

Algoritmos de busca em tempo real aplicados a jogos digitais

Eder L. Trindade Ricardo P. Martins Ferreira Eduardo P. C. Fantini Hugo B. de Paula

Pontifícia Universidade Católica de Minas Gerais

Resumo

Restrições de tempo de resposta e de recursos computacionais oferecem um desafio para o emprego de técnicas sofisticadas de Inteligência Artificial nos jogos. Neste artigo estudamos como aprimorar os algoritmos de busca em tempo real encontrados na literatura. Em especial, investigamos como permitir que vários agentes se desloquem simultaneamente no ambiente de um jogo. Também foi feito um estudo de caso em um jogo real, o *Ultima Online*, para avaliar o comportamento desses algoritmos em jogos digitais.

Palavras Chaves: Inteligência Artificial, Agentes, *Real-time Search*, *pathfinding*, RTA*, *Games*

Contatos dos autores:

eder.lucio.trindade@gmail.com
{poley,hugo}@pucminas.br
eduardofantini@gmail.com

1. Introdução

Os jogos digitais são ambientes propícios para a aplicação de técnicas de inteligência artificial (IA). Por um lado, estes ambientes limitados, onde as regras costumam ser bem definidas, servem como laboratórios para testes [Waveren, 2001]. Por outro lado, a IA ainda tem muito a contribuir para o desenvolvimento dos jogos. Restrições de tempo de resposta e de recursos computacionais oferecem um desafio para o emprego de técnicas sofisticadas de IA em jogos digitais.

O planejamento de trajetória é um dos problemas presentes nos jogos onde técnicas de inteligência artificial são usadas com sucesso. Em muitas situações os movimentos de um agente/personagem não podem ser tratados como deslocamentos simples controlados pelo jogador. Nestes casos, é útil o emprego de algoritmos para o mapeamento do ambiente em um grafo e de algoritmos de busca de caminhos entre a posição atual no grafo e a posição desejada do agente no jogo [Nonnenmacher et al, 2007].

A maioria dos jogos digitais resolve o problema de busca de caminhos com soluções baseadas no algoritmo A* [Millington, 2006]. O algoritmo A* é um algoritmo informado de busca ótima, *off-line*. Nos algoritmos de busca *off-line* [Russell and Norvig, 2004], a fase do planejamento – mapeamento do ambiente e busca no grafo – antecede a fase da execução, quando o agente se desloca pelo caminho escolhido. Os algoritmos de busca *off-line* nem sempre são capazes de atender as necessidades de tempo de

resposta em um jogo. Quando um personagem fica imóvel aguardando o cálculo de uma trajetória, a cadência do jogo pode ficar prejudicada. Assim, uma decisão rápida para o próximo movimento, mesmo que não ótima, em algumas situações é mais importante do que esperar o cálculo do melhor movimento. Estas dificuldades ficam evidentes quando é necessário calcular simultaneamente a trajetória de diversos personagens.

Entre as soluções adotadas na literatura para lidar com os requisitos de tempo de resposta para o problema de encontrar caminhos, os algoritmos de busca *on-line*, conhecidos como algoritmos de busca em tempo real são freqüentemente adotados [Russell and Norvig, 2004]. Apesar de não tratarmos características relacionadas a tempo, adotamos o termo “tempo real” por ser o termo utilizado na literatura. Os algoritmos de busca em tempo real são algoritmos onde as fases de planejamento e de execução são alternadas. O agente planeja e executa alternadamente os movimentos até alcançar o objetivo. Embora não sejam ótimos, se algumas condições forem atendidas, como condições de conexidade do grafo, eles conseguem orientar o agente até o objetivo, mesmo quando o grafo sofre alterações durante o deslocamento do agente. Este comportamento é relevante em jogos digitais porque, muitas vezes, o ambiente é dinâmico.

Neste artigo estudamos como aprimorar os algoritmos de busca em tempo real encontrados na literatura. Em especial, investigamos como permitir que vários agentes se desloquem simultaneamente no ambiente de um jogo. Realizamos um estudo comparativo entre algoritmos de busca em tempo real presentes na literatura e nossas adaptações. Também foi feito um estudo de caso em um jogo real, o *Ultima Online*, para avaliar o comportamento de alguns algoritmos de busca em tempo real. O *Ultima Online* é um jogo digital do tipo MMORPG (*Massive Multiplayer Online Role-Playing Game*) em terceira pessoa [Origin, 2008], o qual possibilita a interação do jogador com um complexo ambiente virtual. Os resultados obtidos nos experimentos quantitativos e qualitativos realizados indicam a eficácia do uso de algoritmos de busca em tempo real em jogos digitais.

O restante do artigo é organizado da seguinte forma. A próxima seção apresenta alguns dos principais algoritmos de busca em tempo real. A Seção 3 apresenta as adaptações propostas e na Seção 4 são apresentados os resultados computacionais e as alterações feitas no emulador do *Ultima Online*, o *RunUO*. Na última seção são apresentadas algumas conclusões e oportunidades para trabalhos futuros.

2. Algoritmos de busca em tempo real - Trabalhos Relacionados

Os algoritmos de busca em tempo real propõem uma solução de compromisso entre a qualidade da decisão de qual é o próximo estado que um agente deve visitar e o tempo para calcular esta decisão [Koenig, 2006]. Existem vários artigos que sugerem a implementação de algoritmos de busca em tempo real em jogos digitais [Koenig, 2006][Koenig, 2007][Sun, 2008]. Entretanto, estes artigos não apresentam nenhuma implementação em um ambiente de jogo real.

A busca em tempo real foi motivada primeiramente pela robótica sensível a tempo [Koenig, 1999] e em seguida por jogos digitais, abrangendo os jogos de RTS (*real-time strategy*), de FPS (*first-person shooter*) e de RPG (*role-playing games*). Em todos eles o tempo desempenha um papel fundamental, uma vez que vários agentes fazem buscas de caminhos simultaneamente e é exigido do jogo um tempo de resposta muito rápido para garantir o *gameplay* [Bulitko et al., 2007]. Encontrar caminhos em jogos é computacionalmente muito caro. No jogo RTS "*Age of Empires II*", por exemplo, o *pathfinding* consome entre 60 e 70% do tempo de simulação [Pottinger, 2000].

O problema de encontrar caminhos em grafos é representado da seguinte forma: considere um Grafo $G(V, E)$ onde V representa um conjunto de vértices, E representa um conjunto de arestas, I representa o vértice inicial e O representa o conjunto de vértices finais. Um caminho é um subconjunto conexo do grafo G . O problema consiste em determinar um caminho de I para O .

Nas subseções seguintes descrevemos dois algoritmos de busca em tempo real clássicos, *Learning Real-Time A** e o *Real-Time A** [Korf 1990].

