toda informação usada para representar um estado do mundo, usado durante o processo de planejamento, se refere ao próprio NPC. Para melhor compreensão desta questão da centralização da informação no NPC, um exemplo será descrito.

Supondo que um NPC receba a tarefa de eliminar o personagem do PC, a representação da meta, relevante para tal tarefa, será apenas "alvo está morto". Desta forma, o que o algoritmo de planejamento deve fazer é apenas encontrar um conjunto de ações que, no final, atribuam o valor "verdadeiro" para o símbolo "alvo está morto". Ou seja, não é armazenada a informação de qual alvo deve morrer, pois isto é irrelevante ao processo de planejamento. O algoritmo, como é baseado no A*, trabalha com um grafo, tal que cada nó armazena informações sobre um estado do mundo. Essa informação sobre o alvo que deve ser eliminado fica armazenada em outra estrutura de dados que pode ser acessada durante o processo, mas não fará parte do "nó". Isto é importante para não se ter nós com informações em excesso, uma vez que, quanto mais informações tem um nó, mais demorada é a comparação entre dois nós.

Para a representação das ações, foi usada uma linguagem semelhante à STRIPS (FIKES e NILSSON, 1971), que será descrita na seção 3.3.1.1, com as seguintes

alterações: incorporação de um custo associado à execução de uma ação; não utilização de listas de símbolos que devem ser inseridos e lista de símbolos que devem ser excluídos do estado do mundo corrente. Ao invés disso, usou-se vetores com elementos do tipo par-valor para precondições e efeitos; procedimentos para checagem de precondições de informações não presentes no estado do mundo e procedimentos para realização de efeitos não relacionados com o processo de planejamento, mas sim com a execução da ação. Por exemplo, quando uma ação *Attack* fizer parte de um plano, no momento da execução do plano, uma animação deve ser exibida.

Orkin (2005) descreve excelentes resultados de seu trabalho, o que pode ser comprovado pelo sucesso do jogo *F.E.A.R* (F.E.A.R, 2005). Este jogo foi produzido pela *Monolith*, empresa em que Orkin trabalha e toda a IA do jogo é controlada pelas técnicas e algoritmos criados pelo pesquisador.

A idéia de Orkin de usar o algoritmo A* como algoritmo de planejamento e o sistema de representação simbólica do estado do mundo e das ações foi utilizada no presente trabalho. Orkin (2005) sugere que a arquitetura provavelmente funcionaria em outros tipos de jogos, uma vez que foi testado apenas em um jogo do tipo FPS.

2.2 Planejamento HTN aplicado a jogos eletrônicos

GOAP é um novo e promissor paradigma para codificação do módulo que controla a IA de jogos eletrônicos MUNOZ-AVILA apud ORKIN (2005). A linguagem de representação de ações e metas deste paradigma é baseado, conforme citado, no sistema STRIPS. Planejamento HTN (*Hierarchical Task-Network*) é uma outra forma de planejamento que trabalha com tarefas (*tasks*) de alto nível, ao invés de ações (EROL *et al.*, 1994) de tal forma que, uma tarefa de alto nível é decomposta em outras tarefas mais simples, até que, eventualmente, todas as tarefas sejam decompostas em ações.

HTN possui duas vantagens sobre a linguagem STRIPS. Primeiramente, porque HTN é, comprovadamente, mais expressiva que STRIPS, ou seja, existem problemas que podem ser expressos em HTN, mas não em STRIPS. A segunda vantagem é que, vários autores afirmam que HTN codifica, de forma natural, o conhecimento estratégico (MUÑOZ-AVILA, 2005).

O trabalho de Muñoz utiliza o planejamento hierárquico para codificar estratégias que, um ou mais NPC's, devem executar para a realização de tarefas no jogo do tipo FPS da *Epic Games* conhecido como *Unreal Tournament.* Neste contexto, NPC's são chamados de *bots.*

Muñoz (2005) apresenta um estudo de caso que demonstra a viabilidade de seu trabalho em jogos do tipo FPS. O estudo é feito sobre uma modalidade do jogo *Unreal Tournament* chamado "Dominação". Nessa modalidade, membros de uma equipe devem dominar certas áreas de um mapa. Quando um membro de uma equipe atinge uma destas áreas, diz-se que tal área foi "dominada" pelo time em questão. Nesse caso, o time ganha um ponto a cada cinco segundos que mantém o domínio da área. O jogo termina quando um dos times atingir uma quantidade mínima de pontos estipulados para a partida.

O servidor do jogo UT (*Unreal Tournament*) provê informações sobre o mundo do jogo e controla toda a dinâmica deste mundo bem como interação entre *bots* e personagens do jogador. Um programa cliente é responsável por decidir quais os comandos serão usados para controlar o comportamento de um *bot* e repassá-los para o servidor. O programa cliente usado no estudo de caso de Muñoz foi o *Javabot* (MUÑOZ-AVILA e FISHER, 2004).

Durante o estudo de caso, o time de *bots* que tinha seus planos gerados por um algoritmo HTN (foi usado o algoritmo SHOP) enfrentou dois outros times: o time padrão composto por *bots* originais presentes no cliente *Javabot* e um time aprimorado, que corresponde ao time padrão com pequenos aprimoramentos no sistema de navegação e táticas de dominação. Estes aprimoramentos foram também usados no time HTN.

Em praticamente todos os testes, o time de robôs controlados pelo algoritmo de planejamento hierárquico apresentou melhores resultados. O trabalho de Muñoz mostra ainda uma segunda iniciativa bem sucedida de utilização de técnicas de planejamento para o controle do comportamento de NPC's em um atual jogo eletrônico.

2.3 Planejamento Multiagente Baseado em Leilões e Reparação de Planos

Krogt (2005) propõe a utilização de três conceitos importantes para resolver um problema de planejamento: planejamento multiagente, leilões e reparação de planos. De

acordo com o autor, grande parte das interessantes aplicações de planejamento é composta por aquelas em que o processo de planejamento acontece através da colaboração entre agentes.

O trabalho de Krogt baseia-se em um grupo de aplicações, por meio das quais agentes comunicam-se entre si, colaboram para a resolução de uma meta, mas precisam manter certa privacidade com relação a seu conhecimento interno. Ou seja, um agente não tem acesso a todo o conhecimento de um outro agente. Considera também que um agente pode não conseguir cumprir uma meta sozinho, visto que seu conjunto de ações pode não permitir a realização de parte desta meta. Por exemplo, quando se tem um problema em que é necessário transportar um pacote de um determinado ponto de uma cidade para um outro determinado ponto de uma outra cidade, e se tem agentes com campos limitados de atuação, pode-se ter uma situação em que nenhum destes agentes é capaz de atingir, sozinho, a meta. Ou seja, supondo que um agente seja capaz de realizar o transporte apenas na área da cidade de origem do pacote, que um outro agente seja capaz de transportar pacotes somente na área da cidade de destino, e que, um terceiro agente realize o transporte de pacotes na área entre a cidade de origem e destino, isoladamente, eles não conseguem movimentar o pacote da origem até o destino. Entretanto, se trabalharem de forma colaborativa, a meta pode ser atingida.

Para preservar a privacidade de cada agente, mas mesmo assim lidar com situações em que um agente sozinho não pode cumprir uma meta, um sistema de leilão foi proposto. Contudo, para que possam colaborar, os agentes devem conhecer as ações dos outros agentes. Um agente A não precisa saber "como" outro agente B realizará tal ação, mas A deve saber que B é capaz de realizá-la. O que cada agente pode fazer é modelado como um serviço.

Um serviço é representado da mesma forma que uma ação. Entretanto, para distingui-lo de uma ação regular, ele é chamado "ação externa". Assim, um agente possui um conjunto de ações e um conjunto de ações externas. Essas ações externas são os serviços que os outros agentes são capazes de realizar.

Nesse sentido, após um plano ter sido gerado por um agente A, quando existir uma ação externa nesse plano, um leilão é proposto para descobrir qual agente fornecerá tal serviço com um menor custo. Desta forma, uma espécie de reparação do plano será feita, uma vez que inicialmente estava destinado ao agente A realizar a ação externa. No final, o plano do agente A é composto apenas pelo conjunto de ações que ele realmente realizará.

Essa idéia de leilões foi utilizada no trabalho descrito nesta dissertação. Diferente da abordagem apresentada por Krogt, neste trabalho é assumido que cada agente é capaz de sozinho, realizar metas. Isso não quer dizer que dois agentes não possam colaborar para resolver a mesma meta, mas sim, que um não necessita do outro para resolvê-la. Caso aconteça a colaboração, o benefício seria uma redução no tempo de realização da meta.

2.4 Planejamento para Produção de Recursos em Jogos do Tipo RTS

Em jogos do tipo RTS, existem dois objetivos centrais: produção de recursos e participação em batalhas táticas. O processo de produção de recursos envolve produzir ou coletar vários tipos de matérias-primas (tais como, ouro e madeira), construções, unidades civis e militares e visa aprimorar a economia e o poder bélico de um raça ou civilização. Em batalhas táticas, o jogador utiliza unidades militares para ganhar territórios ou derrotar um adversário.

O trabalho de Chan (CHAN et. al, 2007) tem como foco o uso técnicas de planejamento para controlar a produção de recursos. De uma forma mais específica, procurou-se desenvolver um mecanismo de seleção de ações que fosse capaz de resolver qualquer meta atingível do tipo produção de recursos de uma forma rápida.

O planejador foi desenvolvido usando uma união do algoritmo MEA(*Means-Ends Analysis*) com um sistema escalonador. Dado o estado inicial e a meta de produção de recursos, o planejador determina um plano seqüencial que resolve a meta usando o menor número possível de ações e recursos. A partir deste plano seqüencial, o escalonador altera a ordem das ações procurando identificar conjuntos de ações que possam ser executados em paralelo. Assim, um plano gerado para um agente, pode ser resolvido, de forma cooperativa, por dois ou três agentes, diminuindo o tempo total para sua realização.

Para a realização de experimentos, foi utilizado o jogo RTS chamado *Wargus* – uma versão de *Warcraft II* para sistemas que não suportam a *engine* original desenvolvida pela *Blizzard*, como, por exemplo, *Linux*. De acordo com Chan (CHAN et. al, 2007), o planejador desenvolvido foi capaz de realizar uma grande quantidade de objetivos em tempo real. O desempenho para atingir uma meta, em termos de ciclos de jogo, foi igual ou

melhor do que experientes jogadores humanos e, significativamente melhor do que os planejadores de propósito geral testados no ambiente *Wargus*.

Na abordagem desenvolvida no jogo Wargus, tem-se que é possível gerenciar uma grande quantidade de complexas metas de produção de recursos através de algoritmos de planejamento e que os resultados produzidos são comparáveis aos resultados de jogadores humanos especialistas no jogo.

2.5 Uso de Campos Potenciais em Jogos do Tipo RTS

Campos potenciais (do inglês, *potential fields*) é uma técnica originada na robótica para controlar a navegação de robôs em ambientes dinâmicos. Algumas tentativas foram feitas no sentido de utilizar a técnica na área de jogos eletrônicos , contudo elas enfrentaram problemas de desempenho e altos custos de implementação. Todavia, recentemente, Hagelback e Johansson (HAGELGBACK e JOHANSSON, XX) consiguiram implementar um sistema multiagente baseado em campos potenciais que apresentou bons resultados em jogos do tipo RTS.

A técnica de campos potenciais foi proposta por Ossama Khatib em 1985. Em seu trabalho ele apresenta um elemento denominado “manipulador” sendo movido em campo de forças. A posição a ser atingida é uma força de atração e os obstáculos são forças que repelem o manipulador. Muitos estudos a respeito de navegação espacial e desvio de obstáculos foram feitos posteriormente.

Hagelback e Johansson realizaram testes no ORTS (*Open Real Time Strategy*) - uma *engine* para jogos do tipo RTS criada com o propósito de ser um ambiente de experimentação para pesquisas na área de inteligência artificial. Os jogos interpretados pela ORTS se assemelham bastante com tradicionais jogos de estratégia comerciais como *Warcraft II* ou *Age of Empires*.

Em seu trabalho com a *engine* ORTS, foram realizados testes com dois tipos de partidas: uma partida com apenas um jogador e com objetivo de coleta e transporte de recursos e uma outra com dois jogadores, tal que o objetivo é destruir as bases adversárias.

Cada NPC (ou trabalhador), base, construção ou elemento do ambiente possui cargas que geram campos potenciais ao seu redor. A esses campos são atribuídos pesos e

quando seus valores são somados, formam um campo potencial total que é utilizado como referencia para a navegação dos NPC’s.

Existem ainda campos potenciais que funcionam como forças de atração. Em uma partida do tipo dois jogadores, uma base inimiga tem cargas que atrai NPC’s do outro time. Em uma partida do tipo um jogador, os recursos a serem coletados possuem cargas que atraem os NPC’s.

O comportamento de cada trabalhador é controlado por uma simples máquina de estado finito. Nenhum controle central ou controlador explícito é necessário, uma vez que a coordenação é gerada através do uso do sistema de cargas.

A abordagem apresentou bons resultados. Em 2008, por exemplo, o sistema de Hagelback e Johansson foi o campeão de uma competição organizada pela Universidade de Alberta (e que utilizou o ORTS), vencendo 98% das partidas que disputou.

3. Conceitos

Neste capítulo são apresentados os principais conceitos utilizados no trabalho desenvolvido. São eles: jogos RTS, algoritmos de busca, planejamento e sistemas multiagentes.

3.1 Jogos RTS

Os jogos eletrônicos são classificados em várias categorias, dentre elas, destacam-se: FPS (*First Person Shoter*), RTS (*Real Time Strategy*), RPG (*Role Playing Game*), Esporte e Corrida. Mais especificamente falando, a categoria RTS descreve jogos em que os jogadores devem gerenciar recursos e traçar estratégias para realização de tarefas básicas que incluem aprimoramento da economia e poder bélico de uma civilização ou sociedade e combates visando expansão territorial.

A classificação RTS foi criada para diferenciar esse tipo de jogo dos tradicionais jogos de estratégia baseados em um sistema de turnos como a série *Civilization* (1991) ou a série *Heroes of Might and Magic* (HEROES, 1995). Neste sistema, a dinâmica do jogo é dividida em porções de tempo bem definidas chamadas turnos ou rodadas. Nesse caso, existe uma separação entre a fase em que se define uma estratégia e a fase de sua execução. Primeiramente planeja-se, depois, na fase de execução, o jogador verifica se sua estratégia funcionou ou não.

Em um jogo RTS, não existe essa noção de turnos. Enquanto um jogador está pensando, o outro pode estar tirando vantagem desta situação agindo, colocando sua estratégia em prática.

Além do fator *real-time* e da importância da concepção de estratégias para realização de objetivos, alguns componentes são comuns em jogos do tipo RTS: coleta e gerenciamento de recursos, criação de bases e construções, possibilidades de evoluções tecnológicas e combates de unidades militares.

Na pesquisa desenvolvida e documentada por esta dissertação, utilizou-se um ambiente de teste chamado *Project Hoshimi*. De acordo com o próprio site oficial do ambiente (PROJECT, 2007), ele é classificado como sendo um jogo. Isto faz sentido, uma vez que trabalha com um conceito de partidas, modos de jogo envolvendo um ou dois times e um sistema de pontuação. O fato de, a versão atual, não contar com a presença de um jogador humano, não o coloca em outra categoria. Na verdade, é totalmente possível alterar a *engine* do *Project Hoshimi* para possibilitar a experiência de jogo em que um time é controlado por um jogador humano. Caso isto seja feito, o melhor gênero para classificá-lo seria RTS. Apesar de não contar com todas as características de um RTS clássico como *Warcraft II* ou a série *Age of Empires*, tem-se no *Project Hoshimi* a noção de recursos (nanorobôs e enzimas AZN), a noção de combate (times podem atacar ou serem atacados por um time especial denominado "Time do Pierre"), não se trabalha com um sistema de turnos e o ambiente é extremamente dinâmico se assemelhando bastante ao ambiente de um clássico jogo do tipo RTS. Além disso, a elaboração de estratégias, bem como a ordem ideal de colocá-las em prática, define o vencedor de uma partida. De qualquer forma, para evitar confusões com os jogos clássicos do tipo RTS, o *Project Hoshimi* será classificado como "jogo de estratégia", conforme já mencionado na seção 1.1 desta dissertação.

3.2 Algoritmos de Busca

Dentre os vários tipos de algoritmos de busca, destacam-se os algoritmos baseados em grafos. Uma vez que grande parte do trabalho realizado utiliza um algoritmo baseado em grafos, a seguir, seguem algumas informações sobre essa classificação, bem como os principais tipos encontrados na literatura e relacionados com a proposta desta pesquisa.

Existem dois tipos de grafos: o grafo direcionado e o grafo não direcionado. Um grafo direcionado G é um par (V,E), tal que V é um conjunto finito e E é uma relação binária em V (CORMEN et al., 2001) . V é denominado o conjunto de vértices de G, e E é chamado de conjunto de arestas de G. Um grafo não direcionado G é um par (V,E) , tal que E constitui um conjunto de pares não ordenados de vértices e V um conjunto finito de vértices. Isto quer dizer que, em um grafo não ordenado, uma aresta é um conjunto e não uma relação binária. Por convenção, a notação (u, v), tal que u e v $\in$ V, é usada ao invés da

tradicional {u,v} utilizada para conjuntos. A Figura 3-1 (CORMEN et al., 2001, p87) exibe exemplos de um grafo direcionado e de um grafo não direcionado.

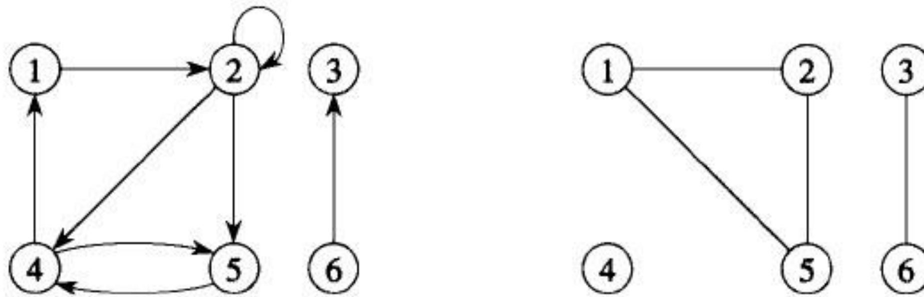

Figura 3-1: Grafo direcionado(a) e grafo não direcionado (b).

No grafo direcionado, percebe-se a presença de setas no final das arestas. Isto já não acontece com os grafos não direcionados.

Realizar uma busca em um grafo significa, sistematicamente, percorrer suas arestas, para visitar seus vértices. Técnicas de buscas em grafos constituem o coração do que é conhecido como *Graph Algorithms* (CORMEN et al., 2001).

Dentre os mais simples algoritmos usados para realizar buscas em um grafo tem-se: busca em largura e busca em profundidade. Entretanto, a partir de idéias simples como as utilizadas nestes dois algoritmos, são criados famosos e populares algoritmos com o algoritmo *Single-Source Shortest-Paths* de Djistra (CORMEN et al., 2001) e o A* (HART, NILSSON, e RAPHAEL, 1968) - elemento central deste trabalho.

3.2.1 Busca em largura

A busca em largura ou, em inglês, *Breadth-first search*, conforme citado na seção anterior, é um dos tipos de algoritmos mais usados para a realização de uma busca em um grafo.

Dado um grafo G = (V,E) e um vértice de origem s, no algoritmo da busca em largura, percorre-se, de forma sistemática, as arestas de V, visando descobrir quais os vértices são acessíveis a partir de s. Nesse caso, para cada vértice acessível v, o caminho que atravessa a árvore gerada a partir de s, corresponde ao menor caminho de s até v. Depois, para cada vértice v, ele descobre novamente a coleção de vértices acessíveis a partir dele, e o processo continua até que o vértice procurado seja encontrado (CORMEN et al., 2001).

Desta forma, se na Figura 3-2 o elemento procurado fosse aquele destacado na cor cinza, a ordem em que os vértices seriam visitados está destacada no interior de cada vértice. Primeiramente, seria visitado o vértice com rótulo 1, depois o de rótulo 2, e assim sucessivamente.

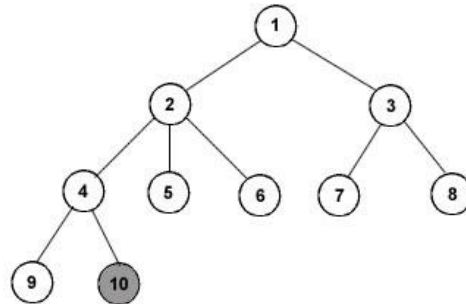

Figura 3-2: Busca em largura.

É importante ressaltar que não existe nenhuma heurística para guiar a busca. Logo, ele faz uma busca exaustiva em todo grafo, não considerando nada que o aproxime de sua meta.

3.2.2 Busca em profundidade

Na busca em profundidade, ou em inglês, *depth-first search*, dado um grafo G = (V,E), percorre-se, assim como na busca em largura, sistematicamente, as arestas de V, no entanto, a ordem em que os vértices são alcançados é diferente. Na busca em profundidade, as arestas são exploradas a partir do mais recente vértice *v* descoberto e que ainda possui arestas inexploradas. Após exploradas todas as arestas usando o critério especificado, caso o elemento buscado ainda não tenha sido descoberto, acontece um retrocesso (*backtracking*) e o procedimento continua a partir das arestas que foram deixadas de lado quando *v* foi encontrado. Se todas as arestas acessíveis a partir de *v* forem percorridas e o elemento não tenha sido encontrado, outro vértice é escolhido como origem, e assim é feito até que não existam mais arestas não percorridas ou até que o elemento buscado seja descoberto (CORMEN et al., 2001).

Usando o mesmo exemplo da seção 3.2.1, a ordem de busca pelo vértice destacado na cor cinza seria a especificada pelos rótulos da Figura 3-3.

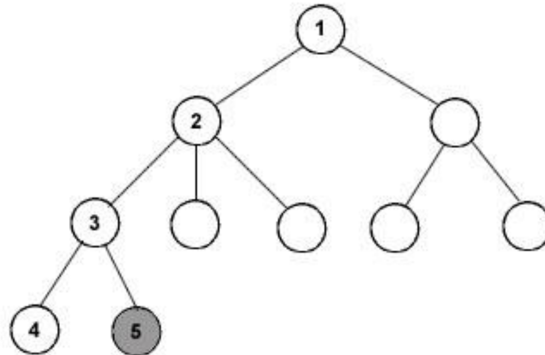

Figura 3-3: Busca em profundidade.

Nesse caso específico, o vértice foi encontrado mais rapidamente usando a busca em profundidade. Contudo, se por exemplo, o vértice buscado fosse aquele localizado a esquerda do vértice de rótulo 2, a busca em largura o encontraria muito mais rapidamente.

Da mesma forma que o algoritmo de busca em largura, a busca em profundidade não utiliza nenhum tipo de heurística para guiá-lo na escolha dos vértices a serem analisados.

3.2.3 Dijkstra

Uma forma particular de busca refere-se àquela em que se procura o menor caminho entre dois pontos. Em um problema do tipo "caminho mais curto", ou em inglês, *shortest path problem,* é fornecido um grafo direcionado em que as arestas são dotadas de pesos, um ponto de origem e um ponto de destino. A tarefa constitui em encontrar o caminho que tem como vértice inicial o ponto de origem, como vértice final, o ponto de destino e que possui o menor somatório dos custos das arestas (CORMEN at al., 2001).

O problema do caminho mais curto possui diversas variações, sendo que, uma delas é denonimada o problema do "caminho mais curto a partir de uma única origem" (*single-source shortest-paths problem*). Nessa variação, deseja-se encontrar o menor caminho para um determinado vértice origem *s* a partir de qualquer outro vértice *v,* sendo que *s* e *v* são vértices de um grafo direcionado e com pesos associados às arestas.

O algoritmo de Dijkstra foi criado para resolver problemas do tipo *single-source shortest-paths* em um grafo G=(V,E) direcionado e ponderado por uma função de peso. O algoritmo de Dijkstra mantém um conjunto S de vértices tal que, para cada vértice $v \in S$, tem-se calculado o caminho de menor custo a partir de *s*. O algoritmo repetidamente seleciona um vértice $u \in V - S$ e que possui a menor estimativa de custo do menor caminho e o insere em S. Por fim, aplica-se um procedimento chamado "relaxamento" a todas as arestas que tem o vértice *u* como origem (CORMEN et al., 2001).