2.1 Learning Real-Time A* - LRTA*

O algoritmo LRTA* [Korf 1990] (*Learning Real-Time A**) é um algoritmo de busca informada e em tempo real. Ele não é ótimo, ou seja, não garante que será encontrado o melhor caminho entre o nó inicial e o nó objetivo. O algoritmo possui conhecimento da localização do vértice objetivo e utiliza esta informação para eleger a próxima posição do agente. No contexto onde realizamos os experimentos utilizamos a distância *Manhattan* como medida heurística da distância até o objetivo.

O LRTA* inicia com um agente localizado no vértice inicial. No passo seguinte, o agente se desloca para o vizinho mais promissor de acordo com a função de custo. A busca termina quando o agente encontra um vértice objetivo. A cada iteração o agente atualiza a estimativa da distância até o objetivo melhorando a informação sobre o grafo. Na fase de planejamento o agente calcula para os nós vizinhos a função de custo

Considere o vértice A como o vértice inicial e o vértice D como o vértice objetivo.

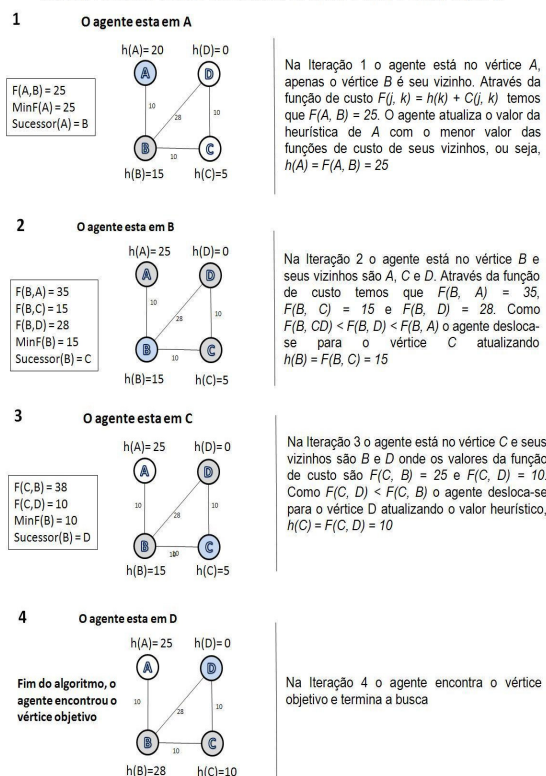


Figura 1: Execução do LRTA*

Constantes e funções:

I	vértice inicial
O	vértice objetivo
$\text{Min}F(j)$	função que retorna o valor do menor custo estimado dos caminhos que partem de j até o objetivo
$\text{Sucessor}(j)$	função que retorna o vértice vizinho de j cujo custo estimado do caminho até o objetivo passando por ele é o menor
$\text{Heuristica}(j)$	função que retorna a heurística da distância do vértice j até o objetivo.

Variáveis:

j	vértice corrente
$h(j)$	valor da heurística do vértice j
V	conjunto de vértices do grafo

Procedimento:

```

[01] Para cada  $j$  em  $V$  faça
[02]    $h(j) = \text{Heuristica}(j)$ 
[03]  $j = I$ 
[04] Enquanto ( $j \neq O$ ) faça
[05] {
[06]    $h(j) = \text{Min}F(j)$ 
[07]    $j = \text{Sucessor}(j)$ 
[08] }
```

Figura 2: Pseudocódigo do LRTA*

$F(j, k) = h(k) + C(j, k)$ onde k é um vértice vizinho, j é o vértice corrente, $h(k)$ representa o valor atual da heurística de k até o objetivo O e $C(j, k)$ é o custo da aresta do vértice corrente j até k . O agente atualiza a heurística do vértice corrente com o menor valor de $F(j, k)$ e move-se para ele. O processo então reinicia.

A Figura 1 ilustra um exemplo de execução do LRTA*. O nó inicial é o vértice A e o nó objetivo é o vértice D. O algoritmo calcula a heurística de cada vértice, consideremos o seguinte estado corrente: $h(A) = 20$, $h(B) = 15$, $h(C) = 5$ e $h(D) = 0$.

2.2 Real-Time A* - RTA*

O Algoritmo RTA* [Korf 1990] (*Real Time A**) é um algoritmo de busca por qualquer caminho e em tempo real semelhante ao LRTA*. A diferença entre o LRTA* e o RTA* é que o RTA* atualiza a heurística do vértice corrente com o segundo melhor valor de $F(j, k)$ mantendo uma informação mais precisa do grafo. A Figura 3 apresenta o pseudocódigo do RTA*.

Constantes e funções:

I	<i>vértice inicial</i>
O	<i>vértice objetivo</i>
$SegundoMinF(j)$	<i>função que retorna o valor o segundo menor custo estimado dos caminhos que partem de j até o objetivo</i>
$Sucessor(j)$	<i>função que retorna o vértice vizinho de j cujo custo estimado do caminho até o objetivo passando por ele é o menor</i>
$Heuristica(j)$	<i>função que retorna a distância de heurística do vértice j até o objetivo.</i>

Variáveis:

j	<i>vértice corrente</i>
$h(j)$	<i>valor da heurística do vértice j</i>
V	<i>conjunto de vértices do grafo</i>

Procedimento:

```

[01] Para cada j em V faça
[02]      $h(j) = Heuristica(j)$ 
[03]  $j = I$ 
[04] Enquanto ( $j \neq O$ ) faça
[05] {
[06]      $h(j) = SegundoMinF(j)$ 
[07]      $j = Sucessor(j)$ 
[08] }
```

Figura 3: Pseudocódigo do RTA*

3. Variações propostas dos algoritmos

Neste trabalho estudamos algoritmos de busca em tempo real e propomos novos algoritmos adotando duas estratégias:

- Investigamos o efeito nos algoritmos da falta de informação sobre o ambiente de busca. Propomos uma variação no algoritmo RTA* melhorando o aprendizado do agente sobre o ambiente;
- Estudamos diversos agentes no processo de busca. Propomos um algoritmo baseado em sistemas multiagentes onde agentes compartilham informação, o que pode agilizar o processo de busca.

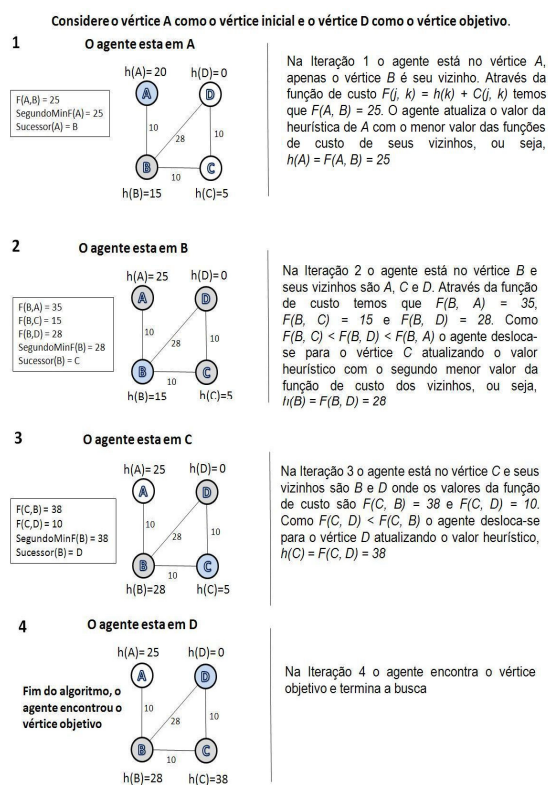


Figura 4: Execução do RTA*

A seguir apresentamos dois algoritmos o *Real Time A* plus*, onde melhoramos o aprendizado do algoritmo *Real Time A** e o *Relay Real Time A**, onde mais de um agente cooperam para agilizar o processo de busca.

3.1 Real-Time A* Plus – RTA*P

O *Real-Time A* Plus* (RTA*P) é uma variação do algoritmo *Real-Time A**, onde a novidade é acrescentar mais informação ao algoritmo. No RTA* o algoritmo analisa todos os vizinhos do vértice corrente para escolher o vértice mais promissor e atualizar a heurística do vértice corrente com o segundo melhor valor de custo. Contudo esse algoritmo descarta as informações previamente calculadas sobre os outros vizinhos. O algoritmo RTAP armazena também qual é o vizinho do segundo melhor valor de custo e qual é o valor do terceiro melhor custo. Caso o agente retorne a um vértice já expandido o agente desloca-se para o vértice de segundo melhor custo e atualiza o valor da heurística do vértice corrente com o terceiro melhor custo, sem a necessidade de realizar nova análise dos vizinhos.

O RTA*P ilustra o comportamento do agente quando aumentamos seu aprendizado sobre o ambiente. Comparamos o RTA*P e o RTA* em diversas situações explorando ambientes com e sem informação. O RTA*P inicia com um agente localizado no vértice inicial, no passo seguinte o agente se desloca para o vizinho mais promissor. A busca encerra quando o agente encontra um vértice objetivo.

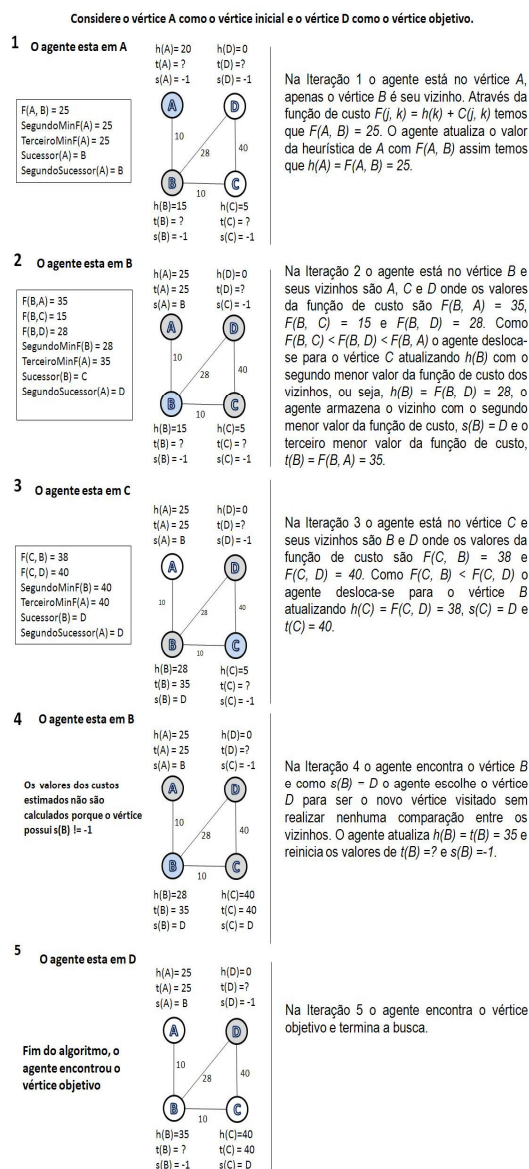


Figura 5: Execução do RTA*P

A cada iteração o agente atualiza a estimativa heurística do vértice. Na fase de planejamento o agente calcula para todos seus vizinhos a função de custo $F(j, k) = h(k) + C(j, k)$ onde j é o vértice corrente, k é um vértice vizinho de j , $h(k)$ representa o valor da heurística da distância de k até O e $C(j, k)$ é o custo da aresta do vértice corrente j até o vizinho k . O algoritmo também armazena qual o vértice $s(j)$ possui o segundo menor valor da função de custo e o valor do terceiro menor valor da função de custo, $t(j)$. Se o vértice j ainda não foi visitado o agente atualiza a heurística do vértice corrente com o segundo menor valor de $F(j, k)$ e move-se para o vizinho de menor $F(j, k)$. Se o vértice k já foi visitado o agente não calcula a função de custo para os vizinhos de k e caminha para o vértice $s(k)$ atualizando a heurística $h(k) = t(k)$. A Figura 5 apresenta um exemplo de execução do RTA*P. A Figura 6 apresenta o algoritmo RTA*P.

Constantes e funções:

I	vértice inicial
O	vértice objetivo
$\text{SegundoMinF}(j)$	função que retorna o valor do segundo menor custo estimado dos caminhos até o objetivo que partem de j
$\text{TerceiroMinF}(j)$	função que retorna o valor do terceiro menor custo estimado dos caminhos até o objetivo que partem de j
$\text{Sucessor}(j)$	função que retorna o vértice vizinho de j cujo custo estimado do caminho até o objetivo passando por ele é o menor
$\text{SegundoSucessor}(j)$	função que retorna o vértice vizinho de j cujo custo estimado do caminho até o objetivo passando por ele é o segundo menor
$\text{Heuristica}(j)$	função que retorna a distância de Manhattan do vértice j até o objetivo.
Variáveis:	
j	vértice corrente
$h(j)$	valor da heurística do vértice j
$s(j)$	vizinho de j com a segundo menor custo estimado
$t(j)$	valor da heurística do vizinho de j com o terceiro menor custo estimado
$\text{vizinho}(j)$	conjunto de vértices vizinhos de j .
V	conjunto de vértices do grafo

Procedimento:

```

[01] Para cada  $j$  em  $V$  faça
[02] {
[03]      $h(j) = \text{Heuristica}(j)$ ,  $t(j) = 0$ ,  $s(j) = -1$ 
[04] }
[05]  $j = I$ 
[06] Enquanto ( $j \neq O$ ) faça
[07] {
[08]     se  $s(j) \neq -1$  então
[09]     {
[10]          $j = s(j)$ ,  $h(j) = t(j)$ ,  $s(j) = -1$ 
[11]     }
[12]     Senão
[13]     {
[14]          $h(j) = \text{SegundoMinF}(j)$ 
[15]          $t(j) = \text{TerceiroMinF}(j)$ 
[16]          $s(j) = \text{SegundoSucessor}(j)$ 
[17]          $j = \text{Sucessor}(j)$ 
[18]     }
[19] }