Para cada vértice $v \in V$, é mantido um atributo d[*v*] que indica um valor limite para o peso do caminho de menor custo que tem como origem *s* e como destino *v*. d[*v*] é chamado de estimativa de caminho de menor custo. O processo de relaxamento de uma aresta (*u,v*) consiste em testar se é possível melhorar o caminho mais curto para *v*, a partir de *u*. Esse processo procura diminuir o valor de d[*v*].

Na Figura 3-4 é apresentado o pseudocódigo do algoritmo de Dijkstra.

```
// Inicializacoes
//    Distância inicial para todos os vertices com relação a origem
//      s é desconhecida. Logo, um valor extremamente alto é utilizado
Para cada vertex v em V
{
    dist[v] = infinito;
    previous[v] = null;
}
// Distância da origem até a origem é 0
dist[s] = 0;

// S = Conjunto de vértices cujos caminhos mais curtos a partir da
// origem s já foram calculados. Inicialmente é nulo
S = null;
// Q = V - S, tal que V é o conjunto de todos os vertices
//   Inicialmente, como S = null, Q = V
Q = GetNosNaoPertencentesS();

Enquanto (Q != null)
{
    u = GetNodeMenorDistancia(Q);
    Q.Remove(u);
    S.Add(u);
    Para cada vizinho v em u.GetVizinhos()
    {
       // Relaxamento da aresta (u,v)
       alt = dist[u] + GetDistancia(u, v);
       Se (alt < dist[v])
       {
           dist[v] = alt;
           previous[v] = u;
       }
     }
}
```

Figura 3-4: Psedocódigo do algoritmo de Dijkstra.

O algoritmo de Dijkstra pode ser considerado um caso especial A* - algoritmo usado como base do trabalho desenvolvido e documentado nesta dissertação – em que o valor da heurística é sempre 0.

3.2.4 A*

Muitos dos problemas de relevância científica estão relacionados a um problema geral de encontrar um caminho através de um grafo (HART, NILSSON, e RAPHAEL, 1968). Normalmente, são utilizadas duas abordagens para a solução deste tipo de problema: uma abordagem matemática e uma abordagem heurística.

Uma solução baseada em uma abordagem matemática geralmente trabalha com propriedades de grafos abstratos e algoritmos que examinam os nós de um grafo visando estabelecer caminhos de menor custo. Nesse tipo de solução, geralmente, não se tem uma preocupação com o custo computacional necessário para executar os algoritmos (HART, NILSSON, e RAPHAEL, 1968).

Em uma abordagem heurística, trabalha-se com conhecimento específico do domínio do problema para aumentar a eficiência computacional de soluções de problemas de buscas em grafos. No algoritmo A*, tem-se o uso de informação específica de domínio incorporada a uma abordagem matemática formal.

Em problemas de busca que utilizam grafos, um passo comum é a identificação do próximo nó que será testado. O A* utiliza uma função de avaliação $f(n)$ para escolher dentre os nós disponíveis, qual deve ser o próximo a ser utilizado. Essa função $f(n)$ é calculada somando-se $g(n)$ com $h(n)$, tal que, $g(n)$ é o custo atual do caminho gerado a partir do nó inicial s até o nó n e $h(n)$ é uma heurística que determina uma estimativa do custo do caminho que tem como início o no n e como final, a meta a ser atingida.

A seguir, tem-se o pseudo-código do algoritmo.

```
// Marque s como aberto e calcule f(s).
s.Status = "aberto";
// Selecione um nó aberto que possua menor valor f(n).
n = GetNoMenorF();

//Se n é igual a meta buscada, marque-o como fechado e termine o algoritmo.
Se (n == meta)
{
    n.Status = "fechado";
    return;
}

//Se n é não é igual a meta e ainda existirem nós marcados como "abertos"
Enquanto (n != meta and ExistemNosAbertos())
{
    // marque n como fechado
    n.Status = "fechado";

    //    identifique os sucessores de n através de um operador T
    sucessores = n.T();
    // Calcule o valor de f(n) de cada sucessor e marque como aberto cada
    //   sucessor ainda nao marcado como fechado
    CalcularF(sucessores);
    Para cada (sucessor em sucessores)
       Se (sucessor.Status != "fechado")
          sucessor.Status = "aberto";

    // Remarque como aberto cada nó fechado ni, desde que ele seja sucessor
    //    de n e que e que f(ni) seja menor agora do que quando ni foi
    //    marcado como fechado.
    Se (sucessor.FAtual < sucessorFAnterior)
        sucessor.Status = "aberto";

   // Selecione um nó aberto que possua menor valor f(n).
   n = GetNoMenorF();
}
```

Figura 3-5: Pseudo-código do algoritmo A*.

O operador T define a lista de sucessores de um nó *n.* Sua implementação varia de domínio para domínio.

Um dado importante é que, sempre que um determinado nó marcado como aberto é recuperado para ser comparado com a meta; escolhe-se aquele com menor valor *f(n)*. Ou seja, escolhe-se aquele que possui a menor soma do custo acumulado até o momento com a estimativa de custo para se atingir a meta. Desta forma, os nós não são escolhidos ao acaso; sua definição é guiada por um processo que garante o menor caminho até o momento.

Conforme já mencionado, o A* é o algoritmo base para o desenvolvimento da pesquisa realizada.

O A* é muito utilizado em problemas do tipo *pathfinding*. Para facilitar a compreensão do algoritmo, um exemplo será apresentado. Supondo-se a existência de um mapa formado por vários quadrados dispostos todos adjacentes uns aos outros, criando

uma formação em grade. A Figura 3-6 apresenta tal mapa. “Obstáculo” representa uma área que não pode ser atravessada, “A” representa o ponto de origem e “B” o ponto em que se deseja chegar. Movimentos nesse mapa são possíveis de um quadrado para outro em qualquer direção: horizontal, vertical ou diagonal.

Inicialmente, “A” é marcado como *s* (nó inicial) e *f(s)* é calculado. Seguindo o algoritmo, o próximo passo é recuperar o elemento *n* marcado como “aberto” e que possui menor *f(n)*. Nesse exemplo, o custo de um nó *g(n)* será calculado como o custo de movimentação de um ponto até outro. Para movimentos na horizontal, o custo será estipulado com 10, e na diagonal como 14. A “Distância de Manhattan” será usada para calcular o valor da heurística. Assim, é contada a quantidade de quadrados na horizontal ou

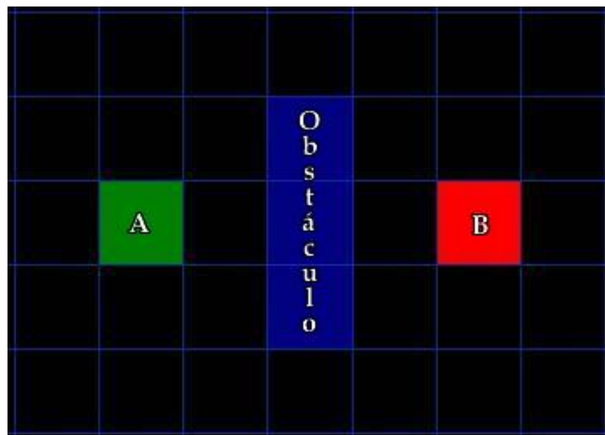

Figura 3-6: Exemplo A* - estado inicial.

vertical (diagonal será desconsidera) que existem entre o ponto corrente e o destino. A esse valor, multiplica-se 10, que é o custo de movimentação na horizontal ou vertical. De qualquer forma, nesse momento inicial, tem-se apenas um elemento marcado como aberto e é ele que será recuperado.

A seguir, deve-se verificar se ele é igual ao ponto em que se deseja atingir. Nesse caso, não é. Logo, ele é marcado como “fechado”, mas a execução do algoritmo continua. Desta forma, devem ser identificados os sucessores de *n*. Eles serão calculados como sendo todos os quadrados adjacentes a *n*.

A Figura 3-7 mostra os sucessores de *n*. Cada sucessor é marcado como aberto e o *f(n)* é calculado para todos eles. Nas figuras, o valor de *f(n)* será apresentado na parte superior esquerda de cada quadrado, o valor de *g(n)* na porção inferior esquerda e *h(n)* na parte inferior direita.

Seguindo o algoritmo, deve-se executar novamente o passo 2. Logo, o quadrado com menor *f(n)* e recuperado. Esse quadrado, nesse momento, é aquele localizado imediatamente a direita do ponto A e com valor de *f(n)* igual a 40.

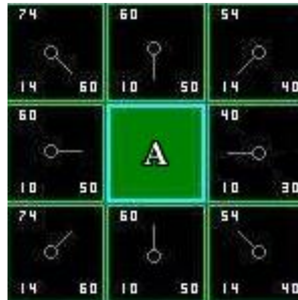

Figura 3-7: Exemplo A* - sucessores de n.

Como ele também não é igual ao destino, devem ser analisados os seus sucessores. Os três quadrados a direita pertencem a uma área que não pode ser atravessada, logo, devem ser descartados. Formalmente, eles até poderiam ser marcados como “abertos”. Se isso fosse feito, o valor de seu custo deveria ser igual a ∞ indicando que jamais deveriam ser selecionados. O ponto imediatamente a esquerda é aquele a partir do qual o quadrado corrente foi selecionado (em um grafo, o ponto “A” seria o nó pai do quadrado (nó) corrente). Os outros quatro quadrados restantes já estão na lista aberta. Resta, então, verificar se passando pelo nó corrente o caminho para estes quatro quadrados não é menor. Faz-se isso verificando os valores de *f(n)* destes quadrados. Considerando, inicialmente, o quadrado imediatamente acima do quadrado corrente, tem-se que seu custo, seria igual a 20. Isso porque, o custo do nó corrente já é 10. Para se deslocar na vertical, tem-se um custo de mais 10. Logo, *g(n)* calculado a partir do quadrado corrente é igual a 20. O valor da heurística não é alterado e permanece igual a 40. Desta forma, o novo *f(n)* seria igual a 60. Este valor é maior do que o valor original e igual a 54. Assim, ele não deve ser remarcado como nó aberto. É importante notar que isso faz todo sentido, para eu chegar a esse quadrado acima do quadrado corrente, é bem mais simples, a partir de “A”, deslocar-se na diagonal, do que, a partir de “A”, deslocar-se, inicialmente para a direita e, em seguida, para cima. Essa situação se repetirá com os outros três quadrados.

Este procedimento é executado até que o ponto destino seja atingido. A Figura 3-8 mostra o estado final após a execução completa do algoritmo.

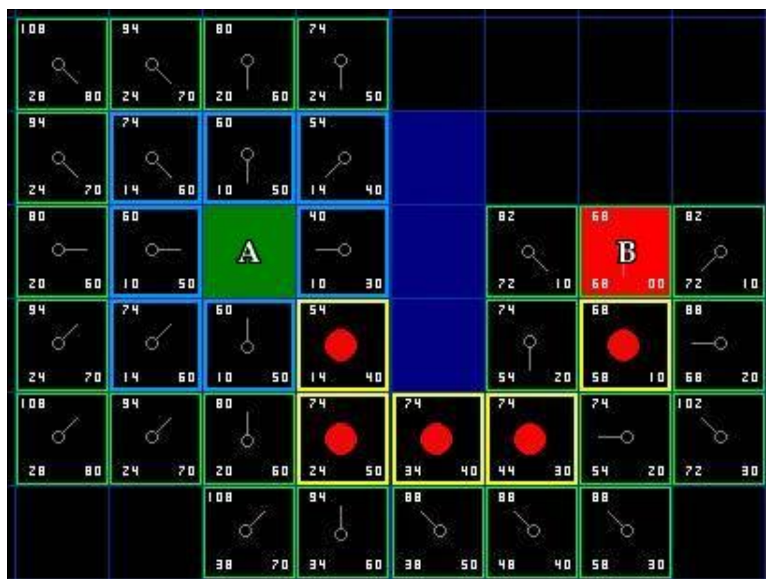

Figura 3-8: Exemplo A* - caminho gerado.

Os círculos em vermelho indicam o menor caminho de “A” até “B”.

3.3 Planejamento

Um problema de planejamento requer três entradas: uma apresentação do estado inicial do mundo, uma meta a ser atingida e uma descrição das ações que podem ser realizadas. O resultado de um processo de planejamento é uma seqüência de ações, que quando executada em um estado inicial qualquer, atingirá uma determinada meta (WELD, 1999).

O planejamento como elemento da área de Inteligência Artificial surgiu das pesquisas sobre busca em espaço de estados, prova de teoremas, teoria de controle e de necessidades práticas de robótica, escalonamento, entre outras (RUSSELL e NORVIG, 2002).

Um dos mais famosos domínios usados para explicar o conceito de planejamento é conhecido como “mundo dos blocos”. O domínio é composto por uma coleção de blocos no formato de cubos dispostos sobre uma mesa. Um bloco pode estar diretamente sobre a mesa ou sobre outro bloco. Existe ainda a figura de um braço robótico capaz de mover um bloco de uma posição para outra, independentemente se este está em cima da mesa ou em cima de outro bloco. A Figura 3-9 ilustra este domínio.

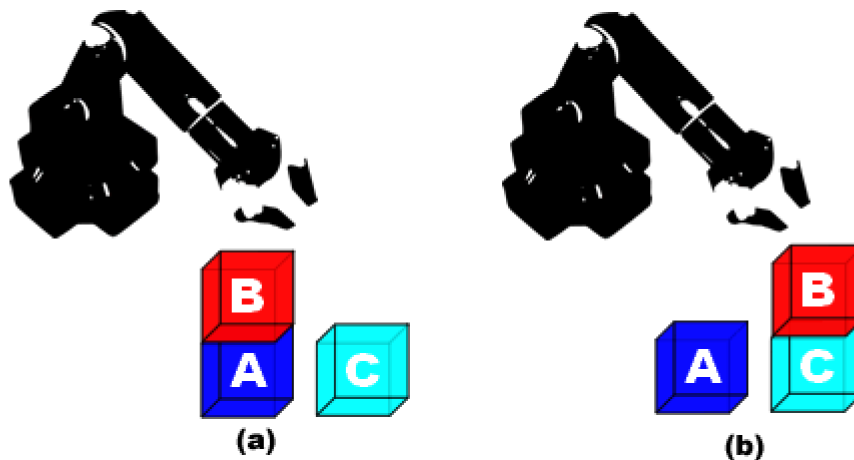

Figura 3-9: Mundo dos blocos.

Pode-se definir um problema de planejamento da seguinte forma:

- Estado inicial: o estado apresentado na Figura 3-9 (a), ou seja, os bloco A e C sobre a mesa, o bloco B sobre o bloco A e o braço robótico livre.
- Meta: os blocos A e C sobre a mesa, o bloco B sobre o bloco C (Figura 3-9(b))
- Ações possíveis: mover um bloco para a mesa ou mover um bloco para cima de outro bloco.

O problema pode ser resolvido executando-se uma única ação: mover o bloco B para cima do bloco C.

Um elemento importante e, que ainda não foi mencionado, é a linguagem de representação. É preciso uma linguagem para descrever os estados do mundo e as ações que podem ser realizadas.

3.3.1 Linguagem de representação

Os dois principais fatores que devem ser considerados na criação de uma linguagem de representação são: expressividade e eficiência. Uma linguagem deve ser expressiva o bastante para descrever grande variedade de problemas, porém restritiva o suficiente para que permita que algoritmos a utilizem eficientemente.

Dentre as várias linguagens criadas para modelar estados do mundo e ações possíveis em um problema de planejamento, destacam-se: STRIPS, ADL e PDDL.

3.3.1.1 STRIPS

STRIPS (*STanford Research Institute Problem Solver*) é, na verdade, mais do que uma simples linguagem de representação; é um solucionador geral de problemas (*problem solver*). Criado em 1971 no Instituto de Pesquisa de Stanford, STRIPS tinha, inicialmente, a função de resolver problemas encontrados no domínios dos robôs (FIKES e NILSSON, 1971).

Em jogos simples ou quebra-cabeças, o estado do mundo é facilmente representado em uma lista ou matriz. Por outro lado, isso não acontece nos ambientes dinâmicos e complexos em que os robôs se encontram inseridos. Para representar tal classe de ambientes, STRIPS utiliza predicados da lógica de primeira ordem.

Neste momento, o foco não é explorar o solucionador de problemas, mas sim a "linguagem STRIPS". Desta forma, a seguir será detalhado como são representadas ações e estados de mundo.

Um estado do mundo é representado por um conjunto de fórmulas bem-formadas. É permitido, por exemplo, o uso de literais proposicionais. No caso dos literais da lógica de primeira ordem, estes devem ser aterrados e não podem incluir funções. A hipótese de mundo-fechado (REITER, 1978) é considerada para tratar o *problema do quadro* (McCARTHY e HAYES, 1969) e significa que qualquer condição que não seja mencionada em um estado é assumida como falsa.

Sendo assim, considerando o "mundo dos blocos" apresentado na seção 3.3, para descrever um estado inicial do mundo onde se tem um robô na posição A, um bloco B na posição b e outro bloco C na posição c, pode-se usar a seguinte fórmula:

```
ATR(A) ∧ AT(B,b) ∧ AT(C,c)
```

Em um problema de planejamento, além do estado inicial, utiliza-se outro estado de mundo que corresponde à meta, ou seja, como deve ficar o mundo após a realização das ações definidas pelo algoritmo de planejamento. Para representar uma meta em que se deseja que o bloco B esteja na posição a e o bloco C na posição b, tem-se:

```
AT(B,a) ∧ AT(C,b)
```

Para transformar um estado de mundo em outro é necessário a aplicação de uma ou mais ações. Na linguagem STRIPS, uma ação é chamada de *operador* e é formada por um conjunto de precondições e um conjunto de efeitos. As precondições descrevem a situação em que um operador pode ser aplicado e os efeitos o que acontecerá com o mundo após a aplicação.

As precondições são também descritas usando fórmulas bem-formadas. Supondo a existência de um operador *push* que empilha um objeto k localizado originalmente na posição m, sobre algo localizado na posição n, algumas precondições devem ser satisfeitas para que *push* possa ser aplicado. Inicialmente, o robô deve estar localizado na posição m (visto que será o robô o responsável por mover o bloco de uma posição para outra). Além disso, o bloco k deve, obviamente, estar em m. Desta forma, as precondições de *push* podem ser descritas como:

```
ATR(m) ∧ AT(k,m)
```

Os efeitos são divididos em duas listas: *add list* e *delete list.* Na *add list* são descritas as cláusulas que serão incorporadas ao mundo, ou seja, aquelas que talvez não fossem verdadeiras antes da aplicação do operador, mas que passarão a ser. Na *delete list* são descritas as cláusulas que não mais serão verdadeiras, ou seja, contém cláusulas que devem ser retiradas do estado do mundo anterior à aplicação do operador.

No caso do operador *push,* o robô não mais estará em m, mas sim em n. Da mesma forma, o objeto k não mais estará na posição m, pois será empilhado sobre n. O operador *push* pode ser descrito, completamente, por:

```
push(k,m,n)
      Precondições: ATR(m)
                  ∧ AT(K,m)
      Delete List: ATR(m)
                  ∧ AT(K,m)
      Add List: ATR(n)
                  ∧ AT(K,n)
```

3.3.1.2 ADL

Com a evolução da área de planejamento, percebeu-se que a linguagem STRIPS não era capaz de representar alguns domínios reais. Desta forma, surgiram diversas variantes desta linguagem. Dentre as mais importantes, destaca-se a ADL ou *Action Description Language* (PEDNAULT, 1989).

Algumas das diferenças mais relevantes são: possibilidade da utilização de quantificadores na meta (na linguagem STRIPS eram permitidos apenas literais aterrados);

efeitos condicionais, ou seja, um efeito acontecerá apenas se uma condição for satisfeita e suporte ao predicado "igualdade" (não presente na linguagem STRIPS).

A Tabela 3-1 apresenta uma comparação mais detalhada entre STRIPS e ADL (RUSSELL e NORVIG, 2002).

Tabela 3-1: Comparação entre STRIPS e ADL.

Linguagem STRIPS	Linguagem ADL
Somente literais positivos nos estados `Poor ∧ Unknown`	Literais positivos e negativos nos estados: `¬Rich ∧ ¬Famous`
Hipótese de mundo fechado, ou seja, literais não mencionados são considerados *falsos*.	Hipótese de mundo aberto, ou seja, literais não mencionados desconhecidos.
O efeito P ∧ ¬Q significa adicionar P e remover Q.	O efeito P ∧ ¬Q significa adicionar P e ¬Q e remover ¬P e Q.
A meta só pode ter literais aterrados: `Rich ∧ Famous`	É permitido o uso de quantificadores na meta: `(∃x) (AT(P1,x) ∧ AT(P2,x))`
A meta é uma conjunção: `Rich ∧ Famous`	A meta permite conjunções e disjunções: `¬Poor ∧ (Rich ∨ Smart)`
Efeitos são conjunções.	Permite efeitos condicionais: `when P: E` Significa que E é um efeito somente se P for satisfeito.
Não suporta verificação de igualdade	O predicado de igualdade `(x = y)` é suportado.
Não suporta tipos.	Variáveis podem ter tipos, como em: `(p: Plane)`.

Dentre todas as diferenças listadas, é importante ressaltar que a possibilidade de representar *efeitos condicionais* permite a redução na quantidade de ações instanciadas para resolver um problema, visto que uma mesma ação pode ser utilizada para diferentes situações.

3.3.1.3 PDDL

PDDL (*Planning Domain Definition Language*) é um padrão de linguagem para definição de problemas e domínios utilizados para planejamento. Além de suas características próprias, PDDL suporta STRIPS e ADL. É uma linguagem desenvolvida

especialmente para a Competição de Planificadores realizada no AIPS-98 (*Artificial Intelligence Planning System*), com o objetivo de unificar a definição de domínios e problemas para sistemas de planejamento, permitindo assim compará-los de forma direta (GHALLAB et al., 1998).

PDDL é derivado de outras formalizações como STRIPS, ADL e outras "linguagens". Apesar de muitos planejadores não suportarem completamente o PDDL, a maioria suporta os conceitos de STRIPS. A linguagem possui as seguintes características de sintaxe:

- Ações representadas no estilo STRIPS.
- Efeitos condicionais.
- Universos dinâmicos, permitindo a criação e destruição de objetos.
- Quantificadores universais.
- Especificação de ações hierárquicas compostas de sub-ações e sub-metas.
- Gerenciamento de múltiplos problemas em múltiplos domínios.

A representação de um domínio em PDDL é dada por uma declaração contendo os seguintes itens:

- Definição do nome do problema.
- Conjunto de características de representação necessárias para definição do domínio.
- Conjunto de proposições do domínio.
- As ações que podem ser utilizadas, contendo:
  - Nome da ação.
  - Parâmetros passados para a ação.
  - Conjunto de pré-condições.
  - Conjunto de efeitos.

Basicamente, PDDL especifica a sintaxe para o processo de planejamento e descreve a base para os atuais planejadores. Possui especificamente um conjunto de predicados, uma definição da representação do domínio e um conjunto de ações, com metas, pré-condições e efeitos.

3.3.2 Planejamento por busca

Conforme mencionado na seção 3.3, um problema de planejamento resume-se na identificação de um conjunto de ações que transforma o estado atual do mundo no estado descrito em uma meta. Uma forma de encontrar estas ações é através da utilização de algoritmos de busca.

A essência de um algoritmo de busca é simples: considera-se, inicialmente, um elemento de uma coleção, se ele corresponde ao elemento que se está procurando, o algoritmo retorna este elemento ou sua posição e nada mais necessita ser feito. Caso contrário, outro elemento deve ser testado. A escolha de qual será este próximo elemento é chamada de *estratégia de busca* (RUSSELL e NORVIG, 2002) e corresponde à lógica do algoritmo. No caso de um problema de planejamento, os elementos presentes no domínio da busca são estados do mundo e, em particular, o elemento procurado é a meta.