```

Figura 6: Pseudocódigo do RTA*P

3.3 Relay Real-Time A* - RRTA*

O *Real Time A** (RRTA*) é um algoritmo de busca informada em tempo real baseado em sistemas multiagentes [Yokoo and Ishida, 1999] [Wooldridge, 2002]. O RRTA* é formado por dois ou mais agentes que dividem informação para encontrar um caminho entre dois nós. Vamos discutir neste artigo o caso com apenas dois agentes. Chamamos o primeiro agente de A e o segundo agente de B. O agente A é o agente responsável por encontrar um caminho entre o nó inicial e o nó final passando pelo nó origem de B. O agente B tem o objetivo de encontrar um caminho entre um nó qualquer do grafo que é seu nó inicial e o nó objetivo. O nó de origem de B deve pertencer ao caminho de A. Assim, o agente B antecipa o conhecimento sobre o grafo para A. Neste artigo vamos utilizar como estratégia para a escolha da posição inicial de B o ponto médio da semi-reta entre o nó inicial e o nó objetivo.

O RRTA* é formado por duas esferas de decisão: uma esfera tática onde o algoritmo coordena os agentes e uma esfera operacional onde os agentes executam a busca.

Na esfera tática o agente A começa a busca no nó inicial. O algoritmo elege um vértice para B iniciar a busca, neste caso, o ponto médio da semi-reta entre o nó inicial e nó objetivo. O agente B tem a função de encontrar o nó objetivo e A, a priori, tem o objetivo de encontrar o vértice por onde B iniciou a busca. Quando o agente A encontrar o nó por onde B partiu ou algum vértice que foi caminho de B o agente A passa a procurar o nó objetivo. A busca termina quando o agente A encontra o nó objetivo.

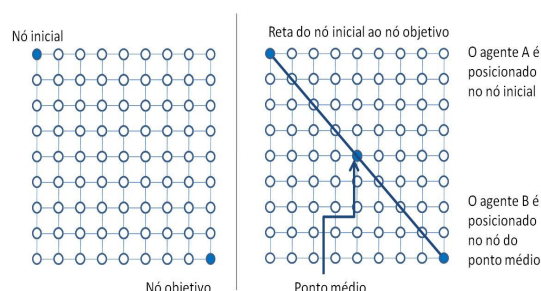


Figura 7: Estratégia de posicionamento dos agentes - RRTA* - Ponto Médio – Dois Agentes

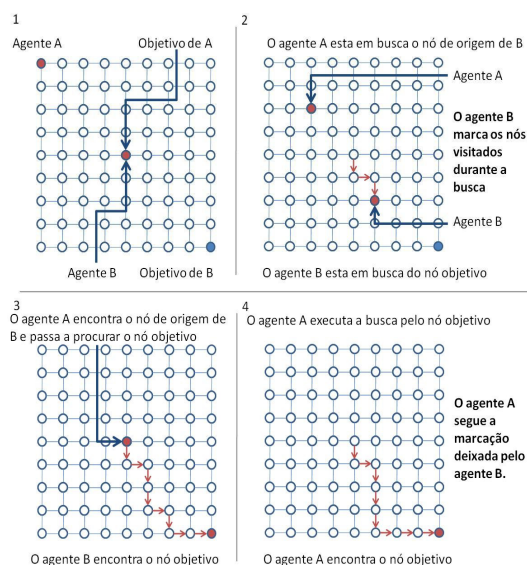


Figura 8: Execução do RRTA*

Na esfera operacional, ambos agentes utilizam o algoritmo RTA* para realizar a busca. O agente B realiza a busca a procura do nó objetivo marcando os nós por onde passou, o agente B marca no nó visitado qual é o próximo nó que ele irá visitar. O Agente A realiza a busca e quando encontra um nó visitado por B segue as marcações deixadas por B sem fazer qualquer verificação, assim o agente A utiliza do conhecimento do grafo adquirido na busca de B. A Figura 7 ilustra a definição das posições iniciais dos agentes.

Nos exemplos adotados neste trabalho o algoritmo traça uma semi-reta entre o nó inicial e o nó objetivo.

O algoritmo calcula o ponto médio desta semi-reta. As coordenadas do ponto médio são definidas pela equação: $X = \frac{x1+x2}{2}$, $Y = \frac{y1+y2}{2}$ onde X representa a ordenada do ponto médio, $x1$ a ordenada do nó inicial e $x2$ a ordenada do nó objetivo, Y representa a abscissa do ponto médio, $y1$ representa a abscissa do nó inicial e $y2$ representa a abscissa do nó objetivo.

Constantes e funções:

I	vértice inicial
O	vértice objetivo
$Estrategia(x, I, O)$	função que posiciona o agente x , esta função é responsável pela estratégia de posicionamento
$RTA^*(x)$	função que executa uma iteração da busca RTA* para o agente x
$Getcorrente(x)$	função que retorna o vértice corrente do agente x
$SetObjetivo(x, j)$	função que substitui o vértice objetivo do agente x por j
$GetObjetivo(x)$	função que retorna o vértice objetivo temporário do agente x
$PontoMarcado(j, x)$	função que retorna se o vértice j está marcado pelo agente x
$Sucessor(x)$	função que retorna o agente que x persegue

Variáveis:

j	vértice corrente
x	agente corrente
A	Conjunto de Agentes

Procedimento:

```

[01] Para cada agente  $x \in A$  faça
[02]    $Estrategia(x, I, O)$ 
[03] Enquanto existir ( $Getcorrente(x) \neq O \forall x \in A$ ) faça
[04] {
[05]   Para cada agente  $x \in A$  faça
[06]   {
[07]      $RTA^*(x)$ 
[08]     se ( $Getcorrente(x) = GetObjetivo(x)$ ) ou
[09]       ( $PontoMarcado(Getcorrente(x), Sucessor(x))$ )
[10]        $SetObjetivo(x, GetObjetivo(Sucessor(x)))$ 
[11]     Se  $Getcorrente(x) = O$ 
[12]        $A = A - x$ 
[13]   }

```

Figura 9: Pseudocódigo do RRTA*

A estratégia para posicionamento dos agentes depende muito do tipo do grafo. Apresentamos um critério mais didático, contudo outros critérios podem ser adotados. A Figura 9 apresenta o pseudocódigo do algoritmo RRTA*, o procedimento RTA* além de executar uma iteração da busca RTA* marca os nós por onde B passou e o agente A quando encontra um nó visitado pelo agente B segue a marcação sem analisar os vizinhos. O primeiro passo do algoritmo é posicionar os agentes, o agente A é posicionado no nó inicial e o agente B é posicionado no nó escolhido. O algoritmo atribui ao agente A a função de encontrar o nó por onde B iniciou a busca e ao agente B a função de encontrar o nó objetivo. No segundo passo os agentes executam a busca utilizando o algoritmo RTA*, o agente B durante a busca deixa marcas que depois poderão ser seguidas pelo agente A.

Quando o agente A encontra o ponto por onde B iniciou a busca ele passa a procurar o nó objetivo. O agente B termina sua busca quando encontra o nó objetivo, quando encontra com o agente A ou quando o agente A encontra o nó objetivo. Quando o agente A encontra as marcas deixadas pelo agente B ele passa a segui-las, assim o agente A aproveita o conhecimento do grafo adquirido por B durante sua busca. A busca termina quando o agente A encontra o nó objetivo. A Figura 8 apresenta a execução do RRTA*.

Realizamos experimentos como o RRTA* apenas em grafos conexos, a estratégia de posicionamento dos agentes utilizando o ponto médio deve ser usada em somente em grafos conexos.

4. Experimentos Computacionais

Foram realizados dois grupos de experimentos. O primeiro grupo trata de um estudo comparativo entre os diferentes algoritmos propostos na literatura e os novos algoritmos propostos neste trabalho em um ambiente estruturado. São ambientes típicos para testes de algoritmos de busca [Koenig, 2006][Koenig, 2007][Sun, 2008]. O segundo grupo trata de experimentos qualitativos onde algoritmos de busca *on-line* são aplicados em um jogo MMORPG - o *Ultima Online* - para avaliar seus comportamentos em um ambiente de jogo real.

Os ambientes estruturados são labirintos representados em grafos 4-conexos gerados aleatoriamente de tamanho entre 100 a 1.000.000 vértices. São labirintos quadrados de tamanhos entre 10 x 10 a 1000 x 1000. Escolhemos o jogo eletrônico *Ultima Online* por sua grande popularidade e por possuir um emulador de servidor - o *RunUO* - de código aberto o qual permite alterações nos algoritmos de busca. Os experimentos foram realizados em uma máquina com processador *Intel Core 2 Duo 2.2 Ghz*, 2 GB de Memória RAM, sistema operacional *Windows XP Professional 2002 Service Pack 2*. Utilizamos como compilador o *Borland C++ Builder 6.0*.

4.1 Experimentos quantitativos em grafos baseados labirintos

Para comparar o desempenho dos algoritmos consideramos o número médio de vértices expandidos e o tempo médio de execução de cada algoritmo. Consideramos como vértices expandidos os vértices por onde os agentes passaram e o tempo de execução de um algoritmo representa o tempo em milissegundos gasto para atingir o objetivo. Estes dados são as médias das execuções do mesmo algoritmo no mesmo grafo. Para cada grafo executamos repetidamente o mesmo algoritmo. Limitamos o tempo total de execução do algoritmo e, por isso, os grafos maiores foram menos executados.

Para a realização dos experimentos foram desenvolvidos dois geradores de grafos: gerador de grafos aleatórios (GCA), e gerador de grafos conexos (GCC). O tamanho do grafo corresponde ao produto do número de colunas pelo número de linhas do grafo. Por exemplo, um grafo de tamanho igual 10.000 vértices possui 100 colunas e 100 linhas.

Definimos como vértices obstáculos os vértices que possuem grau zero. Os geradores recebem como parâmetros os vértices iniciais e finais, o tamanho, o percentual do número de arestas ou o percentual de vértices obstáculos, o grau médio do grafo e o tipo do grafo. O tipo do grafo é definido pelas distâncias entre nós vizinhos que podem ser constantes ou aleatórias.

Algoritmo Gerador do grafo	Algoritmo	Pesos das arestas	Heurística	Número de execuções
GCA	RTA*	Constante	Manhattan	18600
GCA	RTAP*	Constante	Manhattan	18600
GCA	RTA*	Aleatório	Manhattan	18600
GCA	RTAP*	Aleatório	Manhattan	18600
GCA	RTA*	Aleatório	Sem heurística	18600
GCA	RTAP*	Aleatório	Sem heurística	18600
GCC	RTA*	Constante	Manhattan	18600
GCC	RTAP*	Constante	Manhattan	18600
GCC	RRTA*	Constante	Manhattan	18600
GCC	RTA*	Aleatório	Manhattan	18600
GCC	RTAP*	Aleatório	Manhattan	18600
GCC	RRTA*	Aleatório	Manhattan	18600
GCC	RTA*	Aleatório	Sem heurística	18600
GCC	RTAP*	Aleatório	Sem heurística	18600
GCC	RRTA*	Aleatório	Sem heurística	18600
Total de execuções				279000

Tabela 1: Número de execuções dos experimentos quantitativos em grafos baseados em labirintos

O gerador de grafos aleatórios (GCA) garante a geração de grafos com pelo menos um caminho entre o vértice inicial e o vértice final, mas não garante que o grafo seja conexo. Por este motivo, nos grafos gerados pelo o GCA não é possível garantir o funcionamento correto do algoritmo RRTA* utilizando a estratégia de posicionamento pelo ponto médio. No GCA o percentual de vértices obstáculos define a densidade do grafo, uma vez que o obstáculo possui grau zero e conseqüentemente o número de arestas é menor. A

O Gerador de grafos conexos (GCC) garante a existência de caminhos entre quaisquer vértices do grafo, ou seja, todos os vértices estão interligados. Este algoritmo contempla a geração de grafos de tamanhos e densidades diferentes. Os grafos são no máximo 4-conexos, ou seja, cada vértice possui no máximo quatro vizinhos. Os pesos das arestas representam as distâncias entre os vértices que podem ser uniformes ou aleatórias. No GCC a quantidade de arestas define a densidade do grafo, consideramos um grafo com 100% de densidade quando todos os vértices possuírem grau igual a quatro.

Os experimentos foram separados em dois subgrupos distintos, o primeiro são experimentos executados nos

grafos gerados pelo GCA e o segundo nos grafos gerados pelo GCC.

Utilizamos como heurística a distância de *Manhattan* e para cada subgrupo estudamos o comportamento dos algoritmos em grafos onde os pesos das arestas são iguais, em grafos onde os pesos das arestas são aleatórios ou em situações sem heurística. Para os grafos gerados pelo GCA não estudamos o RRTA* porque a estratégia de posicionamento através do ponto médio não é adequada neste tipo de grafo.

A Tabela 1 apresenta o número de execuções dos algoritmos RTA*, RTA*P e RRTA* nos experimentos deste artigo, foram realizados 111600 execuções para o Grupo I e 167400 execuções para o grupo II.