Uma busca pode começar a partir de um elemento inicial e, a cada passo, se aproximar da meta, ou, ter início na meta e se aproximar, passo a passo, do estado inicial. No primeiro caso temos uma busca *progressiva* e no segundo uma busca *regressiva.*

3.3.2.1 Busca Progressiva

Em um algoritmo de planejamento baseado em uma busca progressiva tem-se uma verificação do estado inicial e se este for igual à meta a ser atingida, este próprio estado inicial é a solução. Caso contrário, deve-se buscar por uma sequência finita de ações que transforme o estado inicial na meta.

Desde o início das pesquisas na área de planejamento até o final da década de 90, considerava-se que a eficiência deste tipo de algoritmo era muito ruim para ser considerado aplicável (RUSSELL e NORVIG, 2002). Existem dois grandes motivos que explicam esse fato. O primeiro deles é que ele considera todas as ações possíveis para cada estado. Considerando uma pequena expansão do mundo dos blocos em que o braço mecânico que realiza a movimentação dos blocos pode executar inúmeras outras ações como, por exemplo, parafusar, martelar ou quebrar objetos; estas novas possibilidades são totalmente irrelevantes em uma meta de posicionamento de blocos, contudo, serão testadas pelo algoritmo. Conforme será apresentado a seguir, isso não acontece na Busca Regressiva.

O segundo motivo está relacionado com o melhor caminho para se chegar a uma solução. Considerando-se a existência de um problema em que um avião deve levar pacotes de um aeroporto para outro; de forma mais específica, que um avião deve levar 20 pacotes de um aeroporto A para outro aeroporto B. Supondo também a existência de 5 aviões e 10 aeroportos diferentes, a solução não é complicada. Na verdade, existe uma solução simples: carregar os 20 pacotes em qualquer avião presente no aeroporto A; este por sua vez deve voar até o aeroporto B e descarregar os pacotes. Entretanto, encontrar essa opção não é tão simples, visto que o espaço de busca é grande. Cada um dos 5 aviões pode voar para outros 9 aeroportos e cada um dos 20 pacotes pode ser, tanto carregados para o avião, quanto, também, descarregados (caso já estivessem no avião).

Para atenuar esse último problema faz-se necessário a utilização de uma boa heurística para guiar a busca. Neste ponto tem-se a possibilidade prática de uso de algoritmos de busca progressiva. Recentemente, esse tipo de algoritmo foi usado com êxito em ambientes complexos fazendo uso de boas heurísticas. Ou seja, é possível utilizá-lo eficientemente, contudo o papel de uma heurística para guiar a busca é essencial.

3.3.2.2 Busca Regressiva

Se nos algoritmos baseados em busca progressiva parte-se do estado inicial, na busca regressiva usa-se a meta como ponto de partida. O objetivo passa a ser atingir o estado inicial.

Voltando ao domínio do mundo dos blocos estendido, ou seja, no domínio em que o braço mecânico realiza também ações como parafusar, martelar ou quebrar objetos, no problema de posicionamento de blocos, as novas ações não precisam ser analisadas. Isto porque, a meta é ter os blocos x, y e z nas posições p1, p2 e p3. Partindo da meta, o que se deve descobrir é qual o conjunto de ações é capaz de alterar a posição dos blocos. Desta forma, não serão utilizadas as ações parafusar, martelar ou quebrar objetos, diminuindo consideravelmente a quantidade de ações que podem ser utilizadas em cada transformação de estado.

Um fator importante que deve ser observado em algoritmos de busca regressiva é que as ações devem manter a consistência do estado atual com relação à meta. Ou seja, uma ação não deve desfazer parte da meta que, até aquele momento, encontra-se satisfeita. Por exemplo, no caso do problema do transporte de cargas de um aeroporto para outro,

suponha a utilização, em um primeiro momento, de uma ação que carrega o primeiro pacote para o interior de um avião. Neste ponto, tem-se uma aproximação da meta, visto que, no estado inicial, 20 pacotes deveriam ser carregados e após a utilização da ação, apenas 19. Desta forma, outra ação não deve descarregar esse pacote (pelo menos não no aeroporto de origem) pois estaria se distanciando da meta. Uma ação que não desfaz parte da meta satisfeita é chamada consistente (RUSSELL e NORVIG, 2002).

Apesar de apresentar um desempenho superior a busca progressiva, um algoritmo de busca regressiva deve também fazer uso de uma boa heurística para apresentar grande eficiência. Uma função heurística, nesse caso, tem o papel de estimar a distância do estado atual com relação à meta.

3.3.3 POP (Planejamento de ordem-parcial)

O planejamento baseado em algoritmos de busca regressiva ou progressiva explora uma seqüência linear de ações diretamente conectadas à meta ou ao estado atual. Este tipo de planejamento é chamado *totalmente ordenado* ou *planejamento de ordem-total.* Existe uma outra abordagem que é capaz de trabalhar com a decomposição de problemas, tal que, a meta principal é dividida em sub-metas que podem ser atingidas separadamente e sem ordem fixa de realização. Algoritmos que apresentam essas características são chamados algoritmos de *ordem-parcial* (RUSSELL e NORVIG, 2002).

Um fator que pode influenciar positivamente no desempenho do algoritmo é a possibilidade de trabalhar nas sub-metas sem ordem fixa. Isto porque pode ser mais conveniente escolher atingir, inicialmente, metas mais “importantes” ou mais “simples”.

3.3.4 Planejamento usando grafos

Em busca de soluções mais eficientes e de heurísticas mais precisas, Avrim Blum e Merrick Furst (1997) criaram um novo algoritmo que utilizava a estrutura de um grafo para resolver problemas de planejamento. O *GraphPlan* – nome dado ao algoritmo – precisava de uma classificação e esta foi definida pelos próprios autores como *Planning Graphs.*

Um *Planning Graph* é um grafo direcionado e separado em níveis que possui dois tipos de nós e três tipos de arestas. Os níveis alternam-se entre níveis que contêm nós com proposições e níveis em que os nós são representações de ações. No primeiro nível de um *Planning Graph*, tem-se um nó para cada proposição do estado inicial do mundo. Em

seqüência tem-se um nível com nós que representam ações que poderiam ser aplicadas ao estado inicial. O nível seguinte apresenta o estado do mundo, descrito por proposições, e é determinado pela aplicação das ações do nível anterior no primeiro nível do grafo. Depois tem-se outro nível de possíveis ações e outro nível proposicional e assim por diante.

As arestas de um *Planning Graph* representam relações entre ações e proposições. Os nós presentes no nível-ação i são conectados por "arestas precondicionais" às precondições do nível-proposicional i, por "arestas de inclusão" aos efeitos do tipo "inclusão" do nível-proposicional i + 1 e por "arestas de exclusão" aos efeitos do tipo "exclusão" do nível-proposicional i + 1. Um efeito do tipo inclusão refere-se àquele que adiciona proposições ao estado corrente do mundo e um efeito do tipo exclusão àquele que remove proposições do estado do mundo.

Para exclarecer esta idéia de níveis e tipos de nós e arestas, considere um problema em que uma pessoa deseja comer um pedaço de bolo. Se a pessoa já possui o pedaço de bolo, basta comê-lo. Se ela ainda não o tem, é necessário fazer um bolo. Note que, neste caso, está sendo desconsiderada a possibilidade de comprar o bolo. Sendo assim, as duas ações possíveis são "comer" ou "fazer".

A Figura 3-10 demonstra uma representação do problema "ter e comer o bolo".

```
Estado Inicial: tem(bolo)

Meta: tem(bolo) ∧ comeu(bolo)

Ações:

        comer(bolo):
            tem(bolo)                        Precondições
            ¬ tem(bolo) ∧ comeu(bolo)        Efeitos

        fazer(bolo):
            ¬ tem(bolo)                      Precondições
            tem(bolo)                        Efeitos
```

Figura 3-10: Representação do problema "ter e comer o bolo".

Tem-se, inicialmente, o estado inicial que, neste caso, indica que a pessoa possui o bolo. A seguir, tem-se a meta que define que além de ter o bolo, a pessoa deve comer o bolo. Por fim, são apresentadas as duas ações: comer o bolo e fazer o bolo.

O *Planning Graph* para este problema pode ser escrito conforme apresentado na Figura 3-11. No primeiro nível têm-se duas proposições: `tem(bolo)` e

¬comeu(bolo). O segundo nível apresenta a ação comer(bolo). No terceiro nível têm-se os nós com as proposições tem(bolo), ¬ tem(bolo), comeu(bolo) e ¬comeu(bolo). O quarto nível apresenta duas ações fazer(bolo) e comer(bolo) e, o quinto e último nível, possui as mesmas proposições do nível três. Neste exemplo, as "arestas precondicionais" são representadas por linhas simples, as "arestas de inclusão" por linhas pontilhadas e as "arestas de exclusão" por linhas tracejadas. Ou seja, para se comer o bolo, é precondição necessária ter um pedaço de bolo ou, em outras palavras, precisa-se da existência da proposição tem(bolo). Da mesma forma, quando se come o bolo, o efeito é a exclusão das proposições tem(bolo) e ¬ comeu(bolo) e a inclusão das proposições ¬tem(bolo) e comeu(bolo). As linhas com um círculo preenchido representam a persistência dos estados mediante a não execução de ação alguma. Tem-se aqui, na verdade, um quarto tipo de aresta.

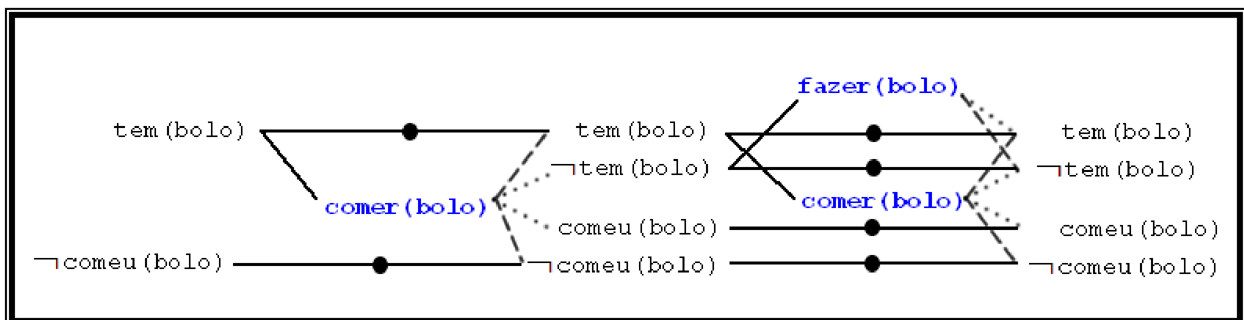

Figura 3-11: Planning Graph do problema "ter e comer o bolo".

Uma vez definido o que é um *Planning Graph*, falta explicar como esta estrutura pode ser utilizada por um algoritmo de planejamento. A resposta é simples: algoritmos de planejamento fazem uma busca no grafo, nível a nível, procurando identificar ações que satisfazem a(s) meta(s). Esta busca deve seguir algumas regras básicas visando gerar planos válidos e consistentes. Nesse caso, o algoritmo deve identificar proposições que são mutualmente exclusivas, ou seja, que não podem ser verdadeiras ao mesmo tempo em um mesmo nível, ações que desfazem efeitos criados por outras ações e ações que modificam o estado de precondições necessárias a ações que já pertencem ao plano. O algoritmo *GraphPlan* lida com todas essas restrições e ainda é capaz de identificar a não existência de um plano válido para um determinado estado inicial e meta proposta.

3.3.5 Planejamento usando Lógica Proposiconal

Outra abordagem utilizada para resolver problemas de planejamento baseia-se no uso de lógica proposicional. Entretanto, do final da década de 60 até o início da década de 90, a técnica foi praticamente abandonada. Isto porque a forma com que a lógica proposicional era utilizada trazia resultados ineficientes uma vez que a geração de um plano era feita através da prova de um teorema. O teorema era formulado da seguinte forma: dados o estado inicial do mundo e um conjunto de axiomas que descreviam o efeito das ações, a meta seria verdadeira em uma situação resultante da aplicação de uma seqüência de ações, ou seja, o estado inicial mais uma seqüência de ações deveria implicar na meta (RUSSELL e NORVIG, 2002).

Um dos problemas da abordagem baseada na prova de teoremas é que ela pode gerar modelos incorretos, isto é, os axiomas são respeitados, mas o plano gerado não é valido (KAUTZ e SELMAN, 1992).

Foi proposta em 1992 uma nova concepção que usava a lógica proposicional mas, sem fazer uso da prova de teoremas. A idéia de Kautz e Selman (1992) foi gerar planos através da verificação da satisfatibilidade de uma fórmula expressa em sua FNC (Forma Normal Conjuntiva) semelhante a:

```
estado inicial ∧ conjunto de axiomas ∧ meta
```

Neste caso, o problema resume-se em encontrar um conjunto de axiomas que garanta que todo modelo gerado a partir deles constitua um plano válido. Nesse sentido, devem ser criados axiomas para definir: o estado inicial do mundo, a meta, as ações (incluindo suas precondições e efeitos) e algumas restrições para assegurar a geração de planos válidos.

Para exemplificar, voltando ao domínio do mundo dos blocos descrito na seção 3.3, considere apenas dois blocos A e B, de tal forma que, inicialmente A se encontra em cima de B e considere como meta o estado inverso atingido no tempo 3, ou seja, que B se encontre acima de A duas unidades de tempo após o estado inicial. Nesse contexto, o estado inicial e a meta poderiam ser expressos da seguinte forma:

```
on(A,B,1) ∧ on(B,Table,1) ∧ clear(A,1) ∧ on(B,A,3)
```

O terceiro argumento de cada proposição (os valores inteiros) indica o tempo, isto é, no tempo 1, A se encontra acima de B, B se encontra sobre a mesa e não existe nada sobre A. Além disso, no tempo 3, deseja-se que B esteja sobre A.

No caso da descrição das ações, deve-se especificar tanto seus efeitos como precondições. Uma forma de fazê-lo é colocar tanto os efeitos como as precondições como sendo implicados pela ação em si. No exemplo proposto, ter-se-ia algo do tipo:

$$\forall\ x,y,z,i\ \mathtt{move}(x,y,z,i) \rightarrow (\mathtt{clear}(x,i) \wedge \mathtt{clear}(z,i) \wedge \mathtt{on}(x,y,i))$$

Por fim, algumas restrições devem ser feitas. Nesse caso, deve-se deixar explícito que uma ação só acontece em um estado de tempo.

$$\forall\ x,y,z,x',y',z',i\ (x \neq x' \vee y \neq y' \vee z \neq z') \rightarrow (\neg \mathtt{move}(x,y,z,i) \vee \neg \mathtt{move}(x',y',z',i))$$

A partir destes axiomas, do estado inicial do mundo e da meta, o único modelo que pode ser gerado para este exemplo é `move(A,b,Table,1)` e `move(B,Table,A,2)`.

Baseado nessas idéias, Kautz e Selman desenvolveram o SATPlan. O SATPlan é precursor de outro famoso algoritmo apresentado em 1998 chamado BlackBox. Este, por sua vez, utiliza a lógica do SATPlan aliada a técnica de geração de planos usando grafos (KAUTZ e SELMAR, 1998).

3.4 Sistemas Multiagentes

Nesta seção serão apresentados os conceitos de Agente e Sistemas Multiagente. Estes conceitos são importantes para compreender o contexto em que são criados os algoritmos que tem como função comprovar as hipóteses da seção 1.1. O *AStarPlanner*, algoritmo que será descrito no Capítulo 5, é o componente planejador da arquitetura de um agente. O *MultiAStarPlanner*, descrito no Capítulo 6, é um algoritmo que utiliza um sistema de leilões para identificar os agentes que devem executar determinadas metas.

3.4.1 Agente

Segundo Russel e Norvig (2002), um agente "é tudo o que pode ser considerado capaz de perceber seu ambiente por meio de sensores e de agir sobre esse ambiente por intermédio de atuadores". Um ser humano, por exemplo, é um tipo de agente. Ele percebe

o ambiente a sua volta através de seus olhos, ouvidos e outros órgãos sensoriais e atua sobre o mesmo através de mãos, pés e outras partes do corpo que funcionam como atuadores.

Um agente robótico poderia utilizar sensores de calor, sensores infravermelhos e câmeras para perceber o mundo e um conjunto de motores para atuar sobre ele. Um agente de software poderia perceber o ambiente através de um conjunto de gerenciadores de eventos, e poderia atuar sobre ele através da execução de métodos que alterassem as características desse ambiente.

Os agentes devem realizar ações independentemente do estado do ambiente ou da interação com outros agentes. Normalmente os agentes apresentam as características de autonomia, mobilidade, reatividade e pró-atividade (REIS, 2003).

A autonomia refere-se ao princípio de que os agentes devem agir baseados em suas próprias decisões, não sendo necessário que exista a interferência de um humano.

Mobilidade consiste na capacidade de locomoção do agente de um local para o outro, em caso de um agente de *software* ele pode se mover de um computador para o outro através de uma rede.

A reatividade é a capacidade de reagir às mudanças do ambiente. Apesar de desejável, um agente não pode ser totalmente reativo, sendo capaz de agir autonomamente para satisfação de seus objetivos.

A pró-atividade é a capacidade de iniciativa, representando o comportamento independente do agente. As ações são realizadas de acordo com as metas e não simplesmente devido às mudanças do ambiente.

No Capítulo 4, será apresentado um ambiente denominado *Project Hoshimi.* Ele constitui um simulador totalmente implementado via software. Nele, um agente é uma entidade denominada nanorobô.

Não existe explicitamente um conceito de sensores físicos, entretanto ao se deslocar sobre uma determinada área de um mapa, um nanorobô possui sensores internos capazes de identificar sobre qual tipo de célula ele se encontra; ele consegue perceber outros agentes a uma distância pré-definida nas configurações do *Project Hoshimi,* ele consegue perceber a presença de áreas não atravessáveis; ele pode também reconhecer pontos especiais no mapa como pontos AZN e pontos HP (*Hoshimi Point*). Da mesma forma, apesar de não existirem atuadores explícitos, um nanorobô pode se movimentar através de pressupostos propulsores, pode atirar em um outro nanorobô, pode coletar enzimas de

pontos AZN e pode transferir enzimas para pontos HP. Um tipo especial de nanorobô chamado *NanoAI* pode criar outros nanorobôs. Existem ainda nanorobôs capazes de alterar a densidade das células próximas a eles.

3.4.2 Sistemas Multiagente

Geralmente, um agente não existirá sozinho em um ambiente, ele fará parte de uma sociedade de agentes, de um Sistema Multiagente. Estes agentes poderão compartilhar suas habilidades, trabalhar em paralelo ou em problemas comuns, operar sobre questões como redundância e tolerância a falhas e cooperar para solucionar problemas (HUNS e STEPHENS, 2001).

Os Sistemas Multiagente são sistemas compostos por dois ou mais agentes, que exibem comportamento autônomo e ao mesmo tempo interagem com outros agentes presentes no sistema. Estes conceitos são inspirados nos humanos, sendo uma proposta para simular comportamento de sociedades e ambientes. Duas características fundamentais destes sistemas são: capacidade de tomadas de decisões autônomas direcionadas para solução de seus objetivos e comunicação entre os agentes. Formalmente, um "Sistema Multiagente é aquele composto por um número de agentes, os quais interagem uns com os outros, tipicamente por troca de mensagens através de alguma infra-estrutura de rede.(WOOLDRIDGE, 2002)". Para que essa interação aconteça com sucesso, serão requeridas, a esses agentes, habilidades de cooperação, coordenação e negociação.

Em sua tese de doutorado, Reis (2003) afirma que "os Sistemas Multiagente incluem diversos agentes que interagem ou trabalham em conjunto, podendo compreender agentes homogêneos ou heterogêneos. Cada agente é basicamente um elemento capaz de resolução autônoma de problemas e opera assincronamente, com respeito aos outros agentes".

Nas duas definições, tem-se a noção de interação entre diversos agentes. Conforme apresentado por Wooldridge (2002), para que um agente possa realizar ações e interagir com outros agentes, é necessário que exista uma infra-estrutura. Basicamente, essa infra-estrutura é composta por: organizações, interações, agentes, um ambiente e esferas de influência. A Figura 3-12 mostra essa infra-estrutura.O ambiente consiste no local onde estão os agentes e seu estado é alterado de acordo com os resultados das ações dos agentes. Para cada agente existe uma interação que ocorre com outros agentes e os efeitos desta

interação pode alterar o comportamento do agente e auxiliar na conquista das metas dos agentes. Cada agente possui uma esfera de influência distinta sobre o ambiente, sendo capaz de influenciar diferentes partes do ambiente.

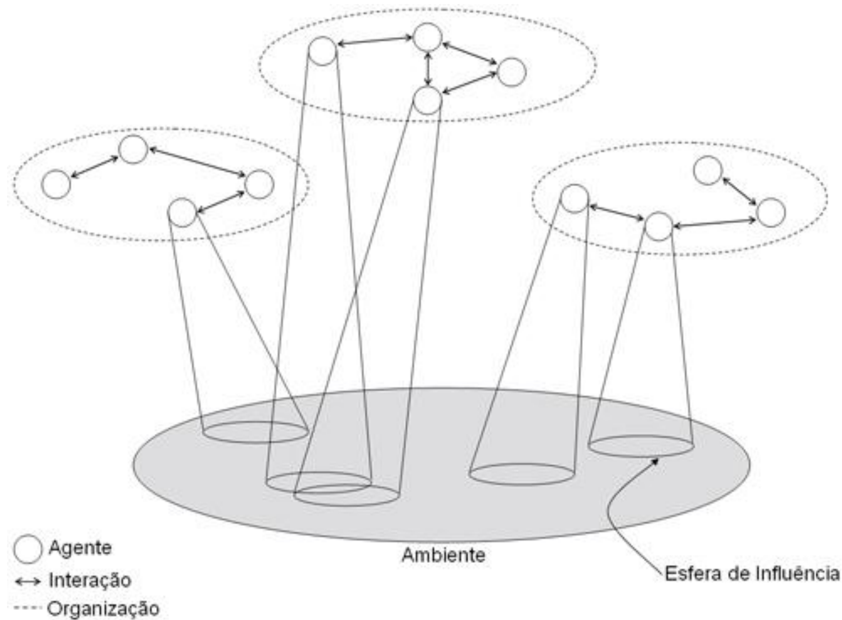

Figura 3-12: Infra-estrutura básica de um Sistema Multiagente (WOOLDRIDGE, 2002, p106).

Apesar de ser uma área relativamente nova, os Sistemas Multiagente ganharam notoriedade e destaque nos últimos anos (REIS, 2003). O interesse em sua pesquisa ocorre principalmente devido a possibilidade de simular comportamentos complexos a partir de simples agentes que agem localmente. O trabalho utiliza conceitos básicos dos Sistemas multiagente, onde ocorrem relações de competição por ações e vários agentes interagem localmente para solução de problemas globais.

4. Project Hoshimi

Project Hoshimi é o nome da plataforma escolhida para testar as hipóteses levantadas nesta dissertação, na seção 1.1. A plataforma é constituída por um simulador de um ambiente multiagente e uma SDK*(Software Developer Kit)* com classes e métodos para controle dos agentes. Essa plataforma foi criada pela Microsoft e utilizada em uma competição mundial conhecida por *Imagine Cup*. Esta, por sua vez, possui várias categorias, dentre elas, a *Programming Battle* que utilizou a *Project Hoshimi* (HOSHIMI, 2006). A plataforma foi usada nas edições de 2005, 2006 e 2007 da competição, sendo que, a cada ano, novas funcionalidades foram acrescentadas. A partir da edição de 2007, a categoria *Programming Battle* passou a usar o nome *Project Hoshimi.*

Nessa categoria, um programador tem à sua disposição uma coleção de classes para gerenciar um time de nanorobôs. Esses nanorobôs são injetados, virtualmente, dentro do corpo de seres humanos (ou de outros seres vivos) com a intenção de, a priori, curar doenças. Existem diversos tipos de nanorobôs, cada um com uma função distinta, cada um com um conjunto de ações possíveis diferentes.