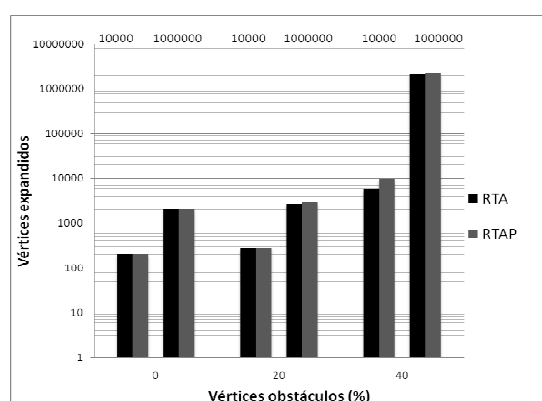


Figura 11: Gráfico da média do número de vértices expandidos para grafos gerados pela GCA com pesos variados entre 1 a 10

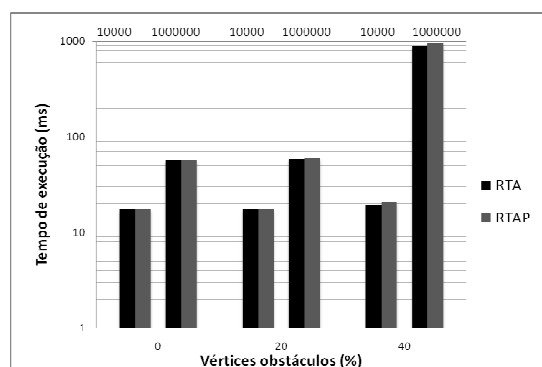


Figura 12: Gráfico da média do tempo de execução para grafos gerados pela GCA com pesos variados entre 1 a 10

Foram gerados 200 grafos para cada configuração, onde 100 têm peso uniforme nas arestas e 100 possuem pesos aleatórios. Cada configuração é definida pelo algoritmo que o criou: GCA, ou GCC; pelo seu número de vértices: 100, 10.000, 1.000.000; e pelo percentual de vértices obstáculos, no caso dos grafos gerados pelo GCA ou pela percentual de arestas, para os grafos gerados pelo GCC.

Os resultados são apresentados em gráficos de barras onde as colunas estão ordenadas de acordo com a legenda, não sendo necessário orientar-se pela cor.

Grupo I: os experimentos do Grupo I foram realizados em grafos com pelo menos um caminho entre o vértice inicial e o vértice final (grafos gerados pelo GCA), mas não necessariamente conexos. Cada vértice possui no máximo quatro vizinhos, a quantidade de vértices obstáculos define a densidade do grafo.

A Figura 11 compara o número de vértices expandidos versus o percentual de vértices obstáculos nos grafos do grupo I. O número de vértices expandidos aumenta quando o percentual de vértices que são obstáculos aumenta. O RTA* e o RTA*P empatam quando o grafo não possui obstáculos. À medida que aumenta o número de vértices que são obstáculos o RTA* apresenta resultados melhores que o RTA*P. Esta diferença é menor em grafos com pesos aleatórios.

A Figura 12 compara o tempo de execução (em milissegundos) com o percentual de vértices obstáculos nos grafos do grupo I. Os algoritmos gastaram mais tempo para solucionar os problemas quando o grafo possui 40% de vértices obstáculos. O RTA* e o RTA*P apresentam comportamentos semelhantes. Não há diferenças nos resultados entre os grafos com arestas com pesos uniformes e os grafos com arestas com pesos aleatórios.

Grupo II: os experimentos do Grupo II foram realizados em grafos conexos gerados pelo algoritmo GCC. Cada vértice possui no máximo quatro vizinhos, a quantidade de arestas define a densidade do grafo. As figuras 13 e 14 comparam o número de vértices expandidos versus o percentual de densidade nos grafos do grupo II. Os gráficos não indicam uma relação direta entre o número de vértices expandidos e a densidade do grafo.

O RTA*P e o RTA* apresentam resultados semelhantes. O RRTA* apresenta número de vértices expandidos maior que o RTA*. Entretanto, se dividirmos o número de vértices expandidos pelo número de agentes temos uma média de vértices expandidos menor que o RTA*. Sem utilizar heurísticas, o RTA*P expande menos vértices que o RTA*.

As figuras 15 e 16 comparam o tempo de execução (em milissegundos) versus o percentual de vértices de densidade nos grafos do grupo II. O RTA* e o RTAP apresentaram desempenhos semelhantes. Não há diferenças nos resultados dos grafos com arestas com pesos uniformes e arestas com pesos aleatórios.

O RTA* gasta mais tempo para executar a busca quando não utiliza heurística. O RTA*P é mais estável e mais eficiente que o RTA* em todas as densidades. Em grafos com pesos aleatórios a diferença entre o RTA*P e o RTA* é menor, mesmo assim o RTA*P é mais eficiente.

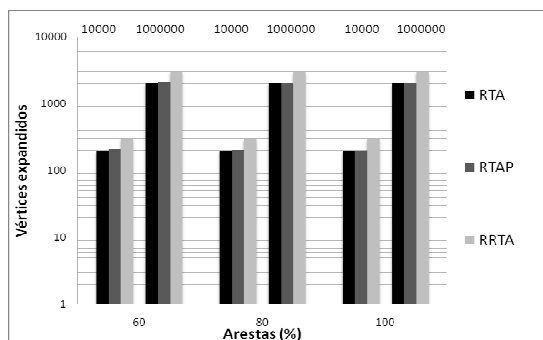


Figura 13: Gráfico da média do número de vértices expandidos para grafos gerados pela GCC com pesos constantes

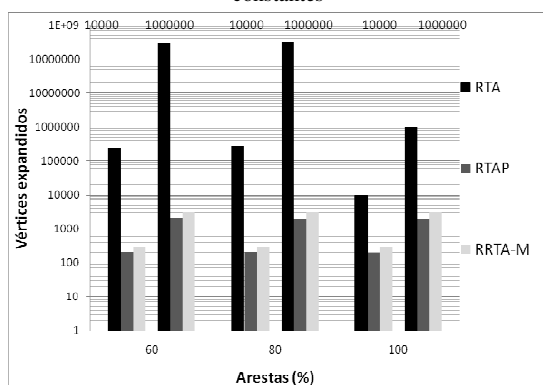


Figura 14: Gráfico da média do número de vértices expandidos para grafos gerados pela GCC com pesos constantes e sem a utilização de heurística

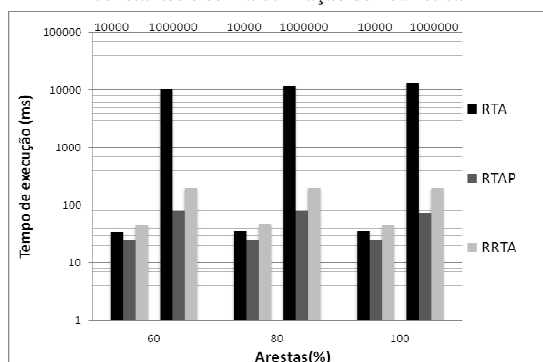


Figura 15: Gráfico da média do tempo de execução para grafos gerados pela GCC com pesos constantes

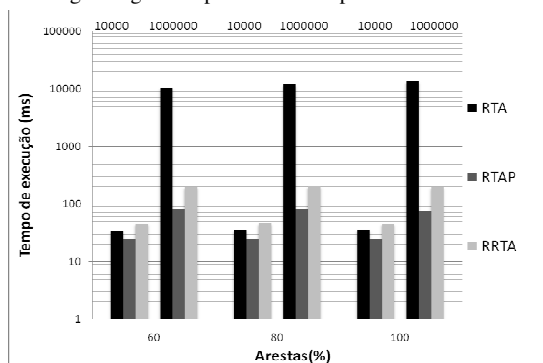


Figura 16: Gráfico da média do tempo de execução para grafos gerados pela GCC com pesos variados entre 1 a 10 e sem a utilização de heurística

4.2 Experimentos realizados no Ultima Online

Existem vários emuladores de servidor para o *Ultima Online*: *UOX*, *POL*, *SPHERE* e *RunUO* [Wikipedia, 2008]. Escolhemos o *RunUO* versão 2 RC2 pelo fato do mesmo ser bastante utilizado. O *RunUO* possui código aberto e recria por meio da plataforma Microsoft.NET o ambiente de um servidor para o *Ultima Online*. No *RunUO* é possível personalizar grande parte das características funcionais do jogo e manipular *NPCs* (*non player characters*). O teste de algoritmos de busca *on-line* implementados no *RunUO* teve como objetivo avaliar qualitativamente, no fluxo do jogo, a capacidade de um agente *NPC* determinar sua trajetória e executá-la.

O *Ultima Online* apresenta situações onde o deslocamento dos *NPCs* é dirigido por algoritmo de busca. Os movimentos de animais de estimação do jogador, denominados *pets*, representam uma destas situações. Os *pets* são agentes *NPCs* que se descolam pelo mapa entre obstáculos em busca de um determinado alvo, o qual pode ser um agressor, um alvo de perseguição móvel ou o seu respectivo dono, o jogador, Figura 17.

O algoritmo de busca utilizado pelo emulador *RunUO* é o A*, com área de busca bastante limitada (38 x 38 quadrantes). O mapa completo do jogo possui 6144 x 4096 quadrantes, sendo que cada um possui até oito vizinhos. A restrição na área de busca é motivada pelo ônus no tempo de resposta e de recursos computacionais.

Ao analisar os algoritmos de busca do *RunUO*, os quais se comportam de forma equivalente aos servidores oficiais de *Ultima Online*, constatamos a combinação de dois algoritmos. O primeiro algoritmo é executado quando não há obstáculos entre o *NPC* e o seu alvo. Este algoritmo é uma simples tentativa de andar em linha reta na direção do alvo, intercalando com movimentos aleatórios quando é necessário um desvio simples. Nos casos onde essa tentativa não obtém sucesso, um algoritmo A* é executado. Se um obstáculo intransponível surge repentinamente na rota do *NPC*, a busca é cancelada e o primeiro algoritmo é executado novamente.

Os algoritmos de busca *on-line* são capazes de encontrar um caminho até o objetivo, se o mesmo existir. O processamento dessa busca não ocasiona lentidão perceptível no servidor. O A* também resolve a busca em todo o mapa do jogo se ampliamos sua área de busca. Contudo percebemos que as respostas do jogo ficam criticamente lentas, principalmente quando mais de vinte agentes executam a busca *off-line* simultaneamente.

A vantagem dos algoritmos *off-line* em relação aos *on-line* é a ótima solução com o menor número de passos entre dois pontos. Por esse motivo, a busca *off-line* é utilizada em grande parte dos jogos. Porém, quando o

jogo possui características de um sistema em tempo real, como os *MMORPG* e *RTS (real-time strategy)*, a busca *off-line* compromete a sua cadência.

A utilização de busca *off-line* consome maior tempo de processamento para o primeiro passo do agente do que a busca *on-line*. Na busca *off-line*, toda a rota é calculada antes do primeiro passo do agente, enquanto que na busca *on-line* cada passo pode ser planejado e executado logo em seguida. Pelo fato dos jogos digitais em tempo real terem características dinâmicas, os alvos também se movimentam e as rotas se modificam, inutilizando grande parte do cálculo da busca *off-line*, a qual deve ser refeita.

O *RunUO* utiliza algoritmo de busca *off-line* com área de busca bastante limitada em torno do agente (Figura 18a) e ignora os obstáculos dinâmicos do trajeto calculado. Porém esses ajustes no algoritmo provocam dois problemas de jogabilidade. O primeiro é a impossibilidade de calcular uma rota quando há obstáculos fixos seccionando a área de busca do agente, Figura 18b. O segundo é a sobreposição de agentes durante e após o caminhar, impossibilitando a seleção unitária dos mesmos para demais atividades *in-game*, Figura 18d.

Implementamos no *RunUO* os algoritmos de busca *on-line* RTA* e LRTA* com o intuito de observar, na perspectiva do jogador, o diferencial em relação ao uso do A*.

Nos experimentos, diferentemente dos pseudocódigos da literatura, não pré-calculamos as heurísticas de cada vértice do mapa, pois no *Ultima Online* cada NPC necessitaria calcular e armazenar 6144 x 4096 valores heurísticos cada vez que uma nova busca fosse iniciada. Então optamos por calcular as heurísticas necessárias em tempo de execução e associamos a cada agente uma árvore binária para armazenar apenas as heurísticas atualizadas durante a busca. Essa árvore começa vazia e à medida que o agente se desloca, as atualizações heurísticas são inseridas na árvore. Quando o agente alcança o seu objetivo ou quando o alvo se desloca, a busca é dada como encerrada ou se prepara para calcular outro trajeto. Nesse instante a árvore binária é reiniciada.

Observamos que os algoritmos *on-line* resolvem os problemas gerados pela utilização de busca *off-line* com área limitada. Os agentes são capazes de caminhar até o seu alvo sempre que existir um caminho válido em qualquer parte do mapa e os agentes jamais se sobrepõem, Figura 18e. Essas soluções melhoram consideravelmente a jogabilidade com os *pets*.

Um problema observado com os algoritmos *on-line* ocorre quando há muitos obstáculos na vizinhança dos NPCs formando uma área côncava. Nesse caso os NPCs parecem, por um momento, estarem perdidos. Este efeito acontece porque os movimentos dos *pets* acompanham o planejamento do algoritmo. Apesar

dessa desvantagem, os agentes encontram seu objetivo, Figura 18c. Este problema pode ser resolvido calibrando o número de iterações que o agente deve planejar a rota antes de se deslocar efetivamente no ambiente do jogo.



Figura 17: Um jogador de *Ultima Online* e seus *pets*.