O principal nanorobô é o *NanoAI* e cada time pode conter apenas um. O *NanoAI* é como uma unidade central de comando, se ele for destruído, todo o time perde seu poder de realizar ações. Ele é o responsável por criar outros nanorobôs. *NanoCollectors* e *NanoContainers* são especializados em coleta, transporte e transferência de enzimas. Diferenciam-se um do outro pela capacidade de armazenamento e pelas ações de ataque/proteção. Têm-se, ainda, os *NanoProtectors* que são tipos especiais de *NanoCollectors* que possuem apenas ações de ataque. Um *NanoNeedle* é responsável pela injeção de enzimas. Para que enzimas possam ser transferidas, *NanoNeedles* devem ser criados em cima de pontos especiais denominados *Hoshimi Points*. Assim, quando um *NanoCollector* ou um *NanoContainer* carregado de enzimas se acopla a um *NanoNeedle*, a injeção de enzimas é iniciada. Por fim, tem-se o *NanoBlocker*, que tem a função de aumentar a densidade das células adjacentes a ele.

O ambiente em que esses nanorobôs existem constitui uma mapa. Como eles são injetados no interior de seres vivos, cada mapa corresponde a uma parte de um ser. Desta

forma, pode-se ter um mapa que representa um coração, um pulmão ou, ainda, uma porção de um tronco de uma árvore.

O programador deve, em um determinado mapa, fazer com que uma coleção de nanorobôs realize um conjunto de objetivos. Existem diversos tipos de objetivos, tais como: exploração, sobrevivência e coleta e transferência de enzimas. Para cada objetivo atingido, o time ganha uma pontuação.

Os mapas possuem pontos com características especiais. Os mais importantes são: AZN e HP. Um ponto do tipo AZN é um local onde um nanorobô pode coletar enzimas. Um ponto do tipo HP é um local onde um nanorobô pode depositar enzimas.

Na Figura 4-1, tem-se um mapa que representa um coração humano. Os pontos brancos circulares são do tipo AZN e os brancos retangulares são do tipo HP.

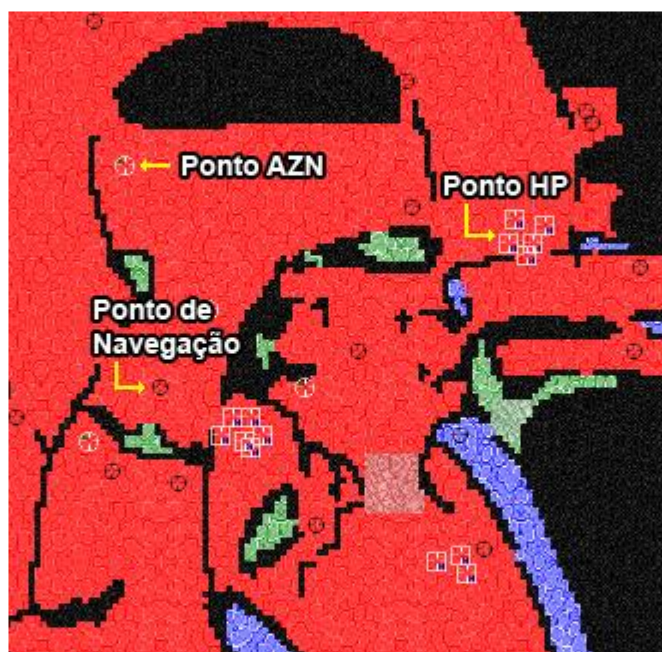

Figura 4-1: Pontos AZN, HP e de Navegação.

Quando enzimas são depositadas em pontos do tipo HP, o time de nanorobôs recebe uma determinada pontuação. Logo, independente dos objetivos, coletar e depositar enzimas é sempre importante.

Existem ainda os chamados *Navigation Points* ou “Pontos de Navegação” que correspondem a locais pelos quais nanorobôs devem passar. São usados em objetivos do tipo exploração. Geralmente, existe um tempo associado a um objetivo deste tipo, isto é, um nanorobô deve atingir um determinado Ponto de Navegação em um determinado tempo. A representação de um Ponto de Navegação é um ponto circular preto, ligeiramente menor que um AZN, como mostrado na Figura 4-1.

Em um mapa, existem células com diferentes densidades. Porções na cor vermelha são células de densidade baixa, na cor azul de densidade média e verde de densidade alta. Quanto maior a densidade, mais lenta é a locomoção dos nanorobôs. Logo, no momento de calcular o caminho mais rápido entre dois pontos, deve-se dar preferência por áreas de densidade baixa. Contudo, robôs do tipo *NanoExplorer* são capazes de se locomoverem na velocidade máxima independente da densidade da célula. A Figura 4-2 exibe um mapa que possui células com os três níveis de densidade.

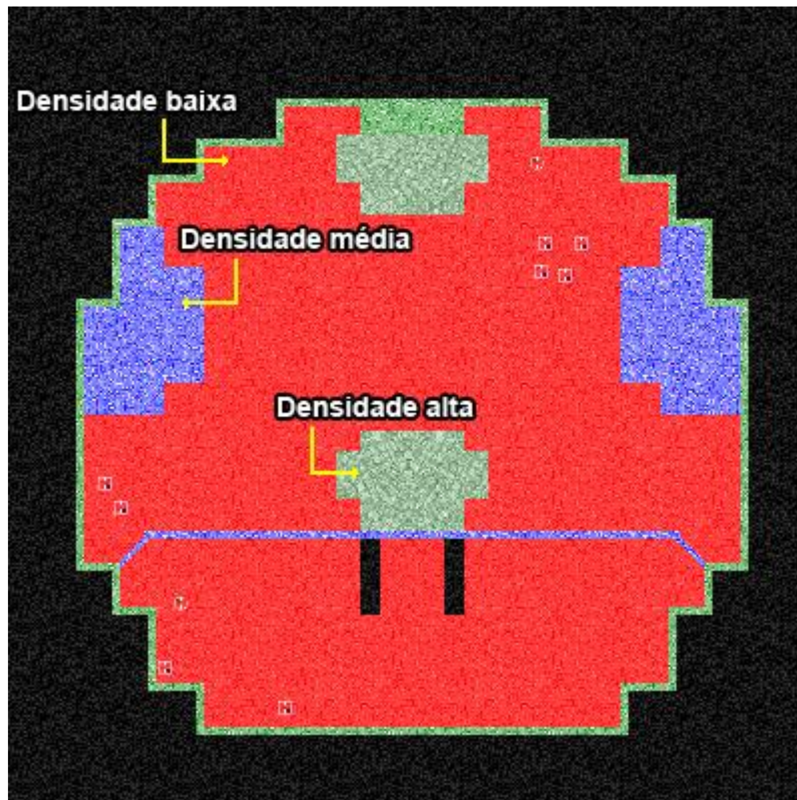

Figura 4-2: Densidade das células.

Além disso, é relevante ressaltar que *NanoBlockers* são capazes de aumentar a densidade das células, sendo esta outra informação que deve ser considerada em algoritmos do tipo *pathfinding*.

Em um mapa, além dos aglomerados comuns de células, têm-se os chamados fluxos sanguíneos ou *bloodstreams*. Na Figura 4-3, destacam-se exemplos deste tipo de elemento. O fluxo é direcional, logo, se o nanorobô se movimenta em sentido contrário a ele, terá uma penalidade de 50% na velocidade de locomoção. Caso se movimente no sentido do fluxo ou em uma direção perpendicular a ele, se deslocará com o dobro da velocidade.

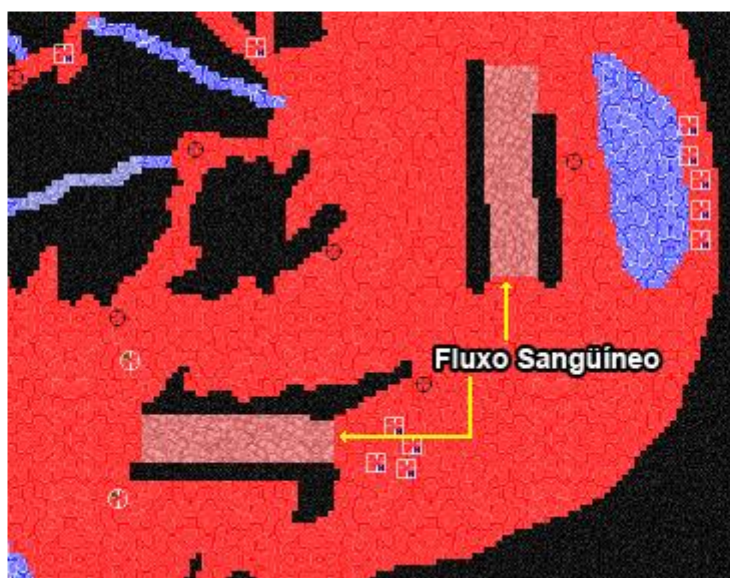

Figura 4-3: Fluxos sanguíneos (*bloodstreams*).

Outra característica do *Project Hoshimi* é a existência de um time de nanorobôs denominado *Time do Pierre*. Sua única função é atrapalhar os demais times a realizarem os objetivos do mapa corrente. Os robôs do *Time do Pierre*, ao se depararem com robôs de outros times, tentarão destruí-los. Nesse sentido, em alguns momentos pode ser interessante tentar destruir esses "inimigos" ou evitar passar próximo deles.

O ambiente multiagente é totalmente controlado por *software*. Vários times de robôs podem coexistir ao mesmo tempo em um mapa, tentando, cada um deles atingir a maior quantidade de objetivos. Entretanto, não existe a possibilidade de um jogador externo, controlar o time e tomar decisões à medida que o tempo evolui. Essa situação, apesar de não existir, pode ser conseguida com pequenas alterações no *software* que controla o ambiente. Desta forma, ter-se-ia um clássico jogo eletrônico em que parte dos agentes seria controlada por *software* e um ou mais times seriam controlados por jogadores externos.

Analisando a natureza do ambiente, ele foi classificado como um jogo de estratégia, conforme descrito no final da seção 3.1. O que define um time vencedor é a estratégia utilizada para cumprir as metas e a escolha das melhores ações. Essa escolha sempre deve levar em consideração o tempo, visto que, o jogo acontece em um intervalo fixo de tempo, ou seja, os times têm uma quantidade de minutos predeterminada no início do jogo para atingir as metas.

Além disso, a todo momento, os agentes devem redefinir suas próximas ações, uma vez que o ambiente é totalmente dinâmico e influenciado por agentes de times diferentes. Por exemplo, um agente pode ter na sua lista de ações a injeção de enzimas em um determinado HP. Todavia, um outro nanorobô deste mesmo time pode passar próximo deste HP e descobrir que ele já foi ocupado por um agente de um outro time. Nesse caso, a ação de injeção de enzimas naquele ponto torna-se impossível, exigindo a definição de uma nova ação.

Do ponto de vista de um sistema multiagente, tem-se um ambiente de processamento e comunicação centralizado no *NanoAI.* Ou seja, toda comunicação entre dois nanorobôs diferentes deve ser feita através do *NanoAI.* Da mesma forma, ele funciona como uma entidade central de processamento. Os nanorobôs têm autonomia para gerarem seus próprios planos e realizarem suas ações, mas dependem de um controle central para isso. É como se a fonte de sua energia fosse proveniente do *NanoAI.* Desta forma, a eliminação deste nanorobô deixa todos os outros incapazes de realizarem ações. Não existe nenhum tipo de protocolo formal de troca de mensagens entre um nanorobô e *NanoAI,* como KIF (WOOLDRIDGE 2002) ou KQML(WOOLDRIDGE, 2002). Na verdade, a interação acontece apenas através de variáveis de escopo global presentes no *NanoAI*. Observa-se também que não existe nenhum mecanismo que defina um aspecto colaborativo ou cooperativo na realização das metas. Essa questão fica sob a responsabilidade do programador.

5. *AStarPlanner*

Conforme mencionado na sessão 1.1, um dos objetivos deste trabalho foi verificar a viabilidade da utilização de um algoritmo de planejamento baseado no tradicional A* em um jogo de estratégia. Na literatura, pode-se encontrar uma tentativa muito bem sucedida realizada por Orkin (2004 e 2005) de aplicar esse algoritmo para gerar planos e definir as ações de NPCs em um jogo do tipo FPS intitulado F.E.A.R (*First Encounter Assault Recon*) (F.E.A.R, 2005).

Em seu trabalho, Jeff Orkin comenta que não foram feitos testes em outros tipos de jogos, como RPG (*Role Playing Game*) ou RTS, mas que, provavelmente, se o fizesse, não enfrentaria problemas. Contudo, em um jogo do tipo FPS, geralmente os planos são compostos de poucas ações e, além disso, a quantidade de ações que um NPC é capaz de realizar é pequena. Supondo-se que um NPC encontre um personagem do jogador e que uma meta de destruir ou eliminar esse personagem seja gerada, os planos gerados poderiam ser algo do tipo:

- Plano 1: atacar com uma determinada arma.
- Plano 2: procurar um lugar protegido e atacar com uma arma de longo alcance.

Não existem muitas outras possibilidades e, nas duas citadas, o maior plano foi composto por duas ações. Esse foi só um exemplo, mas, geralmente, as metas de agentes em um FPS podem ser atingidas com a realização de poucas ações, o que pode ser observado nos exemplos utilizados por Orkin em seus trabalhos e apresentações em conferências.

Em um jogo de estratégia, a quantidade de ações pode ser muito maior. Por exemplo, no *Project Hoshimi* citado no Capítulo 4, dependendo do tipo de representação adotada para descrever o ambiente, ações e metas, pode-se ter um plano com até 20 ações para a realização de uma meta do tipo preenchimento total da capacidade de um *NanoNeedle*. Dependendo da capacidade de armazenamento do *NanoCollector* ou *NanoContainer*, pode-se ter um número ainda maior. Nesse sentido, o planejador poderia até gerar a melhor seqüência de ações para a realização da meta, contudo, o desempenho poderia ser um problema.

O uso do algoritmo A* em jogos eletrônicos não é algo novo. Entretanto, em geral, ele é utilizado com outra finalidade de descobrir o menor caminho entre dois pontos. Supondo que um personagem precise se deslocar de um ponto A do mapa até um ponto B. Geralmente, existem obstáculos ou locais não atravessáveis como uma barreira de chamas. Além disso, poderão existir diversos caminhos possíveis. O A* é aplicado para descobrir o menor deles. Algoritmos desse tipo são classificados como *pathfinding algorithms.*

Em um problema do tipo *pathfinding*, tipicamente, um nó do grafo gerado pelo A* corresponde a uma coordenada em um sistema 2D ou 3D. O que une um nó a outro é uma operação de movimentação. Tem-se, resumidamente, um ponto inicial, um ponto final e uma seqüência de pontos que une a origem ao destino.

Um algoritmo de planejamento pode ser encarado como um problema de busca com um estado inicial, um estado final e uma seqüência de ações que transformam o estado inicial no estado final e identificada por uma busca em um espaço de estados do mundo.

Assim, usando o A* para gerar planos, cada nó do grafo corresponde a um estado do mundo e o que une um nó a outro é uma ação. Assim, tem-se, por exemplo: mover para uma posição x, coletar um determinado recurso, usar um determinado recurso, atirar. Certamente, não se pode representar todo o estado do mundo em um nó, pois geraria problemas de desempenho na comparação de dois estados pelo algoritmo de busca. Por outro lado, deve-se armazenar o mínimo necessário para que bons planos sejam gerados. A solução é utilizar um sistema de representação simbólica do mundo centrada no agente (ORKIN, 2004). Desta forma, todo conhecimento armazenado tem o agente como referência. Esse simples fato de centralização já economiza o trabalho de descobrir a qual objeto uma determinada informação se refere.

Para usar o algoritmo A*, é necessário estipular o custo das ações e a heurística utilizada para calcular a distância que um determinado estado do mundo se encontra da meta. O custo das ações é utilizado para estabelecer a preferência de execução. Supondo que um NPC do jogo F.E.A.R. encontre com o personagem do jogador, ele deve atacá-lo. Ele poderá fazê-lo através de uma ação "atacar" ou de uma ação "atacar a partir de um lugar protegido". Caso seja a intenção dos *designers* do jogo que ele normalmente ataque a partir de lugares protegidos, como por exemplo, quando estiver atrás de uma rocha, o que devem fazer é atribuir um custo da ação maior para "atacar" do que para a ação "atacar a partir de um lugar protegido".

No *Project Hoshimi*, não se identificou esse tipo de situação, ou seja, um momento em que se deveria dar preferência a uma ação. Sendo assim, foi estipulado um custo fixo e igual a 1 para todas elas. Caso, seja verificada essa necessidade de preferência, a única coisa que se deve alterar é o custo das ações, isto é, nada precisa ser modificado no algoritmo.

No caso da heurística, ela é usada para restringir o espaço de busca. Cada ação possui um valor heurístico associado. Esse valor mede a distância que o estado atual se encontrará do estado final (meta), após a execução da ação. Assim, quanto mais uma ação conseguir fazer com que o estado atual se aproxime da meta, maior será sua preferência de utilização em um plano. Desta forma, se existirem duas ações com mesmo custo, mas, com valores heurísticos diferentes, ao invés de testar as duas possibilidades e contar com a sorte para descobrir qual a ação seria mais adequada para gerar um plano eficiente, o A* testará primeiramente, a que possuir um valor heurístico menor.

A heurística utilizada no *AStarPlanner* foi a quantidade de símbolos não satisfeitos com relação à meta. Para melhor entendimento, um exemplo será apresentado. Considerando-se um estado inicial do mundo em que um agente encontra-se na posição (100,20) e que possua o valor da propriedade *IsNeedlePartiallyFull* igual a *false*. Considerando-se ainda uma meta que estipula que um agente deve se localizar na posição (123,53) e ter a propriedade *IsNeedlePartiallyFull* igual a *true*. Uma ação do tipo *Transfer* tem em sua lista de efeitos uma definição de alteração do valor da propriedade *IsNeedlePartiallyFull* para *true*. Desta forma, caso esta ação fosse aplicada ao estado inicial, apenas um símbolo não seria resolvido (aquele relacionado com o posicionamento). Logo, o valor heurístico associado à ação *Transfer*, nesta situação, é 1.

Uma heurística é dita admissível, ou seja, capaz de gerar resultados ótimos, quando nunca superestima o custo para atingir uma meta (RUSSEL e NORVIG, 2002). Uma maneira de garantir a admissibilidade de uma heurística é trabalhar com um *relaxamento* do problema. "O custo de uma solução ótima para um problema relaxado é uma heurística admissível para o problema original" (RUSSEL e NORVIG, 2002, p107). O processo de relaxamento envolve retirar algumas restrições do problema. No *AStarPlanner,* a heurística utilizada faz uso de um relaxamento do problema original, na medida que desconsidera a presença de precondições de uma ação. Isso porque, a análise é feita apenas em termos dos símbolos não satisfeitos com relação a meta, nada é comentado sobre a possibilidade da existência de precondições. No caso de sua existência, ter-se-ia ainda que

realizar algumas outras ações para satisfazer essas precondições, o que faria com que a heurística retornasse um valor abaixo do real custo para atingir a meta.

Contudo, é preciso analisar a questão da independência das metas para garantir a admissibilidade. Supondo uma meta $m1 \wedge m2 \wedge m3$ e as ações *a1*, *a2* e *a3*. Supondo ainda que a ação *a1* resolva a sub-meta *m1*. Logo, o valor de sua heurística seria igual a 2, indicando que com o uso de duas ações resolver-se-ia, por completo, a meta proposta. Porém, caso *a2* ou *a3* resolvam *m2* e *m3* ao mesmo tempo, o custo real seria igual a 1. Neste caso, a heurística para *a1* teria superestimado o custo real da meta.

Fazendo uma análise criteriosa de todas as metas possíveis e da lista de ações disponíveis no ambiente *Project Hoshimi*¸ verificou-se que, em nenhum caso, uma ação seria capaz de resolver dois símbolos das metas propostas. Isso é coerente com os resultados apresentados na seção 5.3.2. Todavia, sabe-se que esta pode ter sido uma particularidade deste ambiente, podendo não ocorrer em outros jogos de estratégia. Mesmo assim, no caso de um jogo, não se está preocupado com a solução ótima sempre, mas sim com planos executados em tempos aceitáveis e capazes de gerar uma sensação, ao jogador, de que os NPCs são realmente inteligentes.

5.1 O Algoritmo

Nessa sessão será descrito o algoritmo *AStarPlanner*. O *AStarPlanner* é um algoritmo que segue uma abordagem de definição de planos a partir de um algoritmo de busca, de ordem total e que não trabalha com conceito de replanejamento.

A idéia básica do algoritmo A* é simples e já foi descrita na sessão 3.2.4. No caso do *AStarPlanner*, optou-se por implementar as listas aberta e fechada através de uma estrutura de dados denominada *heap*. A vantagem da utilização de uma *heap* é que sempre que um elemento é retirado da lista aberta, este deve ser o que possui menor valor de F (custo G do nó + valor da heurística H). Esta é justamente a grande característica de uma *heap*, uma vez que sua cabeça constitui o menor (ou maior) elemento da coleção (CORMEN et al., et al, 2001). Uma outra opção poderia ser trabalhar com uma lista ordenada. A desvantagem é que a coleção teria que estar completamente ordenada. No caso do A* isto não é necessário, uma vez que precisa-se apenas do menor elemento.

A grande diferença do *AStarPlanner* para o A* é que, como ele trabalha em cima de ações que, eventualmente, vão compor planos, ele deve gerenciar situações em que

determinadas condições devem ser satisfeitas para que uma ação possa ser usada. Em outras palavras, existem ações que possuem precondições e isso deve ser tratado pelo algoritmo.

O nó usado no *AStarPlanner* é formado pelos seguintes atributos: *G* (custo do nó), *H* (valor da heurística associada ao nó), *F* (*G* + *H*), *Pai* (referência para o nó pai), *Sucessores* (lista de sucessores), *EstadoDoMundo* (conjunto de pares símbolo-valor que descrevem o estado corrente do mundo), *Acao* (ação utilizada para gerar o estado corrente do mundo) e *EstadoDoMundoAntesDaAcao* (estado do mundo antes da aplicação da ação armazenada no atributo *Acao*).

As entradas do *AStarPlanner* são dois nós: origem e destino. Em ambos os nós, a referência para o nó pai é nula. O *EstadoDoMundo* da origem é o estado corrente do agente ou o estado do agente após a realização do seu último plano (A seção 6.2 descreve com detalhes esse assunto). No caso do destino, o estado do mundo é o conjunto de pares símbolo-valor que descreve a meta. Os valores do atributo *Acao* também são nulos para os dois nós. O valor do atributo *EstadoDoMundoAntesDaAcao* é igual ao *EstadoDoMundo* na origem e no destino.

É importante salientar que o nó destino não será utilizado no plano gerado. Ele será usado apenas para a realização de comparações. Ou seja, durante o processo de planejamento, serão verificadas relações de igualdade entre o valor do atributo *EstadoDoMundo* de vários nós e do nó destino. Desta forma, não existe problema algum em se ter atributos como *Acao* com valor nulo e *EstadoDoMundoAntesDaAcao* com o mesmo valor de *EstadoDoMundo*.

O atributo *Acao* identifica qual ação foi aplicada para a obtenção do estado do mundo especificado em *EstadoDoMundo*. Para facilitar a explicação, um exemplo será apresentado. Supõe-se, inicialmente, que o estado do mundo é composto apenas por um atributo booleano *isEnemyDead* que indica se um oponente está morto ou não, e que em um nó A, o valor de *isEnemyDead* seja igual a *false*. Supõe-se também que em uma ação do tipo *Atack,* na sua lista de efeitos, exista uma alteração do valor de *isEnemyDead* para *true*. Desta forma, um nó B com valor do atributo *isEnemyDead* do *EstadoMundo* igual a *true*, pode ser gerado a partir de A, aplicando-se a ação *Atack.* Resumindo, o valor do atributo *Acao* do nó B é igual a *Atack* e, além disso, o valor do atributo *Pai*, será igual a A, visto que B foi gerado a partir de A.