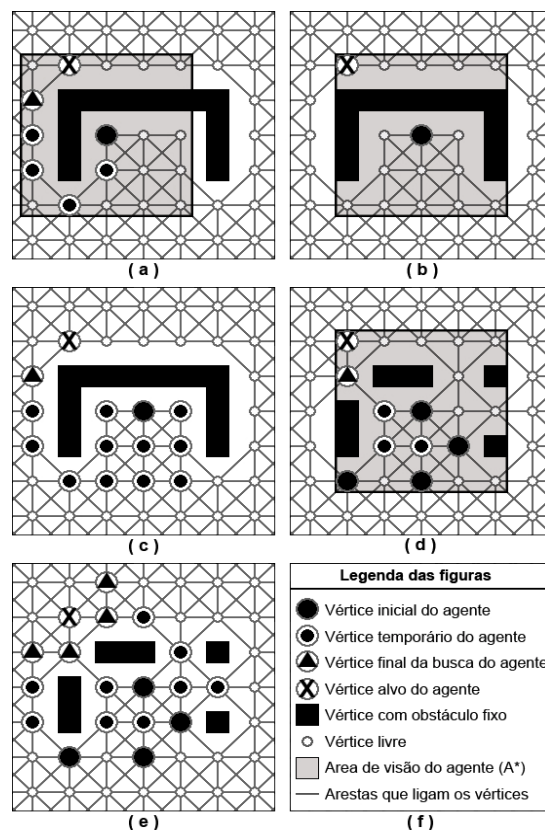


Figura 18: Comparações entre algoritmos *off-line* (a, b, d) e *on-line* (c, e) em situações observadas no *RunUO*.

5. Considerações Finais

Em jogos digitais existem situações onde é necessária a utilização de algoritmos de busca. A maioria dos jogos eletrônicos resolve o problema de busca com soluções baseadas no algoritmo A*. Os algoritmos de busca *off-line* consomem mais recursos que os algoritmos de

busca *on-line*. A utilização de algoritmos de busca *on-line* em jogos digitais economiza processamento e memória viabilizando a busca simultânea de vários agentes em ambientes mais complexos. Outra vantagem é que os algoritmos de busca *on-line* encontram a solução mesmo em ambientes dinâmicos sem a necessidade de recalculer toda a trajetória.

O RRTA* (*Relay Real Time A**) distribui a busca em mais de um agente e é útil em situações onde a busca possa ser dividida, em jogos digitais, com vários jogadores podemos distribuir a busca e também utilizar este algoritmo em situações de perseguição. Apesar do RRTA* apresentar números que indicam que ele é menos eficaz que o RTA*, ele encontra caminhos para mais de um agente. Para conseguir o mesmo resultado, o RTA* teria que executar uma busca para cada agente.

Quando dispomos de heurísticas adequadas o RTA* é mais eficiente que o RTA*P. Entretanto existem situações em que não possuímos heurísticas, por exemplo, em alguns jogos digitais não conhecemos a posição do vértice objetivo. Para estas situações o RTA*P é mais eficiente que o RTA*. O RTA*P é mais estável que o RTA*, em grafos 4-conexos o número de vértices expandidos e o tempo de execução do RTA*P sofrem menos variação que os do RTA* quando acrescentamos, retiramos ou apenas modificamos a heurística. Quando o RTA* apresenta-se mais eficiente que o RTA*P esta diferença não é grande, por isso a utilização do RTA*P em jogos digitais é indicada.

O emulador *RunUO* é um ambiente adequado para teste de algoritmos de busca em jogos pois mostrou ser suficientemente completo e flexível.

Aplicar algoritmos de busca *on-line* em jogos digitais é motivado pelo fato desses experimentos nos levarem a uma melhor observação do comportamento dos agentes da inteligência artificial (IA) em situações diversas. Tais observações são inspirações para novas idéias de implementações. A percepção humana sobre a inteligência de um agente de IA pode ser presumida quando há uma imersão no universo de ação do mesmo, essa especificidade é bastante comum nos jogos. Os algoritmos *on-line* são eficientes nos jogos possibilitando a busca em ambientes dinâmicos e provendo a possibilidade de executar a busca para um número maior de agentes.

5.1 Trabalhos Futuros

Propusemos alguns algoritmos e estudamos o comportamento de algoritmos de busca em tempo real em jogos digitais. Contudo ainda é necessário avaliar estes algoritmos em outros jogos e em contextos diferentes.

A estratégia de posicionamento do RRTA* é fundamental para sua eficiência, é necessário estudar outras estratégias e validá-las no contexto de jogos

digitais além de explorar o potencial do RRTA* em situações de perseguição.

É importante investigar o comportamento de outros algoritmos de busca *on-line* em jogos eletrônicos e estudar a complexidade e a completude dos algoritmos propostos.

Referências

- Bulitko, V., Sturtevant, N., Lu, J., & Yau, T. 2007. *Graph Abstraction in Real-time Heuristic Search*. *Journal of Artificial Intelligence Research JAIR*
- Koenig, S. 2006. *Real-Time Adaptive A**. *AAMAS*
- Koenig, S. 1999. *Exploring unknown environments with real-time search or reinforcement learning*. In *Proceedings of the Neural Information Processing Systems NIPS*.
- Koenig, S., Likhachev, M., and Sun, X., 2007. *Speeding up Moving-Target Search*. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems AAMAS*.
- Korf, R. E. 1990. *Real-time heuristic search*. *Artificial Intelligence*, 189–21.
- Millington, I. 2006. *Artificial Intelligence for Games*. *Elsevier*.
- Nonnenmacher, V., Ferreira, S. and Osorio, F. 2007. *A Constelação: Extensões do A* para aperfeiçoar a IA de NPCs em jogos*. *SBGames*
- Origin – Ultima Online. *Play guide starting up em: http://guide.uo.com/start3d_welcome.html* [Acessado 8 de Agosto de 2008].
- Pottinger, D. C. 2000. *Terrain analysis in real-time strategy games*. In *Proceedings of Computer Game Developers Conference GDC*.
- Rabin, S. 1995. *AI Game Programming Wisdom*.
- Russel, S. and Norvig, P. 2004. *Inteligência Artificial*. *Campus*.
- Sun, X., Koenig, G., and Yeoh, W.. 2008. *Generalized Adaptive A**. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*
- Van Waveren, J. M. P. 2001. *The Quake III Arena Bot. Master's thesis, Delft University of Technology, Delft, Netherlands*.
- Wikipedia - *Ultima Online em: http://en.wikipedia.org/wiki/Ultima_Online* [Acessado 8 de Agosto de 2008].
- Wooldridge, M. 2002. *An Introduction to Multiagent Systems*. *John Wiley Sons, LTD*
- Yokoo, M. And Ishida, T. 1999. *Search Algorithms for Agents, Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence 1999*, 36-44.