No exemplo anterior, o nó B faria parte da lista de sucessores de A. A lista de sucessores, representada pelo atributo *Sucessores*, indica quais são os nós gerados a partir do nó corrente através da aplicação de ações. O objetivo é identificar quais serão os próximos nós testados na intenção de encontrar um nó com o mesmo estado do mundo especificado no nó destino. No *AStarPlanner,* a lista de sucessores é criada verificando-se quais os valores dos símbolos do atributo *EstadoMundo* são diferentes dos valores dos

```
AStarPlanner(origem, destino)
{
        Se (origem = destino)
              return;

        // Adicionar origem na lista aberta
        ListaAberta.Adiciona(origem);

        // Enquanto a lista fechada não contiver o destino e a lista
        // aberta possuir elementos...
        Enquanto (ListaFechada.NãoPossui(destino) && ListaAberta.Count()!=0)
        {
              // Retirar elemento da cabeça da lista aberta
              noCorrente = ListaAberta.GetCabeca();

              // Se ação possuir precondições...
              Se (noCorrente.Acao.TemPrecondicoes())
                    TratarPrecondicoes(noCorrente, destino);

              // Remover cabeça da lista aberta
              ListaAberta.Remove();
              // Adicionar no corrente na lista fechada
              ListaFechada.Add(noCorrente);

              // Para cada sucessor do nó corrente...
              Para Cada (sucessor em noCorrente.Sucessores)
              {
                    // Se sucessor estiver na lista fechada, ignore-o
                    Se (ListaFechada.Possui(sucessor))
                          continue;

                    // Se sucessor estiver na lista aberta
                    Se (ListaAberta.Possui(sucessor)
                    {
                          // Verificar se existe um caminho menor até o sucessor passando pelo
                          // no corrente
                          custoAntigo = sucessor.G;
                          novoCusto = sucessor.GetNovoG()+ noCorrente.G;

                          Se (custoAntigo > novoCusto)
                          {
                                sucessor.Pai = noCorrente;
                                sucessor.G = novoCusto;
                          }
                    }
                    Senão
                          ListaAberta.Adiciona(sucessor);
              }
        }

        Se (ListaFechada.Possui(destino)
        {
              no = ListaFechada.GetNo(destino);
              Enquanto (no.Pai <> null)
              {
                    Solucao.Adiciona(no);
                    no = no.Pai;
              }
        }
}
```

Figura 5-1: Pseudocódigo do *AStarPlanner.*

símbolos da meta proposta. Para cada valor diferente, cria-se novos nós. Cada nó tem um novo estado de mundo gerado através da aplicação de uma ação e deve ser inserido na lista de sucessores. Na Figura 5-1 tem-se o pseudocódigo do *AStarPlanner*:

Em termos de lógica, basicamente, o *AStarPlanner* diferencia-se do A* pela inclusão do teste `Se (noCorrente.Acao.TemPrecondicoes())` e da linha subsequente que é executada quando a interpretação do teste resulta em valor verdade igual a *true*. Na Figura 5-2, apresenta-se o pseudocódigo do método *TratarPrecondicoes*.

```
TratarPrecondicoes(noCorrente)
{
      origem = noCorrente.EstadoDoMundoAntesDaAcao;
      destino = noCorrente.Acao.Precondicoes;

      // Meta a ser atingida são as precondições da ação corrente. MAS, o
      // estado inicial para a busca dessa meta não é o estado corrente, mas
      // sim o estado antes da execução da ação do nó corrente.
      meta = origem.GetDiferencas(destino);

      plano = AStarPlanner(origem, meta);

      plano.No[0].Pai = noCorrente.Pai;
      noCorrente.Pai = plano.No[plano.QuantidadeNos -1];

      noCorrente.G = noCorrente.G + plano.No[plano.QuantidadeNos -1].G;

      Para Cada (no em plano)
            ListaFechada.Adiciona(no);

}
```

Figura 5-2: Pseudocódigo do Método TratarPrecondições.

Uma precondição, nada mais é do que uma nova meta. Logo, foi aplicado recursivamente o algoritmo A* para descobrir a sequência de ações que levariam a um estado em que as precondições fossem satisfeitas.

Contudo, um detalhe deve ser levado em consideração. Supondo-se que uma meta seja satisfeita aplicando-se a seqüência de ações A → B → C, tal que A, B e C são ações e que → determina a ordem de execução. Supondo-se ainda que B possua uma lista de precondições que são satisfeitas por uma nova seqüência de ações D → E, tal que D e E são ações e → determina a ordem de execução. Nesse caso, a ordem completa de execução seria A → D → E → B → C. Logo, no momento em que B fosse selecionada, antes de selecionar C, suas precondições deveriam ser calculadas. É importante salientar que, quando a partir de B, forem geradas as ações D e E, deve-se ter cuidado para que o estado inicial seja o estado após a aplicação de A e não de B. Além disso, o estado inicial para a

aplicação de B deve ser o estado após a aplicação de E, e não após a aplicação de A. Por isso, na primeira linha do algoritmo tem-se como estado inicial o estado do nó corrente antes da execução da ação associada a este nó.

A meta é gerada a partir das diferenças entre o estado do mundo antes da execução da ação (armazenado em origem) e a lista de precondições (armazenada em destino).

Após a geração do plano, é preciso atualizar a estrutura de nós pais. Assim, voltando ao exemplo das ações A,B,C,D e E, o nó pai, que possui a primeira ação do plano gerado para atingir as precondições necessárias para que B possa ser executado, é o antigo nó pai de B, ou seja, A. Não se pode esquecer que cada ação se encontra em um nó. Logo, `plano.No[0].Pai = noCorrente.Pai`. De forma semelhante, o pai de B passa a ser E (última ação do plano), logo `noCorrente.Pai = plano.No[plano.QuantidadeNos - 1]`.

Além disso, como o nó possui precondições, seu custo deve ser atualizado de acordo com os custos dos nós necessários para garantir que as precondições sejam satisfeitas. Como o custo é um valor cumulativo, basta recuperar o valor do último nó do plano gerado para atender as precondições.

Por fim, é necessário adicionar os nós usados para resolver as precondições do nó corrente na lista fechada.

5.2 A Implementação e Arquitetura dos agentes

Nessa seção será descrita a arquitetura dos agentes em que o algoritmo *AStarPlanner* foi usado. Além disso, serão apresentados alguns aspectos técnicos com relação à codificação do componente planejador da arquitetura, ou seja, o *AStarPlanner*.

Conforme mencionado anteriormente, o ambiente de testes escolhido foi o *Project Hoshimi.* O mesmo constitui um ambiente multiagentes, tal que, cada agente é representado por um nanorobô. Desta forma, parte da arquitetura do agente já estava definida e implementada. Contudo, para trabalhar com o *AStarPlanner*, ela necessitou ser estendida. A nova arquitetura assemelha-se bastante àquela definida pelo MIT (*Massachusetts Institute of Technology*) para o agente C4 (BURKE, 2001).

Nessa arquitetura, um agente é composto por uma unidade de memória, componentes de percepção, planejamento, execução de ações e um componente baseado

em um quadro negro (*blackboard*) responsável pela troca de informações entre os demais elementos da arquitetura. A unidade de memória armazena o estado do ambiente que é relevante para a tarefa que estiver sendo executada. O componente de percepção permite a atualização dinâmica do estado do ambiente, o que se faz possível pela presença de sensores. O componente de execução, também conhecido como sistema motor, é responsável pela execução das ações. O componente que determina quais as ações e a ordem que devem ser executadas é um sistema de planejamento.

O componente *blackboard* e o planejador (*AStarPlanner*) não existem no *Project Hoshimi*, logo foram criados. O sistema de percepção foi alterado para permitir que o *blackboard* fosse atualizado. A unidade de memória teve que ser estendida para trabalhar com uma notação simbólica, uma vez que o *AStarPlanner* trabalha com esse tipo de notação. Por fim, o sistema motor não foi alterado.

5.2.1 Implementação do planejador

O subsistema planejador da arquitetura é o *AStarPlanner*. Conforme apresentado na seção 3.2.4, o algoritmo A* trabalha com um grafo. No *AStarPlanner*, os nós do grafo são instâncias de uma classe chamada *AStarPlannerNode*. Além dos tradicionais atributos para armazenar a meta (*m_Goal*) do custo do nó (*m_G*), do valor da heurística (*m_H*), do custo total do nó (*m_F*) e de uma referência para o nó pai (m_*Parent*), tem-se também: *m_Action* que indica qual ação foi executada para se chegar àquele estado do mundo; *m_WorldState* que armazena o estado corrente do mundo; *m_WorldStateBeforeAction* que armazena o estado do mundo antes da execução da ação.

O atributo *m_WorldState* armazena o estado do mundo através de uma tabela *hash* de objetos *WorldStateProperty*. Cada chave corresponde a um símbolo que denota um tipo de conhecimento, como por exemplo, *Position* (símbolo cujo valor descreve a posição atual do agente).

As ações são formadas por listas de precondições e efeitos. Estas listas também foram implementadas com tabelas *hash* as quais seguem exatamente a mesma estrutura da representação de mundo. Isso facilita a identificação de ações que podem ser úteis para solucionar uma meta ou identificar se as precondições de uma ação são ou não satisfeitas por um estado corrente do mundo.

Durante a execução do algoritmo A*, após verificar que o nó corrente não satisfaz a meta e, sabendo-se que o que transforma um nó em outro é a execução de uma ação, o que se deseja saber são as ações que gerarão os nós sucessores (ou vizinhos). Para gerar, efetivamente, o nó vizinho, é necessário simular a execução de uma ação no estado corrente do mundo. Por simular a execução, entende-se verificar se as precondições da ação são satisfeitas e aplicar a lista de efeitos no estado corrente do mundo daquele nó. É importante ressaltar que, antes de aplicar a ação, uma cópia do mundo deve ser armazenada no atributo *m_WorldStateBeforeAction.* Esse valor é utilizado no método "TratarPrecondições", citado no início deste capítulo.

5.3 Testes Realizados

O algoritmo *AStarPlanner* foi utilizado para gerar os planos do time de nanorobôs *CodeSdruxulators2007* que participou da edição de 2007 da *Imagine Cup/Project Hoshimi*. Em 2006, o time *CodeSdruxulators* usou um algoritmo baseado em máquinas de estado finito para gerar a seqüência de ações que cada nanorobô tinha que executar para cumprir uma meta.

Foram feitos dois tipos de testes. No primeiro deles, procurou-se comparar o desempenho na geração dos planos e no segundo, verificar a qualidade dos planos gerados.

5.3.1 Teste de desempenho

Para isolar fatores externos não relacionados com a geração do plano em si, algumas restrições foram feitas para a realização dos testes. A primeira delas foi trabalhar com um único tipo de testes. Foi escolhido um cenário em que se necessitaria de planos com a maior quantidade de ações para cumprir uma meta. Isso porque, queria-se verificar se os planos eram gerados em um tempo aceitável. Desta forma, foi escolhido o teste "preencher totalmente um ponto HP". Isto quer dizer que, um nanorobô deveria coletar enzimas, transportá-las e depositá-las em um ponto HP até que este fosse completamente preenchido. Cada enzima depositada em um ponto HP, confere uma pontuação para o time.

A segunda restrição foi que em todos os testes, o nanorobô *NanoAI* seria criado sempre na mesma posição. No código de ambos os times que participaram da competição,

a posição inicial do nanorobô AI poderia variar, dependendo da posição inicial de outros times presentes na partida ou de características particulares do mapa. A posição inicial é extremamente importante pois é neste ponto que todos os outros nanorobôs serão criados.

Os testes foram feitos com apenas um time no mapa. Na *Imagine Cup*, dois times competiam no mesmo mapa, ao mesmo tempo. Mas, para evitar que um time influenciasse na rota do outro, optou-se por colocar um time por vez em cada mapa.

Outra restrição imposta foi a ordem de pontos HP que seriam visitados pelo *NanoAI*. Cada vez que ele atingisse um ponto HP, um *NanoNeedle* seria criado. Em ambos os times que participaram da competição, essa rota era totalmente variável. Isso porque, dois times participavam de uma mesma partida, logo, caso o time adversário criasse um *NanoNeedle* em um ponto HP, este ponto não poderia ser usado por nenhum outro time, a não ser que o *Time do Pierre* destruísse esse nanorobô. Isto porque, em ponto HP pode existir somente um *NanoNeedle*. Desta forma, se este ponto HP estivesse na rota do time que não criou a *NanoNeedle*, ele deveria ser excluído da mesma.

A rota influencia diretamente no resultado do time. Supondo o mapa da Figura 5-3 e os pontos HP A, B e C, pode-se verificar que, se o *NanoAI* passar pelos pontos seguindo a ordem A → B → C, o tempo necessário para realizar tal traçado será menor do que se percorresse a rota A → C → B. Foram considerados apenas 3 pontos nesse exemplo, extrapolando o número de pontos para a quantidade total existente em um mapa. É simples perceber que o tempo gasto para atingir todos os pontos pode ser drasticamente diferente dependendo do percurso usado. É importante ressaltar que, enquanto o *NanoAI* não passar por um ponto HP, um *NanoNeedle* não poderá ser criado e, sem um *NanoNeedle* criado, não é possível transferir enzimas para aquele ponto. Isto quer dizer que, como cada partida acontece em um intervalo predeterminado de tempo, dois percursos diferentes podem resultar em diferentes quantidades de *NanoNeedles* criados.

Obviamente, foi estabelecido que seriam criados nanorobôs coletores de mesmo tipo, velocidade de movimentação, coleta e transferência de enzimas e mesma capacidade de armazenamento.

Por fim, foram definidos quantos nanorobôs seriam criados e quais os pontos HP que cada nanorobô deveria preencher. Foram criados sempre 7 nanorobôs, quantidade que normalmente era dedicada a este tipo de objetivo pelos times *CodeSdruxulators* e *CodeSdruxulators2007*. Uma fator importante para esse tipo de teste é justamente a definição de qual nanorobô preenche qual ponto HP. Supondo que, no mapa da Figura 5-3,

exista um nanorobô próximo ao ponto A e um outro no canto inferior direito do mapa. É claro que aquele próximo do ponto A o preencheria mais rapidamente, uma vez que o outro deveria, inicialmente, percorrer quase todo o mapa até chegar ao local do *Hoshimi Point*.

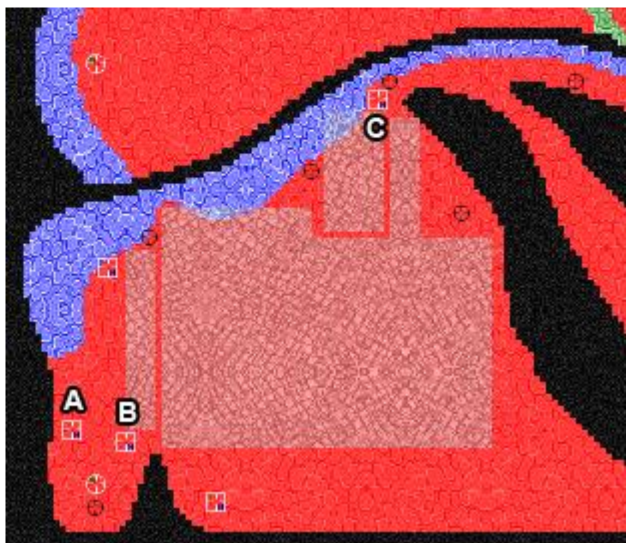

Figura 5-3: Ordem para percorrer pontos do tipo HP.

Conforme pode ser observado na Tabela 5-1, os resultados obtidos em todos os mapas foram exatamente os mesmos, o que comprova que o algoritmo *AStarPlanner* gerou planos eficientemente sem influenciar no desempenho. É importante destacar que, o simulador trabalha com um sistema de turnos e que cada turno tem duração predefinida. Isso quer dizer que todos os cálculos de todos os nanorobôs de um time, devem ser feitos nesse tempo. Caso algum cálculo seja iniciado e não terminado dentro do turno, o time perde o próximo turno, não podendo realizar nenhuma outra ação a não ser cálculos iniciados e não terminados no turno anterior. Por exemplo, supondo que um algoritmo inicie seu processamento no turno 11 e demore 300 milissegundos para retornar um resultado. Nesse caso, o turno 12 será usado apenas para concluir o cálculo deste algoritmo, não sendo permitido nenhum novo cálculo nesse turno.

Tabela 5-1: Tabela comparativa de resultados dos times *CodeSdruxulators* e *CodeSdruxulators2007*.

Mapa	Quantidade de pontos obtidos por *CodeSdruxulators*	Quantidade de pontos obtidos por *CodeSdruxulators2007*
A1	1720	1720
A2	4245	4245
A3	1985	1985
A4	2350	2350

Tabela 5-1: Tabela comparativa de resultados dos times *CodeSdruxulators* e *CodeSdruxulators2007 (continuação).*

Mapa	Quantidade de pontos obtidos por *CodeSdruxulators*	Quantidade de pontos obtidos por *CodeSdruxulators2007*
SC3	2830	2830
SC4	2540	2540
Mashroom D	1760	1760

Nos testes realizados, utilizou-se um turno com 0,2 s ou 200 milissegundos (o mesmo tempo usado nas edições de 2006 e 2007 da *Imagine Cup*).

Ressalta-se que os testes foram feitos com o tipo de objetivo que gera a maior quantidade de ações por plano – preencher um ponto do tipo HP. Nos testes realizados, todos os planos tinham 20 ações, ou seja, não era um plano simples de ser gerado. Foram feitos testes com alguns mapas para estimar o tempo médio para geração de um plano. Na Tabela 5-2**Erro! Fonte de referência não encontrada.** têm-se dados sobre os tempos de todos os planos gerados por todos os robôs para resolver metas em três mapas diferentes. Os dados são: a quantidade de planos gerados, o tempo total para geração dos planos e a média de tempo para cada mapa.

Tabela 5-2: Tempo médio de geração de planos.

Mapa	A2	A3	SC4
Quantidade de Planos	36	32	41
Tempo Total (s)	0,223969	0,198263	0,254357
Média (s)	**0,006221**	**0,006196**	**0,006204**

Constatou-se que, em média, cada plano era gerado em, aproximadamente, 0,006s ou 6 milissegundos. Desta forma, como o turno tinha 200 milissegundos, poderia-se trabalhar com 33 nanorobôs atendendo 33 metas ao mesmo tempo (200/6 = 33,333).

Os testes foram realizados em um *notebook* HP, modelo dv6898, com processador Intel Core 2 Duo 1.83MHz, 4Gb de memória RAM DDR2 333MHz e 2 Mb de memória *cache*.

5.3.2 Testes de Qualidade dos Planos Gerados

Para verificar a qualidade dos planos gerados, foram realizados testes com os mais diversos objetivos possíveis. Ao invés de testar um time inteiro, apenas um nanorobô era testado por vez. A este nanorobô era proposto uma meta e, após gerado o plano, este era analisado. Como o estado inicial do nanorobô era determinante na escolha das ações que seriam usadas para atingir o objetivo, para cada meta, foram testados vários estados iniciais.

A seguir tem-se a Tabela 5-3 com descrição dos testes realizados. Na primeira coluna é apresentado o número do teste, na segunda o meta proposta, na terceira o estado inicial do nanorobô e, na última, a seqüência de ações para cumprir a meta.

Tabela 5-3: Testes de qualidade de planos gerados.

Teste	Meta	Estado inicial	Plano Gerado
1	Ir para posição (130,34)	Posição = (100,23) Tipo posição = comum	Goto (130, 34)
2	Ir para posição (130,34)	Posição = (130,34) Tipo posição = comum	Nulo
3	Transferir enzimas para HP na posição (115, 55)	Posição = (45, 103) Quantidade de enzimas em estoque = 0 Tipo posição = comum	GotoType (AZN) → Collect → Goto (115,34) →Transfer
4	Transferir enzimas para HP na posição (115, 55)	Posição = (45, 103) Quantidade de enzimas em estoque = 20 Tipo posição = comum	Goto → Transfer
5	Transferir enzimas para HP na posição (115, 55)	Posição = (92, 70) Quantidade de enzimas em estoque = 0 Tipo posição = AZN	Collect → Goto (115, 55) → Transfer
6	Transferir enzimas para HP na posição (115, 55)	Posição = (92, 70) Quantidade de enzimas em estoque = 20 Tipo posição = AZN	Goto(115,55) → Transfer
7	Transferir enzimas para HP na posição (115, 55)	Posição = (115,55) Quantidade de enzimas em estoque = 0 Tipo posição = HP	Transfer
8	Matar nanobot do Pierre	Não relevante	Fire
9	Matar nanobot do Pierre na posição (150, 33)	Posição = (150, 35)	Fire
10	Matar nanobot do Pierre na posição (150, 33)	Posição = (100,22)	Goto(150, 33) → Fire
11	Ter 20 AZN em estoque e ir para posição (140, 23)	Posição = (92, 70) Quantidade de enzimas em estoque = 0 Tipo posição = AZN	Collect → Goto (140, 23)

Tabela 5-3: Testes de qualidade de planos gerados.

Teste	Meta	Estado inicial	Plano Gerado
12	Ter 20 AZN em estoque e ir para posição (140, 23)	Posição = (45, 103) Quantidade de enzimas em estoque = 0 Tipo posição = comum	GotoType (AZN) → Collect → Goto (140, 23)
13	Ter 20 AZN em estoque e ir para posição (140, 23)	Posição = (45, 103) Quantidade de enzimas em estoque = 20 Tipo posição = comum	Goto (140,23)
14	Transferir enzimas para qualquer ponto HP	Posição = (115, 55) Tipo da posição = comum Quantidade de enzimas em estoque	GotoType (AZN) → Collect → GotoType (HP) → Transfer

Antes de comentar os resultados, é preciso explicar o que cada ação faz. Uma ação do tipo *Goto*, envia um determinado nanorobô para uma posição específica. Por exemplo, *Goto* (100,34), faz com que um nanorobô se movimente para a posição (100,34).

Existe também a ação *GotoType* em que o importante não é a posição em si, mas o tipo da posição que será alcançada. Supondo a existência de uma ação *GotoType* HP, o nanorobô que for comandando por ela deverá procurar o ponto HP mais próximo de sua localização e se deslocar para lá. Note que, consequentemente, sua posição poderá ser alterada (caso ele já não esteja sobre o ponto HP mais próximo).

Uma ação do tipo *Collect* faz com que sejam coletadas enzimas AZN. Entretanto, essa ação somente pode ser executada sobre um ponto AZN. Além disso, o nanorobô coletor deve possuir espaço suficiente para coletar enzimas, visto que ele possui uma capacidade máxima de armazenamento.

Por fim, uma ação do tipo *Transfer* faz com que sejam transferidas enzimas do nanorobô coletor para um nanorobô do tipo *NanoNeedle* localizado sobre um ponto HP. Desta forma, essa ação só é possível se o nanorobô coletor possuir enzimas, se estiver sobre um ponto HP e se, nesse ponto, existir um *NanoNeedle* do seu time.

É importante salientar que um mesmo tipo de ação pode resultar em diferentes ações concretas. Pode-se ter, por exemplo, uma ação do tipo *Goto* para diferentes posições do mapa. A melhor ação concreta é aquela em que a movimentação ocorre em um menor tempo.

A importância desse tipo de teste é verificar se, em cada meta, a melhor plano foi gerado. Conforme pode ser observado na Tabela 5-3, todos os planos foram gerados de forma que nenhuma ação desnecessária foi incluída em plano algum, somente aquelas estritamente necessárias foram escolhidas. A seguir, serão detalhados alguns casos.

No teste 1, um nanorobô deveria ocupar a posição (130, 34). Como este nanorobô estava na posição (100,23), a única ação que deve ser executada é uma ação de movimentação *Goto*. O teste 2 é praticamente idêntico, mas o nanorobô já está na posição desejada, logo um plano com uma quantidade vazia de ações é gerado. Nesse caso, nenhuma ação precisa ser executada.

No teste 3, tem-se uma tarefa de transferência de enzimas para um ponto HP específico. Como o nanorobô não possuía enzimas, uma ação do tipo *GotoType* (AZN) era necessária para que ele se deslocasse para o ponto AZN mais próximo dele. Após o deslocamento, o próximo passo era coletar enzimas. Desta forma, corretamente, uma ação do tipo *Collect* foi escolhida. Após coletadas as enzimas, era necessário dirigir-se para o ponto HP específico. Assim, uma ação do tipo *Goto* foi selecionada. Finalmente, para que as enzimas pudessem ser transferidas, uma ação *Transfer* foi requerida.

Note que o plano muda totalmente caso o nanorobô já possua enzimas. Essa situação é verificada no teste de número 4. Nesse caso, como não é necessária a coleta de enzimas, conseqüentemente, não é necessário o deslocamento para um ponto AZN, logo, as ações *GotoType* e *Collect* não fizeram parte do plano.

No teste 14, tem-se uma variação do teste 3 em que, foi estipulado apenas que o nanorobô deveria transferir enzimas, não sendo especificada a posição em que isso deveria acontecer. Logo, ao invés da terceira ação ser do tipo *Goto*, o que enviaria o nanorobô para uma posição predefinida, uma ação do tipo *GotoType* (HP) o faz deslocar para o ponto HP mais próximo dele.

No teste 12, tem-se uma meta que envolve posição final e coleta de enzimas. Mais uma vez, o *AStarPlanner* gerou o plano com a menor quantidade de ações possíveis para atingir a meta. Como o nanorobô não possuía nenhum enzima, o primeiro passo deveria ser coletar enzimas. Para tal, foram geradas ações do tipo *GotoType* (AZN) e *Collect.* Por fim, uma ação de movimentação para uma posição específica foi selecionada (*Goto*).

Por fim, no teste 13, esperava-se que a mesma meta do teste 12 fosse atingida. A diferença é que o nanorobô já possuía enzimas em estoque. Logo, a única ação necessária, e que foi corretamente selecionada, foi uma ação do tipo *Goto.*

Conforme apresentado, em todas as situações propostas o algoritmo *AStarPlanner* gerou a melhor seqüência de ações para atingir as metas. Essa determinação da melhor seqüência foi analisada, caso a caso, e manualmente.

6. Multiagente *AStarPlanner*

O outro objetivo deste trabalho é verificar a possibilidade de usar o algoritmo *AStarPlanner* em um planejamento multiagentes baseado em um sistema de leilão. Ressalta-se que não foi encontrada na literatura nenhuma tentativa de utilização de um sistema de leilão em algoritmos de planejamento em jogos de estratégia. A grande questão era, obviamente, o desempenho, ou seja, seria possível executar o *AStarPlanner* várias vezes para uma mesma meta e, mesmo assim, obter um resultado final em um tempo aceitável? E no caso de várias metas simultâneas, ainda assim o desempenho não seria comprometido?

Conforme mencionado na seção 5.3.1, de acordo com os testes realizados, seria possível trabalhar com 33 metas ao mesmo tempo em um único turno do ambiente *Project Hoshimi.* Isso quer dizer que, se fossem utilizados 7 nanorobôs (quantidade de nanorobôs normalmente usada em metas que envolvem coleta de enzimas no time *CodeSdruxulators*), seria possível trabalhar com, aproximadamente, 5 metas simultâneas. Isto porque, cada nanorobô deveria gerar um plano para cada meta. Logo, multiplicando um valor pelo outro, tem-se uma quantidade de 35 planos gerados em 200 milissegundos (valor de cada turno usado na competição *Imagine Cup* na categoria *Project Hoshimi* nos anos de 2006 e 2007).

6.1 O Algoritmo

Nessa seção será apresentado o *MultiAStarPlanner*, um algoritmo de planejamento multiagentes com uso de um sistema de leilões. A idéia básica é, para cada meta gerada, são identificados agentes capazes de resolvê-la e uma solicitação de oferta é enviada para cada um deles. Eles, por sua vez, fazem seus cálculos e enviam suas ofertas para um agente central que identifica o melhor lance e define o vencedor do leilão. É importante ressaltar que, depois de gerado o plano, é calculado o custo de sua execução em uma unidade de tempo. Isto é feito tomando-se cada ação do plano e identificando-se o tempo necessário para sua realização. O somatório destes valores fornece o custo total de execução do plano.

De forma detalhada, tem-se um laço de repetição que percorre todas as metas existentes em um dado momento, independente de seu estado. Para cada meta, o primeiro teste é justamente saber se a meta já foi atingida ou não. Se ela já foi atingida, não há nada a fazer. Caso contrário, um método busca por uma lista de agentes capazes de solucionar tal meta. Esse passo é importante para evitar que um agente participe desnecessariamente de um leilão. Posteriormente, serão apresentados detalhes sobre sua implementação por meio do *Project Hoshimi*.

A Figura 6-1 apresenta o pseudocódigo do *MultiAStarPlanner*.

```
Para cada m em Metas
{
        Se (m.estado = "Não Realizado")
        {
                agentes ← GetListaAgentesCapazesResolverMeta();
                Para cada a em agentes
                {
                        Se a não possui meta alguma
                        {
                                p ← GeraPlano();
                                AgentesPlanos.Adiciona(a,p);
                        }
                        Senão
                        {
                                e ← GetEstadoAposRealizacaoUltimaMeta();
                                p ← GeraPlanoUsandoEstadoMundo(e);
                                AgentesPlanos.Adiciona(a,p);
                        }
                }
                a' ← GetAgentePlanoMenorCusto(AgentesPlanos);
                a'.AdicionaMeta(m);
                m .estado = "Em progresso";
        }
}
```

Figura 6-1: Pseudocódigo do algoritmo *MultiAStarPlanner*.

Para cada agente *a* escolhido, verifica-se se ele já possui alguma meta. Caso não possua, é gerado um plano *p* para a meta em questão e este plano *p*, juntamente com o agente *a*, são armazenados em uma lista denomina-se *AgentesPlanos*.

Se o agente já possuir uma meta, recupera-se o estado do mundo resultante da execução do plano que resolve a última meta de sua lista. Isso é importante uma vez que para calcular a seqüência de ações que resolverá a meta *m*, ele deve considerar o estado do mundo após a execução de seu último plano e não o estado atual. Da mesma forma, *a* e *p* são armazenados na lista *AgentesPlanos*.

A lista *AgentesPlanos* pode ser implementada e mantida em um agente central, que terá como tarefa a realização do próximo passo do algoritmo: identificar o agente *a*' com plano de menor custo de execução. Por fim, o estado da meta *m* é alterado para "Em Progresso" e ela é adicionada a sua lista de metas do agente com plano de menor custo.

6.2 *MultiAStarPlanner* no *Project Hoshimi*

Após criado o algoritmo, o próximo passo foi a sua implementação no ambiente *Project Hoshimi.* Como coletar e transferir enzimas é sempre importante para um time de nanorobôs, quanto maior a quantidade de *NanoNeedles* preenchidas melhor. Logo, em tese, caso um mapa tenha x pontos HP, existirão x metas de preenchimento de *NanoNeedles*. Contudo, não se deve gerar as x metas de uma vez devido a vários fatores. O primeiro deles diz respeito ao desempenho, isto é, não é possível gerar 20 ou mais leilões ao mesmo tempo para um time de 7 ou mais nanorobôs coletores.

Além disso, é muito arriscado assumir que todos os pontos do tipo HP serão preenchidos por um único time. Isso porque, como já mencionado, caso um *NanoNeedle* de um time A for criado sobre um ponto HP específico, um outro time B não poderá criar um *NanoNeedle* na mesma posição. Desta forma, é mais conveniente a geração de metas de preenchimento de pontos HP seguindo um certo intervalo de tempo, ou seja, em um primeiro momento um número x de metas é gerado; após um tempo y, outras x metas são geradas, e assim por diante. Essa técnica foi adotada durante a implementação.

Para a definição da lista de agentes capazes de realizar uma determinada meta, foi utilizado o seguinte critério: foram escolhidos apenas os nanorobôs capazes de executar ações que, em sua lista de efeitos, possuíam os símbolos presentes na meta proposta. Cada símbolo representa uma propriedade de um estado do mundo. Se uma ação não possui um determinado símbolo na lista de efeitos, quer dizer que ela não pode alterar seu valor. Como a idéia de uma ação é transformar os valores de um estado corrente para valores desejados em uma meta, é coerente excluir ações que não são capazes de alterar esses valores.

Para facilitar a busca, foi criada em cada nanorobô, uma tabela *hash* de tal forma que a chave era um símbolo e o valor uma coleção de ações que possuíam aquele símbolo

em sua lista de efeitos. A tabela *hash* possuía um número de entradas igual ao número de símbolos utilizados para descrever um estado do mundo.

A Tabela 6-1 apresenta os principais símbolos usados para representar os estados do mundo relacionados com as metas básicas do *Project Hoshimi.* Foram criados ainda símbolos para trabalhar com metas que foram definidas de acordo com estratégias geradas para a competição *Imagine Cup*. Por exemplo, um símbolo *FollowAI* foi usado para uma meta que estipulava que um nanorobô do tipo *NanoExplorer* deveria seguir o *NanoAI* do time adversário.

Tabela 6-1: Símbolos e ações que os possuem na lista de efeitos.

Símbolo	Coleção de ações
IsNeedleFull	Transfer
IsNeedlePartialyFull	Transfer
IsEnemyDead	Atack
Position	Goto, GotoType
PositionType	Goto, GotoType
HasAZN	Collect, Transfer

Ressalta-se que, além dos símbolos que descrevem o estado do mundo, outros valores foram necessários para a execução das ações. Por exemplo, para descobrir qual seqüência de ações é suficiente para atingir uma meta de preenchimento de um *NanoNeedle*, não é necessário saber onde serão coletadas ou onde serão depositadas as enzimas. O trabalho do planejador é identificar quais ações são importantes. Ou seja, ele trabalha com fatos do tipo: é necessário preencher um *NanoNeedle* e o nanorobô não deve possuir enzimas em estoque. Logo, ele deve procurar por ações que façam com que enzimas sejam coletadas e depois por ações que possibilitem sua transferência. Contudo, após identificadas as ações, alguns valores são atribuídos e armazenados no *Blackboard* do nanorobô para uso no momento de sua execução. Por exemplo, uma ação do tipo *GotoType*, altera a posição do nanorobô. Logo, para sua execução, é necessário saber quais as coordenadas x e y devem ser atingidas pelo nanorobô.

Na implementação dos métodos **GeraPlano** e **GeraPlanoUsandoEstadoMundo**, a diferença, conforme já destacado na seção 6.1, é o uso ou não do estado do mundo atual. O *AStarPlanner*, algoritmo responsável pela geração do plano, trabalha com nós do tipo *AStarPlannerNode*. Este tipo de nó possui várias propriedades, dentre elas a *WorldState* que armazena o estado do mundo. No caso do método **GeraPlano**, é usado o estado atual

do robô. Já no caso do **GeraPlanoUsandoEstadoMundo**, é utilizado o parâmetro deste método, que possui o último estado do mundo que o nanorobô assumirá antes da execução da meta em questão. Isto é, supondo que o nanorobô não possua meta alguma, a primeira vez será invocado o método GeraPlano. Se este nanorobô entrar em um outro leilão antes do término da execução deste plano, deverá ser invocado o método **GeraPlanoUsandoEstadoMundo**, e deverá ser passado como parâmetro o estado do mundo posterior a execução do primeiro plano. Se existissem duas metas na lista de metas do nanorobô, deveria ser usado o estado do mundo resultante da execução do último plano responsável pela execução da última meta da lista de metas.

6.3 Testes realizados

Nesta seção serão discutidos os testes realizados com o algoritmo *MultiAStarPlanner*. Três tipos de testes foram feitos: testes considerando um número variável, e relativamente pequeno de nanorobôs, testes com turnos menores que 200 milissegundos e testes com uma grande quantidade de robôs. Em cada um destes testes foram analisadas questões como desempenho e qualidade dos planos gerados.

Para averiguar estas questões, os resultados do *MultiAStarPlanner* foram comparados com os resultados de um time com um simples algoritmo denominado Simple*AStarPlanner*. No Simple*AStarPlanner*, os planos individuais de cada robô do time também eram definidos pelo *AStarPlanner*. Contudo, não existe neste, a noção de leilão. Assim, quando uma meta é gerada, o primeiro nanorobô que não estiver fazendo coisa alguma, assumirá a tarefa de solucioná-la. O termo primeiro é usado pois, podem existir vários nanorobôs ociosos, logo, o primeiro que perceber a existência da meta será aquele que tentará atingí-la.

O Simple*AStarPlanner* não trabalha com os nanorobôs mais aptos, ou seja, não escolhe os que resolvem mais rapidamente uma meta, mas, por outro lado, nunca desperdiça turnos. É importante lembrar que, como mencionado na seção 5.3.1, quando um algoritmo não consegue ser executado dentro do tempo de um turno, o turno seguinte é desperdiçado. Esta questão é muito importante, principalmente, nos testes feitos com turnos menores que 200 milissegundos.

Com relação aos testes em si, da mesma forma que nos testes com o *AStarPlanner*, foram definidas restrições para impedir que fatores não relacionados com o algoritmo *MultiAStarPlanner* influenciassem nos resultados. Basicamente, foram as mesmas restrições usadas para o *AStarPlanner*.

Mais uma vez optou-se por trabalhar com apenas um tipo de teste. Foi escolhido o tipo "preencher totalmente um ponto HP" por ser este o objetivo a gerar os maiores planos.

Novamente, optou-se pela criação do nanorobô *NanoAI* sempre na mesma posição e foi imposta uma rota fixa que seria usada por ele para percorrer pontos do tipo HP. Testaram-se dois tipos de rotas fixas. No primeiro tipo, foi criado um algoritmo para gerar uma seqüência de pontos que maximizasse a quantidade de pontos HP percorridos em uma determinada quantidade de tempo. Este algoritmo, geralmente escolhia pontos HP próximos uns dos outros. Foi criado também um algoritmo para gerar uma rota totalmente aleatória. Assim, o primeiro ponto HP gerado poderia, por exemplo, ser aquele localizado ao extremo norte do mapa e o segundo ao extremo sul.

Os testes foram realizados com apenas um time no mapa. Esta decisão foi tomada, pois quando dois times estão no mesmo mapa, o que geralmente define o vencedor não é o algoritmo de planejamento, mas outros tipos de algoritmos, tais como algoritmos para gerar o ponto de aparecimento do *NanoAI*, ou para gerar rota de preenchimento de pontos HP, ou para inserção de *NanoBlockers* para dificultar a navegação do time adversário. É claro que o planejamento é de suma importância, pois se as metas não forem executadas, o time não marca pontos. Entretanto, questões como posição inicial do *NanoAI* e rota para percorrer pontos do tipo HP são vitais para a vitória em uma competição com dois times no mesmo mapa.

O que aconteceria se fossem colocados os dois times no mesmo mapa seria: os dois sairiam do mesmo ponto inicial e percorreriam a mesma rota. Desta forma, o primeiro *NanoAI* a ser criado, chegaria primeiro no primeiro ponto do tipo HP. Ele colocaria um *NanoNeedle* nesta posição e o *NanoAI* do time adversário seria forçado a perseguir o próximo ponto HP da lista. Ele obviamente chegaria primeiro obrigando o outro *NanoAI* a procurar o próximo ponto HP e, assim por diante. Assim, o time que desse a sorte de passar por pontos HP mais próximos de pontos AZN coletaria enzimas mais rapidamente, tendo uma maior chance de ganhar a partida.

Desta vez, não foram definidos os nanorobôs que transportariam enzimas para cada ponto HP. Isto porque essa é justamente a tarefa dos algoritmos *MultiAStarPlanner* e Simple*AStarPlanner*: definir qual nanorobô deverá resolver cada meta.

Um outra diferença com relação aos testes do *AStarPlanner* é que, nos testes do *MultiAStarPlanner*, apesar se manterem constantes algumas características dos robôs coletores tais como, velocidade de movimentação, coleta e transferência de enzimas, em cada teste, utilizou-se nanorobôs com três capacidades de armazenamento diferentes. Para o *MultiAStarPlanner* isto é muito bom, pois ele tentará trabalhar sempre com os nanorobôs de maior capacidade. Para o Simple*AStarPlanner*, não faz a menor diferença, visto que não existe a noção de "o nanorobô mais apto para atingir um objetivo".

Com relação aos mapas usados, foram escolhidos sete com diferentes características. Todos eles se encontram no Apêndice A, localizado na página 82. Dentre eles, tem-se os quatro mapas usados na final mundial da edição de 2007 da competição *Imagine Cup*, e, dois mapas usados em competições de aquecimento da *Imagine Cup* 2007. Procurou-se trabalhar com mapas com grandes e pequenas áreas contíguas de células de baixa densidade, mapas com grandes quantidades de áreas não atravessáveis e mapas com pequenas áreas não atravessáveis, mapas com pequenos e grandes aglomerados de pontos HP, mapas com grande e baixa quantidade de pontos HP, mapas que apresentavam células de todas as densidades e mapas que não possuíam essas características.

6.3.1 Testes com 3, 5 e 7 nanorobôs

O primeiro tipo de teste realizado foi aquele que utilizou tempo de turno igual a 200 milissegundos e quantidade máxima de nanorobôs destinados a objetivos do tipo coleta/transferência de enzimas igual a sete. Ou seja, situações semelhantes àquelas que os times *CodeSdruxulators* e *CodeSdruxulators2007* encontraram nas edições de 2006 e 2007 da *Imagine Cup*.

Conforme mencionado anteriormente, foram testados sempre, dois tipos de rotas: uma otimizada visando maximizar a quantidade de pontos HP percorridos bem como o tempo usado para realizar o percurso, e outra rota totalmente aleatória.

A Tabela 6-2 apresenta os testes realizados usando a rota otimizada. Em destaque na cor cinza claro, estão os testes em que o *MultiAStarPlanner* obteve resultado superior.

Os valores estão expressos em termos da quantidade de pontos obtidos em uma partida. Ganha-se pontos pela realização de objetivos e pela coleta e transferência de enzimas.

Tabela 6-2: Testes *MultiAStarPlanner*/Simple*AStarPlanner* com rota otimizada.

Rota Otimizada								
	Simple*AStarPlanner*				*MultiAStarPlanner*			
	Quantidade de nanorobôs				Quantidade de nanorobôs			
Mapa	3	5	7	**Total**	3	5	7	**Total**
A1	880 pts.	1090 pts.	1620 pts.	**3590 pts.**	880 pts.	1130 pts.	1595 pts.	**3605** pts.
A2	2930 pts.	3550 pts.	3850 pts.	**10330 pts.**	2960 pts.	3665 pts.	3975 pts.	**10600 pts.**
A3	1125 pts.	1425 pts.	1610 pts.	**4160 pts.**	1050 pts.	1350 pts.	1785 pts.	**4185 pts.**
A4	1555 pts.	1920 pts.	2210 pts.	**5685 pts.**	1565 pts.	1955 pts.	2150 pts.	**5670 pts.**
SC3	2355 pts.	2565 pts.	2675 pts.	**7595 pts.**	2255 pts.	2605 pts.	2710 pts.	**7570 pts.**
SC4	1435 pts.	1670 pts.	2160 pts.	**5265 pts.**	1745 pts.	2025 pts.	2195 pts.	**5965 pts.**
Mashroom D	1415 pts.	1570 pts.	1640 pts.	**4625 pts.**	1445 pts.	1710 pts.	1660 pts.	**4815 pts.**
Total	**11695 pts.**	**13790 pts.**	**15765 pts.**	**41250 pts.**	**11900 pts.**	**14440 pts.**	**16070 pts.**	**42410 pts.**

Dos sete mapas testados, o *MultiAStarPlanner* obteve desempenho superior em cinco deles, perdendo apenas em dois, e mesmo assim, com uma diferença irrisória: 15 pontos no mapa A4 e 25 no mapa SC3. Considerando o total geral de pontos, o *MultiAStarPlanner* venceu o Simple*AStarPlanner* por 42410 a 41250 pontos. A diferença é pequena, contudo ela poderia ser maior devido a alguns fatores que serão comentados na seção 7.1.

A Figura 6-2 mostra os mapas A4 e SC3.

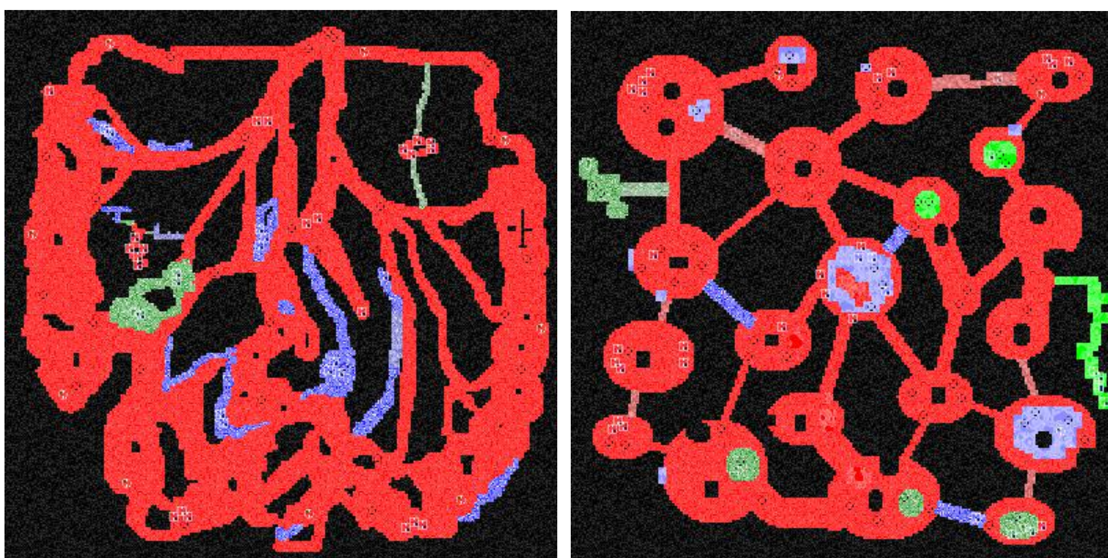

Figura 6-2: Mapa A4 (à esquerda) e SC3 (à direita).

Analisando a figura, é possível deduzir porque o desempenho nesses mapas não foi tão bom. A questão é que, de todos os mapas testados, estes são os que apresentam maiores áreas não atravessáveis na parte central. Isso quer dizer que, o custo de execução dos planos não é calculado adequadamente. O motivo disso é que, no *MultiAStarPlanner*, no momento do cálculo do custo de uma ação *Goto* ou *GotoType*, é utilizada a distância Manhattan. A Figura 6-3 exemplifica bem o erro.

Por exemplo, supondo que um nanorobô deve calcular o custo de movimentação do ponto “origem” até o ponto “destino”, o cálculo será realizado considerando a rota A. Entretanto, deveria considerar a rota B. Ou seja, isso influencia diretamente na escolha dos nanorobôs que devem tentar atingir as metas.

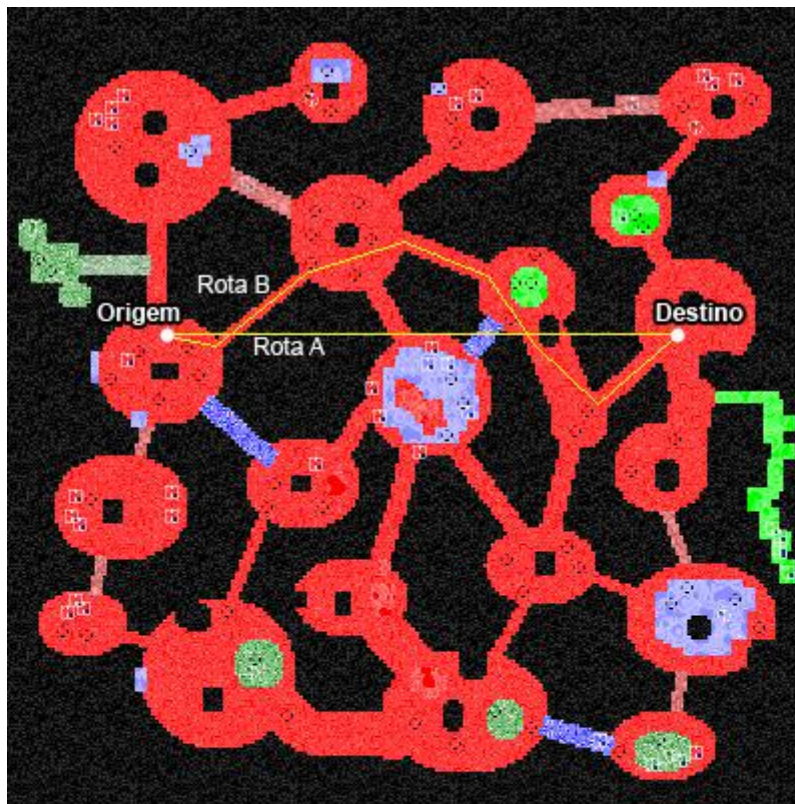

Figura 6-3: Rotas A e B.

Supondo ainda a existência de dois nanorobôs X e Y e considerando o fato do uso da distância Manhattan para definir o custo de ações de movimentações, cálculos errados podem enviar os nanorobôs para destinos incorretos. Se o nanorobô X fosse a melhor opção para cumprir uma meta U e o nanorobô Y o indicado para atingir a meta V, erros na definição do custo das ações *Goto* e *GotoType* poderiam enviar X para cumprir a meta V.

Por conseqüência, poderia fazer com que Y tentasse atingir a meta U, ou seja, um único erro influenciaria as ações de dois nanorobôs.

Quando analisados os resultados em termos de quantidade de nanorobôs usados em cada mapa, tem-se que o *MultiAStarPlanner* sempre obteve maior desempenho. Com três nanorobôs fez 11900 pontos contra 11695 do Simple*AStarPlanner*; com cinco fez 14440 pontos, contra 13790; com sete fez 16070 pontos contra 15765.

A Tabela 6-3 apresenta os testes realizados usando a rota aleatória. Mais uma vez, em destaque na cor cinza claro, estão os testes em que o *MultiAStarPlanner* obteve resultado superior.

Nota-se que, desta vez que o *MultiAStarPlanner* não obteve um resultado tão bom comparando os mapas isoladamente, ou a quantidade de nanorobôs. Mesmo assim, o resultado geral, que soma a pontuação de todos os testes incluindo todas as quantidades de nanorobôs e todos os mapas, o placar ainda é favorável para o *MultiAStarPlanner*, ou seja, 110 pontos mais do que o Simple*AStarPlanner*.

Tabela 6-3: Testes *MultiAStarPlanner*/Simple*AStarPlanner* com rota aleatória.

Rota Aleatória								
	Simple*AStarPlanner*				*MultiAStarPlanner*			
	Quantidade de nanorobôs				Quantidade de nanorobôs			
Mapa	3	5	7	**Total**	3	5	7	**Total**
A1	680 pts.	680 pts.	680 pts.	**2040 pts.**	690 pts.	690 pts.	690 pts.	**2070 pts.**
A2	1870 pts.	1920 pts.	1870 pts.	**5660 pts.**	1920 pts.	1840 pts.	1870 pts.	**5630 pts.**
A3	1250 pts.	1300 pts.	1250 pts.	**3800 pts.**	1285 pts.	1340 pts.	1350 pts.	**3975 pts.**
A4	1285 pts.	1500 pts.	1360 pts.	**4145 pts.**	1275 pts.	1360 pts.	1380 pts.	**4015 pts.**
SC3	1795 pts.	1780 pts.	1730 pts.	**5305 pts.**	1730 pts.	1740 pts.	1750 pts.	**5220 pts.**
SC4	1545 pts.	1630 pts.	1495 pts.	**4670 pts.**	1430 pts.	1595 pts.	1535 pts.	**4560 pts.**
Mashroom D	1375 pts.	1510 pts.	1630 pts.	**4515 pts.**	1415 pts.	1630 pts.	1630 pts.	**4675 pts.**
Total	**9800 pts.**	**10320 pts.**	**10015 pts.**	**30135 pts.**	**9745 pts.**	**10195 pts.**	**10205 pts.**	**30145 pts.**

Analisando os mapas, em apenas três deles o *MultiAStarPlanner* obteve melhor resultado. Quando a comparação é feita em termos da quantidade de nanorobôs utilizados, observou-se que o *MultiAStarPlanner* apresentou desempenho superior ao *AStarPlanner,* apenas utilizando sete nanorobôs.

Estes resultados se devem ao fato de que, como a ordem de preenchimento de pontos é aleatória, em vários casos, o primeiro ponto HP se encontrava muito distante do segundo, que por sua vez se encontrava distante do terceiro, e assim por diante. Quanto mais distantes os pontos, pior a qualidade do custo de ações do tipo *Goto* e *GotoType.*

6.3.2 Testes com 20 nanorobôs

Uma vez realizados os testes com uma quantidade de nanorobôs próxima da que seria utilizada em uma competição, decidiu-se testar um caso extremo: 20 nanorobôs cumprindo metas do tipo "preenchimento de um ponto HP". O teste foi feito, até porque, em um time real, além dos sete nanorobôs envolvidos em metas de coleta/transferência de enzimas, outros nanorobôs estariam presentes no time. Apesar de eles trabalharem em metas cujos planos capazes de atingi-las serem compostos por poucas ações e demandarem pouco tempo de cálculo, deveriam ser considerados.

Novamente, foram feitos testes com uma rota otimizada e com uma rota aleatória. Os resultados para testes com rota otimizada são apresentados na Tabela 6-4.

Tabela 6-4: *MultiAStarPlanner*/SimpleStarPlanner com 20 nanorobôs c/ rota otimizada.

Rota Aleatória		
20 nanorobôs		
Mapa	**Simple*AStarPlanner***	***MultiAStarPlanner***
A1	2300 pts.	2450 pts.
A2	4780 pts.	4780 pts.
A3	4260 pts.	4580 pts.
A4	4680 pts.	4680 pts.
SC3	4150 pts.	3905 pts.
SC4	3930 pts.	4060 pts.
Mashroom D	1760 pts.	1760 pts.

Em cinza claro, estão os testes em que o *MultiAStarPlanner* apresentou resultado superior ao Simple*AStarPlanner*. Em um tom de cinza ainda mais claro com um padrão pontilhado, os resultados iguais. Em sete mapas, houve apenas um resultado não positivo para o *MultiAStarPlanner*. Nos outros seis, três foram empates e em três o *MultiAStarPlanner* fez mais pontos que o Simple*AStarPlanner*. Isto quer dizer que, mesmo perdendo alguns turnos, no geral o uso do *MultiAStarPlanner* ainda é melhor. Isto acontece porque, no Simple*AStarPlanner*, todos os nanorobôs, independentemente de sua melhor qualificação para resolver planos, são usados. No *MultiAStarPlanner*, sempre que possível, nanorobôs com maior capacidade de armazenamento são utilizados. Nos testes com 20

nanorobôs, 12 têm capacidade de armazenamento igual a 20, 5 tem capacidade de armazenamento igual a 25 e 3 têm capacidade de armazenamento igual a 50.

No caso de rotas não otimizadas, os resultados para o *MultiAStarPlanner* são ainda melhores. A Tabela 6-5 apresenta os valores.

Tabela 6-5: *MultiAStarPlanner*/SimpleStarPlanner com 20 nanorobôs c/ rota aleatória.

Rota Aleatória		
20 nanorobôs		
Mapa	**Simple*AStarPlanner***	***MultiAStarPlanner***
A1	840 pts.	865 pts.
A2	2360 pts.	2540 pts.
A3	1680 pts.	2010 pts.
A4	1860 pts.	2120 pts.
SC3	2040 pts.	2160 pts.
SC4	2440 pts.	2560 pts.
Mashroom D	1760 pts.	1760 pts.

Para rotas aleatórias, em apenas um mapa houve empate, no restante, vitória do *MultiAStarPlanner*. Ressalta-se que é uma rota aleatória no sentido de definição do percurso de pontos HP, ou seja, não segue um critério de otimização de percurso. Ela continua sendo uma rota fixa, ou seja, se o algoritmo for executado 300 vezes, nas 300 a rota será a mesma.

Um dado interessante é que, mesmo com os erros de cálculo de custo em operações de movimentação, o algoritmo *MultiAStarPlanner* quando comparado ao Simple*AStarPlanner*, no caso de grande quantidade de nanorobôs, apresentou melhores resultados.

O resultado total do teste é o seguinte:

- para rotas otimizadas - 25860 para Simple*AStarPlanner* e 26215 para *MultiAStarPlanner*;
- rotas otimizadas – 12980 para Simple*AStarPlanner* e 14015 para *MultiAStarPlanner*;
- total – 38840 para Simple*AStarPlanner* e 40230 para *MultiAStarPlanner*

6.3.3 Testes com pequenos turnos

A idéia dos testes com pequenos turnos foi verificar o que aconteceria em situações extremas relacionadas ao tempo dos turnos. Quando o *Project Hoshimi* foi criado, um dos fatores que foram pensados foi um valor de tempo de turno suficiente para que os competidores implementassem seus algoritmos, mas não tão grande a ponto de facilitar a implementações de algoritmos não otimizados. O que foi feito neste trabalho foi reduzir o tempo de turno em 6 vezes, ou seja, ao invés de usar um turno de 200 milissegundos, foi utilizado um turno de, aproximadamente, 33 milissegundos.

Nos testes com rotas otimizadas, no geral, o *MultiAStarPlanner*, mesmo perdendo inúmeros turnos, conseguiu apresentar um resultado, ligeiramente melhor que o Simple*AStarPlanner*. A Tabela 6-6 mostra os resultados destes testes.

Já nos testes com rotas aleatórias, o resultado não foi tão bom. De vinte e um testes realizados, em apenas cinco o *MultiAStarPlanner* obteve melhores resultado. Considerando os valores totais por número de nanorobôs, nenhum resultado é favorável ao *MultiAStarPlanner*. Isso também acontece com os acumulados dos mapas. Em nenhum mapa o *MultiAStarPlanner* apresentou resultado superior ao *SimpleAStarPlanner*.

Tabela 6-6: Testes com turnos de 33 milissegundos e rotas otimizadas.

Rota Otimizada								
	Simple*AStarPlanner*				***MultiAStarPlanner***			
	Quantidade de nanorobôs				**Quantidade de nanorobôs**			
Mapa	3	5	7	**Total**	3	5	7	**Total**
A1	1335 pts.	2700 pts.	3090 pts.	**7125 pts.**	1500 pts.	2430 pts.	3225 pts.	**7155 pts.**
A2	5170 pts.	6225 pts.	7250 pts.	**18645 pts.**	5165 pts.	6280 pts.	7280 pts.	**18725 pts.**
A3	2585 pts.	4015 pts.	5620 pts.	**12220 pts.**	2340 pts.	4045 pts.	5675 pts.	**12060 pts.**
A4	3235 pts.	4895 pts.	6270 pts.	**14400 pts.**	3295 pts.	4905 pts.	6230 pts.	**14430 pts.**
SC3	3400 pts.	4230 pts.	4990 pts.	**12620 pts.**	3335 pts.	4210 pts.	4980 pts.	**12525 pts.**
SC4	3430 pts.	4715 pts.	5240 pts.	**13385 pts.**	3370 pts.	4890 pts.	5140 pts.	**13400 pts.**
Mashroom D	1415 pts.	1570 pts.	1640 pts.	**4625 pts.**	1445 pts.	1710 pts.	1660 pts.	**4815 pts.**
Total	**20570 pts.**	**28350 pts.**	**34100 pts.**	**83020 pts.**	**20450 pts.**	**28470 pts.**	**34190 pts.**	**83110 pts.**

A Tabela 6-7 mostra os valores dos testes com rotas aleatórias.

Tabela 6-7: Testes com turnos de 33 milissegundos e rotas aleatórias.

Rota Aleatória								
	Simple*AStarPlanner*				*MultiAStarPlanner*			
	Quantidade de nanorobôs				**Quantidade de nanorobôs**			
Mapa	3	5	7	**Total**	3	5	7	**Total**
A1	1895 pts.	1980 pts.	1890 pts.	**5765 pts.**	1520 pts.	1530 pts.	1530 pts.	**4580 pts.**
A2	4190 pts.	4370 pts.	4380 pts.	**12940 pts.**	3975 pts.	4480 pts.	4410 pts.	**12865 pts.**
A3	2840 pts.	2870 pts.	2710 pts.	**8420 pts.**	2670 pts.	2820 pts.	2840 pts.	**8330 pts.**
A4	3190 pts.	4045 pts.	4120 pts.	**11355 pts.**	3185 pts.	3945 pts.	4070 pts.	**11200 pts.**
SC3	3185 pts.	3450 pts.	3430 pts.	**10065 pts.**	3050 pts.	3280 pts.	3270 pts.	**9600 pts.**
SC4	3545 pts.	4490 pts.	4430 pts.	**12465 pts.**	3380 pts.	4295 pts.	4420 pts.	**12095 pts.**
Mashroom D	1375 pts.	1510 pts.	1630 pts.	**4515 pts.**	1415 pts.	1630 pts.	1630 pts.	**4675 pts.**
Total	**20220 pts.**	**22715 pts.**	**22590 pts.**	**65525 pts.**	**19195 pts.**	**21980 pts.**	**22170 pts.**	**63345 pts.**

Apesar do ruim desempenho do *MultiAStarPlanner* nesse tipo de teste, essa situação já era esperada, visto que o tempo do turno foi drasticamente reduzido. Entretanto, vale ressaltar que, mesmo nessas circunstâncias extremas, a diferença na pontuação total entre o SimpleAStarPlanner e o MultiAStarPlanner não foi superior a 3%.

7. Conclusão

Neste trabalho verificou-se a viabilidade de utilização de um algoritmo baseado no A* (HART, NILSSON, e RAPHAEL, 1968) para gerar planos em jogos de estratégia, e a possibilidade de uso desse algoritmo numa abordagem multiagente baseada em um sistema de leilões que seria usado para identificar o agente mais apto para cumprir uma determinada meta. A seguir serão feitas considerações finais sobre o trabalho desenvolvido e perspectivas de trabalhos futuros serão especificadas.

7.1 Considerações Finais

Nessa seção serão apresentadas reflexões sobre os resultados obtidos por ambos os algoritmos usados nesse trabalho: *AStarPlanner* e *MultiAStarPlanner*. O objetivo é verificar se as hipóteses propostas na seção 1.1 foram comprovadas.

A primeira hipótese era verificar a possibilidade de utilização de um algoritmo de planejamento baseado no A* em um jogo de estratégia. Conforme os resultados apresentados nas seções 5.3.1 e 5.3.2, pode-se dizer que é totalmente possível. O algoritmo não somente gerou os planos eficientemente, como também apresentou sempre a melhor seqüência de ações para atingir as metas propostas.

Algo que não pode ser totalmente mensurado é a melhora em termos de manutenção de código. Os resultados do *AStarPlanner* foram comparados com os resultados de um algoritmo que gerava planos baseados em máquinas de estado finito. Os valores foram os mesmos, contudo, algo que não está explícito é o esforço necessário para realizar alterações nos códigos. Supondo que novas metas sejam inseridas, novas ações sejam disponibilizadas, no caso do algoritmo baseado em máquinas de estado finito, todas as máquinas implementadas deverão ser revistas e, possivelmente alteradas. Ou seja, o trabalho é considerável.

No caso do *AStarPlanner*, a única coisa que deve ser feita é uma tradução das novas ações para a sintaxe simbólica usada pelo algoritmo. Como os planos são gerados

dinamicamente, não existe necessidade de revisão. Este é, sem dúvidas, um dos melhores diferenciais do algoritmo.

A hipótese dois propunha a utilização do *AStarPlanner* em um planejador multiagente baseado em leilões para jogos de estratégia. Como nada havia sido proposto antes na literatura, era algo que deveria ser investigado.

A conclusão foi que também é possível. Foram feitos inúmeros testes e documentados nas seções 6.3.1, 6.3.2 e 6.3.3. Foram propostos testes em situações semelhantes as que aconteceriam em uma partida da competição *Imagine Cup*, categoria *Project Hoshimi*. Além disso, foram feitos também testes extrapolando os limites idealizados para a competição, usando grande quantidade de nanorobôs executando metas que geravam longos planos e partidas com pequenos turnos.

No caso de testes com parâmetros semelhantes aos da competição, o *MultiAStarPlanner* quase sempre se saiu melhor que o *AStarPlanner*. Nos testes com rotas otimizadas, em cinco dos mapas e em todas as quantidades de nanorobôs, o resultado final foi melhor no *MultiAStarPlanner* que no ASimple*AStarPlanner*. Além disso, nos únicos dois mapas em que o *MultiAStarPlanner* não obteve melhor desempenho, o problema foi o erro no custo das ações de movimentação.

Uma opção para resolver esse problema, seria usar também o algoritmo A* para gerar, o melhor caminho entre dois pontos do mapa e assim, calcular de uma forma mais precisa, o custo das ações de movimentação. Contudo, para ação gerada que envolvesse uma ação do tipo *Goto* ou *GotoType*, ter-se-ia a execução desse algoritmo. Esse fato inviabilizaria sua utilização no ambiente *Project Hoshimi*. De fato, no time *CodeSdruxulators*, utilizava-se o A* para calcular a rota otimizada, ou seja, uma rota que maximizava a quantidade de pontos HP atingidos e o tempo para fazê-lo. Todavia, o algoritmo estava consumindo muitos turnos, logo outra abordagem teve que ser desenvolvida.

No caso de rotas aleatórias com três, cinco e sete nanorobôs, os resultados não foram tão bons quanto no caso das rotas otimizadas mas, mesmo assim, a soma total de pontos nos sete mapas, usando as três quantidade de nanorobôs, ainda foi ligeiramente melhor no *MultiAStarPlanner*.

No caso dos testes com uma grande quantidade de nanorobôs, constatou-se que independentemente de alguns poucos turnos desperdiçados, o resultado final foi ainda melhor que nos testes com poucos nanorobôs. Na verdade, o experimento com esse

parâmetro foi proposto para verificar o desempenho e, de forma inesperada, apresentou melhores resultados em termos de pontuação. Dos quatorze testes realizados, que envolveram rotas otimizadas e aleatórias e sete mapas diferentes, apenas em um deles, o *MultiAStarPlanner,* fez menos pontos que o Simple*AStarPlanner*.

Nos testes com turnos reduzidos, aparentemente, os resultados foram ruins para o *MultiAStarPlanner*. Contudo, uma análise mais cuidadosa mostra que, usando rotas otimizadas, ele apresentou uma maior pontuação total. Os testes feitos com rotas aleatórias foram feitos para medir desempenho e situações em que o algoritmo poderia se sair melhor ou pior. Em uma partida real, não há porque trabalhar com uma rota não otimizada. Desta forma, mesmo trabalhando com um turno 6 vezes menor do que o natural, ainda foi possível extrair resultados positivos.

Por fim, além do uso da distância Manhattan, outro fator contribuiu para problemas no desempenho do *MultiAStarPlanner* no *Project Hoshimi*. Quando um leilão é gerado, é definido o melhor nanorobô para realizar uma meta, ou seja, aquele que a realizaria no menor tempo. Porém, não é feita uma verificação de estimativa de término da partida. Supondo-se que, faltando 200 turnos para terminar a partida, um nanorobô A é designado para cumprir uma meta *m* e que esse agente ainda esteja realizando ações relacionadas à outra meta a qual será atingida após 210 turnos. Para resolver a nova meta a qual ele foi designado, o tempo estimado é de 200 turnos, ou seja, para finalizar todas as suas atividades, ele gastará cerca de 410 turnos. Considerando-se que no mapa existe outro nanorobô B que não está realizando tarefa alguma e que, caso ele fosse realizar a meta *m*, gastaria 500 turnos. Nessa situação, tem-se que, nem o nanorobô A, nem o nanorobô B realizarão a meta. O B não realizará porque quem ganharia o leilão seria A, pois 410 é menor que 500. Por outro lado, A também não realizará a meta pois, antes mesmo de acabar suas ações relacionadas a uma meta anterior a *m*, a partida terminará. Ou seja, tem-se uma situação em que um agente ficará ocioso, mesmo com metas a serem cumpridas e que nenhum outro agente no mapa conseguirá cumpri-la antes do término da partida.

Para resolver esse problema, basta verificar quantos turnos faltam para o final da partida. Algo do tipo, "se o tempo para iniciar a realização do plano for maior que o tempo restante da partida, esse nanorobô não poderá fazer parte do leilão". Isso não é possível, pois os planos têm erros no custo de execução das ações do tipo movimentação devido a utilização da distância Manhattan. Desta forma, quando um valor de 50 turnos é estipulado para se movimentar de um ponto A para outro B, este pode ser, na verdade, igual a 200.

Pode-se ainda, ter um deslocamento em um menor tempo, pois a distância Manhattan não considera fluxos sanguíneos. Isso quer dizer que, se entre esse ponto A e B existisse um fluxo no sentido A para B, o custo de deslocamento, em turnos, seria inferior a 50. Logo, poderiam ser excluídos do leilão nanorobôs aptos para realizar metas, e, em um caso extremo, seriam excluídos os nanorobôs mais aptos a realizá-las.

7.2 Trabalhos Futuros

Como trabalhos futuros, sugere-se a implementação de um algoritmo de escalonamento. Quando um nanorobô é designado para executar mais de duas metas, um algoritmo de escalonamento poderia reduzir o custo de execução, principalmente porque por trabalhar com rotas o tempo todo. Se ao invés de ir para os pontos A, B, C e D para resolver uma meta A, e depois para E,F,G para resolver uma meta B, e por fim, para os pontos H, I e J para resolver uma meta C, ele fosse primeiro para E,F,G, depois para H,I,J e, finalmente para A,B,C e D, o tempo total de execução poderia ser bem menor.

Além disso, os autores pensam em testar o *AStarPlanner* com outras estratégias de planejamento multiagente. O sistema de leilões funcionou, mas é necessário investigar outras possibilidades.

Por fim, pretende-se trabalhar com um algoritmo multiagente em que os planos são gerados e cumpridos por mais de um agente. O *MultiAStarPlanner* trabalha com um agente por meta. A intenção seria fazer com que vários agentes colaborassem não somente no momento da execução do plano, mas também na sua concepção.

Referências

ADVENTURE for the Atari 2600 by Atari. 1979. Disponível em: <http://www.atariguide.com/0/068.htm>. Acesso em: 13 Fevereiro 2008.

BIOSHOCK: Site oficial. 2007. Disponível em: <http://www.2kgames.com/bioshock/enter.html>. Acesso em 02 Junho 2008.

BLUM, A., FURST,M. *Fast Planning Through Planning Graph Analysis*. Artificial Intelligence, 90:281-300, 1997.

BURKE, R., at al. *CreatureSmarts: The Art and Architecture of a Virtual Brain*. In Proceedings of the Game Developers Conference, San Jose, CA, 2001.

CIVILIZATION. 1991. Disponível em: http://en.wikipedia.org/wiki/Civilization_(computer_game). Acesso em: 30 Setembro 2008.

CORMEN, T. H. et al. *Introduction to Algorithms.* MIT Press, 2 edição, 2001.

CORMEN, T. H. et al. *Introduction to Algorithms.* MIT Press, 2 edição, 2001, p87.

DOOM II. Site Oficial. 1994. Disponível em: < http://www.idsoftware.com/games/doom/doom2/> Acesso em: 13 Fevereiro 2008.

ENDURO for the Atari 2600 by Activision. 1983. Disponível em: < http://www.atariguide.com/0/018.htm>. Acesso em: 13 Fevereiro 2008.

EROL, et. al. *HTN planning: Complexity and expressivity*. *AAAI-94 Proceedings.* AAAI Press, 1994.

F.E.A.R: Site da produtora. 2005. Disponível em: < http://www.lith.com/games.asp?id=2>. Acesso em: 20 Março 2008.

FIKES, R. E., NILSSON, N. J., 1971. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*. 2nd IJCAI, Londres, 1971.

FORZA Motorsport 2: Site Oficial. 2007. Disponível em: < http://forzamotorsport.net/>. Acesso em 13 Fevereiro 2008.

GHALLAB, M., HOWE, A., KNOBLOCK, C, MCDERMOTT, D., RAM, A., RAM, A., VELOSO, M., WELD, D., WILKINS, D. *PDDL - The Planning Domain Definition Language*. Manual da competição AIPS-98, 1998.

GEARS of War: Site Oficial. 2006. Disponível em:
< http://gearsofwar.xbox.com/>. Acesso em: 13 Fevereiro 2008.

HALO 3 pulls in US$ 170 million worth of sales in 24 hours. Disponível em: <http://xbox360.qj.net/Halo-3-pulls-in-US-170-million-worth-of-sales-in-24-hours/pg/49/aid/103428> Acesso em: Acesso em: 20 Março 2008.

HAWES, N.A. Anytime Deliberation for Computer Game Agents. Tese de Doutorado, Universidade de Birmingham, Birmingham, 2003.

HART, P. E., NILSSON, N. J., RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems
Science and Cybernetics, 1968, p. 100–107.

HEROES of Might and Magic: A Strategic Quest. 1995. Disponível em:
http://en.wikipedia.org/wiki/Heroes_of_Might_and_Magic:_A_Strategic_Quest. Acesso em: 30 Setembro 2008.

HUNS, M. N, STEPHENS, L.M. *Multiagent Systems and Societies of Agents.* Em: WEISS, G., editor, Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, The MIT Press, 3 ed., 2001, p. 79-114.

KAUTZ, H., SELMAR, B. *Planning as satisfiability*. Em: Proceedings of the Tenth European Conference on Artificial Intelligence *(ECAI'92)*, pages 359-363. 1992.

KAUTZ, H., SELMAR, B. *BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving*. Working notes of the Workshop on Planning as Combinatorial Search, AIPS-98, Pittsburgh, PA, 1998.

KROGT, Roman et. al. *Multiagent Planning through Plan Repair*. AAMAS 2005: 1337-1338.

McCARTHY, J. and HAYES, P. J. *Some Philosophical Problems from the Standpoint of Artificial Intelligence*. Machine Intelligence 4, Edinburgh University Press, 1969, p. 463-502..

MUÑOZ-AVILA, H., FISHER, T., 2004. *Strategic Planning for Unreal Tournament Bots*. AAAI Challenges in Game AI Workshop Technical Report, AAAI Press, 2004.

MUÑOZ-AVILA, H.. et. al. *Hierarchical Plan Representations for Encoding Strategic Game AI*. Proceedings of Artificial Intelligence and Interactive Digital Entertainment, AAAI Press, 2005.

NDP - 2007 U.S. Video Game And PC Game Sales Exceed $18.8 Billion Marking Third Consecutive Year Of Record-Breaking Sales. Disponível em: <http://www.npd.com/press/releases/press_080131b.html. Acesso em: 20 Março 2008.

ORKIN, Jeff. *Symbolic Representation of Game World State: Toward Real-Time Planning in Games*. Em: AAAI Workshop on Challenges in Game AI, San Jose, CA, 2004.

ORKIN, Jeff . *Agent Architecture Considerations for Real-Time Planning in Games*. Em: AIIDE 2005 Proceedings, Marina Del Rey, CA, 2005.

PEDNAULT, E. P. *ADL: Exploring the middle ground between STRIPS and the situation calculus*. Em: Proc. of 1st Int. Conf. on Principles of Knowledge Representation and Reasoning, 1989, p. 324-332.

PROJECT Hoshimi, 2006. Site Oficial do *Project Hoshimi*. Disponível em: <http://www.project-hoshimi.com//>. Acesso em: 10 Novembro de 2007.

REITER, R. *On closed world data bases*. Em: Logic and Data Bases, pag. 119-140. Plenum Press. New York, 1978.

REIS, L. P. Coordenação em Sistemas Multi-Agente: Aplicações na Gestão Universitária e Futebol Robótico. Tese de Doutorado, FEUP, 2003.

RUSSEL, S. J., NORVIG, P., 2002. *Artificial Intelligence: A Modern Approach*, 2nd Edition, Prentice Hall, 2002.

WELD, D. S. 1999. *Recent Advances in AI Planning*. AI Magazine 20(2), 93 – 123.

WOOLDRIDGE, M. *An Introduction to Multiagent Systems*. Wiley, 2002.

Anexo A: Mapas usados nos testes dos algoritmos *AStarPlanner* e *MultiAStarPlanner*

A seguir, serão exibidas imagens de todos os mapas usados nos testes dos algoritmos *AStarPlanner* e *MultiAStarPlanner*.

Mapa A1: mapa com grande concentração de pontos HP em uma área de densidade média. Mapa que possui grande quantidade de barreiras (na cor preta), impedindo uma navegação em linha reta em quase todos os pontos do mapa.

Figura A-1: Mapa A1.

Mapa A2: mapa com grande quantidade de pontos HP e dois fluxos sanguíneos. Apesar de possuir grandes áreas não atravessáveis na parte central, não existem lá muitos pontos HP.

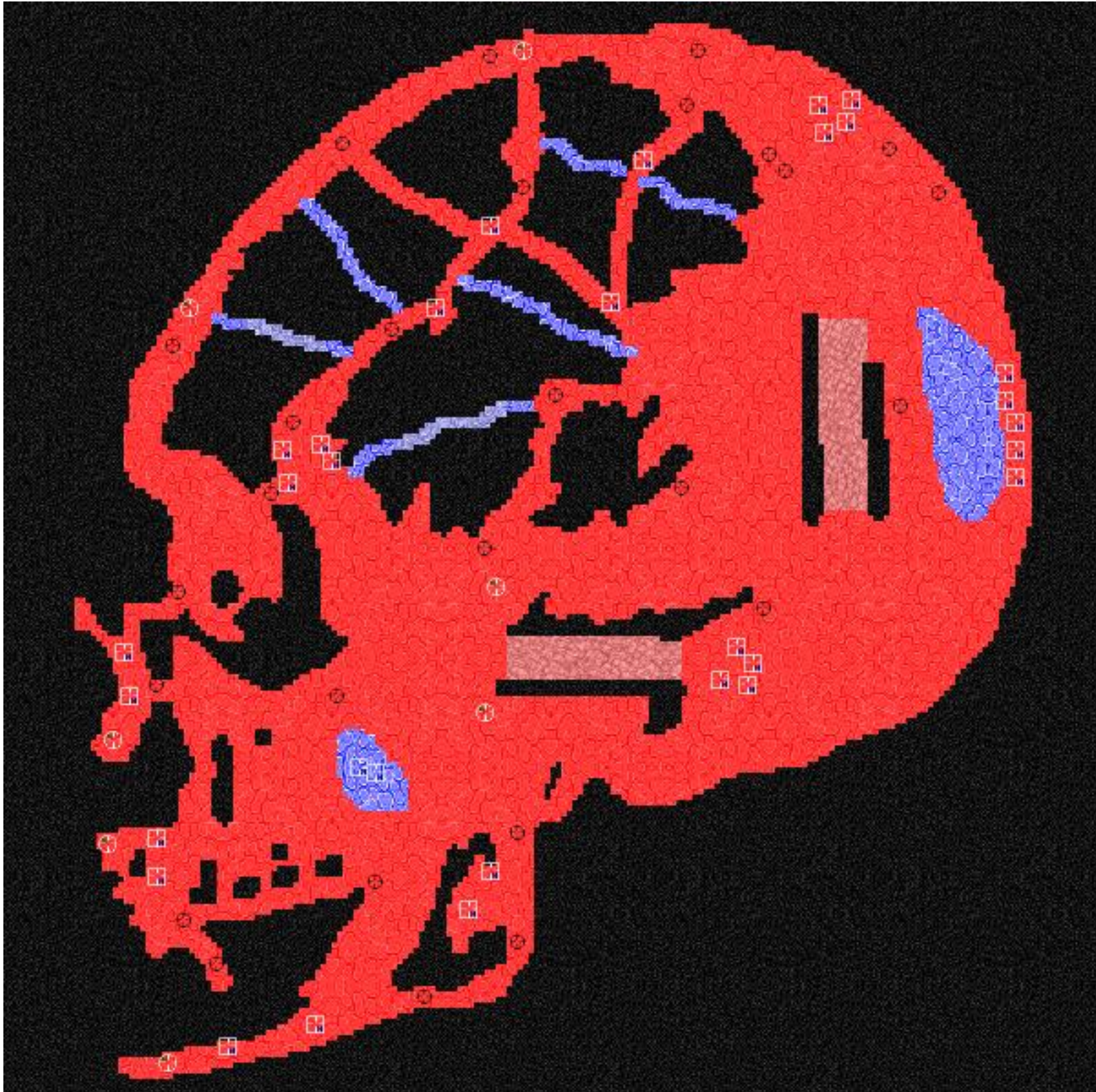

Figura A-2: Mapa A2.

Mapa A3: mapas com vários aglomerados de pontos HP, alguns presentes em áreas de densidade média. O principal deles localiza-se na parte inferior esquerda do mapa.

Figura A-3: Mapa A3.

Mapa A4: um dos mapas com maior quantidade de áreas não atravessáveis e com aglomerados de pontos HP próximos dessas localizações. Por isso, esse mapa prejudica bastante o cálculo de custos de ações do tipo movimentação e que utilizam distância Manhattan.

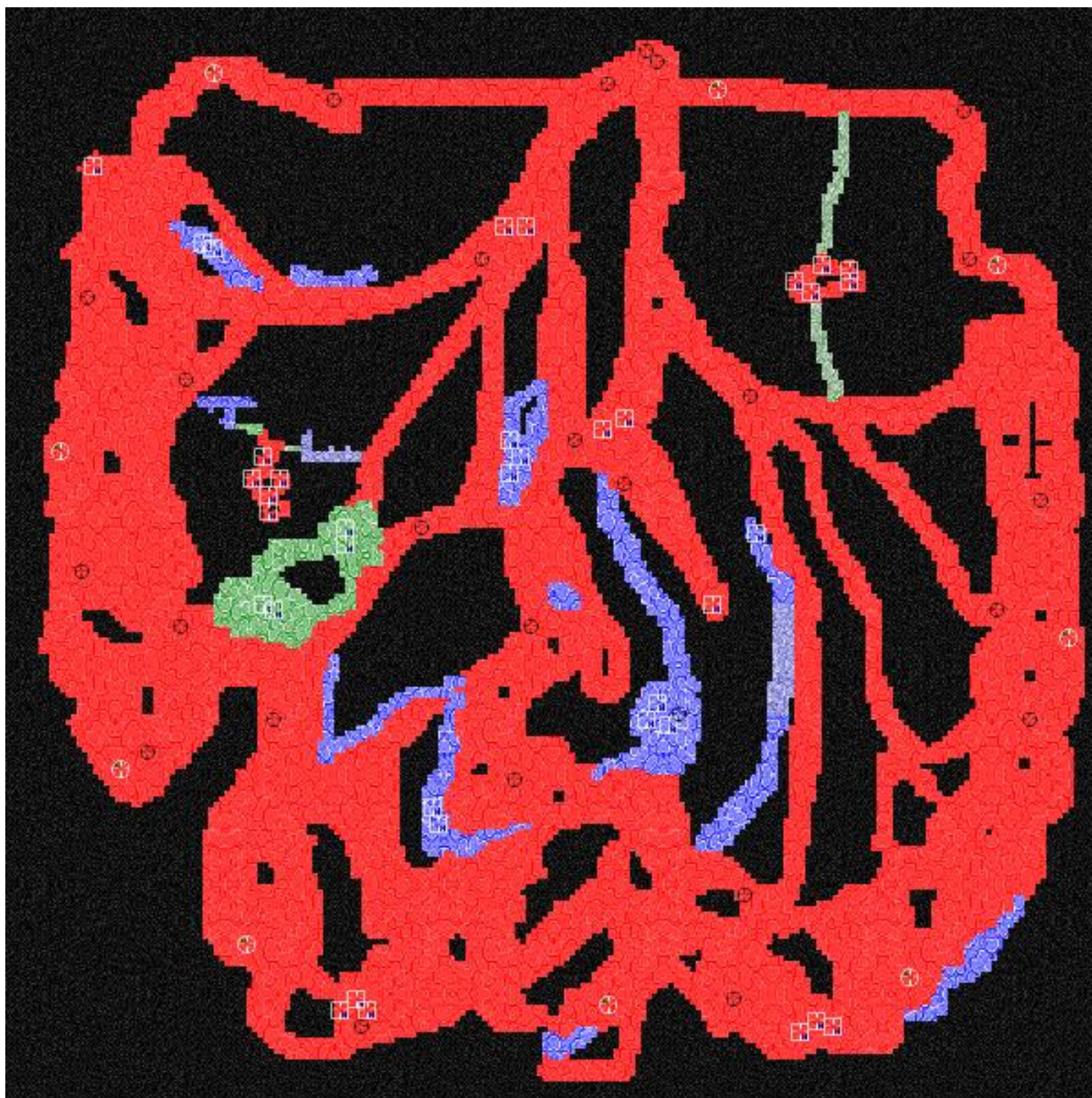

Figura A-4: Mapa A4.

Mapa Mashroom D: um dos mapas mais simples. Apesar de apresentar áreas de baixa, média e alta densidades, todos os pontos AZN e HP localizam-se em áreas de baixa densidade. Além disso, é um mapa que favorece bastante o cálculo do custo de ações do tipo movimentação com uso de distancia Manhattan.

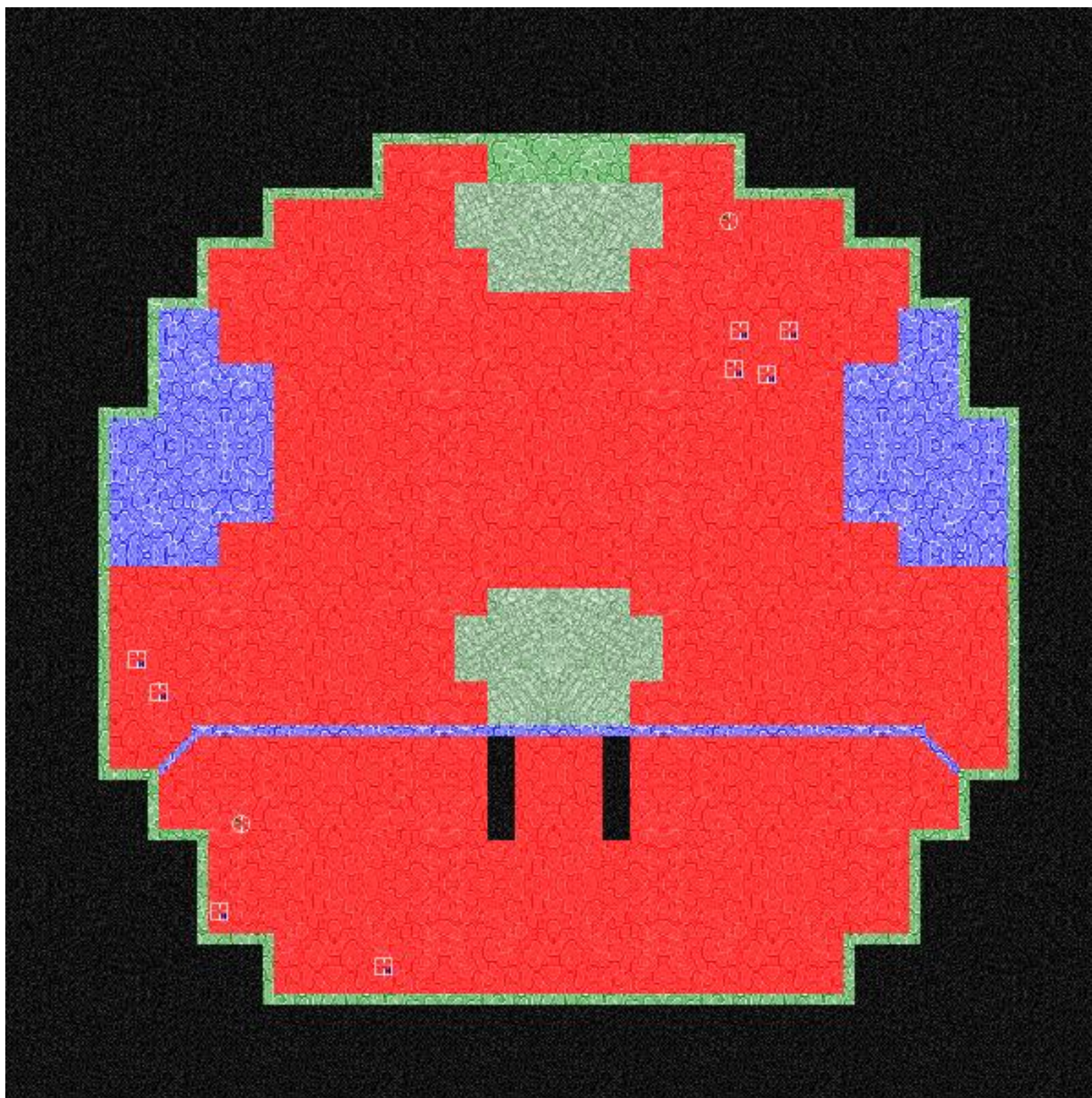

Figura A-5: Mapa Mashroom D.

Mapa SC3: este é o mapa que mais possui áreas não atravessáveis. Não existem grandes áreas atravessáveis contiguas, sendo o mapa formado por pequenas ilhas.

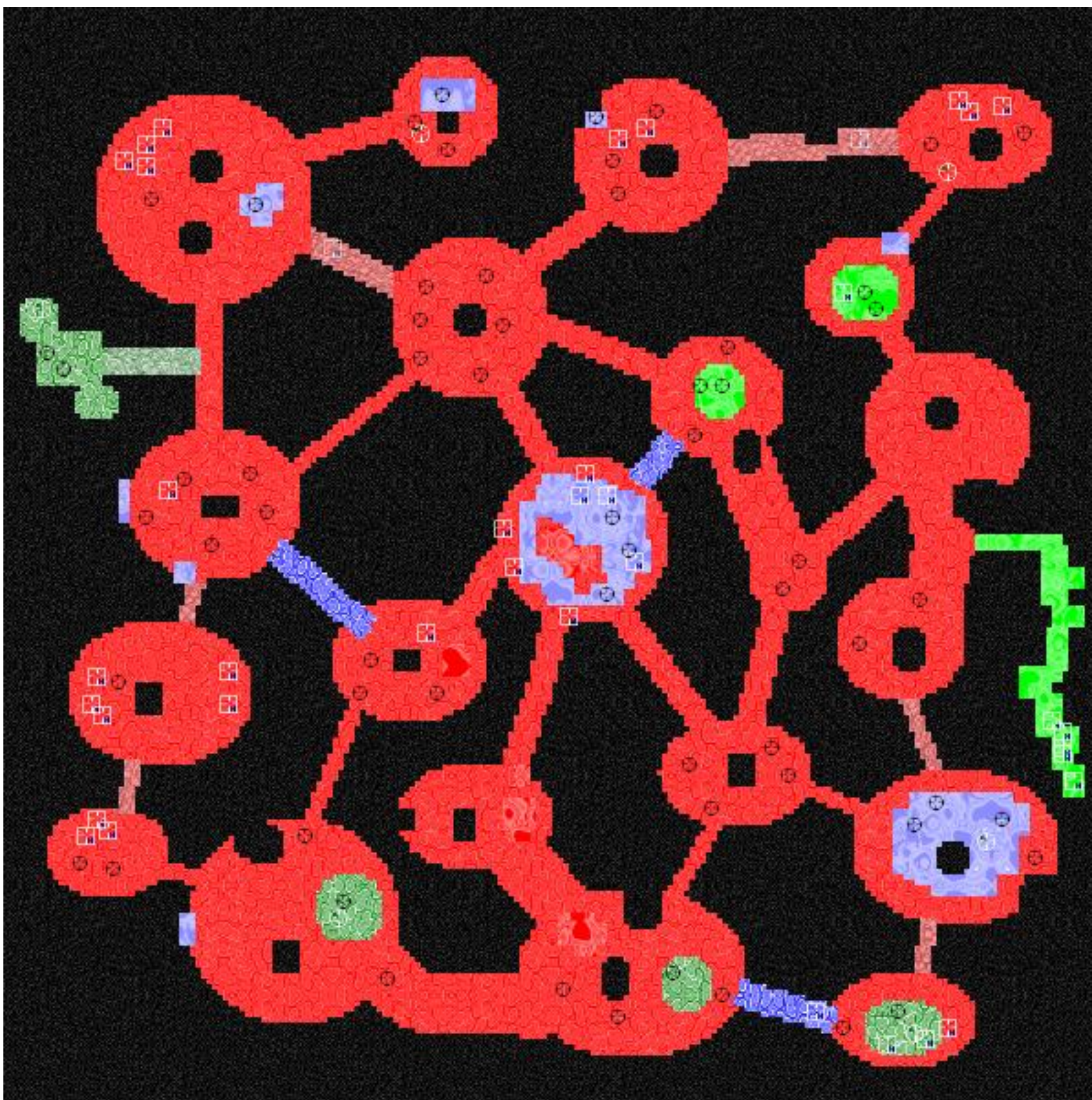

Figura A-6: Mapa SC3.

Mapa SC4: mapa sem aglomerados de pontos HP. Entretanto, grande parte dos pontos HP que se encontram próximos, estão separados por áreas não atravessáveis.

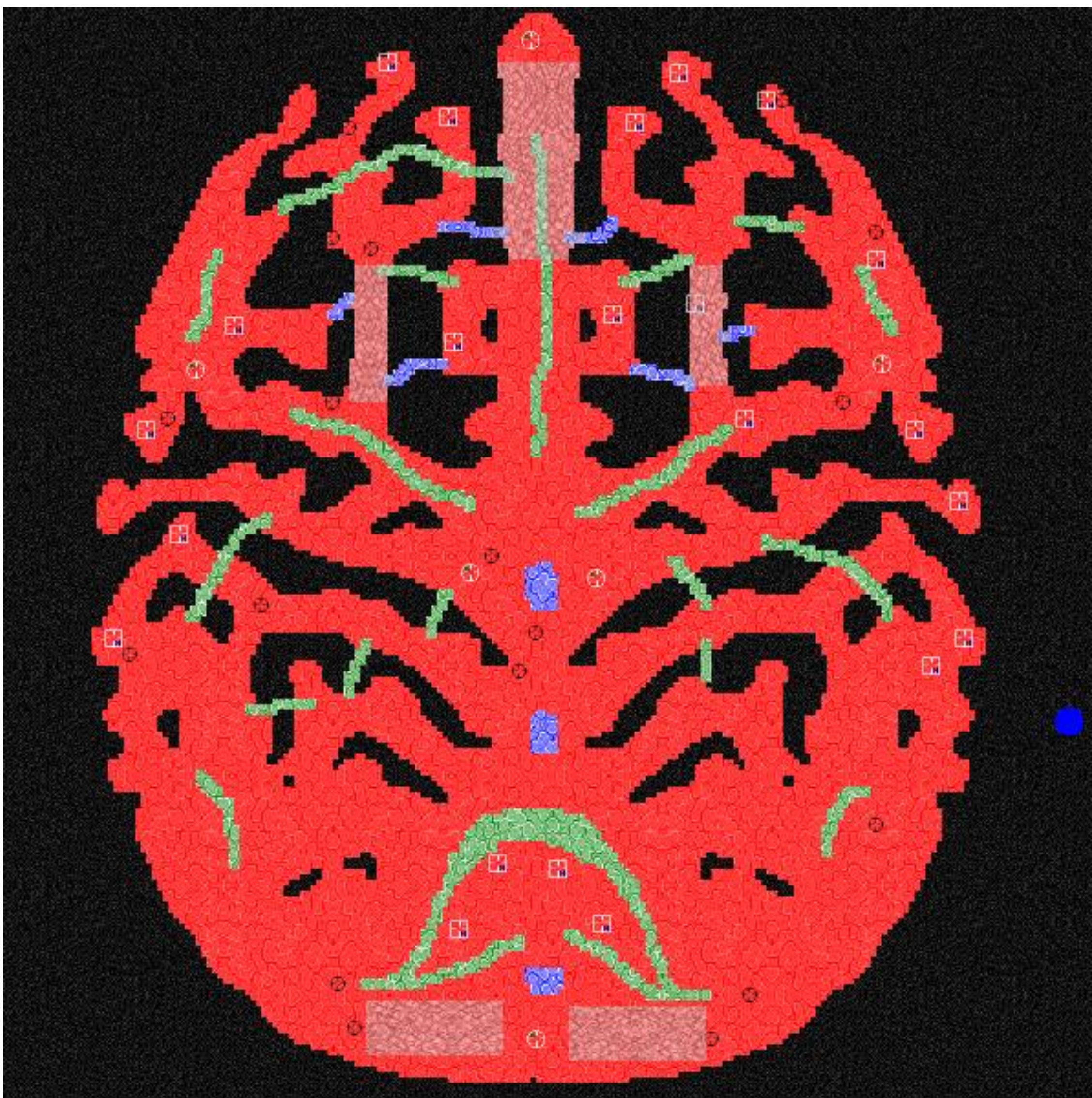

Figura A-7: Mapa SC4.